Бесплатная электронная книга

УЧУСЬ

Java Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#java

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: java-language

It is an unofficial and free Java Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Java Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Java Language

замечания

Язык программирования Java ...

- **Универсальный** : он предназначен для использования для написания программного обеспечения в самых разных областях приложений и не имеет специализированных функций для любого конкретного домена.
- **Class-based** : его структура объектов определена в классах. У экземпляров классов всегда есть те поля и методы, которые указаны в их определениях классов (см. Классы и объекты). Это противоречит неклассическим языкам, таким как JavaScript.
- **Статически типизированный** : компилятор проверяет во время компиляции, что типы переменных соблюдаются. Например, если метод ожидает аргумент типа `String` , этот аргумент на самом деле должен быть строкой при вызове метода.
- **Объектно-ориентированная** : большинство вещей в программе Java являются экземплярами класса, то есть пучками состояний (полей) и поведения (методы, которые работают с данными и образуют *интерфейс* объекта для внешнего мира).
- **Portable** : он может быть скомпилирован на любой платформе с помощью `javac` и полученные файлы классов могут запускаться на любой платформе с JVM.

Java предназначен для того, чтобы позволить разработчикам приложений «писать один раз, работать где угодно» (WORA), что означает, что скомпилированный Java-код может работать на всех платформах, поддерживающих Java, без необходимости перекомпиляции.

Java-код скомпилирован в байт-код (`.class`), которые, в свою очередь, интерпретируются виртуальной машиной Java (JVM). Теоретически байт-код, созданный одним компилятором Java, должен работать одинаково на любом JVM, даже на другом компьютере. JVM может (и в реальных программах) выбирать для компиляции в собственные машинные команды части байт-кода, которые выполняются часто. Это называется компиляцией «точно в срок» (JIT) ».

Java-версии и версии

Существует три «издания» Java, определенные Sun / Oracle:

- *Java Standard Edition (SE)* - это издание, предназначенное для общего использования.
- *Java Enterprise Edition (EE)* добавляет ряд возможностей для создания сервисов

уровня предприятия на Java. Java EE рассматривается отдельно .
- *Java Micro Edition (ME)* основана на подмножестве *Java SE* и предназначена для использования на небольших устройствах с ограниченными ресурсами.

Существует отдельная тема для выпусков Java SE / EE / ME .

Каждое издание имеет несколько версий. Ниже перечислены версии Java SE.

Установка Java

Существует отдельная тема по установке Java (стандартная версия) .

Компиляция и запуск Java-программ

Существуют отдельные темы:

- Компиляция исходного кода Java
- Развертывание Java, включая создание JAR-файлов
- Запуск приложений Java
- Путь Класса

Что дальше?

Вот ссылки на темы, чтобы продолжить изучение и понимание языка программирования Java. Эти темы - основы программирования Java, чтобы вы начали.

- Примитивные типы данных в Java
- Операторы в Java
- Строки в Java
- Основные элементы управления в Java
- Классы и объекты в Java
- Массивы в Java
- Стандарты кода Java

тестирование

Хотя Java не имеет поддержки для тестирования в стандартной библиотеке, существуют сторонние библиотеки, которые предназначены для поддержки тестирования. Две наиболее популярные библиотеки для тестирования модулей:

- JUnit (официальный сайт)
- TestNG (официальный сайт)

Другой

- Шаблоны проектирования для Java описаны в шаблонах проектирования .
- Программирование для Android распространяется на Android .
- Технологии Java Enterprise Edition описаны в Java EE .
- Технологии Oracle JavaFX рассматриваются в JavaFX .

1. В разделе « **Версии** » дата *окончания срока службы (бесплатно)* заключается в том, что Oracle перестанет публиковать дальнейшие обновления Java SE на своих общедоступных сайтах загрузки. Клиенты, которым требуется постоянный доступ к критическим исправлениям ошибок и исправлениям безопасности, а также общее обслуживание Java SE, могут получить долгосрочную поддержку через поддержку Oracle Java SE .

Версии

Версия Java SE	Кодовое имя	Окончание срока службы (бесплатно [1])	Дата выхода
Java SE 9 (ранний доступ)	*Никто*	будущее	2017-07-27
Java SE 8	паук	будущее	2014-03-18
Java SE 7	дельфин	2015-04-14	2011-07-28
Java SE 6	мустанг	2013-04-16	2006-12-23
Java SE 5	тигр	2009-11-04	2004-10-04
Java SE 1.4	Мерлин	до 2009-11-04	2002-02-06
Java SE 1.3	Пустельга	до 2009-11-04	2000-05-08
Java SE 1.2	Детская площадка	до 2009-11-04	1998-12-08
Java SE 1.1	*Никто*	до 2009-11-04	1997-02-19
Java SE 1.0	дуб	до 2009-11-04	1996-01-21

Examples

Создание первой программы Java

Создайте новый файл в текстовом редакторе или в IDE с именем `HelloWorld.java` . Затем вставьте этот блок кода в файл и сохраните:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Запуск в прямом эфире на Ideone

Примечание. Если Java распознает это как `public class` (а не выдает ошибку времени компиляции), имя файла должно совпадать с именем класса (`HelloWorld` в этом примере) с расширением `.java` . Перед этим также должен быть модификатор `public` доступа.

Соглашения об именах рекомендуют, чтобы классы Java начинались с символа верхнего регистра и находились в формате case camel (в котором первая буква каждого слова заглавная). Эти соглашения рекомендуются против знаков подчеркивания (`_`) и доллара (`$`).

Чтобы скомпилировать, откройте окно терминала и перейдите в каталог `HelloWorld.java` :

```
cd /path/to/containing/folder/
```

Примечание: cd - это команда терминала для изменения каталога.

Введите `javac` а затем имя файла и расширение следующим образом:

```
$ javac HelloWorld.java
```

Общеизвестно, что ошибка `'javac' is not recognized as an internal or external command, operable program or batch file.` даже если вы установили `JDK` и сможете запускать программу из `IDE` ех. `eclipse` и т. д. Так как путь по умолчанию не добавляется в среду.

В случае, если вы получите это на окнах, чтобы решить проблему, сначала попробуйте перейти на ваш путь `javac.exe` , скорее всего, это ваш `C:\Program Files\Java\jdk(version number)\bin` . Затем попробуйте запустить его ниже.

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

Раньше, когда мы `javac` он был таким же, как и команда выше. Только в этом случае ваша `OS` знала, где находится `javac` . Итак, давайте расскажем об этом сейчас, поэтому вам не нужно вводить весь путь каждый раз. Нам нужно добавить это к нашей `PATH`

Чтобы изменить `PATH` среды `PATH` в Windows XP / Vista / 7/8/10:

- Панель управления ⇒ Система ⇒ Расширенные настройки системы
- Переключиться на вкладку «Дополнительно» ⇒ Переменные среды
- В «Системные переменные» прокрутите вниз, чтобы выбрать «ПУТЬ» ⇒ Изменить

Вы не можете отменить это, поэтому будьте осторожны. Сначала скопируйте существующий путь в блокнот. Затем, чтобы получить точную PATH для вашего `javac` просмотрите вручную папку, в которой находится `javac` и щелкните по адресной строке, а затем скопируйте ее. Он должен выглядеть примерно так `c:\Program Files\Java\jdk1.8.0_xx\bin`

В поле «Variable value» вставьте этот **IN FRONT** из всех существующих каталогов, за которым следует точка с запятой (;). **НЕ УДАЛЯЙТЕ** любые существующие записи.

```
Variable name  : PATH
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

Теперь это должно решить.

Для Linux-систем попробуйте здесь .

Примечание. *Команда `javac` вызывает компилятор Java.*

Затем компилятор сгенерирует файл байт-кода `HelloWorld.class` который может быть запущен на виртуальной машине Java (JVM) . Компилятор языка программирования Java, `javac` , читает исходные файлы, написанные на языке программирования Java, и компилирует их в файлы классов `bytecode` . При желании компилятор также может обрабатывать аннотации, найденные в исходных и классных файлах, с помощью API `Pluggable` Annotation Processing. Компилятор является инструментом командной строки, но также может быть вызван с использованием Java Compiler API.

Чтобы запустить вашу программу, введите `java` а затем имя класса, которое содержит `main` метод (`HelloWorld` в нашем примере). Обратите внимание, что `.class` опущен:

```
$ java HelloWorld
```

Примечание. *Команда `java` запускает приложение Java.*

Это будет выводиться на консоль:

    Привет, мир!

Вы успешно закодировали и создали свою первую Java-программу!

Примечание. Чтобы распознавать Java-команды (`java` , `javac` и т. Д.), Вам необходимо убедиться:

- Установлен JDK (например, Oracle , OpenJDK и другие источники)

- Ваши переменные среды правильно настроены

Вам нужно будет использовать компилятор (`javac`) и исполнитель (`java`), предоставленный вашей JVM. Чтобы узнать, какие версии вы установили, введите `java -version` и `javac -version` в командной строке. Номер версии вашей программы будет напечатан в терминале (например, `1.8.0_73`).

Более пристальный взгляд на программу Hello World

Программа Hello World содержит один файл, который состоит из определения класса `HelloWorld` , `main` метода и оператора внутри `main` метода.

```
public class HelloWorld {
```

Ключевое слово `class` начинает определение класса для класса с именем `HelloWorld` . Каждое приложение Java содержит хотя бы одно определение класса (дополнительная информация о классах).

```
    public static void main(String[] args)  {
```

Это метод точки входа (определяемый его именем и сигнатурой `public static void main(String[])`), из которого `JVM` может запускать вашу программу. Каждая программа Java должна иметь один. Это:

- `public` : это означает, что метод может быть вызван из любой точки мира извне программы. См. « Видимость» для получения дополнительной информации об этом.
- `static` : означает, что он существует и может выполняться сам по себе (на уровне класса без создания объекта).
- `void` : означает, что он не возвращает значение. ***Примечание.*** *Это не похоже на C и C ++, где ожидается код возврата, такой как* `int` *(путь Java -* `System.exit()` *).*

Этот основной метод позволяет:

- Массив (обычно называемый `args`) `String` s передается как аргументы основной функции (например, из аргументов командной строки).

Почти все это требуется для метода точки входа Java.

Необязательные детали:

- Имя `args` - это имя переменной, поэтому его можно назвать чем угодно, хотя обычно его называют `args` .

- Является ли его тип параметра массивом (`String[] args`) или Varargs (`String... args`), не имеет значения, потому что массивы могут быть переданы в varargs.

Примечание. В одном приложении может быть несколько классов, содержащих метод точки входа (`main`). Точка входа приложения определяется именем класса, переданным в качестве аргумента в команду `java` .

Внутри основного метода мы видим следующее утверждение:

```
        System.out.println("Hello, World!");
```

Давайте разберем эту инструкцию по элементам:

Элемент	Цель
`System`	это означает, что последующее выражение вызовет класс `System` из пакета `java.lang` .
`.`	это «точечный оператор». Операторы точек предоставляют вам доступ к членам классов [1] ; т.е. его поля (переменные) и его методы. В этом случае этот оператор точки позволяет ссылаться на `out` статического поля в `System` класса.
`out`	это имя статического поля типа `PrintStream` внутри класса `System` содержащего стандартные функции вывода.
`.`	это еще один оператор точки. Этот оператор точки обеспечивает доступ к методу `println` в переменной `out` .
`println`	это имя метода в классе PrintStream. Этот метод, в частности, печатает содержимое параметров в консоли и вставляет новую строку после.
`(`	эта скобка указывает, что к способу обращается (а не к полю) и начинает параметры передаются в метод `println` .
`"Hello, World!"`	это строковый литерал, который передается как параметр, в метод `println` . Двойные кавычки на каждом конце ограничивают текст как String.
`)`	эта скобка означает закрытие параметров, передаваемых в метод `println` .
`;`	эта точка с запятой знаменует конец утверждения.

Примечание. *Каждый оператор в Java должен заканчиваться точкой с запятой (;).*

Тело метода и тело класса затем закрываются.

```
    }  // end of main function scope
}      // end of class HelloWorld scope
```

Вот еще один пример, демонстрирующий парадигму ОО. Давайте моделируем футбольную команду с одним (да, одним!) Участником. Их может быть больше, но мы обсудим это, когда мы перейдем к массивам.

Сначала давайте определим наш класс `Team` :

```
public class Team {
    Member member;
    public Team(Member member) {  // who is in this Team?
        this.member = member;  //  one 'member' is in this Team!
    }
}
```

Теперь давайте определим наш `Member` класса:

```
class Member {
    private String name;
    private String type;
    private int level; // note the data type here
    private int rank; // note the data type here as well

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }
}
```

Почему мы здесь используем `private` ? Ну, если кто-то хочет узнать ваше имя, они должны попросить вас прямо, вместо того, чтобы влезть в карман и вытащить карту социального страхования. Это `private` делает что-то вроде этого: оно предотвращает доступ внешних объектов к вашим переменным. Вы можете возвращать только `private` членов через функции getter (показано ниже).

Положив все это вместе и добавив геттеры и основной метод, как обсуждалось ранее, мы имеем:

```
public class Team {
    Member member;
    public Team(Member member) {
        this.member = member;
    }

    // here's our main method
    public static void main(String[] args) {
       Member myMember = new Member("Aurieel", "light", 10, 1);
       Team myTeam = new Team(myMember);
       System.out.println(myTeam.member.getName());
```

```
        System.out.println(myTeam.member.getType());
        System.out.println(myTeam.member.getLevel());
        System.out.println(myTeam.member.getRank());
    }
}

class Member {
    private String name;
    private String type;
    private int level;
    private int rank;

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }

    /* let's define our getter functions here */
    public String getName() { // what is your name?
        return this.name; // my name is ...
    }

    public String getType() { // what is your type?
        return this.type; // my type is ...
    }

    public int getLevel() { // what is your level?
        return this.level; // my level is ...
    }

    public int getRank() { // what is your rank?
        return this.rank; // my rank is
    }
}
```

Выход:

```
Aurieel
light
10
1
```

Запуск на идеон

Еще раз, `main` метод внутри класса `Test` является точкой входа в нашу программу. Без `main` метода мы не можем сообщить виртуальной машине Java (JVM), откуда начать выполнение программы.

---

1 - Поскольку класс `HelloWorld` мало `HelloWorld` классом `System` , он может получать доступ только к `public` данным.

Прочитайте Начало работы с Java Language онлайн: https://riptutorial.com/ru/java/topic/84/начало-работы-с-java-language

глава 2: 2D-графика в Java

Вступление

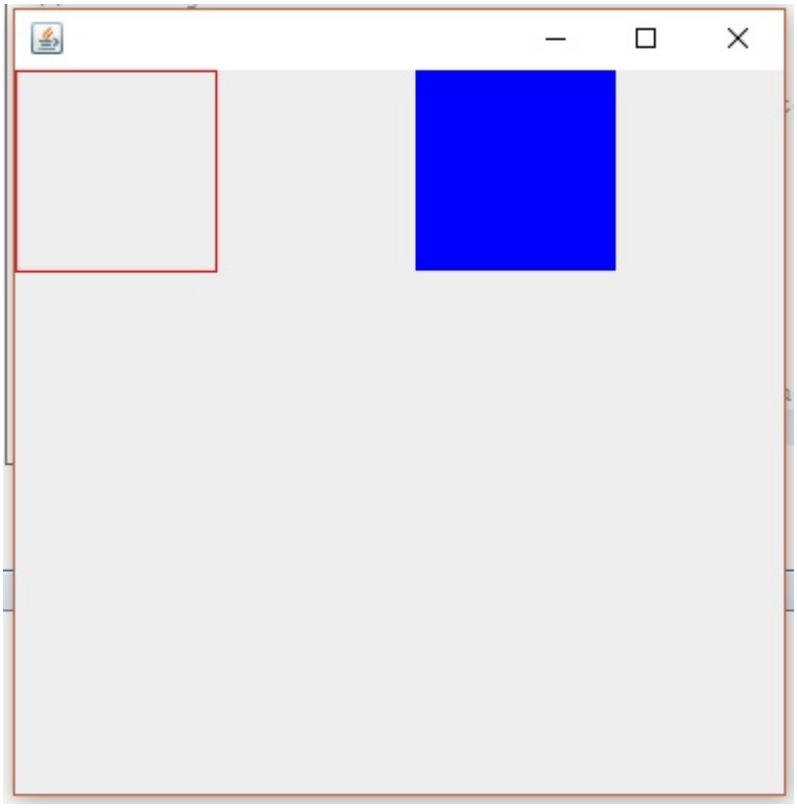
Графика - это визуальные изображения или рисунки на какой-либо поверхности, такие как стена, холст, экран, бумага или камень, чтобы сообщать, иллюстрировать или развлекать. Он включает в себя: графическое представление данных, как при автоматизированном проектировании и производстве, в наборе и графике, а также в учебном и развлекательном программном обеспечении. Изображения, созданные компьютером, называются компьютерной графикой.

Java 2D API является мощным и сложным. Существует несколько способов сделать 2D-графику на Java.

Examples

Пример 1: Рисование и заливка прямоугольника с использованием Java

Это пример, который печатает прямоугольник и заполняет цвет в прямоугольнике.



https://i.stack.imgur.com/dlC5v.jpg

Большинство методов класса Graphics можно разделить на две основные группы:

1. Методы рисования и заполнения, позволяющие отображать основные фигуры, текст и изображения

2. Методы настройки атрибутов, которые влияют на то, как отображается этот рисунок и заполнение

Пример кода. Давайте начнем с небольшого примера рисования прямоугольника и заполнения цвета в нем. Там мы объявляем два класса, один класс - MyPanel, а другой - Test. В классе MyPanel мы используем drawRect () и fillRect () mathods для рисования прямоугольника и заполнения цвета в нем. Мы устанавливаем цвет методом setColor (Color.blue). Во втором классе мы тестируем нашу графику, которая является тестовым классом, мы создаем фрейм и помещаем MyPanel с p = новым объектом MyPanel () в нем. При запуске Test Class мы видим прямоугольник и синий цветный прямоугольник.

Первый класс: MyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel extends JPanel, which will eventually be placed in a JFrame
public class MyPanel extends JPanel {
    // custom painting is performed by the paintComponent method
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        // cast Graphics to Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.red); // sets Graphics2D color
        // draw the rectangle
        g2.drawRect(0,0,100,100); // drawRect(x-position, y-position, width, height)
        g2.setColor(Color.blue);
        g2.fillRect(200,0,100,100); // fill new rectangle with color blue
    }
}
```

Второй класс: тест

```
import javax.swing.;
import java.awt.;
public class Test { //the Class by which we display our rectangle
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // get the content area of Panel.
        Container c = f.getContentPane();
        // set the LayoutManager
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // add MyPanel object into container
        c.add(p);
        // set the size of the JFrame
        f.setSize(400,400);
        // make the JFrame visible
        f.setVisible(true);
        // sets close behavior; EXIT_ON_CLOSE invokes System.exit(0) on closing the JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

```
    public static void main(String args[ ]){
        Test t = new Test();
    }
}
```

Дополнительные пояснения относительно макета границы:
https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent ()

- Это основной метод рисования
- По умолчанию он сначала рисует фон
- После этого он выполняет обычную роспись (круг рисования, прямоугольники и т. Д.),

Graphic2D относится к классу Graphic2D

Примечание . Java 2D API позволяет вам легко выполнять следующие задачи:

- Рисуйте линии, прямоугольники и любую другую геометрическую форму.
- Заполните эти фигуры сплошными цветами или градиентами и текстурами.
- Нарисуйте текст с опциями для тонкого контроля над шрифтом и процессом рендеринга.
- Нарисуйте изображения, опционально применяя операции фильтрации.
- Применяйте операции, такие как компоновка и преобразование во время любой из вышеперечисленных операций рендеринга.

Пример 2: Обивка для рисования и наполнения

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20,20);
        g2.fillOval(50,50,20,20);
    }
}
```

g2.drawOval (int x, int y, int height, int width);
Этот метод будет рисовать овал в указанных положениях x и y с заданной высотой и шириной.

g2.fillOval (int x, int y, int height, int width); Этот метод будет заполнять овал в указанных положениях x и y с заданной высотой и шириной.

Прочитайте 2D-графика в Java онлайн: https://riptutorial.com/ru/java/topic/10127/2d-графика-в-java

глава 3: Apache Commons Lang

Examples

Внедрить метод equals ()

Чтобы реализовать метод `equals` объекта легко, вы можете использовать класс `EqualsBuilder` .

Выбор полей:

```
@Override
public boolean equals(Object obj) {

    if(!(obj instanceof MyClass)) {
        return false;
    }
    MyClass theOther = (MyClass) obj;

    EqualsBuilder builder = new EqualsBuilder();
    builder.append(field1, theOther.field1);
    builder.append(field2, theOther.field2);
    builder.append(field3, theOther.field3);

    return builder.isEquals();
}
```

Использование отражения:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, false);
}
```

логический параметр указывает, должны ли равны проверять переходные поля.

Используя отражение, избегая некоторых полей:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");
}
```

Внедрить метод hashCode ()

Чтобы реализовать метод `hashCode` объекта легко, вы можете использовать класс `HashCodeBuilder` .

Выбор полей:

```
@Override
public int hashCode() {

    HashCodeBuilder builder = new HashCodeBuilder();
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.hashCode();
}
```

Использование отражения:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, false);
}
```

boolean указывает, следует ли использовать преходящие поля.

Используя отражение, избегая некоторых полей:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");
}
```

Внедрить метод toString ()

Чтобы реализовать метод `toString` объекта легко, вы можете использовать класс `ToStringBuilder` .

Выбор полей:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.toString();
}
```

Пример результата:

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

Явное указание имен для полей:

```
@Override
```

```
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append("field1",field1);
    builder.append("field2",field2);
    builder.append("field3",field3);

    return builder.toString();
}
```

Пример результата:

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

Вы можете изменить стиль с помощью параметра:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this,
            ToStringStyle.MULTI_LINE_STYLE);
    builder.append("field1", field1);
    builder.append("field2", field2);
    builder.append("field3", field3);

    return builder.toString();
}
```

Пример результата:

```
ar.com.bna.lang.MyClass@ebbf5c[
  field1=<null>
  field2=0
  field3=false
]
```

Есть несколько стилей, например JSON, no Classname, short и т. Д. ...

Через отражение:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}
```

Вы также можете указать стиль:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
}
```

Прочитайте Apache Commons Lang онлайн: https://riptutorial.com/ru/java/topic/3338/apache-

commons-lang

глава 4: API Reflection

Вступление

Отражение обычно используется программами, которые требуют возможности исследовать или модифицировать поведение среды выполнения приложений, запущенных в JVM. Java Reflection API используется для этой цели, где он позволяет проверять классы, интерфейсы, поля и методы во время выполнения, не зная их имена во время компиляции. А также позволяет создавать новые объекты и вызывать методы с использованием отражения.

замечания

Спектакль

Имейте в виду, что отражение может снизить производительность, использовать его только тогда, когда ваша задача не может быть выполнена без отражения.

Из учебника Java API Reflection :

> Поскольку отражение включает типы, которые динамически разрешены, некоторые оптимизации виртуальной машины Java не могут быть выполнены. Следовательно, рефлексивные операции имеют более низкую производительность, чем их неотражающие аналоги, и их следует избегать в разделах кода, которые часто называются приложениями, чувствительными к характеристикам.

Examples

Вступление

основы

API Reflection позволяет проверять структуру класса кода во время выполнения и динамически вызывать код. Это очень мощно, но это также опасно, поскольку компилятор не может статически определять, действительны ли динамические вызовы.

Простым примером могло бы стать создание публичных конструкторов и методов данного класса:

```
import java.lang.reflect.Constructor;
```

```
import java.lang.reflect.Method;

// This is a object representing the String class (not an instance of String!)
Class<String> clazz = String.class;

Constructor<?>[] constructors = clazz.getConstructors(); // returns all public constructors of
String
Method[] methods = clazz.getMethods(); // returns all public methods from String and parents
```

С помощью этой информации можно указать объект и динамически вызвать разные методы.

Отражение и общие типы

Информация общего типа доступна для:

- параметры метода, используя `getGenericParameterTypes()` .
- методы возвращают типы, используя `getGenericReturnType()` .
- **публичные** поля, используя `getGenericType` .

В следующем примере показано, как извлечь информацию об общем типе во всех трех случаях:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("Method parameter:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "\t");

        System.out.println("Method return type:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "\t");

        System.out.println("Field type:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "\t");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "\t");
            }
```

```
        }

    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }

}
```

Это приводит к следующему результату:

```
Method parameter:
    java.util.Map<java.lang.String, java.lang.Double>
        java.lang.String
        java.lang.Double
Method return type:
    java.util.List<java.lang.Number>
        java.lang.Number
Field type:
    java.util.Map<java.lang.String, java.util.Map<java.lang.Integer,
java.util.List<java.lang.String>>>
        java.lang.String
        java.util.Map<java.lang.Integer, java.util.List<java.lang.String>>
            java.lang.Integer
            java.util.List<java.lang.String>
                java.lang.String
```

Вызов метода

Используя отражение, метод объекта может быть вызван во время выполнения.

В этом примере показано, как вызвать методы объекта `String` .

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

String s = "Hello World!";

// method without parameters
// invoke s.length()
Method method1 = String.class.getMethod("length");
int length = (int) method1.invoke(s); // variable length contains "12"

// method with parameters
// invoke s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // variable substring contains "World!"
```

Получение и настройка полей

Используя Reflection API, можно изменить или получить значение поля во время выполнения. Например, вы можете использовать его в API для извлечения разных полей

на основе фактора, например ОС. Вы также можете удалить модификаторы, такие как `final` чтобы поля модификации были окончательными.

Для этого вам нужно будет использовать метод Class # getField () таким образом, который показан ниже:

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Get the field in class Field "modifiers". Note that it does not
// need to be static
Field modifiersField = Field.class.getDeclaredField("modifiers");

// Allow access from anyone even if it's declared private
modifiersField.setAccessible(true);

// Get the modifiers on the "NAME" field as an int.
int existingModifiersOnNameField = nameField.getModifiers();

// Bitwise AND NOT Modifier.FINAL (16) on the existing modifiers
// Readup here https://en.wikipedia.org/wiki/Bitwise_operations_in_C
// if you're unsure what bitwise operations are.
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// Set the value of the modifiers field under an object for non-static fields
modifiersField.setInt(nameField, newModifiersOnNameField);

// Set it to be accessible. This overrides normal Java
// private/protected/package/etc access control checks.
nameField.setAccessible(true);

 // Set the value of "NAME" here. Note the null argument.
 // Pass null when modifying static fields, as there is no instance object
nameField.set(null, "Hacked by reflection...");

// Here I can directly access it. If needed, use reflection to get it. (Below)
System.out.println(SomeClass.NAME);
```

Получение полей намного проще. Мы можем использовать Field # get () и его варианты, чтобы получить его значение:

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Set accessible for private fields
nameField.setAccessible(true);

// Pass null as there is no instance, remember?
String name = (String) nameField.get(null);
```

Обратите внимание:

При использовании класса # getDeclaredField используйте его для получения поля в самом классе:

```
class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}
```

Здесь `HackMe#iAmDeclared` является объявленным полем. Однако `HackMe#someState` не является объявленным полем, поскольку он унаследован от своего суперкласса Hacked.

Конструктор вызовов

Получение объекта-конструктора

Класс `Constructor` можно получить из объекта `Class` следующим образом:

```
Class myClass = ... // get a class object
Constructor[] constructors = myClass.getConstructors();
```

Если переменная `constructors` будет иметь один экземпляр `Constructor` для каждого публичного конструктора, объявленного в классе.

Если вы знаете точные типы параметров конструктора, к которому хотите получить доступ, вы можете отфильтровать конкретный конструктор. Следующий пример возвращает публичный конструктор данного класса, который принимает параметр `Integer` as:

```
Class myClass = ... // get a class object
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

Если конструктор не соответствует заданным аргументам конструктора, `NoSuchMethodException` .

Новый экземпляр с использованием объекта «Конструктор»

```
Class myClass = MyObj.class // get a class object
Constructor constructor = myClass.getConstructor(Integer.class);
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

Получение констант перечисления

Предоставление этого перечисления в качестве примера:

```
enum Compass {
    NORTH(0),
```

```
    EAST(90),
    SOUTH(180),
    WEST(270);
    private int degree;
    Compass(int deg){
        degree = deg;
    }
    public int getDegree(){
        return degree;
    }
}
```

В Java класс enum похож на любой другой класс, но имеет определенные определенные константы для значений enum. Кроме того, у него есть поле, которое представляет собой массив, который содержит все значения и два статических метода с именами `values()` и `valueOf(String)` .
Мы можем это увидеть, если мы используем Reflection для печати всех полей этого класса

```
for(Field f : Compass.class.getDeclaredFields())
    System.out.println(f.getName());
```

выход будет:

К СЕВЕРУ
ВОСТОК
ЮЖНЫЙ
WEST
степень
ENUM $ ЗНАЧЕНИЯ

Поэтому мы могли бы изучить классы enum с Reflection, как и любой другой класс. Но API Reflection предлагает три метода enum-specific.

проверка перечислений

```
Compass.class.isEnum();
```

Возвращает true для классов, представляющих тип перечисления.

получение значений

```
Object[] values = Compass.class.getEnumConstants();
```

Возвращает массив всех значений перечисления, таких как Compass.values (), но без необходимости экземпляра.

постоянная проверка enum

```
for(Field f : Compass.class.getDeclaredFields()){
```

```
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

Выводит список всех полей класса, которые являются значениями перечисления.

Получить класс с его (полностью квалифицированным) именем

Учитывая `String` содержащую имя класса, к объекту `Class` можно получить доступ с помощью `Class.forName` :

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Java SE 1.2

Он может быть указан, если класс должен быть инициализирован (второй параметр `forName`) и какой `ClassLoader` должен использоваться (третий параметр):

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Вызов перегруженных конструкторов с использованием отражения

Пример: вызывать разные конструкторы путем передачи соответствующих параметров

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("Default constructor");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("Constructor :String => "+a);
    }
    public static void main(String args[]) throws Exception {

        NewInstanceWithReflection object =
(NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
Class[] {String.class});
        NewInstanceWithReflection object1 =
(NewInstanceWithReflection)constructor.newInstance(new Object[]{"StackOverFlow"});
```

```
    }
}
```

выход:

```
Default constructor
Constructor :String => StackOverFlow
```

Объяснение:

1. Создать экземпляр класса с использованием `Class.forName` : он вызывает конструктор по умолчанию
2. Вызывать `getDeclaredConstructor` класса, передавая тип параметров как `Class array`
3. После получения конструктора создайте `newInstance` , передав значение параметра как `Object array`

Неправильное использование API Reflection для изменения частных и конечных переменных

Отражение полезно, когда оно правильно используется для правильной цели. Используя отражение, вы можете получить доступ к закрытым переменным и повторно инициализировать конечные переменные.

Ниже приведен фрагмент кода, который **не** рекомендуется.

```
import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){
                    field.setAccessible(true);
                    field.set(a, "StackOverFlow");
                    System.out.println("A.name="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("age")){
                    field.set(a, 20);
                    System.out.println("A.age="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("rep")){
                    field.setAccessible(true);
                    field.set(a,"New Reputation");
                    System.out.println("A.rep="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("count")){
                    field.set(a,25);
                    System.out.println("A.count="+field.get(a));
                }
            }
```

```
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class A {
    private String name;
    public int age;
    public final String rep;
    public static int count=0;

    public A(){
        name = "Unset";
        age = 0;
        rep = "Reputation";
        count++;
    }
}
```

Выход:

```
A.name=StackOverFlow
A.age=20
A.rep=New Reputation
A.count=25
```

Объяснение:

В обычном сценарии `private` переменные не могут быть доступны за пределами объявленного класса (без методов getter и setter). `final` переменные не могут быть повторно назначены после инициализации.

`Reflection` разрывов обоих барьеров можно злоупотреблять, чтобы изменить как частные, так и конечные переменные, как описано выше.

`field.setAccessible(true)` - ключ к достижению желаемой функциональности.

Конструктор вызовов вложенного класса

Если вы хотите создать экземпляр внутреннего вложенного класса, вам необходимо предоставить объект класса охватывающего класса в качестве дополнительного параметра с классом getDeclaredConstructor .

```
public class Enclosing{
    public class Nested{
    public Nested(String a){
            System.out.println("Constructor :String => "+a);
        }
    }
    public static void main(String args[]) throws Exception {
        Class<?> clazzEnclosing = Class.forName("Enclosing");
        Class<?> clazzNested = Class.forName("Enclosing$Nested");
```

```
        Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();
        Constructor<?> constructor = clazzNested.getDeclaredConstructor(new
Class[]{Enclosing.class, String.class});
        Nested objInner = (Nested)constructor.newInstance(new Object[]{objEnclosing,
"StackOverFlow"});
    }
}
```

Если вложенный класс является статическим, вам не понадобится этот закрытый экземпляр.

Динамические прокси

Динамические прокси-серверы не имеют особого отношения к Reflection, но они являются частью API. Это в основном способ создания динамической реализации интерфейса. Это может быть полезно при создании макетов.
Динамический прокси - это экземпляр интерфейса, который создается с помощью так называемого обработчика вызовов, который перехватывает все вызовы методов и позволяет обрабатывать их вызов вручную.

```
public class DynamicProxyTest {

    public interface MyInterface1{
        public void someMethod1();
        public int someMethod2(String s);
    }

    public interface MyInterface2{
        public void anotherMethod();
    }

    public static void main(String args[]) throws Exception {
        // the dynamic proxy class
        Class<?> proxyClass = Proxy.getProxyClass(
                ClassLoader.getSystemClassLoader(),
                new Class[] {MyInterface1.class, MyInterface2.class});
        // the dynamic proxy class constructor
        Constructor<?> proxyConstructor =
            proxyClass.getConstructor(InvocationHandler.class);

        // the invocation handler
        InvocationHandler handler = new InvocationHandler(){
            // this method is invoked for every proxy method call
            // method is the invoked method, args holds the method parameters
            // it must return the method result
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
                String methodName = method.getName();

                if(methodName.equals("someMethod1")){
                    System.out.println("someMethod1 was invoked!");
                    return null;
                }
                if(methodName.equals("someMethod2")){
                    System.out.println("someMethod2 was invoked!");
```

```
                    System.out.println("Parameter: " + args[0]);
                    return 42;
                }
                if(methodName.equals("anotherMethod")){
                    System.out.println("anotherMethod was invoked!");
                    return null;
                }
                System.out.println("Unkown method!");
                return null;
            }
        };

        // create the dynamic proxy instances
        MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
        MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

        // and invoke some methods
        i1.someMethod1();
        i1.someMethod2("stackoverflow");
        i2.anotherMethod();
    }
}
```

Результатом этого кода является следующее:

```
someMethod1 was invoked!
someMethod2 was invoked!
Parameter: stackoverflow
anotherMethod was invoked!
```

Злые Java-хаки с отражением

API Reflection можно использовать для изменения значений частных и конечных полей даже в библиотеке по умолчанию JDK. Это можно использовать для управления поведением некоторых известных классов, как мы увидим.

Что не возможно

Давайте сначала начнем с единственного ограничения, это единственное поле, которое мы не можем изменить с помощью Reflection. Это Java `SecurityManager` . Он объявлен в java.lang.System как

```
private static volatile SecurityManager security = null;
```

Но он не будет указан в классе System, если мы запустим этот код

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

Thats из-за `fieldFilterMap` в `sun.reflect.Reflection` который содержит карту и поле безопасности в `System.class` и защищает их от любого доступа с помощью Reflection. Поэтому мы не смогли деактивировать `SecurityManager` .

Сумасшедшие струны

Каждая строка Java представлена JVM как экземпляр класса `String` . Однако в некоторых ситуациях JVM экономит кучу пространства, используя тот же экземпляр для строк. Это происходит для строковых литералов, а также для строк, которые были «интернированы», вызывая `String.intern()` . Поэтому, если у вас есть `"hello"` в вашем коде несколько раз, это всегда один и тот же экземпляр объекта.

Строки должны быть неизменными, но можно использовать «злобное» отражение, чтобы изменить их. Пример ниже показывает, как мы можем изменить символы в String, заменив его поле `value` .

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[])  {
        System.out.println("hello");
    }
}
```

Таким образом, этот код напечатает «вы воняете!»

1 = 42

Та же идея может быть использована с классом Integer

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
    public static void main(String args[])  {
        System.out.println(Integer.valueOf(1));
    }
}
```

Все верно

И, согласно этому сообщению stackoverflow, мы можем использовать отражение, чтобы сделать что-то действительно злое.

```
public class Evil {
```

```
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}
```

Обратите внимание, что то, что мы делаем здесь, приведет к тому, что JVM будет вести себя необъяснимыми способами. Это очень опасно.

Прочитайте API Reflection онлайн: https://riptutorial.com/ru/java/topic/629/api-reflection

глава 5: API стека

Вступление

До Java 9 доступ к кадрам стека потоков был ограничен внутренним классом `sun.reflect.Reflection` . В частности, метод `sun.reflect.Reflection::getCallerClass` . Некоторые библиотеки полагаются на этот метод, который устарел.

Альтернативный стандарт API , теперь предоставляется в JDK 9 через `java.lang.StackWalker` класса, и предназначен , чтобы быть эффективными, позволяя ленивый доступ к кадрам стеки. Некоторые приложения могут использовать этот API для прохождения стека выполнения и фильтрации по классам.

Examples

Печать всех кадров стека текущего потока

Следующее выводит все кадры стека текущего потока:

```
1  package test;
2
3  import java.lang.StackWalker.StackFrame;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.util.List;
7  import java.util.stream.Collectors;
8
9  public class StackWalkerExample {
10
11    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
12        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
13        fooMethod.invoke(null, (Object[]) null);
14    }
15 }
16
17 class FooHelper {
18    protected static void foo() {
19        BarHelper.bar();
20    }
21 }
22
23 class BarHelper {
24    protected static void bar() {
25        List<StackFrame> stack = StackWalker.getInstance()
26                .walk((s) -> s.collect(Collectors.toList()));
27        for(StackFrame frame : stack) {
28            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
29        }
30    }
```

```
31 }
```

Выход:

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

Распечатать текущий класс вызывающего абонента

Следующее выводит текущий класс вызывающего абонента. Обратите внимание, что в этом случае `StackWalker` необходимо создать с помощью опции `RETAIN_CLASS_REFERENCE` , чтобы экземпляры `Class` сохранялись в объектах `StackFrame` . В противном случае произойдет исключение.

```
public class StackWalkerExample {

    public static void main(String[] args) {
        FooHelper.foo();
    }

}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {

System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());
    }
}
```

Выход:

```
class test.FooHelper
```

Отображение отражения и других скрытых фреймов

Пара других опций позволяет трассировать трассировки стека реализации и / или отражения. Это может быть полезно для целей отладки. Например, мы можем добавить параметр `SHOW_REFLECT_FRAMES` к экземпляру `StackWalker` при создании, чтобы также были напечатаны кадры для отражающих методов:

```
package test;

import java.lang.StackWalker.Option;
import java.lang.StackWalker.StackFrame;
```

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;
import java.util.stream.Collectors;

public class StackWalkerExample {

    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
        fooMethod.invoke(null, (Object[]) null);
    }
}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {
        // show reflection methods
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
                .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
        }
    }
}
```

Выход:

```
test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main
```

Обратите внимание, что номера строк для некоторых методов отражения могут быть недоступны, поэтому `StackFrame.getLineNumber()` может возвращать отрицательные значения.

Прочитайте API стека онлайн: https://riptutorial.com/ru/java/topic/9868/api-стека

глава 6: AppDynamics и TIBCO BusinessWorks для легкой интеграции

Вступление

Поскольку AppDynamics стремится обеспечить способ измерения производительности приложений, скорость разработки, доставки (развертывания) приложений является важным фактором в достижении успеха DevOps. Мониторинг приложения TIBCO BW с AppD обычно прост и не требует много времени, но при развертывании больших наборов приложений быстрые инструменты являются ключевыми. В этом руководстве показано, как применять все ваши приложения BW на одном шаге без изменения каждого приложения перед его развертыванием.

Examples

Пример инструментария всех приложений BW в одном шаге для Appdynamics

1. Найдите и откройте свой файл BWengine.tra TIBCO BW под TIBCO_HOME / bw / 5.12 / bin / bwengine.tra (среда Linux)

2. Найдите строку, которая гласит:

*** Общие переменные. Измените их только. ***

3. Добавьте следующую строку сразу после этого раздела tibco.deployment =% tibco.deployment%

4. Перейдите в конец файла и добавьте (замените? Вашими значениями по мере необходимости или удалите флаг, который не применяется): java.extended.properties = -javaagent: /opt/appd/current/appagent/javaagent.jar - Dappdynamics.http.proxyHost =? - Dappdynamics.http.proxyPort =? -Dappdynamics.agent.applicationName =? - Dappdynamics.agent.tierName =? -Dappdynamics.agent.nodeName =% tibco.deployment% -Dappdynamics.controller.ssl.enabled =? -Dappdynamics.controller.sslPort =? - Dappdynamics.agent.logs.dir =? -Dappdynamics.agent.runtime.dir =? - Dappdynamics.controller.hostName =? -Dappdynamics.controller.port =? - Dappdynamics.agent.accountName =? -Dappdynamics.agent.accountAccessKey =?

5. Сохраните файл и переустановите. Все ваши приложения теперь должны быть автоматизированы во время развертывания.

Прочитайте AppDynamics и TIBCO BusinessWorks для легкой интеграции онлайн: https://riptutorial.com/ru/java/topic/10602/appdynamics-и-tibco-businessworks-для-легкой-интеграции

глава 7: Autoboxing

Вступление

Autoboxing - это автоматическое преобразование, которое компилятор Java делает между примитивными типами и соответствующими классами обертки объектов. Пример, преобразование int -> Integer, double -> Double ... Если преобразование идет другим путем, это называется распаковкой. Как правило, это используется в коллекциях, которые не могут содержать объекты, отличные от объектов, где необходимы примитивные типы бокса, прежде чем устанавливать их в коллекции.

замечания

Автобоксирование может иметь проблемы с производительностью при частом использовании в вашем коде.

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
- Целое автоматическое разблокирование и авто-бокс дает проблемы с производительностью?

Examples

Использование int и Integer взаимозаменяемо

Поскольку вы используете общие типы с классами утилит, вы часто можете обнаружить, что типы номеров не очень полезны при указании в качестве типов объектов, поскольку они не равны их примитивным аналогам.

```
List<Integer> ints = new ArrayList<Integer>();
```

Java SE 7

```
List<Integer> ints = new ArrayList<>();
```

К счастью, выражения, которые оценивают `int` могут использоваться вместо `Integer` когда это необходимо.

```
for (int i = 0; i < 10; i++)
    ints.add(i);
```

`ints.add(i);` утверждение эквивалентно:

```
ints.add(Integer.valueOf(i));
```

И сохраняет свойства из значения `Integer#valueOf` например, с теми же `Integer` объектами, которые кешируются JVM, когда он находится в диапазоне кеширования чисел.

Это также относится к:

- `byte` И `Byte`
- `short` И `Short`
- `float` И `Float`
- `double` И `Double`
- `long` И `Long`
- `char` И `Character`
- `boolean` И `Boolean`

Однако следует проявлять осторожность в неоднозначных ситуациях. Рассмотрим следующий код:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
ints.remove(1); // ints is now [1, 3]
```

Интерфейс `java.util.List` содержит как метод `remove(int index)` (метод интерфейса `List`), так и `remove(Object o)` (метод, унаследованный от `java.util.Collection`). В этом случае бокс не происходит и не вызывается `remove(int index)` .

Еще один пример странного поведения кода Java, вызванного автобоксированием Целые числа со значениями в диапазоне от `-128` до `127` :

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

Это происходит потому, что `>=` оператор неявно вызывает `intValue()` который возвращает `int` while `==` сравнивает **ссылки** , а не значения `int` .

По умолчанию Java кэширует значения в диапазоне `[-128, 127]` , поэтому оператор `==` работает, потому что `Integers` в этом диапазоне ссылаются на одни и те же объекты, если их значения одинаковы. Максимальное значение кэшируемого диапазона можно определить с `-XX:AutoBoxCacheMax` опции `-XX:AutoBoxCacheMax` JVM. Итак, если вы запустите программу с помощью `-XX:AutoBoxCacheMax=1000` , следующий код напечатает `true` :

```
Integer a = 1000;
```

```
Integer b = 1000;
System.out.println(a == b); // true
```

Использование Boolean в выражении if

Из-за автоматического распаковки в операторе `if` можно использовать `Boolean` выражение:

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

Это работает `while` , `do while` и условие в операторах `for` .

Обратите внимание, что если `Boolean` равно `null` , в преобразовании будет `NullPointerException` .

Автоматическая распаковка может привести к NullPointerException

Этот код компилирует:

```
Integer arg = null;
int x = arg;
```

Но во время выполнения он будет сбой при исключении `java.lang.NullPointerException` во второй строке.

Проблема в том, что примитивный `int` не может иметь `null` значение.

Это минималистический пример, но на практике он часто проявляется в более сложных формах. `NullPointerException` не очень интуитивно понятное и часто мало помогает в поиске таких ошибок.

Положитесь на автобоксинг и автоматическую распаковку с осторожностью, убедитесь, что значения unboxed не будут иметь `null` значений во время выполнения.

Память и вычислительные накладные расходы на автобоксинг

Автобоксинг может иметь значительные накладные расходы памяти. Например:

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

как правило, потребляют значительный объем памяти (около 60 кб для 6 тыс. фактических данных).

Кроме того, целые числа в штучной упаковке обычно требуют дополнительных округлений в памяти и, таким образом, делают кэширование процессора менее эффективным. В приведенном выше примере доступ к памяти распространяется на пять разных местоположений, которые могут находиться в совершенно разных областях памяти: 1. объект `HashMap` , 2. объект `Entry[] table` , 3. объект `Entry` , 4. элемент ввода `key` слов (бокс примитивного ключа), 5. объект `value` entry (бокс примитивного значения).

```
class Example {
  int primitive; // Stored directly in the class `Example`
  Integer boxed; // Reference to another memory location
}
```

Чтение в `boxed` требует двух обращений к памяти, доступ к `primitive` только один.

При получении данных с этой карты, казалось бы, невинный код

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

эквивалентно:

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

Как правило, приведенный выше код вызывает *сбор и сбор мусора* объекта `Integer` для каждой операции `Map#get(Integer)` . (Подробнее см. Примечание ниже).

Чтобы уменьшить эти накладные расходы, несколько библиотек предлагают оптимизированные коллекции для примитивных типов, которые *не* требуют бокса. В дополнение к тому, чтобы избежать накладных расходов на бокс, для этой коллекции потребуется примерно 4 раза меньше памяти на запись. В то время как Java Hotspot *может* оптимизировать автобоксинг, работая с объектами в стеке вместо кучи, невозможно оптимизировать накладные расходы памяти и вызванную память.

В потоках Java 8 также есть оптимизированные интерфейсы для примитивных типов данных, таких как `IntStream` которые не требуют бокса.

Примечание: типичная среда выполнения Java поддерживает простой кэш `Integer` и другой примитивный объект-оболочку, который используется фабричными методами `valueOf` и автобоксированием. Для `Integer` диапазон по умолчанию этого кеша составляет от -128 до +127. Некоторые JVM предоставляют параметр командной строки JVM для изменения размера и диапазона кеша.

Различные случаи Когда Integer и int могут использоваться взаимозаменяемо

Случай 1: используется вместо аргументов метода.

Если метод требует, чтобы объект класса-оболочки был аргументом. Затем, в качестве взаимозаменяемого аргумента может быть передана переменная соответствующего примитивного типа и наоборот.

Пример:

```
int i;
Integer j;
void ex_method(Integer i)//Is a valid statement
void ex_method1(int j)//Is a valid statement
```

Случай 2: При передаче возвращаемых значений:

Когда метод возвращает примитивную переменную типа, тогда объект соответствующего класса-оболочки может быть передан как возвращаемое значение взаимозаменяемо и наоборот.

Пример:

```
int i;
Integer j;
int ex_method()
{...
return j;}//Is a valid statement
Integer ex_method1()
{...
return i;//Is a valid statement
}
```

Случай 3: При выполнении операций.

Всякий раз, когда выполняются операции над числами, переменная примитива и объект соответствующего класса-оболочки могут использоваться взаимозаменяемо.

```
int i=5;
Integer j=new Integer(7);
int k=i+j;//Is a valid statement
Integer m=i+j;//Is also a valid statement
```

Pitfall : не забудьте инициализировать или присвоить значение объекту класса-оболочки.

При использовании объекта класса-оболочки и примитивной переменной взаимозаменяемо никогда не забывать или пропустить инициализацию или присвоение значения объекту класса-оболочки иначе он может привести к исключению нулевого указателя во время

выполнения.

Пример:

```
public class Test{
    Integer i;
    int j;
    public void met()
    {j=i;//Null pointer exception
    SOP(j);
    SOP(i);}
    public static void main(String[] args)
    {Test t=new Test();
    t.go();//Null pointer exception
    }
```

В приведенном выше примере значение объекта не назначено и неинициализировано, и, таким образом, во время выполнения программа будет запущена в исключение нулевого указателя. Так же, как ясно из приведенного выше примера, значение объекта никогда не должно оставаться неинициализированным и неназначенным.

Прочитайте Autoboxing онлайн: https://riptutorial.com/ru/java/topic/138/autoboxing

глава 8: BigDecimal

Вступление

Класс BigDecimal предоставляет операции для арифметики (добавление, вычитание, умножение, деление), масштабирование, округление, сравнение, хэширование и преобразование формата. BigDecimal представляет собой неизменяемые десятичные числа с произвольной точностью. Этот класс должен использоваться при необходимости высокоточного вычисления.

Examples

Объекты BigDecimal являются неизменяемыми

Если вы хотите рассчитать с помощью BigDecimal, вы должны использовать возвращаемое значение, потому что объекты BigDecimal являются неизменяемыми:

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a will still be 42.23

BigDecimal c = a.add(b); // c will be 52.231
```

Сравнение BigDecimals

Метод `compareTo` должен использоваться для сравнения `BigDecimals` :

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0));    // a is greater, returns 1
a.compareTo(new BigDecimal(5));    // a is equal, returns 0
a.compareTo(new BigDecimal(10));   // a is less, returns -1
```

Обычно вы **не** должны использовать метод `equals` поскольку он считает, что два `BigDecimals` равны, только если они равны по стоимости и также **масштабируются** :

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5));       // value and scale are equal, returns true
a.equals(new BigDecimal(5.00));    // value is equal but scale is not, returns false
```

Математические операции с BigDecimal

В этом примере показано, как выполнять основные математические операции с помощью BigDecimals.

1.Addition

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a + b
BigDecimal result = a.add(b);
System.out.println(result);
```

Результат: 12

2.Subtraction

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a - b
BigDecimal result = a.subtract(b);
System.out.println(result);
```

Результат: 2

3.Multiplication

При умножении двух `BigDecimal` s результат будет иметь масштаб, равный сумме шкал операндов.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//Equivalent to result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

Результат: 36.89931

Чтобы изменить масштаб результата, используйте метод перегруженного множителя, который позволяет передавать `MathContext` - объект, описывающий правила для операторов, в частности, режим точности и округления результата. Дополнительные сведения о доступных режимах округления см. В документации Oracle.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//Equivalent to result = a * b
```

```
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

Результат: 36.90

4.Division

Разделение немного сложнее, чем другие арифметические операции, например, рассмотрим приведенный ниже пример:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

Мы ожидаем, что это даст нечто похожее: 0.7142857142857143, но мы получим:

Результат: **java.lang.ArithmeticException: Неограничивающее десятичное расширение; нет точного представимого десятичного результата.**

Это будет отлично работать, когда результат будет завершающим десятичным, скажем, если бы я хотел разделить 5 на 2, но для тех чисел, которые при делении будут давать не заканчивающийся результат, мы получим `ArithmeticException` . В сценарии реального мира невозможно предсказать значения, которые будут встречаться во время деления, поэтому нам нужно указать **масштаб** и **режим округления** для деления BigDecimal. Для получения дополнительной информации о режиме масштабирования и округления см. Документацию Oracle .

Например, я мог бы сделать:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a / b (Upto 10 Decimal places and Round HALF_UP)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

Результат: 0.7142857143

5.Remainder или модуль

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
```

```
System.out.println(result);
```

Результат: 5

6.Power

```
BigDecimal a = new BigDecimal("5");

//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

Результат: 9765625

7.Max

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

Результат: 7

8.Min

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

Результат: 5

9. Переместите точку влево

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 2 places left of current position
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

Результат: 52.3449843776

10.Переведите точку вправо

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 3 places right of current position
BigDecimal result = a.movePointRight(3);
System.out.println(result);
```

Результат: 5234498.43776

Существует множество дополнительных параметров и комбинаций параметров для вышеупомянутых примеров (например, существует 6 вариантов метода разделения), этот набор является неисчерпывающим списком и охватывает несколько базовых примеров.

Использование BigDecimal вместо float

В связи с тем, что тип float представлен в памяти компьютера, результаты операций с использованием этого типа могут быть неточными - некоторые значения сохраняются в виде приближений. Хорошими примерами этого являются денежные расчеты. Если требуется высокая точность, следует использовать другие типы. например, Java 7 предоставляет BigDecimal.

```
import java.math.BigDecimal;

public class FloatTest {

public static void main(String[] args) {
    float accountBalance = 10000.00f;
    System.out.println("Operations using float:");
    System.out.println("1000 operations for 1.99");
    for(int i = 0; i<1000; i++){
        accountBalance -= 1.99f;
    }
    System.out.println(String.format("Account balance after float operations: %f",
accountBalance));

    BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
    System.out.println("Operations using BigDecimal:");
    System.out.println("1000 operations for 1.99");
    BigDecimal operation = new BigDecimal("1.99");
    for(int i = 0; i<1000; i++){
        accountBalanceTwo = accountBalanceTwo.subtract(operation);
    }
    System.out.println(String.format("Account balance after BigDecimal operations: %f",
accountBalanceTwo));
}
```

Вывод этой программы:

```
Operations using float:
1000 operations for 1.99
```

```
Account balance after float operations: 8009,765625
Operations using BigDecimal:
1000 operations for 1.99
Account balance after BigDecimal operations: 8010,000000
```

Для стартового баланса 10000,00, после 1000 операций за 1.99, мы ожидаем, что баланс будет равен 8010,00. Использование типа float дает нам ответ около 8009,77, что неприемлемо неточно в случае денежных расчетов. Использование BigDecimal дает нам правильный результат.

BigDecimal.valueOf ()

Класс BigDecimal содержит внутренний кеш часто используемых чисел, например, от 0 до 10. Методы BigDecimal.valueOf () предоставляются в предпочтении конструкторам с похожими параметрами типа, то есть в приведенном ниже примере a предпочтительнее b.

```
BigDecimal a = BigDecimal.valueOf(10L); //Returns cached Object reference
BigDecimal b = new BigDecimal(10L); //Does not return cached Object reference

BigDecimal a = BigDecimal.valueOf(20L); //Does not return cached Object reference
BigDecimal b = new BigDecimal(20L); //Does not return cached Object reference


BigDecimal a = BigDecimal.valueOf(15.15); //Preferred way to convert a double (or float) into
a BigDecimal, as the value returned is equal to that resulting from constructing a BigDecimal
from the result of using Double.toString(double)
BigDecimal b = new BigDecimal(15.15); //Return unpredictable result
```

Инициализация BigDecimals со значением нуля, один или десять

BigDecimal предоставляет статические свойства для чисел 0, один и десять. Рекомендуется использовать их вместо использования фактических чисел:

- `BigDecimal.ZERO`
- `BigDecimal.ONE`
- `BigDecimal.TEN`

Используя статические свойства, вы избегаете ненужного создания экземпляра, также у вас есть буквальный код, а не «магическое число».

```
//Bad example:
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);

//Good Example:
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

Прочитайте BigDecimal онлайн: https://riptutorial.com/ru/java/topic/1667/bigdecimal

глава 9: BigInteger

Вступление

Класс `BigInteger` используется для математических операций с большими целыми числами с слишком большими величинами для примитивных типов данных. Например, 100-факториал составляет 158 цифр - намного больше, чем может представлять `long` . `BigInteger` предоставляет аналоги всем примитивным целочисленным операторам Java и всем соответствующим методам из `java.lang.Math` а также нескольким другим операциям.

Синтаксис

- BigInteger variable_name = new BigInteger ("123456789012345678901"); // десятичное целое в виде строки
- BigInteger variable_name = new BigInteger ("10101011010101001010100110001100111010110001111100001010110100101", 2) // двоичное целое в виде строки
- BigInteger variable_name = new BigInteger ("ab54a98ceb1f0800", 16) // шестнадцатеричное целое число в виде строки
- BigInteger variable_name = new BigInteger (64, new Random ()); // генератор псевдослучайных чисел, обеспечивающий 64 бита для построения целого числа
- BigInteger variable_name = new BigInteger (новый байт [] {0, -85, 84, -87, -116, -21, 31, 10, -46}); // подписали двухдополнительное представление целого (big endian)
- BigInteger variable_name = new BigInteger (1, новый байт [] {- 85, 84, -87, -116, -21, 31, 10, -46}); // Непрерывное представление целых чисел без знака (положительное целое число)

замечания

`BigInteger` неизменен. Поэтому вы не можете изменить свое состояние. Например, следующее не будет работать, поскольку `sum` не будет обновляться из-за неизменности.

```
BigInteger sum = BigInteger.ZERO;
for(int i = 1; i < 5000; i++) {
   sum.add(BigInteger.valueOf(i));
}
```

Назначьте результат переменной `sum` чтобы она работала.

```
sum = sum.add(BigInteger.valueOf(i));
```

Java SE 8

В официальной документации `BigInteger` говорится, что реализации `BigInteger` должны поддерживать все целые числа от -2 ^{2147483647} до 2 ^{2147483647} (эксклюзивные). Это означает, что `BigInteger` s может иметь более 2 *миллиардов* бит!

Examples

инициализация

Класс `java.math.BigInteger` обеспечивает аналоги операций для всех примитивных целочисленных операторов Java и для всех соответствующих методов из `java.lang.Math` . Поскольку пакет `java.math` не будет автоматически доступен, вам может потребоваться импортировать `java.math.BigInteger` прежде чем вы сможете использовать простое имя класса.

Чтобы преобразовать значения `long` или `int` в `BigInteger` используйте:

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

или, для целых чисел:

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

который *расширит* целое число `intValue` до long, используя расширение бита знака для отрицательных значений, так что отрицательные значения останутся отрицательными.

Чтобы преобразовать числовую `String` в `BigInteger` используйте:

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

Следующий конструктор используется для преобразования строкового представления `BigInteger` в указанном радиусе в `BigInteger` .

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString , binaryRadix);
```

Java также поддерживает прямое преобразование байтов в экземпляр `BigInteger` . В настоящее время может использоваться только подписанная и беззнаковая кодировка большого конца:

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

Это будет генерировать экземпляр `BigInteger` со значением -128, поскольку первый бит интерпретируется как бит знака.

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

Это будет генерировать экземпляр `BigInteger` со значением 128, поскольку байты интерпретируются как беззнаковое число, а знак явно установлен в 1, положительное число.

Существуют предопределенные константы для общих значений:

- `BigInteger.ZERO` - значение «0».
- `BigInteger.ONE` - значение «1».
- `BigInteger.TEN` - значение «10».

Существует также `BigInteger.TWO` (значение «2»), но вы не можете использовать его в своем коде, потому что он является `private` .

Сравнение BigIntegers

Вы можете сравнить `BigIntegers` же, как вы сравниваете `String` или другие объекты в Java.

Например:

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

Выход:

```
Not Equal
```

Замечания:

В общем, **не** используйте использование оператора `==` для сравнения BigIntegers

- `==` operator: сравнивает ссылки; т.е. имеют ли два значения один и тот же объект
- Метод `equals()` : сравнивает содержимое двух BigIntegers.

Например, BigIntegers **не** следует сравнивать следующим образом:

```
if (firstBigInteger == secondBigInteger) {
  // Only checks for reference equality, not content equality!
}
```

Это может привести к неожиданному поведению, поскольку оператор `==` проверяет только ссылочное равенство. Если оба BigIntegers содержат один и тот же контент, но не относятся к одному и тому же объекту, **это не сработает.** Вместо этого сравните BigIntegers, используя методы `equals` , как описано выше.

Вы также можете сравнить свой `BigInteger` с постоянными значениями, такими как 0,1,10.

например:

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

Вы также можете сравнить два BigIntegers с помощью метода `compareTo()` , как `compareTo()` ниже: `compareTo()` возвращает 3 значения.

- **0:** Когда оба **равны** .
- **1:** Когда первое **больше** второго (одно в скобках).
- **-1:** Когда первое **меньше** секунды.

```
BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //code when both are equal.
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //code when reallyBig is greater than reallyBig1.
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //code when reallyBig is less than reallyBig1.
}
```

Примеры математических операций BigInteger

BigInteger находится в неизменяемом объекте, поэтому вам нужно назначить результаты любой математической операции новому экземпляру BigInteger.

Дополнение: *10 + 10 = 20*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

выход: 20

Субстрат: *10 - 9 = 1*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

выход: 1

Отдел: *10/5 = 2*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

выход: 2

Отдел: *17/4 = 4*

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

выход: 4

Умножение: *10 * 5 = 50*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

выход: 50

Мощность: *10 ^ 3 = 1000*

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

выход: 1000

Остаток: *10% 6 = 4*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

выход: 4

GCD: наибольший общий делитель (GCD) для `12` и `18` - `6` .

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");

System.out.println(value1.gcd(value2));
```

Выход: 6

Максимум два BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

Выход: 11

Минимум двух BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

Выход: 10

Двоичные логические операции на BigInteger

BigInteger поддерживает двоичные логические операции, доступные также для типов `Number` . Как и во всех операциях, они реализуются путем вызова метода.

Двоичный или:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

Выход: 11 (что эквивалентно `10 | 9`)

Двоичные и:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

Выход: 8 (что эквивалентно `10 & 9`)

Binary Xor:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.xor(val2);
```

Выход: 3 (что эквивалентно `10 ^ 9`)

RightShift:

```
BigInteger val1 = new BigInteger("10");

val1.shiftRight(1);   // the argument be an Integer
```

Выход: 5 (эквивалентно `10 >> 1`)

Сдвиг влево:

```
BigInteger val1 = new BigInteger("10");

val1.shiftLeft(1);   // here parameter should be Integer
```

Выход: 20 (эквивалентно `10 << 1`)

Двоичная инверсия (нет):

```
BigInteger val1 = new BigInteger("10");

val1.not();
```

Выход: 5

*NAND (And-Not): **

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.andNot(val2);
```

Выход: 7

Создание случайных BigIntegers

Класс `BigInteger` имеет конструктор, предназначенный для генерации случайных `BigIntegers` , учитывая экземпляр `java.util.Random` и `int` который определяет, сколько бит будет иметь `BigInteger` . Его использование довольно просто - когда вы вызываете конструктор `BigInteger(int, Random)` следующим образом:

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

то вы получите `BigInteger` , значение которого находится в `bitCount` от 0 (включительно) и до 2 `bitCount` (исключение).

Это также означает, что `new BigInteger(2147483647, sourceOfRandomness)` может вернуть все положительные значения `BigInteger` с достаточным временем.

Какова будет `sourceOfRandomness` ? Например, `new Random()` достаточно хорош в большинстве случаев:

```
new BigInteger(32, new Random());
```

Если вы хотите отказаться от скорости для более качественных случайных чисел, вместо этого вы можете использовать `new SecureRandom ()` :

```
import java.security.SecureRandom;

// somewhere in the code...
new BigInteger(32, new SecureRandom());
```

Вы даже можете реализовать алгоритм «на лету» с анонимным классом! Обратите внимание , что **выкатывает свой собственный алгоритм ГСЧ закончится вас с низким качеством случайностью,** поэтому всегда обязательно использовать алгоритм , который , как доказано , чтобы быть достойной , если вы не хотите, чтобы в результате `BigInteger` (ы) быть предсказуемым.

```
new BigInteger(32, new Random() {
    int seed = 0;

    @Override
    protected int next(int bits) {
        seed = ((22695477 * seed) + 1) & 2147483647; // Values shamelessly stolen from
Wikipedia
        return seed;
    }
});
```

Прочитайте BigInteger онлайн: https://riptutorial.com/ru/java/topic/1514/biginteger

глава 10: BufferedWriter

Синтаксис

- новый BufferedWriter (Writer); // Конструктор по умолчанию
- BufferedWriter.write (int c); // Записывает один символ
- BufferedWriter.write (String str); // Записывает строку
- BufferedWriter.newLine (); // Записывает разделитель строк
- BufferedWriter.close (); // Закрывает BufferedWriter

замечания

- Если вы попытаетесь написать из `BufferedWriter` (используя `BufferedWriter.write()`) после закрытия `BufferedWriter` (используя `BufferedWriter.close()`), это вызовет `IOException` .
- Конструктор `BufferedWriter(Writer)` НЕ выбрасывает `IOException` . Однако конструктор `FileWriter(File)` `FileNotFoundException` , которое расширяет `IOException` . Таким образом, `IOException` также поймает `FileNotFoundException` , никогда не требуется второй оператор catch, если вы не планируете делать что-то другое с `FileNotFoundException` .

Examples

Напишите строку текста в файл

Этот код записывает строку в файл. Важно закрыть автора, так что это делается в блоке `finally` .

```
public void writeLineToFile(String str) throws IOException {
  File file = new File("file.txt");
  BufferedWriter bw = null;
  try {
    bw = new BufferedWriter(new FileWriter(file));
    bw.write(str);
  } finally {
    if (bw != null) {
      bw.close();
    }
  }
}
```

Также обратите внимание, что `write(String s)` не помещает символ новой строки после того, как строка была записана. Для этого используйте `newLine()` .

Java SE 7

Java 7 добавляет пакет `java.nio.file` и пытается использовать ресурсы :

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

Прочитайте BufferedWriter онлайн: https://riptutorial.com/ru/java/topic/3063/bufferedwriter

глава 11: ByteBuffer

Вступление

Класс `ByteBuffer` был введен в java 1.4 для облегчения работы с двоичными данными. Он особенно подходит для использования с данными примитивного типа. Это позволяет создавать, но также и последующую манипуляцию `byte[]` s на более высоком уровне абстракции

Синтаксис

- byte [] arr = новый байт [1000];
- ByteBuffer buffer = ByteBuffer.wrap (arr);
- ByteBuffer buffer = ByteBuffer.allocate (1024);
- ByteBuffer buffer = ByteBuffer.allocateDirect (1024);
- byte b = buffer.get ();
- байт b = buffer.get (10);
- short s = buffer.getShort (10);
- buffer.put ((byte) 120);
- buffer.putChar ('a');

Examples

Основное использование - создание ByteBuffer

Существует два способа создания `ByteBuffer` , где можно снова подразделить.

Если у вас уже есть `byte[]` , вы можете *«обернуть»* его в `ByteBuffer` чтобы упростить обработку:

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

Это будет возможность для кода, который обрабатывает сетевые взаимодействия на низком уровне

Если у вас нет уже существующего `byte[]` , вы можете создать `ByteBuffer` над массивом, специально выделенным для буфера следующим образом:

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

Если код-путь чрезвычайно критичен по производительности и вам нужен **прямой доступ к системной памяти** , `ByteBuffer` может даже выделять *прямые* буферы с помощью `#allocateDirect()`

Основное использование - запись данных в буфер

Учитывая `ByteBuffer` экземпляр можно записывать данные примитивного типа к нему с помощью *относительного* и *абсолютного* `put` . Поразительное различие заключается в том, что помещение данных с использованием *относительного* метода отслеживает индекс, в который данные вставляются для вас, в то время как абсолютный метод всегда требует указания индекса для `put` данных.

Оба метода позволяют *«цепочки»* вызовов. При достаточно большом буфере можно сделать следующее:

```
buffer.putInt(0xCAFEBABE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBABE);
```

что эквивалентно:

```
buffer.putInt(0xCAFEBABE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBABE);
```

Обратите внимание, что метод, базирующийся на `byte` не указан специально. Кроме того , обратите внимание , что это справедливо и для передачи одновременно `ByteBuffer` и `byte[]` , чтобы `put` . Кроме этого, все примитивные типы имеют специализированные `put` методы.

Дополнительная заметка: индекс, указанный при использовании абсолютного значения `put*` , всегда учитывается в `byte` .

Основное использование - использование DirectByteBuffer

`DirectByteBuffer` - это специальная реализация `ByteBuffer` , у которой нет `byte[]` .

Мы можем выделить такой ByteBuffer, вызывая:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

Эта операция будет выделять 16 байт памяти. Содержимое прямых буферов *может* находиться вне обычной кучи мусора.

Мы можем проверить, является ли ByteBuffer прямым, вызывая:

```
directBuffer.isDirect(); // true
```

Основные характеристики `DirectByteBuffer` том, что JVM будет пытаться изначально работать с выделенной памятью без дополнительной буферизации, поэтому выполняемые на ней операции могут быть быстрее, чем выполняемые на ByteBuffers с лежащими под ним массивами.

Рекомендуется использовать `DirectByteBuffer` с тяжелыми операциями ввода-вывода, которые полагаются на скорость выполнения, например, в режиме реального времени.

Мы должны знать, что если мы попытаемся использовать метод `array()` мы получим `UnsupportedOperationException` . Таким образом, это хорошая практика, чтобы проверить, есть ли у нашего ByteBuffer (байтовый массив), прежде чем мы попытаемся получить к нему доступ:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Другое использование прямого байтового буфера - это взаимодействие через JNI. Поскольку буфер прямого байта не использует `byte[]` , а представляет собой фактический блок памяти, можно получить доступ к этой памяти непосредственно через указатель в собственном коде. Это может сэкономить массу проблем и накладных расходов при сортировке между Java и собственным представлением данных.

Интерфейс JNI определяет несколько функций для обработки буферов с прямым байтом: поддержка NIO .

Прочитайте ByteBuffer онлайн: https://riptutorial.com/ru/java/topic/702/bytebuffer

глава 12: CompletableFuture

Вступление

CompletableFuture - это класс, добавленный в Java SE 8, который реализует интерфейс Future от Java SE 5. Помимо поддержки интерфейса Future, он добавляет множество методов, которые позволяют асинхронный обратный вызов, когда будущее будет завершено.

Examples

Преобразование метода блокировки в асинхронный

Следующий метод займет секунду или два в зависимости от вашего подключения, чтобы получить веб-страницу и подсчитать длину текста. Какими бы ни были потоковые вызовы, они будут блокироваться в течение этого периода времени. Также он вызывает исключение, которое полезно позже.

```
public static long blockingGetWebPageLength(String urlString) {
    try (BufferedReader br = new BufferedReader(new InputStreamReader(new
URL(urlString).openConnection().getInputStream()))) {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            sb.append(line);
        }
        return sb.toString().length();
    } catch (IOException ex) {
        throw new RuntimeException(ex);
    }
}
```

Это преобразует его в метод, который немедленно возвращается, перемещая вызов метода блокировки в другой поток. По умолчанию метод supplyAsync будет запускать поставщика в общий пул. Для метода блокировки это, вероятно, не очень хороший выбор, поскольку вы могли бы исчерпать потоки в этом пуле, поэтому я добавил дополнительный параметр сервиса.

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

Чтобы использовать функцию в асинхронном режиме, следует использовать методы, которые принимают ламду, которую вызывают с результатом поставщика, когда она

завершается, например thenAccept. Также важно использовать метод исключения или обработки для регистрации любых исключений, которые могли произойти.

```
public static void main(String[] args) {

    asyncGetWebPageLength("https://stackoverflow.com/")
            .thenAccept(l -> {
                System.out.println("Stack Overflow returned " + l);
            })
            .exceptionally((Throwable throwable) -> {
                Logger.getLogger("myclass").log(Level.SEVERE, "", throwable);
                return null;
            });

}
```

Простой пример CompletingFuture

В приведенном ниже примере, `calculateShippingPrice` метод вычисляет стоимость доставки, которая занимает некоторое время обработки. Например, в реальном мире это может быть обращение к другому серверу, который возвращает цену, основанную на весе продукта и способе доставки.

Путем моделирования этого методом async через `CompletableFuture` мы можем продолжить разную работу в методе (т.е. рассчитать затраты на упаковку).

```
public static void main(String[] args) {
    int price = 15; // Let's keep it simple and work with whole number prices here
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // Here, we get the future
        .thenAccept(shippingPrice -> { // And then immediately work on it!
            // This fluent style is very useful for keeping it concise
            System.out.println("Your total price is: " + (price + shippingPrice));
        });
    System.out.println("Please stand by. We are calculating your total price.");
}


public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync is a factory method that turns a given
        // Supplier<U> into a CompletableFuture<U>

        // Let's just say each 200 grams is a new dollar on your shipping costs
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // Now let's simulate some waiting time...
        } catch(InterruptedException e) { /* We can safely ignore that */ }

        return shippingCosts; // And send the costs back!
    });
}
```

Прочитайте CompletableFuture онлайн:
https://riptutorial.com/ru/java/topic/10935/completablefuture

глава 13: Enum, начиная с номера

Вступление

Java не разрешает имя enum начинать с номера типа 100A, 25K. В этом случае мы можем добавить код с помощью _ (underscore) или любого разрешенного шаблона и проверить его.

Examples

Enum с именем при начале

```
public enum BookCode {
    _10A("Simon Haykin", "Communication System"),
    _42B("Stefan Hakins", "A Brief History of Time"),
    E1("Sedra Smith", "Electronics Circuits");

    private String author;
    private String title;

    BookCode(String author, String title) {
        this.author = author;
        this.title = title;
    }

    public String getName() {
        String name = name();
        if (name.charAt(0) == '_') {
            name = name.substring(1, name.length());
        }
        return name;
    }

    public static BookCode of(String code) {
        if (Character.isDigit(code.charAt(0))) {
            code = "_" + code;
        }
        return BookCode.valueOf(code);
    }
}
```

Прочитайте Enum, начиная с номера онлайн: https://riptutorial.com/ru/java/topic/10719/enum--начиная-с-номера

глава 14: FileUpload для AWS

Вступление

Загрузите файл в ведро AWS s3 с использованием API Spring Spring.

Examples

Загрузить файл в корзину s3

> Здесь мы создадим отдых APi, который возьмет файл-объект в качестве параметра multipart из front-end и загрузит его в ведро S3 с помощью java rest API.

Требование : - секретный ключ и ключ доступа для ведра s3, где вы хотите загрузить файл.

код: - DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
public class DocumentController {

    private static String bucketName = "pharmerz-chat";
    //   private static String keyName        = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
MediaType.MULTIPART_FORM_DATA)
    public URL uploadFileHandler(@RequestParam("name") String name,
                                 @RequestParam("file") MultipartFile file) throws IOException
{

/******* Printing all the possible parameter from @RequestParam *************/

        System.out.println("*****************************");

        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType()" + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        /*************Parameters to b pass to s3 bucket put Object **************/
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();
```

```
// Credentials for Aws
        AWSCredentials credentials = new BasicAWSCredentials("AKIA*************",
"zr**********************");

        /***************** DocumentController.uploadfile(credentials);
*************************/


        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                    bucketName, keyName, is, new ObjectMetadata()));


            URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
            // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.)));
            System.out.println("***********************************");
            System.out.println(url);

            return url;

        } catch (AmazonServiceException ase) {
            System.out.println("Caught an AmazonServiceException, which " +
                    "means your request made it " +
                    "to Amazon S3, but was rejected with an error response" +
                    " for some reason.");
            System.out.println("Error Message:    " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code:   " + ase.getErrorCode());
            System.out.println("Error Type:       " + ase.getErrorType());
            System.out.println("Request ID:       " + ase.getRequestId());
        } catch (AmazonClientException ace) {
            System.out.println("Caught an AmazonClientException, which " +
                    "means the client encountered " +
                    "an internal error while trying to " +
                    "communicate with S3, " +
                    "such as not being able to access the network.");
            System.out.println("Error Message: " + ace.getMessage());
        }

        return null;

    }


}
```

Функция переднего конца

```
var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
 "async": true,
 "crossDomain": true,
 "url": "http://url/",
```

```
 "method": "POST",
 "headers": {
   "cache-control": "no-cache"
 },
 "processData": false,
 "contentType": false,
 "mimeType": "multipart/form-data",
 "data": form
}

$.ajax(settings).done(function (response) {
 console.log(response);
});
```

Прочитайте FileUpload для AWS онлайн: https://riptutorial.com/ru/java/topic/10589/fileupload-для-aws

глава 15: FTP (протокол передачи файлов)

Синтаксис

- Соединение FTPClient (хост InetAddress, int port)
- Логин FTPClient (имя пользователя String, пароль String)
- Отключение FTPClient ()
- FTPReply getReplyStrings ()
- boolean storeFile (String remote, InputStream local)
- Хранилище OutputStreamFileStream (String remote)
- boolean setFileType (int fileType)
- boolean completePendingCommand ()

параметры

параметры	подробности
хозяин	Либо имя хоста, либо IP-адрес FTP-сервера
порт	Порт FTP-сервера
имя пользователя	Имя пользователя FTP-сервера
пароль	Пароль FTP-сервера

Examples

Подключение и вход в FTP-сервер

Чтобы начать использовать FTP с Java, вам нужно будет создать новый `FTPClient` а затем подключиться и войти на сервер с помощью `.connect(String server, int port)` и `.login(String username, String password)` .

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//Import all the required resource for this project.

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com"; //Server can be either host name or IP address.
        int port = 21;
        String user = "Username";
        String pass = "Password";
```

```
        FTPClient ftp = new FTPClient;
        ftp.connect(server, port);
        ftp.login(user, pass);
    }
}
```

Теперь у нас есть основы. Но что, если у нас есть ошибка подключения к серверу? Мы хотим знать, когда что-то пойдет не так, и получите сообщение об ошибке. Давайте добавим код, чтобы поймать ошибки при подключении.

```
try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.printIn("Operation failed. Server reply code: " + replyCode)
        return;
    }
    ftp.login(user, pass);
} catch {

}
```

Давайте сломаем то, что мы только что сделали, шаг за шагом.

```
showServerReply(ftp);
```

Это относится к функции, которую мы будем делать на более позднем этапе.

```
int replyCode = ftp.getReplyCode();
```

Это захватывает код ответа / ошибки с сервера и сохраняет его как целое.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

Это проверяет код ответа, чтобы узнать, была ли ошибка. Если произошла ошибка, она просто выведет «Операция не выполнена. Код ответа сервера:», за которой следует код ошибки. Мы также добавили блок try / catch, который мы добавим на следующем шаге. Затем давайте создадим функцию, которая проверяет `ftp.login()` наличие ошибок.

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
        return;
    } else {
        System.out.println("LOGGED IN SERVER");
    }
```

Давайте также сломаем этот блок.

```
boolean success = ftp.login(user, pass);
```

Это не просто попытка входа на FTP-сервер, но и сохранение результата в виде логического.

```
showServerReply(ftp);
```

Это проверит, посылал ли сервер нам какие-либо сообщения, но сначала нам нужно создать функцию на следующем шаге.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Этот оператор проверяет, успешно ли мы вошли в систему; если это так, он напечатает « LOGGED IN SERVER», иначе он напечатает «Не удалось войти в сервер». Это наш скрипт:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                    System.out.println("Operation failed. Server reply code: " + replyCode);
                    return;
                }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch {

        }
    }
```

```
}
```

Теперь давайте создадим полный блок Catch, если мы столкнемся с любыми ошибками всего процесса.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

Завершенный блок блокировки теперь будет печатать «Ой, что-то пошло не так». и stacktrace, если есть ошибка. Теперь наш последний шаг - создать `showServerReply()` мы использовали некоторое время.

```
private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
    for (String aReply : replies) {
        System.out.println("SERVER: " + aReply);
        }
    }
}
```

Эта функция принимает `FTPClient` как переменную и называет ее «ftp». После этого он хранит любые серверные ответы с сервера в массиве строк. Затем он проверяет, сохранены ли какие-либо сообщения. Если он есть, он печатает каждый из них как «СЕРВЕР: [ответ]». Теперь, когда мы выполнили эту функцию, это завершенный скрипт:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
```

```
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch (IOException ex) {
            System.out.println("Oops! Something went wrong.");
            ex.printStackTrace();
        }
    }
}
```

Сначала нам нужно создать новый `FTPClient` и попробовать подключиться к серверу и войти в него с помощью `.connect(String server, int port)` и `.login(String username, String password)` . Важно подключиться и войти в систему с помощью блока try / catch, если наш код не сможет подключиться к серверу. Нам также необходимо создать функцию, которая проверяет и отображает любые сообщения, которые мы можем получать с сервера, когда мы пытаемся подключиться и войти в систему. Мы будем называть эту функцию « `showServerReply(FTPClient ftp)` ».

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

public static void main(String[] args) {
    // SET THESE TO MATCH YOUR FTP SERVER //
    String server = "www.server.com";
    int port = 21;
    String user = "username"
    String pass = "password"

FTPClient ftp = new FTPClient
try {
    ftp.connect(server, port)
    showServerReply(ftp);
    int replyCode = ftpClient.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
            System.out.println("Operation failed. Server reply code: " + replyCode);
            return;
        }
    boolean success = ftp.login(user, pass);
```

```
    showServerReply(ftp);
    if (!success) {
        System.out.println("Failed to log into the server");
            return;
        } else {
            System.out.println("LOGGED IN SERVER");
        }
    } catch (IOException ex) {
        System.out.println("Oops! Something went wrong.");
        ex.printStackTrace();
        }
    }
}
```

После этого вы должны теперь подключить свой FTP-сервер к вам Java-скрипту.

Прочитайте FTP (протокол передачи файлов) онлайн:
https://riptutorial.com/ru/java/topic/5228/ftp--протокол-передачи-файлов-

глава 16: HttpURLConnection

замечания

- Использование HttpUrlConnection на Android требует, чтобы вы добавили разрешение на доступ в Интернет для своего приложения (в `AndroidManifest.xml`).
- Существуют также другие Java-HTTP-клиенты и библиотеки, такие как OkHttp Square, которые более удобны в использовании и могут обеспечить лучшую производительность или больше возможностей.

Examples

Получить тело ответа из URL-адреса в виде строки

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //add headers to the connection, or check the status if desired..

    // handle error response code it occurs
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));

    StringBuilder response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

Это будет загружать текстовые данные из указанного URL-адреса и возвращать их как строку.

Как это работает:

- Во-первых, мы создаем `HttpUrlConnection` из нашего URL-адреса с `new URL(url).openConnection()` . Мы `UrlConnection` это возвращает `HttpUrlConnection` , поэтому у

нас есть доступ к таким вещам, как добавление заголовков (например, User Agent) или проверка кода ответа. (Этот пример не делает этого, но его легко добавить.)

- Затем создайте `InputStream` на основе кода ответа (для обработки ошибок)
- Затем создайте `BufferedReader` который позволяет нам читать текст из `InputStream` мы получаем из соединения.
- Теперь мы добавляем текст в `StringBuilder` последовательно.
- Закройте `InputStream` и верните строку, которую мы теперь имеем.

Заметки:

- Этот метод будет генерировать `IoException` в случае сбоя (например, сетевую ошибку или отсутствие подключения к Интернету), а также исключить *исключение* `MalformedUrlException` если данный URL-адрес недействителен.
- Его можно использовать для чтения с любого URL-адреса, который возвращает текст, например веб-страницы (HTML), API REST, которые возвращают JSON или XML и т. Д.
- См. Также: Прочитать URL-адрес строки в нескольких строках кода Java .

Использование:

Очень просто:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Данные POST

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
```

```
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
        if (in != null) in.close();
    }
}
```

Это будет POST-данных для указанного URL-адреса, а затем прочитайте ответ по очереди.

Как это устроено

- Как обычно, мы получаем `HttpURLConnection` из `URL` .
- Задайте тип содержимого с помощью `setRequestProperty` , по умолчанию это `application/x-www-form-urlencoded`
- `setDoOutput(true)` сообщает подключению, что мы будем отправлять данные.
- Затем мы получаем `OutputStream` , вызывая `getOutputStream()` и записывая данные на него. Не забудьте закрыть его после завершения.
- Наконец, мы читаем ответ сервера.

Удалить ресурс

```
   public static void delete (String urlString, String contentType) throws IOException {
        HttpURLConnection connection = null;

        try {
            URL url = new URL(urlString);
            connection = (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);
            connection.setRequestMethod("DELETE");
            connection.setRequestProperty("Content-Type", contentType);

            Map<String, List<String>> map = connection.getHeaderFields();
            StringBuilder sb = new StringBuilder();
            Iterator<Map.Entry<String, String>> iterator =
responseHeader.entrySet().iterator();
            while(iterator.hasNext())
            {
                Map.Entry<String, String> entry = iterator.next();
                sb.append(entry.getKey());
                sb.append('=').append('"');
                sb.append(entry.getValue());
                sb.append('"');
                if(iterator.hasNext())
                {
                    sb.append(',').append(' ');
                }
            }
            System.out.println(sb.toString());
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (connection != null) connection.disconnect();
        }
    }
```

Это УДАЛИТЬ ресурс в указанном URL-адресе, а затем распечатает заголовок ответа.

Как это устроено

- мы получаем `HttpURLConnection` из `URL` .
- Задайте тип содержимого с помощью `setRequestProperty` , по умолчанию это `application/x-www-form-urlencoded`
- `setDoInput(true)` сообщает о соединении, которое мы намерены использовать для ввода URL-адреса.
- `setRequestMethod("DELETE")` для выполнения HTTP DELETE

Наконец, мы печатаем заголовок ответа сервера.

Проверьте, существует ли ресурс

```
/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}
```

Объяснение:

Если вы просто проверяете, существует ли ресурс, лучше использовать запрос HEAD, чем GET. Это позволяет избежать накладных расходов на передачу ресурса.

Обратите внимание, что метод возвращает `true` если код ответа `200` . Если вы ожидаете ответа на перенаправление (т. Е. 3XX), то, возможно, этот способ может быть усилен для их соблюдения.

Пример:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Прочитайте HttpURLConnection онлайн:
https://riptutorial.com/ru/java/topic/156/httpurlconnection

глава 17: InputStreams и OutputStreams

Синтаксис

- int read (byte [] b) выбрасывает IOException

замечания

Обратите внимание, что большую часть времени вы НЕ используете `InputStream` s напрямую, но используете `BufferedStream` или похожи. Это связано с тем, что `InputStream` считывает из источника каждый раз при вызове метода чтения. Это может вызвать значительное использование ЦП в контекстных коммутаторах в ядре и из него.

Examples

Чтение InputStream в строку

Иногда вы можете читать байтовый ввод в String. Для этого вам нужно будет найти что-то, что преобразует между `byte` и «родной Java» UTF-16 Codepoints, используемым как `char` . Это делается с помощью `InputStreamReader` .

Чтобы немного ускорить процесс, «обычным» является выделение буфера, так что у нас не слишком много накладных расходов при чтении с Input.

Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
     StringWriter writer = new StringWriter();

     char[] buffer = new char[1024];
     try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
           int n;
           while ((n = reader.read(buffer)) != -1) {
                // all this code does is redirect the output of `reader` to `writer` in
                // 1024 byte chunks
                writer.write(buffer, 0, n);
           }
     }
     return writer.toString();
}
```

Преобразование этого примера в Java SE 6 (и более низкий) совместимый код не учитывается как упражнение для читателя.

Запись байтов в OutputStream

Запись байтов в `OutputStream` одному байту за раз

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Запись байтового массива

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Написание раздела массива байтов

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Закрытие потоков

Большинство потоков необходимо закрыть, когда вы закончите с ними, иначе вы можете ввести утечку памяти или оставить файл открытым. Важно, чтобы потоки были закрыты, даже если выбрано исключение.

Java SE 7

```
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

Помните: try-with-resources гарантирует, что ресурсы были закрыты при выходе из блока, независимо от того, происходит ли это с обычным потоком управления или из-за исключения.

Java SE 6

Иногда попробовать-с-ресурсами не вариант, или, может быть, вы поддерживаете более старую версию Java 6 или ранее. В этом случае правильная обработка заключается в использовании блока `finally` :

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} catch (IOException e) {
    //handle this however you want
}
finally {
    try {
        if(out != null)
            out.close();
    } catch (IOException e) {
        //typically not much you can do here...
    }
}
```

Обратите внимание, что закрытие потока оболочки также закрывает его базовый поток. Это означает, что вы не можете обернуть поток, закрыть оболочку и продолжить использование исходного потока.

Копирование входного потока в выходной поток

Эта функция копирует данные между двумя потоками -

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Пример -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Обтекание потоков ввода / вывода

`OutputStream` и `InputStream` имеют много разных классов, каждый из которых обладает уникальной функциональностью. Обертывая поток вокруг другого, вы получаете функциональность обоих потоков.

Вы можете обтекать поток сколько угодно раз, просто обратите внимание на порядок.

Полезные комбинации

Запись символов в файл при использовании буфера

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Сжатие и шифрование данных перед записью в файл при использовании буфера

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

Список оберток потока ввода / вывода

обертка	Описание
BufferedOutputStream / BufferedInputStream	В то время как `OutputStream` записывает данные по одному байту за один раз, `BufferedOutputStream` записывает данные в куски. Это уменьшает количество системных вызовов, что повышает производительность.
DeflaterOutputStream / DeflaterInputStream	Выполняет сжатие данных.
InflaterOutputStream / InflaterInputStream	Выполняет декомпрессию данных.
CipherOutputStream / CipherInputStream	Шифрует / расшифровывает данные.
DigestOutputStream / DigestInputStream	Генерирует дайджест сообщений для проверки целостности данных.
CheckedOutputStream / CheckedInputStream	Создает CheckSum. CheckSum - это более тривиальная версия Message Digest.
DataOutputStream / DataInputStream	Позволяет писать примитивные типы данных и строки. Предназначен для написания байтов. Независимая платформа.
PrintStream	Позволяет писать примитивные типы данных и строки. Предназначен для написания байтов. Платформа зависит.
OutputStreamWriter	Преобразует OutputStream в Writer. OutputStream имеет дело с байтами, в то время как Writers имеет дело с символами

обертка	Описание
PrintWriter	Автоматически вызывает OutputStreamWriter. Позволяет писать примитивные типы данных и строки. Строго для написания символов и лучше всего писать символы

Пример DataInputStream

```
package com.streams;
import java.io.*;
public class DataStreamDemo {
  public static void main(String[] args) throws IOException {
    InputStream input = new FileInputStream("D:\\datastreamdemo.txt");
    DataInputStream inst = new DataInputStream(input);
    int count = input.available();
    byte[] arr = new byte[count];
    inst.read(arr);
    for (byte byt : arr) {
      char ki = (char) byt;
      System.out.print(ki+"-");
    }
  }
}
```

Прочитайте InputStreams и OutputStreams онлайн:
https://riptutorial.com/ru/java/topic/110/inputstreams-и-outputstreams

глава 18: Java Pitfalls - использование исключений

Вступление

Некоторые нарушения языка программирования Java могут выполнять программу для получения неправильных результатов, несмотря на то, что они правильно составлены. Основная цель этого раздела - перечислить общие **ошибки,** связанные с **обработкой исключений** , и предложить правильный способ избежать таких ошибок.

Examples

Pitfall - Игнорирование или сбой исключений

В этом примере речь идет об умышленном игнорировании или «раздавливании» исключений. Или, точнее, речь идет о том, как поймать и обработать исключение таким образом, чтобы его игнорировать. Тем не менее, прежде чем мы опишем, как это сделать, мы должны сначала указать, что исключение от раздавливания, как правило, не является правильным способом борьбы с ними.

Исключения обычно забрасываются (кем-то), чтобы уведомить другие части программы о том, что произошло какое-то значительное (то есть «исключительное») событие. Вообще (хотя и не всегда) исключение означает, что что-то пошло не так. Если вы закодируете свою программу для выдачи исключения, есть вероятность, что проблема снова появится в другой форме. Чтобы ухудшить ситуацию, когда вы выкалываете исключение, вы выбрасываете информацию в объекте исключения и связанной с ним трассировке стека. Скорее всего, это затруднит выяснение того, каков был исходный источник проблемы.

На практике, при использовании функции автоматической коррекции IDE, «исправление» ошибки компиляции, вызванной необработанным исключением, часто происходит сбой при сбоях. Например, вы можете увидеть такой код:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Ясно, что программист принял предложение IDE, чтобы ошибка компиляции исчезла, но предложение было неуместным. (Если файл открылся не сработал, программа, скорее всего, что-то с этим `inputStream` . С приведенной выше «коррекцией» программа может потерпеть неудачу позже, например, с помощью `NullPointerException` поскольку `inputStream`

теперь имеет значение `null` .)

Сказав это, вот пример умышленного подавления исключения. (В целях аргумента предположим, что мы определили, что прерывание при показе самообороны является безвредным.) Комментарий говорит читателю, что мы сознательно раздавили исключение и почему мы это сделали.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

Другой общепринятый способ подчеркнуть, что мы *намеренно* подавляем исключение, не говоря о том, почему нужно указывать это с именем переменной исключения, например:

```
try {
    selfie.show();
} catch (InterruptedException ignored) {  }
```

Некоторые IDE (например, IntelliJ IDEA) не будут отображать предупреждение о пустом блоке catch, если имя переменной установлено на `ignored` .

Pitfall - Catching Throwable, Exception, Error или RuntimeException

Общим шаблоном мыслей для неопытных программистов на Java является то, что исключения - это «проблема» или «бремя», и лучший способ справиться с этим - как можно скорее поймать их всех [1] . Это приводит к следующему коду:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

Вышеприведенный код имеет значительный недостаток. `catch` на самом деле поймает больше исключений, чем ожидает программист. Предположим, что значение `fileName` равно `null` из-за ошибки в другом месте приложения. Это заставит конструктор `FileInputStream` выкинуть `FileInputStream` `NullPointerException` . Обработчик поймает это и сообщит пользователю:

```
    Could not open file null
```

что бесполезно и запутанно. Хуже того, предположим, что это был код «процесс ввода», который выдал неожиданное исключение (отмечено или не отмечено!). Теперь пользователь получит сообщение об ошибке для проблемы, которая не возникла при

открытии файла, и вообще не может быть связана с I / O.

Корень проблемы заключается в том, что программист закодировал обработчик для `Exception` . Это почти всегда ошибка:

- Catching `Exception` поймает все проверенные исключения и большинство непроверенных исключений.
- Catching `RuntimeException` большинство непроверенных исключений.
- `Error` Catching `Error` будет проверять неконтролируемые исключения, которые сигнализируют внутренние ошибки JVM. Эти ошибки, как правило, не подлежат восстановлению и не должны быть пойманы.
- Catching `Throwable` поймает все возможные исключения.

Проблема с улавливанием слишком широкого набора исключений заключается в том, что обработчик обычно не может обрабатывать все из них соответствующим образом. В случае `Exception` и т. Д. Программисту сложно предсказать, что *можно* поймать; т.е. чего ожидать.

В общем, правильное решение , чтобы иметь дело с исключениями , которые выбрасываются. Например, вы можете поймать их и обработать их на месте:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

или вы можете объявить их как `thrown` приложенным методом.

Очень мало ситуаций, когда ловушка `Exception` подходит. Единственное, что возникает обычно, это что-то вроде этого:

```
public static void main(String[] args) {
    try {
        // do stuff
    } catch (Exception ex) {
        System.err.println("Unfortunately an error has occurred. " +
                           "Please report this to X Y Z");
        // Write stacktrace to a log file.
        System.exit(1);
    }
}
```

Здесь мы действительно хотим иметь дело со всеми исключениями, поэтому `Throwable Exception` (или даже `Throwable`) верна.

---

1 - Также известен как Покемон Исключение обработки .

Pitfall - Бросание Throwable, Exception, Error или RuntimeException

Хотя `Throwable` `Exception` `Throwable` , `Exception` , `Error` и `RuntimeException` является плохим, бросать их еще хуже.

Основная проблема заключается в том, что когда ваше приложение должно обрабатывать исключения, наличие исключений верхнего уровня затрудняет различение различных условий ошибки. Например

```
try {
    InputStream is = new FileInputStream(someFile);  // could throw IOException
    ...
    if (somethingBad) {
        throw new Exception();  // WRONG
    }
} catch (IOException ex) {
    System.err.println("cannot open ...");
} catch (Exception ex) {
    System.err.println("something bad happened");  // WRONG
}
```

Проблема в том, что, поскольку мы `Exception` экземпляр `Exception` , мы вынуждены его поймать. Однако, как описано в другом примере, catching `Exception` является плохим. В этой ситуации становится трудно различать «ожидаемый» случай `Exception` который генерируется, если `somethingBad` `true` , и непредвиденный случай, когда мы фактически поймаем неконтролируемое исключение, такое как `NullPointerException` .

Если исключение верхнего уровня разрешено распространять, мы сталкиваемся с другими проблемами:

- Теперь мы должны помнить все разные причины, по которым мы выбрали верхний уровень и дискриминируем / обрабатываем их.
- В случае `Exception` и `Throwable` нам также необходимо добавить эти исключения в предложение `throws` методов, если мы хотим, чтобы исключение распространялось. Это проблематично, как описано ниже.

Короче говоря, не бросайте эти исключения. Бросьте более конкретное исключение, которое более подробно описывает «исключительное событие», которое произошло. Если вам нужно, определите и используйте специальный класс исключений.

Объявление Throwable или Exception в «бросках» метода проблематично.

Заманчиво заменить длинный список исключенных исключений в предложение `throws` метода с `Exception` или даже Throwable. Это плохая идея:

1. Это заставляет вызывающего абонента обрабатывать (или распространять) `Exception` .

2. Мы больше не можем полагаться на компилятор, чтобы рассказать нам о конкретных проверенных исключениях, которые необходимо обработать.
3. Обработка `Exception` должным образом затруднено. Трудно понять, какие фактические исключения могут быть пойманы, и если вы не знаете, что можно поймать, трудно понять, какая стратегия восстановления подходит.
4. Обработка `Throwable` еще сложнее, так как теперь вы также должны справляться с потенциальными сбоями, которые никогда не должны восстанавливаться.

Этот совет означает, что некоторые другие шаблоны следует избегать. Например:

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

Вышеупомянутые попытки регистрировать все исключения по мере их прохождения, без окончательного обращения с ними. К сожалению, до Java 7, `throw ex;` заявление заставило компилятор думать, что любое `Exception` может быть выброшено. Это может заставить вас объявить вложенный метод как `throws Exception` . Начиная с Java 7 компилятор знает, что набор исключений, которые могут быть (переброшены), меньше.

Pitfall - Catching InterruptedException

Как уже указывалось в других ловушках, ловя все исключения, используя

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Поставляется с множеством разных проблем. Но одна из проблем заключается в том, что это может привести к взаимоблокировкам, поскольку он разбивает систему прерываний при написании многопоточных приложений.

Если вы начинаете нить, вы также должны быть в состоянии остановить ее внезапно по разным причинам.

```
Thread t = new Thread(new Runnable() {
    public void run() {
         while (true) {
             //Do something indefinetely
         }
    }
}

t.start();
```

```
//Do something else

// The thread should be canceld if it is still active.
// A Better way to solve this is with a shared variable that is tested
// regularily by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
   t.interrupt();
   t.join();
}

//Continue with program
```

`t.interrupt()` приведет к `t.interrupt()` InterruptedException в этом потоке, чем предназначен для отключения потока. Но что, если Thread должен очистить некоторые ресурсы до того, как он полностью остановится? Для этого он может поймать InterruptedException и выполнить некоторую очистку.

```
 Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                //Do something indefinetely
            }
        } catch (InterruptedException ex) {
            //Do some quick cleanup

            // In this case a simple return would do.
            // But if you are not 100% sure that the thread ends after
            // catching the InterruptedException you will need to raise another
            // one for the layers surrounding this code.
            Thread.currentThread().interrupt();
        }
    }
}
```

Но если у вас есть выражение catch-all в вашем коде, InterruptedException также будет поймано им, и прерывание не продолжится. Который в этом случае может привести к тупиковой ситуации, поскольку родительский поток ждет бесконечно, чтобы этот ада остановился на `t.join()` .

```
 Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                }
                catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Dead code as the interrupt exception was already caught in
            // the inner try-catch
            Thread.currentThread().interrupt();
```

```
        }
    }
}
```

Так что лучше поймать Исключения отдельно, но если вы настаиваете на использовании catch-all, по крайней мере, поймите InterruptedException индивидуально заранее.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                } catch (InterruptedException ex) {
                    throw ex; //Send it up in the chain
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Some quick cleanup code

            Thread.currentThread().interrupt();
        }
    }
}
```

Pitfall - Использование исключений для нормального управления потоком

Существует мантра, которую некоторые специалисты Java обычно читают:

> «Исключения должны использоваться только в исключительных случаях».

(Например: http://programmers.stackexchange.com/questions/184654)

Суть этого в том, что это плохая идея (в Java) использовать исключения и обработку исключений для реализации нормального управления потоком. Например, сравните эти два способа обращения с параметром, который может быть нулевым.

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

В этом примере мы (по дизайну) рассматриваем случай, когда `word` равно `null` как будто это пустое слово. Две версии имеют значение `null` либо с использованием обычных *if ... else,* либо *try ... catch* . Как нам решить, какая версия лучше?

Первый критерий - читаемость. Хотя читаемость трудно объективно определить количественно, большинство программистов согласятся с тем, что существенный смысл первой версии легче распознать. Действительно, чтобы действительно понять вторую форму, вам нужно понять, что `Math.min` `NullPointerException` не может быть `String.substring` методами `Math.min` или `String.substring` .

Второй критерий - эффективность. В версиях Java до Java 8 вторая версия значительно (на порядки) медленнее первой версии. В частности, построение объекта исключения влечет за собой захват и запись стековых кадров, на всякий случай, если требуется стек.

С другой стороны, существует множество ситуаций, когда использование исключений является более читаемым, более эффективным и (иногда) более правильным, чем использование условного кода для обработки «исключительных» событий. Действительно, есть редкие ситуации, когда необходимо использовать их для «не исключительных» событий; т.е. события, которые происходят относительно часто. Для последнего стоит взглянуть на способы сокращения накладных расходов на создание объектов исключений.

Pitfall - чрезмерные или неуместные стеки

Одним из наиболее неприятных вещей, которые могут сделать программисты, является разброс вызовов `printStackTrace()` во всем их коде.

Проблема заключается в том, что `printStackTrace()` будет писать stacktrace для стандартного вывода.

- Для приложения, предназначенного для конечных пользователей, которые не являются Java-программистами, stacktrace в лучшем случае неинформативна и в худшем случае вызывает тревогу.
- Вероятно, для серверного приложения никто не будет смотреть на стандартный вывод.

Лучше всего не вызывать `printStackTrace` напрямую, или если вы вызываете его, сделайте это так, чтобы трассировка стека была записана в файл журнала или файл ошибки, а не на консоль конечного пользователя.

Один из способов сделать это - использовать структуру ведения журнала и передать объект исключения в качестве параметра события журнала. Однако даже регистрация исключений может быть вредной, если это было сделано вредно. Рассмотрим следующее:

```
public void method1() throws SomeException {
    try {
```

```
        method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}
```

Если исключение `method2` в `method2` , вы, вероятно, увидите два экземпляра одной и той же stacktrace в файле журнала, что соответствует тому же отказу.

Короче говоря, либо регистрируйте исключение, либо повторно бросайте его (возможно, завернутое с другим исключением). Не делай того и другого.

Pitfall - прямое подклассирование «Throwable»

`Throwable` имеет два прямых подкласса: `Exception` и `Error` . Хотя можно создать новый класс, который напрямую расширяет `Throwable` , это нецелесообразно, так как многие приложения предполагают, что существуют только `Exception` и `Error` .

Более того, нет практической выгоды для прямого подкласса `Throwable` , поскольку полученный класс, по сути, просто проверенное исключение. В противном случае `Exception` подкласса приведет к такому же поведению, но будет более четко передавать ваши намерения.

Прочитайте Java Pitfalls - использование исключений онлайн:
https://riptutorial.com/ru/java/topic/5381/java-pitfalls---использование-исключений

глава 19: Java Pitfalls - синтаксис языка

Вступление

Некоторые нарушения языка программирования Java могут выполнять программу для получения неправильных результатов, несмотря на то, что они правильно составлены. Основной темой этой темы является список распространенных ошибок с их причинами и предложение правильного способа избежать таких проблем.

замечания

В этом разделе рассматриваются конкретные аспекты синтаксиса языка Java, которые либо подвержены ошибкам, либо не могут использоваться определенным образом.

Examples

Pitfall - игнорирование видимости метода

Даже опытные разработчики Java склонны думать, что Java имеет только три модификатора защиты. На самом деле у этого языка четыре! **Частый** (видимо, обычный) уровень видимости часто забывается.

Вы должны обратить внимание на то, какие методы вы публикуете. Публичные методы в приложении - это видимый API приложения. Это должно быть как можно меньше и компактнее, особенно если вы пишете библиотеку многократного использования (см. Также принцип SOLID). Важно также учитывать наглядность всех методов и использовать только защищенный или пакетный доступ, если это необходимо.

Когда вы объявляете методы, которые должны быть **закрытыми** как общедоступные, вы раскрываете внутренние детали реализации этого класса.

Следствием этого является то, что вы проверяете только публичные методы своего класса - на самом деле вы можете тестировать **только** общедоступные методы. Плохая практика заключается в повышении видимости частных методов, чтобы иметь возможность запускать единичные тесты против этих методов. Тестирование общедоступных методов, которые вызывают методы с более ограничительной видимостью, должно быть достаточным для тестирования всего API. Вы **никогда не** должны расширять свой API более публичными методами только для модульного тестирования.

Pitfall - Отсутствие «перерыва» в случае «переключения»

Эти проблемы с Java могут быть очень смущающими и иногда оставаться неоткрытыми до

тех пор, пока они не будут запущены в производство. Эффективное поведение в операторах switch часто полезно; однако отсутствие ключевого слова «break», когда такое поведение нежелательно, может привести к катастрофическим результатам. Если вы забыли поставить «break» в «case 0» в приведенном ниже примере кода, программа напишет «Zero», а затем «One», так как поток управления внутри здесь будет проходить весь оператор «switch» до тех пор, пока он достигает «перерыва». Например:

```
public static void switchCasePrimer() {
        int caseIndex = 0;
        switch (caseIndex) {
            case 0:
                System.out.println("Zero");
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            default:
                System.out.println("Default");
        }
}
```

В большинстве случаев чище решение было бы использовать интерфейсы и перемещать код с определенным поведением в отдельные реализации (*состав над наследованием*)

Если оператор switch неизбежен, рекомендуется документировать «ожидаемые» всплытия, если они происходят. Таким образом, вы показываете коллегам-разработчикам, что знаете о недостающем разрыве и что это ожидаемое поведение.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Pitfall - Пункты с запятой и отсутствующие фигурные скобки

Это ошибка, которая вызывает реальную путаницу для начинающих Java, по крайней мере, в первый раз, когда они это делают. Вместо того, чтобы писать это:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

они случайно пишут это:

```
if (feeling == HAPPY);
```

```
    System.out.println("Smile");
else
    System.out.println("Frown");
```

и недоумевают, когда компилятор Java сообщает им, что `else` неуместно. Компилятор Java с интерпретированием выше:

```
if (feeling == HAPPY)
    /*empty statement*/ ;
System.out.println("Smile");   // This is unconditional
else                           // This is misplaced.  A statement cannot
                               // start with 'else'
System.out.println("Frown");
```

В других случаях ошибки компиляции не будут, но код не будет делать то, что намеревается программист. Например:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

только печатает «Привет» один раз. Опять же паразитная точка с запятой означает, что тело цикла `for` представляет собой пустой оператор. Это означает, что следующий вызов `println` является безусловным.

Другой вариант:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

Это даст ошибку «Не могу найти символ» для `i` . Наличие ложной точки с запятой означает, что вызов `println` пытается использовать `i` вне его области видимости.

В этих примерах есть прямолинейное решение: просто удалите ложную точку с запятой. Однако из этих примеров можно извлечь более глубокие уроки:

1. Точка с запятой в Java не является «синтаксическим шумом». Наличие или отсутствие точки с запятой может изменить смысл вашей программы. Не добавляйте их в конце каждой строки.

2. Не доверяйте отступу вашего кода. В языке Java лишние пробелы в начале строки игнорируются компилятором.

3. Используйте автоматический индентор. Все IDE и многие простые текстовые редакторы понимают, как правильно отступать код Java.

4. Это самый важный урок. Следуйте последним рекомендациям стиля Java и поместите фигурные скобки вокруг операторов «then» и «else» и оператора body цикла. Открытая скобка (`{`) не должна быть на новой строке.

Если программист следовал правилам стиля, то пример `if` с неуместными точками с запятой выглядел бы так:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
    System.out.println("Frown");
}
```

Это выглядит странно для опытного глаза. Если вы автоматически отступом этот код, он, вероятно, будет выглядеть так:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

который должен выделиться как неудачный даже для новичков.

Pitfall - Оставляя брекеты: проблемы с «болтающимися, если» и «болтающимися»

В последней версии руководства по стилю Java Java указано, что операторы «then» и « else» в операторе `if` всегда должны быть заключены в «фигурные скобки» или «фигурные скобки». Аналогичные правила применяются к телам различных операторов цикла.

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                  // <- close brace
```

Синтаксис языка Java на самом деле не требуется. В самом деле, если «то» часть оператора `if` является единственным выражением, то законно оставить фигурные скобки

```
if (a)
    doSomething();
```

или даже

```
if (a) doSomething();
```

Однако есть опасность игнорировать правила стиля Java и оставлять фигурные скобки. В частности, вы значительно увеличиваете риск того, что код с ошибочным отступом будет неверно истолкован.

Проблема «болтаться»:

Рассмотрим пример кода сверху, переписанный без брекетов.

```
if (a)
   doSomething();
   doSomeMore();
```

Этот код, *кажется, говорит,* что вызовы `doSomething` и `doSomeMore` будут возникать *тогда и только тогда, когда* `a true` . Фактически, код имеет неправильный отступ. Спецификация языка Java, что `doSomeMore()` представляет собой отдельный оператор, следующий за оператором `if` . Правильный отступ выглядит следующим образом:

```
if (a)
   doSomething();
doSomeMore();
```

Проблема «болтаться еще»

Вторая проблема возникает, когда мы добавляем `else` к миксу. Рассмотрим следующий пример с отсутствующими фигурными скобками.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

Вышеприведенный код *говорит,* что `doZ` будет вызываться, когда `a` является `false` . Фактически, отступ неверен еще раз. Правильный отступ для кода:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

Если код был написан в соответствии с правилами стиля Java, это выглядело бы так:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
      doZ();
   }
}
```

Чтобы проиллюстрировать, почему это лучше, предположите, что вы случайно ошиблись в коде. У вас может получиться что-то вроде этого:

```
if (a) {                          if (a) {
   if (b) {                           if (b) {
      doX();                             doX();
   } else if (c) {                    } else if (c) {
      doY();                             doY();
} else {                          } else {
   doZ();                             doZ();
}                                     }
}                                 }
```

Но в обоих случаях ошибочный код «выглядит неправильно» для глаз опытного Java-программиста.

Pitfall - перегрузка вместо переопределения

Рассмотрим следующий пример:

```
public final class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = (firstName == null) ? "" : firstName;
        this.lastName = (lastName == null) ? "" : lastName;
    }

    public boolean equals(String other) {
        if (!(other instanceof Person)) {
            return false;
        }
        Person p = (Person) other;
        return firstName.equals(p.firstName) &&
                lastName.equals(p.lastName);
    }

    public int hashcode() {
        return firstName.hashCode() + 31 * lastName.hashCode();
    }
}
```

Этот код не будет вести себя так, как ожидалось. Проблема в том, что методы `equals` и `hashcode` для `Person` не переопределяют стандартные методы, определенные `Object` .

- Метод `equals` имеет неправильную подпись. Он должен быть объявлен как `equals(Object)` не `equals(String)` .
- Метод `hashcode` имеет неправильное имя. Это должен быть `hashCode()` (обратите внимание на капитал **C**).

Эти ошибки означают, что мы объявили случайные перегрузки, и они не будут использоваться, если `Person` используется в полиморфном контексте.

Однако есть простой способ справиться с этим (начиная с Java 5). Используйте аннотацию `@Override` всякий раз, когда вы *планируете* переопределить ваш метод:

Java SE 5

```
public final class Person {
    ...

    @Override
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashcode() {
        ....
    }
}
```

Когда мы добавим `@Override` аннотации к объявлению метода, компилятор проверяет , что метод *не* переопределить (или реализацию) метод , объявленный в суперклассе или интерфейсе. Итак, в приведенном выше примере компилятор даст нам две ошибки компиляции, которых должно быть достаточно, чтобы предупредить нас об ошибке.

Pitfall - Октальные литералы

Рассмотрим следующий фрагмент кода:

```
// Print the sum of the numbers 1 to 10
int count = 0;
for (int i = 1; i < 010; i++) {    // Mistake here ....
    count = count + i;
}
System.out.println("The sum of 1 to 10 is " + count);
```

Новичок Java может быть удивлен, узнав, что указанная выше программа печатает неверный ответ. Он фактически печатает сумму чисел от 1 до 8.

Причина в том, что целочисленный литерал, начинающийся с нуля ('0'), интерпретируется компилятором Java как восьмеричный литерал, а не десятичный литерал, как вы могли ожидать. Таким образом, `010` - это восьмеричное число 10, которое равно 8 десятичным.

Pitfall - объявление классов с теми же именами, что и стандартные классы

Иногда программисты, которые не знакомы с Java, ошибаются в определении класса с именем, которое совпадает с широко используемым классом. Например:

```
package com.example;

/**
 * My string utilities
```

```
 */
public class String {
    ....
}
```

Затем они задаются вопросом, почему возникают неожиданные ошибки. Например:

```
package com.example;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Если вы скомпилируете и затем попытаетесь запустить указанные выше классы, вы получите сообщение об ошибке:

```
$ javac com/example/*.java
$ java com.example.Test
Error: Main method not found in class test.Test, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Кто-то, смотрящий на код для класса `Test` , увидит объявление `main` и посмотрит на его подпись и задается вопросом, о чем жалуется команда `java` . Но на самом деле, команда `java` говорит правду.

Когда мы объявляем версию `String` в том же пакете, что и `Test` , эта версия имеет приоритет перед автоматическим импортом `java.lang.String` . Таким образом, подпись метода `Test.main` самом деле

```
void main(com.example.String[] args)
```

вместо

```
void main(java.lang.String[] args)
```

и команда `java` не будет распознавать *это* как метод точки входа.

Занятие. Не определяйте классы, которые имеют то же имя, что и существующие классы в `java.lang` , или другие обычно используемые классы в библиотеке Java SE. Если вы это сделаете, вы настроитесь на всевозможные неясные ошибки.

Pitfall - использование '==' для проверки логического

Иногда новый программист Java будет писать такой код:

```
public void check(boolean ok) {
```

```
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

Опытный программист заметил бы это как неуклюжий и хотел бы переписать его как:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

Тем не менее, с ошибкой `ok == true` чем просто неуклюжесть. Рассмотрим этот вариант:

```
public void check(boolean ok) {
    if (ok = true) {           // Oooops!
        System.out.println("It is OK");
    }
}
```

Здесь программист ошибся `==` as `=` ... и теперь код имеет тонкую ошибку. Выражение `x = true` безоговорочно присваивает `true x` а затем оценивается как `true` . Другими словами, теперь метод `check` будет печатать «Все в порядке» независимо от параметра.

Урок здесь состоит в том, чтобы избавиться от привычки использовать `== false` и `== true` . В дополнение к тому, чтобы быть многословным, они делают ваше кодирование более склонным к ошибкам.

Примечание. Возможная альтернатива `ok == true` которая позволяет избежать ошибок, заключается в использовании условий Yoda ; т.е. поставить литерал в левой части реляционного оператора, как в `true == ok` . Это работает, но большинство программистов, вероятно, согласятся с тем, что условия Yoda выглядят странно. Конечно, `ok` (или `!ok`) является более кратким и более естественным.

Pitfall - импорт подстановок может сделать ваш код хрупким

Рассмотрим следующий частичный пример:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Предположим, что когда вы впервые разработали код против версии 1.0 `somelib` и версии 1.0 `otherlib` . Затем в какой-то более поздний момент вам нужно обновить свои

зависимости до более поздних версий, и вы решите использовать версию `otherlib` версии 2.0. Также предположим, что одно из изменений, которые они сделали для `otherlib` между 1.0 и 2.0, заключалось в том, чтобы добавить класс `Context` .

Теперь, когда вы перекомпилируете `Test` , вы получите ошибку компиляции, сообщающую вам, что `Context` является неоднозначным импортом.

Если вы знакомы с кодовой базой, это, вероятно, всего лишь незначительные неудобства. Если нет, то у вас есть некоторая работа, чтобы сделать, чтобы решить эту проблему, здесь и потенциально в другом месте.

Проблема здесь в подстановочных импортах. С одной стороны, использование подстановочных знаков может сделать ваши классы на несколько строк короче. С другой стороны:

- Совместимые с обновлением изменения в других частях вашей кодовой базы, стандартных библиотек Java или сторонних библиотек могут привести к ошибкам компиляции.
- Читаемость страдает. Если вы не используете IDE, выяснить, какой из подстановочных импортов тянет в названный класс, может быть сложно.

Урок состоит в том, что плохой идеей использовать подстановочный импорт в коде, который должен быть долговечным. Конкретные (несимметричные) импорт не требуют больших усилий для поддержки, если вы используете среду IDE, и это стоит того.

Pitfall: использование 'assert' для проверки аргумента или пользователя

Иногда вопрос о StackOverflow заключается в том, целесообразно ли использовать `assert` для проверки аргументов, предоставляемых методу, или даже входов, предоставленных пользователем.

Простой ответ заключается в том, что он не подходит.

Лучшие альтернативы включают:

- Выбрасывание исключения IllegalArgumentException с использованием настраиваемого кода.
- Использование методов `Preconditions` доступных в библиотеке Google Guava.
- Используя `Validate` методы , доступные в библиотеке Apache Commons Lang3.

Это то, что предлагает Java Language Specification (JLS 14.10, для Java 8) по этому вопросу:

> Как правило, проверка утверждения включена во время разработки и тестирования программ и отключена для развертывания для повышения производительности.

Поскольку утверждения могут быть отключены, программы не должны предполагать, что выражения, содержащиеся в утверждениях, будут оцениваться. Таким образом, эти булевы выражения, как правило, не содержат побочных эффектов. Оценка такого булевского выражения не должна влиять на состояние, которое видно после завершения оценки. Это не является незаконным, если логическое выражение, содержащееся в утверждении, имеет побочный эффект, но оно, как правило, неприемлемо, поскольку это может привести к изменению поведения программы в зависимости от того, были ли включены или запрещены утверждения.

В свете этого утверждения не должны использоваться для проверки аргументов в публичных методах. Проверка аргументов обычно является частью контракта метода, и этот контракт должен быть подтвержден, если утверждения включены или отключены.

Вторая проблема с использованием утверждений для проверки аргументов заключается в том, что ошибочные аргументы должны приводить к соответствующему исключению во время выполнения (например, `IllegalArgumentException` , `ArrayIndexOutOfBoundsException` или `NullPointerException`). Ошибка утверждения не приведет к соответствующему исключению. Опять же, не запрещено использовать утверждения для проверки аргументов в публичных методах, но это, как правило, неуместно. Предполагается, что `AssertionError` никогда не будет поймано, но это можно сделать, поэтому правила для операторов try должны обрабатывать утверждения, появляющиеся в блоке try так же, как и текущее обращение с инструкциями throw.

Pitfall авто-распаковки нулевых объектов в примитивы

```
public class Foobar {
    public static void main(String[] args) {

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

Ловушка здесь заключается в том, что `null` сравнивается с `false` . Поскольку мы сравниваем примитивное `boolean` с `Boolean` , Java пытается *распаковать* `Boolean Object` в примитивный эквивалент, готовый для сравнения. Однако, поскольку это значение равно `null` , `NullPointerException` .

Java неспособна сравнивать примитивные типы с `null` значениями, что вызывает `NullPointerException` во время выполнения. Рассмотрим примитивный случай условия `false == null` ; это создало бы ошибку *времени компиляции* `incomparable types: int and <null>` .

Прочитайте Java Pitfalls - синтаксис языка онлайн:
https://riptutorial.com/ru/java/topic/5382/java-pitfalls---синтаксис-языка

глава 20: JavaBean

Вступление

JavaBeans (TM) - это образец для разработки API классов Java, который позволяет использовать экземпляры (бобы) в различных контекстах и использовать различные инструменты *без* явного написания кода Java. Шаблоны состоят из соглашений для определения геттеров и сеттеров для *свойств* , для определения конструкторов и для определения API-интерфейсов слушателей событий.

Синтаксис

- **Правила именования свойств JavaBean**
- Если свойство не является логическим, следует использовать префикс метода getter. Например, getSize () является допустимым именем получателя JavaBeans для свойства с именем «размер». Имейте в виду, что вам не нужно иметь переменный именованный размер. Имя свойства выводится из геттеров и сеттеров, а не через любые переменные в вашем классе. То, что вы возвращаете из getSize (), зависит от вас.
- Если свойство является логическим, префикс метода getter либо get, либо is. Например, getStopped () или isStopped () являются действительными именами JavaBeans для логического свойства.
- Должен быть установлен префикс метода setter. Например, setSize () является допустимым именем JavaBean для свойства с именем size.
- Чтобы заполнить имя метода getter или setter, измените первую букву имени свойства на верхний регистр и затем добавьте его в соответствующий префикс (get, is или set).
- Сигнатуры метода Setter должны быть помечены как public, с типом возвращаемого типа и аргументом, который представляет тип свойства.
- Подписи метода Getter должны быть отмечены как public, не принимать аргументы и иметь тип возвращаемого значения, который соответствует типу аргумента метода setter для этого свойства.
- **Правила именования слушателей JavaBean**
- Имена методов прослушивателя, используемые для «регистрации» слушателя с источником события, должны использовать префикс add, за которым следует тип слушателя. Например, addActionListener () является допустимым именем для метода, который источник события должен будет разрешить другим пользователям регистрироваться для событий Action.
- В именах методов прослушивателя, используемых для удаления («отменить регистрацию»), слушатель должен использовать префикс remove, за которым следует тип слушателя (с использованием тех же правил, что и метод добавления регистрации).

- Тип слушателя, который нужно добавить или удалить, должен быть передан как аргумент метода.
- Имена методов прослушивателя должны заканчиваться словом «Слушатель».

замечания

Для того чтобы класс был Java Bean, должен следовать этому стандарту - в целом:

- Все его свойства должны быть частными и доступны только через геттеры и сеттеры.
- Он должен иметь открытый конструктор без аргументов.
- Необходимо реализовать интерфейс `java.io.Serializable` .

Examples

Базовый Java-компонент

```
public class BasicJavaBean implements java.io.Serializable{

    private int value1;
    private String value2;
    private boolean value3;

    public BasicJavaBean(){}

    public void setValue1(int value1){
        this.value1 = value1;
    }

    public int getValue1(){
        return value1;
    }

    public void setValue2(String value2){
        this.value2 = value2;
    }

    public String getValue2(){
        return value2;
    }

    public void setValue3(boolean value3){
        this.value3 = value3;
    }

    public boolean isValue3(){
        return value3;
    }
}
```

Прочитайте JavaBean онлайн: https://riptutorial.com/ru/java/topic/8157/javabean

глава 21: Java-агенты

Examples

Изменение классов с помощью агентов

Во-первых, убедитесь, что используемый агент имеет следующие атрибуты в Manifest.mf:

```
Can-Redefine-Classes: true
Can-Retransform-Classes: true
```

Запуск Java-агента позволит агенту получить доступ к классу Instrumentation. С помощью Instrumentation вы можете вызвать *addTransformer (трансформатор ClassFileTransformer)* . ClassFileTransformers позволит вам переписать байты классов. Класс имеет только один метод, который поставляет ClassLoader, который загружает класс, имя класса, экземпляр java.lang.Class, это ProtectionDomain и, наконец, байты самого класса.

Это выглядит так:

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
          ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

Модификация класса исключительно из байтов может занять возраст. Чтобы исправить это, есть библиотеки, которые могут быть использованы для преобразования байтов класса в нечто более пригодное для использования.

В этом примере я буду использовать ASM, но другие альтернативы, такие как Javassist и BCEL, имеют схожие функции.

```
ClassNode getNode(byte[] bytes) {
    // Create a ClassReader that will parse the byte array into a ClassNode
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        // This populates the ClassNode
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
        cr = null;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return cn;
}
```

Отсюда можно внести изменения в объект ClassNode. Это позволяет невероятно легко изменять доступ к полям / методам. Плюс с API-интерфейсом ASM, изменяющим байт-код методов, это легкий ветерок.

После завершения редактирования вы можете преобразовать ClassNode обратно в байты со следующим методом и вернуть их в метод *преобразования* :

```
public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :
ClassWriter.COMPUTE_FRAMES);
    cn.accept(cw);
    byte[] b = cw.toByteArray();
    return b;
}
```

Добавление агента во время выполнения

Агенты могут быть добавлены в JVM во время выполнения. Чтобы загрузить агента, вам нужно использовать *VirtualMachine.attatch (* Attach API) Attach API *(String id)* . Затем вы можете загрузить скомпилированную фразу агента следующим способом:

```
public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attach(vm.id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Это не будет вызывать *premain ((String agentArgs, Инструментарий INST)* в загруженном агенте, но вместо этого будет вызывать *agentmain (String agentArgs, Инструментарий INST).* Это требует *Агент-класс* должны быть установлено в средстве manifest.mf.

Настройка базового агента

Класс Premain будет содержать метод *«premain (String agentArgs Instrumentation inst)»*

Вот пример:

```
import java.lang.instrument.Instrumentation;

public class PremainExample {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println(agentArgs);
    }
}
```

Когда они скомпилированы в файл jar, откройте манифест и убедитесь, что он имеет атрибут Premain-Class.

Вот пример:

```
Premain-Class: PremainExample
```

Чтобы использовать агент с другой java-программой «myProgram», вы должны определить агента в аргументах JVM:

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

Прочитайте Java-агенты онлайн: https://riptutorial.com/ru/java/topic/1265/java-агенты

глава 22: Java-версии, версии, выпуски и дистрибутивы

Examples

Различия между Java SE JRE или Java SE JDK

Sun / Oracle выпуски Java SE представлены в двух формах: JRE и JDK. Говоря простыми словами, JRE поддерживают запуск Java-приложений, а JDK также поддерживают разработку Java.

Java Runtime Environment

Java Runtime Environment или JRE-дистрибутивы состоят из набора библиотек и инструментов, необходимых для запуска и управления приложениями Java. Инструменты в типичной современной JRE включают:

- `java` команда для запуска Java-программы в JVM (виртуальная машина Java)
- Команда `jjs` для запуска движка Nashorn Javascript.
- Команда `keytool` для манипуляции Java-хранилищами.
- Команда `policytool` для редактирования политик безопасности изолированной `policytool` .
- Инструменты `pack200` и `unpack200` для упаковки и распаковки файла «pack200» для веб-развертывания.
- `orbd` , `rmid` , `rmiregistry` и `tnameserv` которые поддерживают приложения Java CORBA и RMI.

Установщики Desktop JRE включают плагин Java, подходящий для некоторых веб-браузеров. Это преднамеренно исключено из «Server JRE» installers.linux syscall read benchmarku

Начиная с версии Java 7 6, установщики JRE включили JavaFX (версия 2.2 или новее).

Java Development Kit

Набор Java Development Kit или JDK включает инструменты JRE и дополнительные инструменты для разработки программного обеспечения Java. Дополнительные инструменты обычно включают:

- Команда `javac` , которая компилирует исходный код Java («.java») в байт-код файлов («.class»).

- Инструменты для создания JAR-файлов, таких как `jar` и `jarsigner`
- Средства разработки, такие как:
  - `appletviewer` для запуска апплетов
  - `idlj` компилятор CORBA IDL для Java
  - `javah` генератор-заглушка JNI
  - `native2ascii` для преобразования набора символов исходного кода Java
  - `schemagen` генератор схемы Java-XML (часть JAXB)
  - `serialver` генерирует строку версии Serialization Java Object.
  - инструменты поддержки `wsgen` и `wsimport` для JAX-WS
- Диагностические инструменты, такие как:
  - `jdb` базовый отладчик Java
  - `jmap` и `jhat` для демпинга и анализа кучи Java.
  - `jstack` для получения дампа потока потоков.
  - `javap` для изучения файлов «.class».
- Инструменты управления приложениями и мониторинга, такие как:
  - `jconsole` - консоль управления,
  - `jstat` , `jstatd` , `jinfo` и `jps` для мониторинга приложений

Типичная установка Sun / Oracle JDK также включает ZIP-файл с исходным кодом библиотек Java. До появления Java 6 это был единственный общедоступный исходный код Java.

Начиная с Java 6, полный исходный код для OpenJDK доступен для загрузки с сайта OpenJDK. Обычно он не входит в пакеты JDK (Linux), но доступен как отдельный пакет.

В чем разница между Oracle Hotspot и OpenJDK

Ортогональная JRE по сравнению с JDK-дихотомией, есть два типа выпусков Java, которые широко доступны:

- Релизы Oracle Hotspot - это те, которые вы загружаете с сайтов загрузки Oracle.
- Выпуски OpenJDK - это те, которые построены (как правило, сторонними поставщиками) из исходных репозиториев OpenJDK.

В функциональных терминах существует небольшая разница между выпуском Hotspot и выпуском OpenJDK. В Hotspot есть несколько дополнительных «корпоративных» функций, которые могут использовать клиенты Oracle (оплачивающие) Java, но кроме того, одна и та же технология присутствует как в Hotspot, так и в OpenJDK.

Еще одно преимущество Hotspot над OpenJDK заключается в том, что выпуски исправлений для Hotspot, как правило, доступны чуть раньше. Это также зависит от того, насколько гибким является ваш поставщик OpenJDK; например, сколько времени потребуется команде разработчиков дистрибутива Linux для подготовки и QA новой сборки OpenJDK и получить ее в своих публичных хранилищах.

С другой стороны, выпуски Hotspot недоступны из репозиториев пакетов для большинства дистрибутивов Linux. Это означает, что сохранение вашего программного обеспечения Java на современном компьютере Linux обычно более эффективно, если вы используете Hotspot.

Различия между Java EE, Java SE, Java ME и JavaFX

Технология Java - это язык программирования и платформа. Язык программирования Java - это высокоуровневый объектно-ориентированный язык с особым синтаксисом и стилем. Java-платформа - это особая среда, в которой работают приложения языка Java.

Существует несколько платформ Java. Многие разработчики, даже давние разработчики языка Java, не понимают, как разные платформы связаны друг с другом.

Языковые платформы Java

Существует четыре платформы языка программирования Java:

- Платформа Java, стандартная версия (Java SE)
- Платформа Java, Enterprise Edition (Java EE)
- Java Platform, Micro Edition (Java ME)
- Java FX

Все платформы Java состоят из виртуальной машины Java (VM) и интерфейса прикладного программирования (API). Виртуальная машина Java - это программа для конкретной аппаратной и программной платформы, которая запускает приложения для Java-технологий. API представляет собой набор программных компонентов, которые можно использовать для создания других программных компонентов или приложений. Каждая платформа Java предоставляет виртуальную машину и API, что позволяет приложениям, написанным для этой платформы, работать на любой совместимой системе со всеми преимуществами языка программирования Java: независимость от платформы, мощность, стабильность, простота разработки и безопасность.

Java SE

Когда большинство людей думает о языке программирования Java, они думают о Java SE API. API Java SE обеспечивает основные функциональные возможности языка программирования Java. Он определяет все, от базовых типов и объектов языка программирования Java до классов высокого уровня, которые используются для создания сетей, обеспечения безопасности, доступа к базе данных, разработки графического

интерфейса пользователя (GUI) и анализа XML.

В дополнение к основному API платформа Java SE состоит из виртуальной машины, средств разработки, технологий развертывания и других библиотек классов и наборов инструментов, обычно используемых в приложениях Java.

Java EE

Платформа Java EE построена поверх платформы Java SE. Платформа Java EE обеспечивает среду API и среду выполнения для разработки и запуска широкомасштабных многоуровневых, масштабируемых, надежных и безопасных сетевых приложений.

Java ME

Платформа Java ME предоставляет API и небольшую виртуальную машину для запуска приложений Java для программирования на небольших устройствах, таких как мобильные телефоны. API - это подмножество Java SE API, а также специальные библиотеки классов, полезные для разработки небольших приложений. Приложения Java ME часто являются клиентами служб платформы Java EE.

Java FX

Технология Java FX - это платформа для создания богатых интернет-приложений, написанных на Java FX ScriptTM. Java FX Script - это статически типизированный декларативный язык, который скомпилирован в байт-код Java-технологии, который затем может быть запущен на виртуальной машине Java. Приложения, написанные для платформы Java FX, могут включать и связываться с языковыми классами Java-программирования и могут быть клиентами служб платформы Java EE.

- Взято из документации Oracle

Версии Java SE

История версий Java SE

В следующей таблице приведены сроки значительных основных версий платформы Java SE.

Java SE Version [1]	Кодовое имя	Окончание срока службы (бесплатно [2])	Дата выхода
Java SE 9 (ранний доступ)	*Никто*	будущее	2017-07-27 (по оценкам)
Java SE 8	*Никто*	будущее	2014-03-18
Java SE 7	дельфин	2015-04-14	2011-07-28
Java SE 6	мустанг	2013-04-16	2006-12-23
Java SE 5	тигр	2009-11-04	2004-10-04
Java SE 1.4.2	Богомол	до 2009-11-04	2003-06-26
Java SE 1.4.1	Хоппер / Кузнечик	до 2009-11-04	2002-09-16
Java SE 1.4	Мерлин	до 2009-11-04	2002-02-06
Java SE 1.3.1	Божья коровка	до 2009-11-04	2001-05-17
Java SE 1.3	Пустельга	до 2009-11-04	2000-05-08
Java SE 1.2	Детская площадка	до 2009-11-04	1998-12-08
Java SE 1.1	бенгальский огонь	до 2009-11-04	1997-02-19
Java SE 1.0	дуб	до 2009-11-04	1996-01-21

Примечания:

1. Ссылки на онлайн-копии документации соответствующих выпусков на веб-сайте Oracle. Документация для многих старых версий больше не доступна в Интернете, хотя ее обычно можно загрузить из Oracle Java Archives.

2. Большинство исторических версий Java SE прошли официальные даты окончания жизни. Когда версия Java проходит эту веху, Oracle перестает предоставлять для нее бесплатные обновления. Обновления по-прежнему доступны для клиентов с контрактами на поддержку.

Источник:

- Дата выпуска JDK от Roedy Green от Canadian Mind Products

Основные сведения о версии Java SE

Версия Java SE	Особенности
Java SE 8	Лямбда-выражения и потоки, созданные с помощью MapReduce. Двигатель Nashorn Javascript. Аннотации по типам и повторяющиеся аннотации. Неподписанные арифметические расширения. Новые API дат и времени. Статически связанные библиотеки JNI. Пусковая установка JavaFX. Удаление PermGen.
Java SE 7	Строковые переключатели, *try-with-resource* , алмаз (`<>`), улучшенные числовые литералы и улучшения обработки / восстановления исключений. Расширения библиотеки параллелизма. Расширенная поддержка для родных файловых систем. Timsort. Криптографические алгоритмы ECC. Улучшена поддержка 2D-графики (GPU). Вставляемые аннотации.
Java SE 6	Значительные улучшения производительности платформы JVM и Swing. API языка сценариев и движок Javascript Mozilla Rhino. JDBC 4.0. API компилятора. JAXB 2.0. Поддержка веб-сервисов (JAX-WS)
Java SE 5	Generics, аннотации, авто-бокс, классы `enum` , varargs, улучшенные `for` циклов и статический импорт. Спецификация модели памяти Java. Улучшения Swing и RMI. Добавление пакета `java.util.concurrent.*` И `Scanner` .
Java SE 1.4	Ключевое слово `assert` . Классы регулярных выражений. Цепочка исключений. NIO API - неблокирующий ввод-вывод, `Buffer` и `Channel` . `java.util.logging.*` API. API ввода-вывода изображений. Интегрированный XML и XSLT (JAXP). Интегрированная безопасность и криптография (JCE, JSSE, JAAS). Встроенный Java Web Start. API настроек.
Java SE 1.3	Включен JVM HotSpot. Интеграция CORBA / RMI. Интерфейс именования и интерфейса Java (JNDI). Рамка отладчика (JPDA). API JavaSound. Proxy API.
Java SE 1.2	Ключевое слово `strictfp` . Swing API. Плагин Java (для веб-браузеров). CORBA. Структура коллекций.
Java SE 1.1	Внутренние классы. Отражение. JDBC. RMI. Unicode / символьные потоки. Поддержка интернационализации. Капитальный ремонт модели событий AWT. JavaBeans.

Источник:

- Википедия: история версий Java

Прочитайте Java-версии, версии, выпуски и дистрибутивы онлайн:
https://riptutorial.com/ru/java/topic/8973/java-версии--версии--выпуски-и-дистрибутивы

глава 23: JAXB

Вступление

JAXB или Java Architecture for XML Binding (JAXB) - это программная среда, которая позволяет разработчикам Java сопоставлять классы Java с представлениями XML. Эта страница познакомит читателей с JAXB, используя подробные примеры о его функциях, предоставляемых главным образом для маршалинга и неархивирования объектов Java в xml-формате и наоборот.

Синтаксис

- JAXB.marshall (объект, fileObjOfXML);
- Object obj = JAXB.unmarshall (fileObjOfXML, className);

параметры

параметр	подробности
fileObjOfXML	`File` объект XML-файла
имя класса	Имя класса с расширением `.class`

замечания

Используя инструмент XJC, доступный в JDK, java-код для структуры xml, описанный в xml-схеме (файл `.xsd`), может быть сгенерирован автоматически, см. Тему XJC .

Examples

Написание XML-файла (сортировка объекта)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Используя аннотацию `XMLRootElement` , мы можем пометить класс как корневой элемент XML-файла.

```
import java.io.File;
import javax.xml.bind.JAXB;

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}
```

`marshal()` используется для записи содержимого объекта в файл XML. Здесь `user` объект и новый объект `File` передаются в качестве аргументов `marshal()` .

При успешном выполнении это создает XML-файл с именем `UserDetails.xml` в пути класса с содержимым ниже.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Чтение XML-файла (unmarshalling)

Чтобы прочитать XML-файл с именем `UserDetails.xml` с приведенным ниже содержимым

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Нам нужен класс POJO с именем `User.java` как `User.java` ниже.

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;
```

```
    // getters and setters
}
```

Здесь мы создали переменные и имя класса в соответствии с узлами XML. Чтобы сопоставить их, мы используем аннотацию `XmlRootElement` в классе.

```
public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName());   // prints Jon Skeet
            System.out.println(user.getUserID()); // prints 8884321
        } catch (Exception e) {
            System.err.println("Exception occurred while reading the XML!");
        }
    }
}
```

Здесь метод `unmarshal()` используется для анализа XML-файла. В качестве двух аргументов требуется имя файла XML и тип класса. Затем мы можем использовать методы getter объекта для печати данных.

Использование XmlAdapter для генерации желаемого формата XML

Когда желаемый формат XML отличается от объектной модели Java, реализация XmlAdapter может использоваться для преобразования объекта модели в объект XML-формата и наоборот. В этом примере показано, как поместить значение поля в атрибут элемента с именем поля.

```
public class XmlAdapterExample {

    @XmlAccessorType(XmlAccessType.FIELD)
    public static class NodeValueElement {

        @XmlAttribute(name="attrValue")
        String value;

        public NodeValueElement() {
        }

        public NodeValueElement(String value) {
            super();
            this.value = value;
        }

        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }
    }
```

```
    public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

        @Override
        public NodeValueElement marshal(String v) throws Exception {
            return new NodeValueElement(v);
        }

        @Override
        public String unmarshal(NodeValueElement v) throws Exception {
            if (v==null) return "";
            return v.getValue();
        }
    }

    @XmlRootElement(name="DataObject")
    @XmlAccessorType(XmlAccessType.FIELD)
    public static class DataObject {

        String elementWithValue;

        @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
        String elementWithAttribute;
    }

    public static void main(String[] args) {
        DataObject data = new DataObject();
        data.elementWithValue="value1";
        data.elementWithAttribute ="value2";

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        JAXB.marshal(data, baos);

        String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

        System.out.println(xmlString);
    }
}
```

Автоматическая конфигурация отображения XML / XML (@XmlAccessorType)

Аннотация `@XmlAccessorType` определяет, будут ли поля / свойства автоматически сериализованы в XML. Обратите внимание, что аннотации полей и методов `@XmlElement` , `@XmlAttribute` или `@XmlTransient` имеют приоритет над настройками по умолчанию.

```
public class XmlAccessTypeExample {

@XmlAccessorType(XmlAccessType.FIELD)
static class AccessorExampleField {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
```

```
}

@XmlAccessorType(XmlAccessType.NONE)
static class AccessorExampleNone {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PROPERTY)
static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("\nField:");
        JAXB.marshal(new AccessorExampleField(), System.out);
        System.out.println("\nNone:");
        JAXB.marshal(new AccessorExampleNone(), System.out);
        System.out.println("\nProperty:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);
        System.out.println("\nPublic:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // outer class end
```

Выход

```
Field:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleField>
    <field>value1</field>
</accessorExampleField>
```

```
None:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleNone/>

Property:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleProperty>
    <getter>getter</getter>
</accessorExampleProperty>

Public:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExamplePublic>
    <field>value1</field>
    <getter>getter</getter>
</accessorExamplePublic>
```

Конфигурация отображения XML вручную / свойства

Аннотации `@XmlElement` , `@XmlAttribute` или `@XmlTransient` и другие в пакете `javax.xml.bind.annotation` позволяют программисту указать, какие и как помеченные поля или свойства должны быть сериализованы.

```
@XmlAccessorType(XmlAccessType.NONE) // we want no automatic field/property marshalling
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";

    @XmlElement(name="different name")
    private String oneName="different name value";

    @XmlTransient
    private String transientField = "will not get serialized ever";


    @XmlElement
    public String getModifiedTransientValue() {
        return transientField.replace(" ever", ", unless in a getter");
    }

    public void setModifiedTransientValue(String val) {} // empty on purpose


    public static void main(String[] args) {
        try {
            JAXB.marshal(new ManualXmlElementsExample(), System.out);
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        }
    }
}
```

Указание экземпляра XmlAdapter для (повторного) использования существующих данных

Иногда необходимо использовать конкретные экземпляры данных. Отдых нежелателен, и ссылка на `static` данные будет иметь запах кода.

Можно указать `XmlAdapter` экземпляр по `Unmarshaller` должен использовать, что позволяет пользователю использовать `XmlAdapter` ей, не равна нулю-Arg конструктор и / или передавать данные к адаптеру.

пример

Пользовательский класс

Следующий класс содержит имя и образ пользователя.

```
import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }
```

```
}
```

адаптер

Чтобы избежать одновременного создания одного и того же изображения в памяти (а также загрузки данных снова), адаптер сохраняет изображения на карте.

Java SE 7

Для действительного кода Java 7 замените метод `getImage` на

```
public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        }
        imageCache.put(url, image);
        reverseIndex.put(image, url);
    }
    return image;
}
```

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // using a single lookup using Java 8 methods
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null,
ex);
                return null;
            }
        });
    }
```

```
    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }

}
```

Примеры XML

Следующие 2 xmls для *Jon Skeet* и его земля 2, которые выглядят одинаково и поэтому используют один и тот же аватар.

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r=PG"
```

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet (Earth 2)"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r=PG"
```

Использование адаптера

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// specifiy the adapter instance to use for every
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// unmarshal second xml using the same adapter instance
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// yields true, since image is reused
System.out.println(result1.getImage() == result2.getImage());
```

Связывание пространства имен XML с сериализуемым классом Java.

Это пример файла `package-info.java` который связывает пространство имен XML с сериализуемым классом Java. Это должно быть помещено в тот же пакет, что и классы Java, которые должны быть сериализованы с использованием пространства имен.

```
/**
 * A package containing serializable classes.
 */
@XmlSchema
(
    xmlns =
    {
        @XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
    },
    namespace = MySerializableClass.NAMESPACE,
    elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;
```

Использование XmlAdapter для обрезки строки.

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

А в package-info.java добавьте следующее объявление.

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type =
String.class)
package com.example.xml.jaxb.bindings;// Packge where you intend to apply trimming filter

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

Прочитайте JAXB онлайн: https://riptutorial.com/ru/java/topic/147/jaxb

глава 24: JAX-WS

Examples

Основная аутентификация

Способ вызова JAX-WS с базовой аутентификацией немного неочевиден.

Вот пример, где `Service` является представлением класса сервиса, а `Port` - это служебный порт, к которому вы хотите получить доступ.

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

Прочитайте JAX-WS онлайн: https://riptutorial.com/ru/java/topic/4105/jax-ws

глава 25: JMX

Вступление

Технология JMX предоставляет инструменты для создания распределенных, основанных на Web, модульных и динамических решений для управления и мониторинга устройств, приложений и сетей, основанных на услугах. По дизайну этот стандарт подходит для адаптации устаревших систем, внедрения новых решений для управления и мониторинга и подключения к будущим.

Examples

Простой пример с платформой MBean Server

Предположим, у нас есть сервер, который регистрирует новых пользователей и приветствует их некоторым сообщением. И мы хотим контролировать этот сервер и изменять некоторые его параметры.

Во-первых, нам нужен интерфейс с нашими методами контроля и контроля

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

И некоторая простая реализация, которая позволит нам увидеть, как она работает и как мы ее влияем

```
public class UserCounter implements UserCounterMBean, Runnable {
    private AtomicLong sleepTime = new AtomicLong(10000);
    private AtomicInteger userCount = new AtomicInteger(0);
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");
    private AtomicBoolean interrupted = new AtomicBoolean(false);

    @Override
    public long getSleepTime() {
        return sleepTime.get();
    }
```

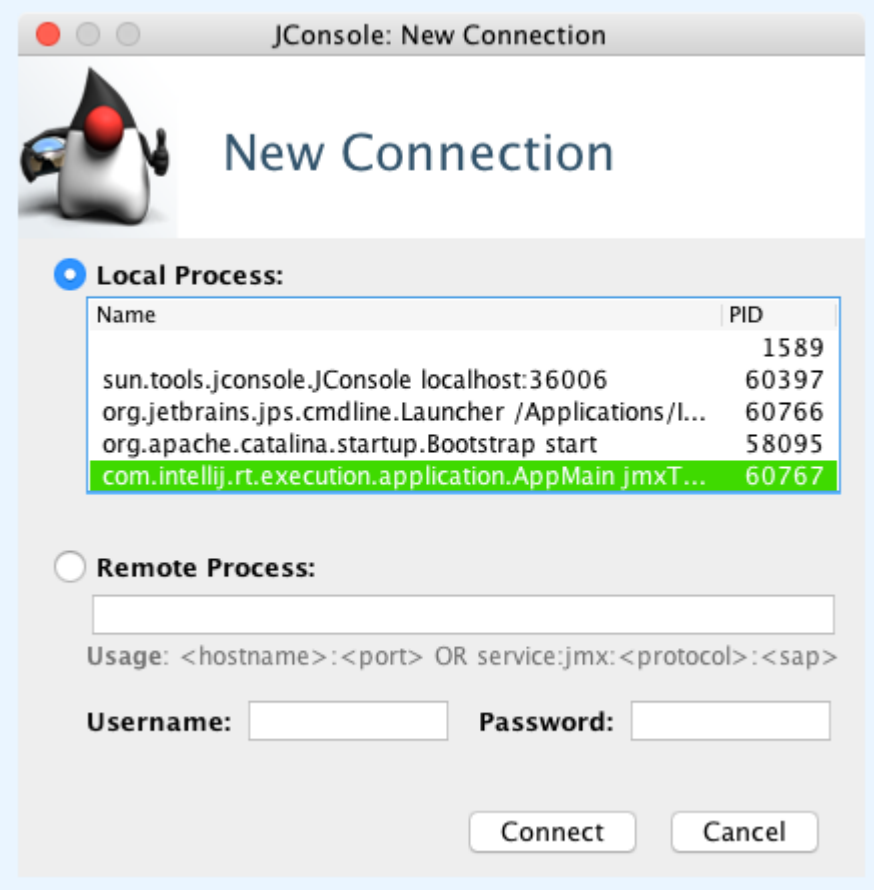
```
    @Override
    public void setSleepTime(long sleepTime) {
        this.sleepTime.set(sleepTime);
    }

    @Override
    public int getUserCount() {
        return userCount.get();
    }

    @Override
    public void setUserCount(int userCount) {
        this.userCount.set(userCount);
    }

    @Override
    public String getGreetingString() {
        return greetingString.get();
    }

    @Override
    public void setGreetingString(String greetingString) {
        this.greetingString.set(greetingString);
    }

    @Override
    public void stop() {
        this.interrupted.set(true);
    }

    @Override
    public void run() {
        while (!interrupted.get()) {
            try {
                System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
                Thread.sleep(sleepTime.get());
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

Для простого примера с локальным или удаленным управлением нам необходимо зарегистрировать наш MBean:

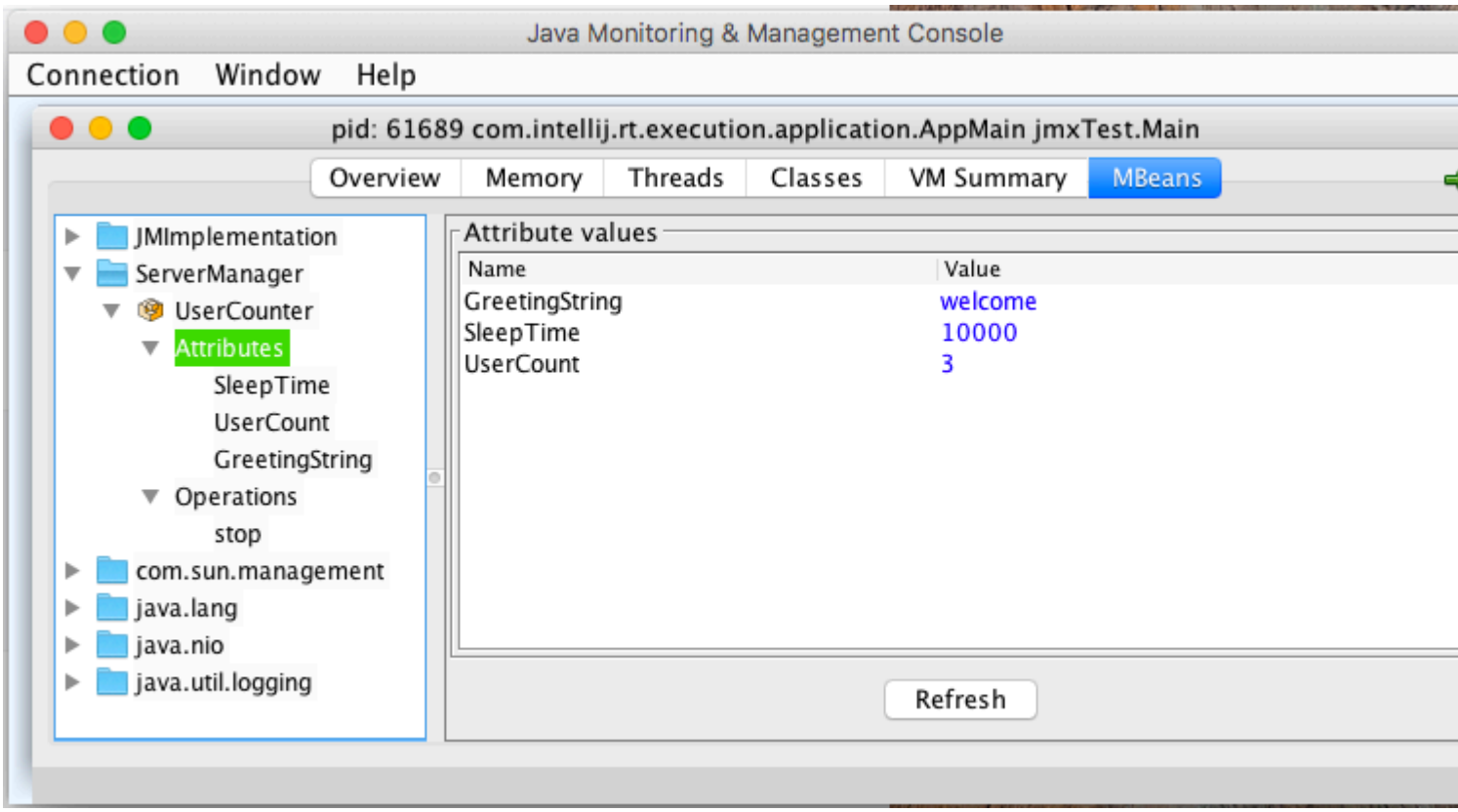
```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
```

```
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

        final Thread thread = new Thread(userCounter);
        thread.start();
        thread.join();
    }
}
```

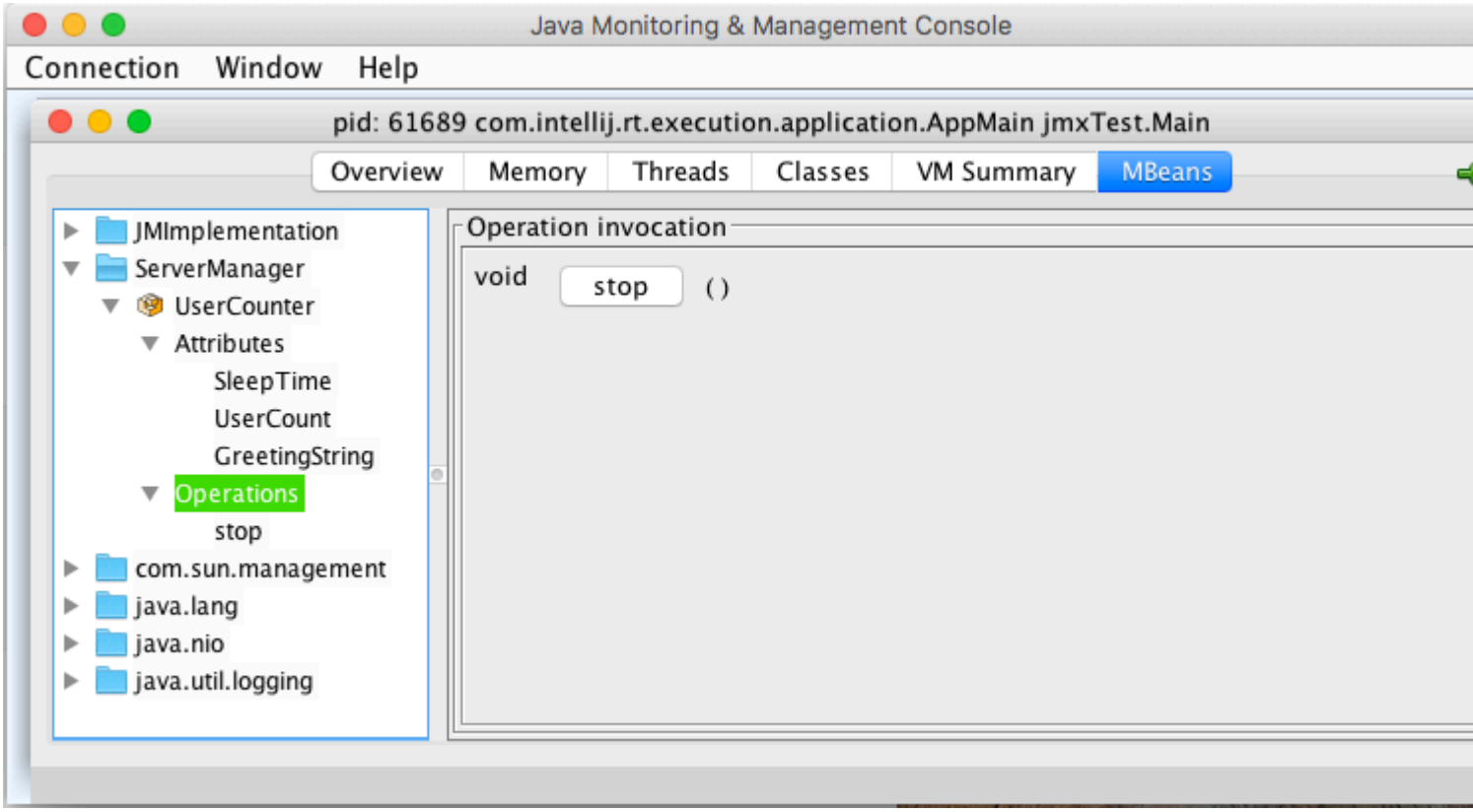
После этого мы можем запустить наше приложение и подключиться к нему через jConsole, который можно найти в каталоге `$JAVA_HOME/bin` . Во-первых, нам нужно найти наш локальный процесс Java с нашим приложением



затем перейдите на вкладку MBeans и найдите MBean, который мы использовали в нашем основном классе как `ObjectName` (в приведенном выше примере это `ServerManager`). В разделе « `Attributes` » мы видим атрибуты. Если вы указали только метод get, атрибут будет доступен для чтения, но не может быть записан. Если вы указали методы get и set, атрибут будет доступен для чтения и записи.



Указанные методы могут быть вызваны в разделе « `Operations` ».



Если вы хотите использовать удаленное управление, вам понадобятся дополнительные параметры JVM, например:

```
-Dcom.sun.management.jmxremote=true //true by default
-Dcom.sun.management.jmxremote.port=36006
```

```
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

Эти параметры можно найти в главе 2 руководств JMX . После этого вы сможете удаленно подключиться к вашему приложению через jConsole с помощью `jconsole host:port` или с указанием `host:port` или `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` в jConsole GUI.

Полезные ссылки:

- Руководства JMX
- Лучшие практики JMX

Прочитайте JMX онлайн: https://riptutorial.com/ru/java/topic/9278/jmx

глава 26: JNDI

Examples

RMI через JNDI

В этом примере показано, как JNDI работает в RMI. Он имеет две роли:

- для предоставления серверу API-интерфейса bind / unbind / rebind в реестре RMI
- для предоставления клиенту API поиска / списка в реестре RMI.

Реестр RMI является частью RMI, а не JNDI.

Чтобы сделать это простым, мы будем использовать `java.rmi.registry.CreateRegistry()` для создания реестра RMI.

1. Server.java (сервер JNDI)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.Hashtable;

/**
 * JNDI Server
 * 1.create a registry on port 1234
 * 2.bind JNDI
 * 3.wait for connection
 * 4.clean up and end
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
            registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
            jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void bindJNDI(String name, Object obj) throws NamingException {
        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }

    public static void unInitJNDI() throws NamingException {
        ctx.close();
    }

    public static void main(String[] args) throws NamingException, IOException {
        initJNDI();
        NMessage msg = new NMessage("Just A Message");
        bindJNDI("/neohope/jndi/test01", msg);
        System.in.read();
        unbindJNDI("/neohope/jndi/test01");
        unInitJNDI();
    }
}
```

2. Client.java (клиент JNDI)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.init context
 * 2.lookup registry for the service
 * 3.use the service
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}
```

3. NMessage.java (класс сервера RMI)

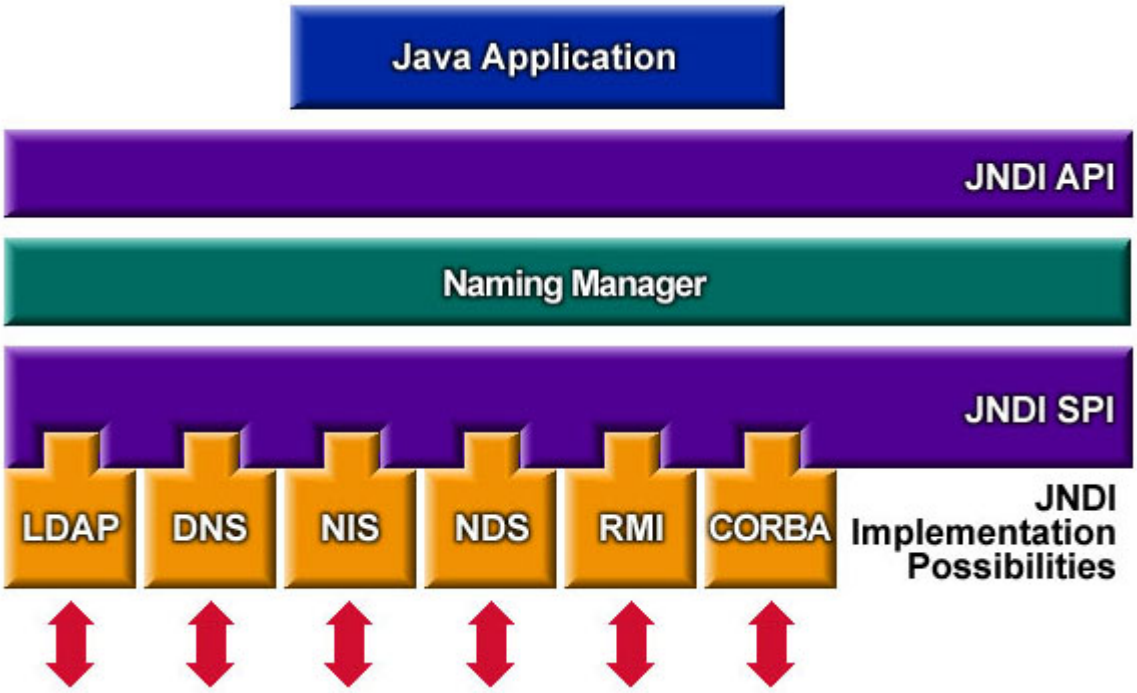
```
package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;
```

```
/**
 * NMessage
 * RMI server class
 * must implements Remote and Serializable
 */
public class NMessage implements Remote, Serializable {
    public String message = "";

    public NMessage(String message)
    {
        this.message = message;
    }
}
```

Как запустить eaxmple:

1. создавать и запускать сервер
2. создавать и запускать клиент

Вводить



Интерфейс **именования и каталогов Java (JNDI)** - это Java API для службы каталогов, которая позволяет клиентам программного обеспечения Java обнаруживать и искать данные и объекты через имя. Он разработан, чтобы быть независимым от какой-либо конкретной реализации именования или службы каталогов.

Архитектура JNDI состоит из интерфейса **API** (Application Programming Interface) и **SPI** (интерфейс поставщика услуг). Приложения Java используют этот API для доступа к различным службам именования и каталогов. SPI позволяет подключать различные службы имен и каталогов прозрачно, позволяя Java-приложению использовать API технологии JNDI для доступа к своим службам.

Как видно из рисунка выше, JNDI поддерживает LDAP, DNS, NIS, NDS, RMI и CORBA. Конечно, вы можете продлить его.

Как это устроено

В этом примере Java RMI использует JNDI API для поиска объектов в сети. Если вы хотите найти объект, вам нужно как минимум две части информации:

- Где найти объект

Реестр RMI управляет привязками имен, он сообщает вам, где найти объект.

- Имя объекта

Что такое имя объекта? Обычно это строка, она также может быть объектом, который реализует интерфейс Name.

Шаг за шагом

1. Сначала вам нужен реестр, который управляет привязкой имени. В этом примере мы используем `java.rmi.registry.LocateRegistry` .

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. И клиенту, и серверу нужен Контекст. Сервер использует контекст для привязки имени и объекта. Клиент использует контекст для поиска имени и получения объекта.

```
//We use com.sun.jndi.rmi.registry.RegistryContextFactory as the InitialContextFactory
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
//the registry usrl is "rmi://localhost:1234"
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
InitialContext ctx = new InitialContext(jndiProperties);
```

3. Сервер связывает имя и объект

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. Клиент ищет объект по названию "/ neohope / jndi / test01"

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Теперь клиент может использовать объект

6. Когда сервер заканчивается, его необходимо очистить.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

Прочитайте JNDI онлайн: https://riptutorial.com/ru/java/topic/5720/jndi

глава 27: JShell

Вступление

JShell - это интерактивный REPL для Java, добавленный в JDK 9. Он позволяет разработчикам мгновенно оценивать выражения, тестировать классы и экспериментировать с языком Java. Ранний доступ для jdk 9 можно получить по адресу: http://jdk.java.net/9/

Синтаксис

- $ jshell - Запустите JShell REPL
- jshell> / <command> - Запустить заданную команду JShell
- jshell> / exit - Выход JShell
- jshell> / help - список команд JShell
- jshell> <java_expression> - Оценить данное выражение Java (точка с запятой необязательна)
- jshell> / vars OR / methods OR / types - См. список переменных, методов или классов, соответственно.
- jshell> / open <файл> - читать файл в качестве ввода в оболочку
- jshell> / edit <идентификатор> - отредактируйте фрагмент в заданном редакторе
- jshell> / set editor <command> - установить команду, которая будет использоваться для редактирования фрагментов с использованием / edit
- jshell> / drop <идентификатор> - удалить фрагмент
- jshell> / reset - Сбросить JVM и удалить все фрагменты

замечания

JShell требует Java 9 JDK, который в настоящее время (март 2017) можно загрузить как ранние снимки доступа из jdk9.java.net . Если при попытке запустить команду `jshell` вы получите сообщение об ошибке, начинающееся с « `Unable to locate an executable` , убедитесь, что `JAVA_HOME` установлен правильно.

Импорт по умолчанию

Следующие пакеты автоматически импортируются при запуске JShell:

```
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
```

```
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*
```

Examples

Вход и выход JShell

Запуск JShell

Прежде чем пытаться запустить JShell, убедитесь, что переменная окружения `JAVA_HOME` указывает на установку JDK 9. Чтобы запустить JShell, выполните следующую команду:

```
$ jshell
```

Если все пойдет хорошо, вы увидите `jshell>` .

Выход из JShell

Чтобы выйти из JShell, запустите следующую команду из приглашения JShell:

```
jshell> /exit
```

Выражения

В JShell вы можете оценивать выражения Java с запятой или без нее. Они могут варьироваться от базовых выражений и операторов до более сложных:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Петли и условные обозначения тоже прекрасны:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

Важно отметить, что **выражения внутри блоков должны иметь точки с запятой!**

переменные

Вы можете объявлять локальные переменные в JShell:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Имейте в виду, что переменные могут быть переоформлены разными типами; это совершенно справедливо в JShell:

```
jshell> String var = "hi"
jshell> int var = 3
```

Чтобы просмотреть список переменных, введите `/vars` в приглашении JShell.

Методы и классы

Вы можете определить методы и классы в JShell:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

Не требуется модификаторов доступа. Как и в других блоках, точки с запятой требуются внутри тел метода. Имейте в виду, что, как и в случае с переменными, можно переопределить методы и классы. Чтобы просмотреть список методов или классов, введите `/methods` или `/types` в приглашении JShell, соответственно.

Редактирование фрагментов

Основной единицей кода, используемой JShell, является **фрагмент** или **исходная запись** . Каждый раз, когда вы объявляете локальную переменную или определяете локальный метод или класс, вы создаете фрагмент, чье имя является идентификатором переменной / метода / класса. В любой момент вы можете отредактировать фрагмент, который вы создали с помощью команды `/edit` . Например, допустим, я создал класс `Foo` с помощью одного метода, `bar` :

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Теперь я хочу заполнить тело моего метода. Вместо того, чтобы переписывать весь класс, я могу его отредактировать:

```
jshell> /edit Foo
```

По умолчанию редактор swing появится с наиболее доступными функциями. Однако вы можете изменить редактор, который использует JShell:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Обратите внимание, что если **новая версия фрагмента содержит любые синтаксические ошибки, она может быть не сохранена.** Аналогично, фрагмент создается только в том случае, если оригинальное объявление / определение является синтаксически правильным; следующее не работает:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

Тем не менее, фрагменты могут быть скомпилированы и, следовательно, доступны для редактирования, несмотря на некоторые ошибки времени компиляции, такие как несогласованные типы - следующие работы:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Наконец, фрагменты могут быть удалены с помощью команды `/drop` :

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  System.out.println(i)
|
```

Чтобы удалить все фрагменты, тем самым переиздав состояние JVM, используйте `\reset` :

```
jshell> int i = 2

jshell> String s = "hi"

jshell> /reset
|  Resetting state.

jshell> i
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  i
|  ^
```

```
jshell> s
|  Error:
|  cannot find symbol
|    symbol:   variable s
|  s
|  ^
```

Прочитайте JShell онлайн: https://riptutorial.com/ru/java/topic/9511/jshell

глава 28: JSON в Java

Вступление

JSON (JavaScript Object Notation) представляет собой легкий, текстовый, независимый от языка формат обмена данными, который легко людям и машинам читать и писать. JSON может представлять два структурированных типа: объекты и массивы. JSON часто используется в приложениях Ajax, конфигурациях, базах данных и веб-службах RESTful. Java API для JSON Processing предоставляет портативные API для анализа, генерации, преобразования и запроса JSON.

замечания

В этом примере основное внимание уделяется разбору и созданию JSON в Java с использованием различных библиотек, таких как библиотека **Google Gson** , Jackson Object Mapper и другие.

Примеры использования других библиотек можно найти здесь: Как разбирать JSON в Java

Examples

Кодирование данных как JSON

Если вам нужно создать `JSONObject` и поместить в него данные, рассмотрите следующий пример:

```
// Create a new javax.json.JSONObject instance.
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// Add a second object.
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// Create a new JSONArray with some values
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// Add another month as the fifth element, leaving the 4th element unset.
someMonths.put(4, "May");

// Add the array to our object
object.put("months", someMonths);

// Encode
```

```
String json = object.toString();

// An exercise for the reader: Add pretty-printing!
/* {
       "foo":"bar",
       "temperature":21.5,
       "year":2016,
       "message":{"Hello":"world"},
       "months":["January","February","March",null,"May"]
   }
*/
```

Декодирование данных JSON

Если вам нужно получить данные из `JSONObject` , рассмотрите следующий пример:

```
String json =
"{\"foo\":\"bar\",\"temperature\":21.5,\"year\":2016,\"message\":{\"Hello\":\"world\"},\"months\":[\"Ja


// Decode the JSON-encoded string
JSONObject object = new JSONObject(json);

// Retrieve some values
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// Retrieve another object
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// Retrieve an array
JSONArray someMonths = object.getJSONArray("months");
// Get some values from the array
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

optXXX vs getXXX

`JSONObject` и `JSONArray` есть несколько методов, которые очень полезны при работе с возможностью того, что значение, которое вы пытаетесь получить, не существует или имеет другой тип.

```
JSONObject obj = new JSONObject();
obj.putString("foo", "bar");

// For existing properties of the correct type, there is no difference
obj.getString("foo");        // returns "bar"
obj.optString("foo");        // returns "bar"
obj.optString("foo", "tux"); // returns "bar"

// However, if a value cannot be coerced to the required type, the behavior differs
obj.getInt("foo");      // throws JSONException
obj.optInt("foo");      // returns 0
obj.optInt("foo", 123); // returns 123
```

```
// Same if a property does not exist
obj.getString("undefined");        // throws JSONException
obj.optString("undefined");        // returns ""
obj.optString("undefined", "tux"); // returns "tux"
```

Те же правила применяются к `getXXX` / `optXXX` `JSONArray` .

Объект для JSON (библиотека Gson)

Допустим, у вас есть класс под названием `Person` с просто `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Код:

```
Gson g = new Gson();

Person person = new Person("John");
System.out.println(g.toJson(person)); // {"name":"John"}
```

Конечно, банда Gson должна быть на пути к классу .

JSON To Object (Библиотека Gson)

Допустим, у вас есть класс под названием `Person` с просто `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Код:

```
Gson gson = new Gson();
String json = "{\"name\": \"John\"}";

Person person = gson.fromJson(json, Person.class);
System.out.println(person.name); //John
```

У вас должна быть библиотека gson в вашем пути к классам.

Извлечение одного элемента из JSON

```
String json = "{\"name\": \"John\", \"age\":21}";

JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21
```

Использование Mapper объекта Jackson

Модель Pojo

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Пример: String to Object

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Пример: объект для строки

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

подробности

Необходима операция импорта:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Зависимость Maven: jackson-databind

Экземпляр `ObjectMapper`

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- `ObjectMapper` является поточно

- рекомендуется: иметь общий статический экземпляр

Десериализация:

```
<T> T readValue(String content, Class<T> valueType)
```

- `valueType` необходимо указать - возврат будет такого типа
- Броски
  - `IOException` - в случае проблемы ввода-вывода низкого уровня
  - `JsonParseException` - если базовый ввод содержит недопустимый контент
  - `JsonMappingException` - если входная структура JSON не соответствует структуре объекта

Пример использования (jsonString - входная строка):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Метод сериализации:

String writeValueAsString (значение объекта)

- Броски
  - `JsonProcessingException` в случае ошибки
  - Примечание: до версии 2.1 предложение бросков включено в IOException; 2.1 удалили его.

Итерация JSON

`JSONObject` над свойствами `JSONObject`

```
JSONObject obj = new JSONObject("{\"isMarried\":\"true\", \"name\":\"Nikita\",
\"age\":\"30\"}");
Iterator<String> keys = obj.keys();//all keys: isMarried, name & age
while (keys.hasNext()) {                        //as long as there is another key
      String key = keys.next();                 //get next key
      Object value = obj.get(key);              //get next value by key
      System.out.println(key + " : " + value);//print key : value
}
```

Итерировать значения `JSONArray`

```
JSONArray arr = new JSONArray();        //Initialize an empty array
//push (append) some values in:
arr.put("Stack");
arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) {//iterate over all values
```

```
    Object value = arr.get(i);          //get value
    System.out.println(value);          //print each value
}
```

JSON Builder - методы цепочки

Вы можете использовать цепочку методов при работе с `JSONObject` и `JSONArray` .

Пример JSONObject

```
JSONObject obj = new JSONObject();//Initialize an empty JSON object
//Before: {}
obj.put("name","Nikita").put("age","30").put("isMarried","true");
//After: {"name":"Nikita","age":30,"isMarried":true}
```

JSONArray

```
JSONArray arr = new JSONArray();//Initialize an empty array
//Before: []
arr.put("Stack").put("Over").put("Flow");
//After: ["Stack","Over","Flow"]
```

JSONObject.NULL

Если вам нужно добавить свойство с `null` значением, вы должны использовать предопределенный статический конечный `JSONObject.NULL` а не стандартную Java- `null` ссылку.

`JSONObject.NULL` - это контрольное значение, используемое для явного определения свойства с пустым значением.

```
JSONObject obj = new JSONObject();
obj.put("some", JSONObject.NULL);   //Creates: {"some":null}
System.out.println(obj.get("some"));//prints: null
```

Заметка

```
JSONObject.NULL.equals(null); //returns true
```

Что является **явным нарушением** контракта `Java.equals()` :

> Для любого ненулевого опорного значения x, x.equals (NULL) должен возвращать ложь

JsonArray в список Java (Gson Library)

Вот простой JsonArray, который вы хотели бы преобразовать в Java `ArrayList` :

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

Теперь передайте список `JsonArray` 'следующему методу, который возвращает соответствующий Java `ArrayList` :

```
public ArrayList<String> getListString(String jsonList){
    Type listType = new TypeToken<List<String>>() {}.getType();
    //make sure the name 'list' matches the name of 'JsonArray' in your 'Json'.
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);
    return list;
}
```

Вы должны добавить следующую зависимость maven к вашему файлу `POM.xml` :

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.7</version>
</dependency>
```

Или у вас должен быть jar `com.google.code.gson:gson:jar:<version>` в вашем пути к классам.

Уничтожьте коллекцию JSON в коллекции объектов, используя Jackson

Предположим , у вас есть класс POJO `Person`

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

И вы хотите разобрать его в массив JSON или карту объектов Person. Из-за стирания типа вы не можете создавать классы `List<Person>` и `Map<String, Person>` непосредственно во время выполнения *(и, следовательно, использовать их для десериализации JSON)* . Чтобы преодолеть это ограничение, Джексон предлагает два подхода - `TypeFactory` и `TypeReference` .

TypeFactory

Подход, использованный здесь, заключается в использовании фабрики (и ее статической функции полезности) для создания вашего типа для вас. Требуемые параметры - это коллекция, которую вы хотите использовать (список, набор и т. Д.) И класс, который вы

хотите сохранить в этой коллекции.

TypeReference

Тип ссылочного подхода кажется более простым, поскольку он экономит вам немного ввода и выглядит более чистым. TypeReference принимает параметр типа, в котором вы передаете желаемый тип `List<Person>` . Вы просто создаете экземпляр объекта TypeReference и используете его в качестве контейнера типа.

Теперь давайте посмотрим, как фактически десериализовать JSON в объект Java. Если ваш JSON отформатирован как массив, вы можете десериализовать его как список. Если существует более сложная вложенная структура, вам необходимо десериализовать карту. Мы рассмотрим примеры обоих.

Удаление десериализации массива JSON

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]"
```

Подход TypeFactory

```
CollectionType listType =
    factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

Подход типа Type

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};
List<Person> list = mapper.readValue(jsonString, listType);
```

Десериализация карты JSON

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

Подход TypeFactory

```
CollectionType mapType =
    factory.constructMapLikeType(Map.class, String.class, Person.class);
List<Person> list = mapper.readValue(jsonString, mapType);
```

Подход типа Type

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

подробности

Используемая инструкция импорта:

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;
```

Используемые экземпляры:

```
ObjectMapper mapper = new ObjectMapper();
TypeFactory factory = mapper.getTypeFactory();
```

Заметка

Хотя подход `TypeReference` может выглядеть лучше, он имеет несколько недостатков:

1. `TypeReference` должен быть `TypeReference` с использованием анонимного класса
2. Вы должны предоставить общую экспликацию

Несоблюдение этого может привести к потере аргумента общего типа, что приведет к неудаче десериализации.

Прочитайте JSON в Java онлайн: https://riptutorial.com/ru/java/topic/840/json-в-java

глава 29: LinkedHashMap

Вступление

Класс LinkedHashMap - это таблица Hash и реализация Linked list интерфейса Map с предсказуемым порядком итерации. Он наследует класс HashMap и реализует интерфейс карты.

Важными моментами для класса Java LinkedHashMap являются: LinkedHashMap содержит значения, основанные на ключе. Он содержит только уникальные элементы. Он может иметь один нулевой ключ и несколько нулевых значений. Это то же самое, что и HashMap, поддерживает порядок вставки.

Examples

Класс Java LinkedHashMap

Ключевые моменты: -

- Является ли таблица Hash и связанный список интерфейсом Map с предсказуемым порядком итерации.
- наследует класс HashMap и реализует интерфейс карты.
- содержит значения на основе ключа.
- только уникальные элементы.
- может иметь один нулевой ключ и несколько нулевых значений.
- то же, что и HashMap, поддерживает порядок вставки.

Методы: -

- void clear ().
- boolean containsKey (ключ объекта).
- Объект get (ключ объекта).
- protected boolean removeEldestEntry (Map.Entry старший)

Пример :-

```
public static void main(String arg[])
    {
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
        lhm.put("Ramesh", "Intermediate");
```

```
        lhm.put("Shiva", "B-Tech");
        lhm.put("Santosh", "B-Com");
        lhm.put("Asha", "Msc");
        lhm.put("Raghu", "M-Tech");

        Set set = lhm.entrySet();
        Iterator i = set.iterator();
        while (i.hasNext()) {
            Map.Entry me = (Map.Entry) i.next();
            System.out.println(me.getKey() + " : " + me.getValue());
        }

        System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));
        System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));
    }
```

Прочитайте LinkedHashMap онлайн: https://riptutorial.com/ru/java/topic/10750/linkedhashmap

глава 30: log4j / log4j2

Вступление

Apache Log4j - это утилита ведения журнала на основе Java, это одна из нескольких фреймворков регистрации Java. В этом разделе показано, как настроить и настроить Log4j на Java с подробными примерами по всем возможным аспектам использования.

Синтаксис

- Logger.debug («текст для журнала»); // Регистрация информации об отладке
- Logger.info («текст для регистрации»); // Регистрация общей информации
- Logger.error («текст для регистрации»); // Запись информации об ошибке
- Logger.warn («текст для регистрации»); // Предупреждения о регистрации
- Logger.trace («текст для регистрации»); // Запись информации о трассировке
- Logger.fatal («текст для регистрации»); // Регистрация фатальных ошибок
- Использование Log4j2 с протоколированием параметров:
- Logger.debug («Debug params {} {} {}", param1, param2, param3); // Регистрация отладки с параметрами
- Logger.info («Info params {} {} {}", param1, param2, param3); // Запись информации с параметрами
- Logger.error («Параметры ошибки {} {} {}", param1, param2, param3); // Ошибка регистрации с параметрами
- Logger.warn ("Warn params {} {} {}", param1, param2, param3); // Запись предупреждений с параметрами
- Logger.trace («Параметры трассировки {} {} {}", param1, param2, param3); // Ведение журнала с параметрами
- Logger.fatal («Фатальные параметры {} {} {}", param1, param2, param3); // Регистрация фатальных данных с параметрами
- Logger.error («Caught Exception:», ex); // Исключение журнала с сообщением и stacktrace (будет автоматически добавлено)

замечания

Конец жизни для Log4j 1 достигнут

5 августа 2015 года Комитет по управлению проектами в области лесозаготовок объявил, что Log4j 1.x достигло конца жизни. Полный текст объявления можно найти в блоге Apache. **Пользователям Log4j 1 рекомендуется обновить до Apache Log4j 2** .

От: http://logging.apache.org/log4j/1.2/

Examples

Как получить Log4j

Текущая версия (log4j2)

Использование Maven:
Добавьте в свой файл `POM.xml` следующую зависимость:

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Использование Айви:

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

Использование Gradle:

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
  compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

Получение log4j 1.x

Примечание: Log4j 1.x достигло конца жизни (EOL) (см. Примечания).

Использование Maven:

Объявите эту зависимость в файле `POM.xml` :

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Использование Айви:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usign Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Использование Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Добавление вручную в сборку пути:

Загрузить с веб-сайта проекта Log4j

Как использовать Log4j в Java-коде

Сначала необходимо создать `final static logger` объект `final static logger` :

```
final static Logger logger = Logger.getLogger(classname.class);
```

Затем вызовите методы ведения журнала:

```
//logs an error message
logger.info("Information about some param: " + parameter); // Note that this line could throw
a NullPointerException!

//in order to improve performance, it is advised to use the `isXXXEnabled()` Methods
if( logger.isInfoEnabled() ){
    logger.info("Information about some param: " + parameter);
}

// In log4j2 parameter substitution is preferable due to readability and performance
// The parameter substitution only takes place if info level is active which obsoletes the use
of isXXXEnabled().
logger.info("Information about some param: {}" , parameter);

//logs an exception
logger.error("Information about some error: ", exception);
```

Настройка файла свойств

Log4j дает вам возможность записывать данные в консоль и файл одновременно.

Создайте файл `log4j.properties` и вставьте эту базовую конфигурацию:

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\log4j-application.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Если вы используете maven, поместите этот файл свойства в путь:

```
/ProjectFolder/src/java/resources
```

Базовый файл конфигурации log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

Это базовая конфигурация log4j2.xml с консольным приложением и корневым регистратором. В макете шаблона указывается, какой шаблон следует использовать для регистрации операторов.
Чтобы отладить загрузку log4j2.xml, вы можете добавить `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` атрибута `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` в теге конфигурации вашего log4j2.xml.
Вы также можете добавить интервал монитора, чтобы он снова загрузил конфигурацию после указанного интервала времени. Интервал мониторинга может быть добавлен в тег конфигурации следующим образом: `monitorInterval = 30` . Это означает, что конфигурация будет загружаться каждые 30 секунд.

Миграция с log4j 1.x на 2.x

Если вы хотите перенести из существующего log4j 1.x в свой проект на log4j 2.x, удалите все существующие зависимости log4j 1.x и добавьте следующую зависимость:

Log4j 1.x API Bridge

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}
```

Мост ведения **журнала Apache Commons** Если ваш проект использует журнал регистрации Apache, который использует log4j 1.x, и вы хотите перенести его на log4j 2.x, то добавьте следующие зависимости:

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
```

```
}
```

Примечание. Не удаляйте никаких существующих зависимостей записи в архивах Apache

Ссылка: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Свойства-Файл для входа в БД

Для этого примера вам понадобится драйвер JDBC, совместимый с системой, в которой работает база данных. С открытым исходным кодом, который позволяет подключаться к базам данных DB2 в системе IBM, можно найти здесь: JT400

Несмотря на то, что этот пример специфичен для DB2, он работает практически для любой другой системы, если вы меняете драйвер и адаптируете URL JDBC.

```
# Root logger option
log4j.rootLogger= ERROR, DB

# Redirect log messages to a DB2
# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL (!!! adapt to your target system !!!)
log4j.appender.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# Set Database Driver (!!! adapt to your target system !!!)
log4j.appender.DB.driver=com.ibm.as400.access.AS400JDBCDriver

# Set database user name and password
log4j.appender.DB.user=USER
log4j.appender.DB.password=PASSWORD

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO DB.TABLENAME VALUES('%d{yyyy-MM-
dd}','%d{HH:mm:ss}','%C','%p','%m')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
```

Выйти из фильтра по уровню (log4j 1.x)

Вы можете использовать фильтр для регистрации только сообщений «ниже», чем, например, уровень `ERROR` . **Но фильтр не поддерживается PropertyConfigurator. Поэтому вы должны перейти на конфигурацию XML, чтобы использовать его** . См. Log4j-Wiki об фильтрах .

Пример "конкретный уровень"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
                    <param name="ConversionPattern" value="%m%n"/>
```

```
            </layout>
            <filter class="org.apache.log4j.varia.LevelMatchFilter">
                    <param name="LevelToMatch" value="info" />
                    <param name="AcceptOnMatch" value="true"/>
            </filter>
            <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

Или "Диапазон уровней"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
                    <param name="ConversionPattern" value="%m%n"/>
            </layout>
            <filter class="org.apache.log4j.varia.LevelRangeFilter">
                    <param name="LevelMax" value="info"/>
                    <param name="LevelMin" value="info"/>
                    <param name="AcceptOnMatch" value="true"/>
            </filter>
</appender>
```

Прочитайте log4j / log4j2 онлайн: https://riptutorial.com/ru/java/topic/2472/log4j---log4j2

глава 31: NIO - Сеть

замечания

`SelectionKey` определяет различные выбираемые операции и информацию между его Селектором и Каналом . В частности, вложение может использоваться для хранения связанной с подключением информации.

Обработка `OP_READ` довольно прямолинейна. Однако следует соблюдать осторожность при работе с `OP_WRITE` : большую часть времени данные могут быть записаны в сокеты, чтобы событие продолжало стрелять. Обязательно зарегистрируйте `OP_WRITE` только до того, как вы захотите записать данные (см. `OP_WRITE` ответ).

Кроме того , `OP_CONNECT` должен быть отменен после того , как канал был подключен (потому что, ну, *это* связано. См это и что ответы на SO). Следовательно, удаление `OP_CONNECT` после `finishConnect()` преуспело.

Examples

Использование Selector для ожидания событий (пример с OP_CONNECT)

NIO появился на Java 1.4 и представил концепцию «Каналы», которые должны быть быстрее, чем обычные входы / выходы. По сетевому интерфейсу `SelectableChannel` является самым интересным, поскольку он позволяет контролировать различные состояния канала. Он работает так же, как и системный вызов C `select()` : мы пробуждаемся, когда происходят определенные типы событий:

- полученное соединение (`OP_ACCEPT`)
- реализовано соединение (`OP_CONNECT`)
- данные доступны в считываемом FIFO (`OP_READ`)
- данные могут быть `OP_WRITE` для записи FIFO (`OP_WRITE`)

Он позволяет разделить между *обнаружением* ввода-вывода сокетов (что-то можно прочитать / записать / ...) и *выполнить* ввод-вывод (чтение / запись / ...). В частности, все обнаружение ввода-вывода может выполняться в одном потоке для нескольких сокетов (клиентов), в то время как операции ввода-вывода могут обрабатываться в пуле потоков или в другом месте. Это позволяет легко масштабировать приложение до количества подключенных клиентов.

В следующем примере показаны основы:

1. Создать `Selector`
2. Создание `SocketChannel`

3. Зарегистрируйте `SocketChannel` для `Selector`
4. Цикл с `Selector` для обнаружения событий

```
Selector sel = Selector.open(); // Create the Selector
SocketChannel sc = SocketChannel.open(); // Create a SocketChannel
sc.configureBlocking(false); // ... non blocking
sc.setOption(StandardSocketOptions.SO_KEEPALIVE, true); // ... set some options

// Register the Channel to the Selector for wake-up on CONNECT event and use some description
as an attachement
sc.register(sel, SelectionKey.OP_CONNECT, "Connection to google.com"); // Returns a
SelectionKey: the association between the SocketChannel and the Selector
System.out.println("Initiating connection");
if (sc.connect(new InetSocketAddress("www.google.com", 80)))
    System.out.println("Connected"); // Connected right-away: nothing else to do
else {
    boolean exit = false;
    while (!exit) {
        if (sel.select(100) == 0) // Did something happen on some registered Channels during
the last 100ms?
            continue; // No, wait some more

        // Something happened...
        Set<SelectionKey> keys = sel.selectedKeys(); // List of SelectionKeys on which some
registered operation was triggered
        for (SelectionKey k : keys) {
            System.out.println("Checking "+k.attachment());
            if (k.isConnectable()) { // CONNECT event
                System.out.print("Connected through select() on "+k.channel()+" -> ");
                if (sc.finishConnect()) { // Finish connection process
                    System.out.println("done!");
                    k.interestOps(k.interestOps() & ~SelectionKey.OP_CONNECT); // We are
already connected: remove interest in CONNECT event
                    exit = true;
                } else
                    System.out.println("unfinished...");
            }
            // TODO: else if (k.isReadable()) { ...
        }
        keys.clear(); // Have to clear the selected keys set once processed!
    }
}
System.out.print("Disconnecting ... ");
sc.shutdownOutput(); // Initiate graceful disconnection
// TODO: emtpy receive buffer
sc.close();
System.out.println("done");
```

Дала бы следующий результат:

```
Initiating connection
Checking Connection to google.com
Connected through 'select()' on java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> done!
Disconnecting ... done
```

Прочитайте NIO - Сеть онлайн: https://riptutorial.com/ru/java/topic/5513/nio---сеть

глава 32: NumberFormat

Examples

NumberFormat

В разных странах существуют разные форматы чисел, и, учитывая это, мы можем использовать разные форматы, используя язык Java. Использование языка может помочь в форматировании

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

используя вышеуказанный формат, вы можете выполнять различные задачи

1. Номер формата

   ```
   numberFormat.format(10000000.99);
   ```

2. Формат валюты

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Формат Процент

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Контрольное число цифр

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Прочитайте NumberFormat онлайн: https://riptutorial.com/ru/java/topic/7399/numberformat

глава 33: ServiceLoader

замечания

`ServiceLoader` может использоваться для получения экземпляров классов, расширяющих данный тип (= служба), которые указаны в файле, упакованном в файл `.jar` . Сервис, который расширен / реализован, часто является интерфейсом, но это не требуется.

Расширяющиеся / реализующие классы должны предоставить конструктор нулевого аргумента для `ServiceLoader` для их создания.

Чтобы быть обнаруженным `ServiceLoader` текстовый файл с именем полного имени типа внедренной службы должен храниться внутри каталога `META-INF/services` в файле jar. Этот файл содержит одно полное имя класса, реализующего службу на строку.

Examples

Служба регистрации

В следующем примере показано, как создать экземпляр класса для ведения журнала через `ServiceLoader` .

обслуживание

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Реализация услуги

Следующая реализация просто записывает сообщение в `System.err`

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
```

```
        System.err.println(message);
    }

    @Override
    public void close() {
    }

}
```

Следующая реализация записывает сообщения в текстовый файл:

```
package servicetest.logger;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import servicetest.Logger;

public class FileLogger implements Logger {

    private final BufferedWriter writer;

    public FileLogger() throws IOException {
        writer = new BufferedWriter(new FileWriter("log.txt"));
    }

    @Override
    public void log(String message) throws IOException {
        writer.append(message);
        writer.newLine();
    }

    @Override
    public void close() throws IOException {
        writer.close();
    }

}
```

META-INF / услуги / servicetest.Logger

В файле `META-INF/services/servicetest.Logger` перечислены имена реализации `Logger` .

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

использование

Следующий `main` метод записывает сообщение всем доступным регистраторам. Регистраторы создаются с использованием `ServiceLoader` .

```
public static void main(String[] args) throws Exception {
    final String message = "Hello World!";

    // get ServiceLoader for Logger
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);

    // iterate through instances of available loggers, writing the message to each one
    Iterator<Logger> iterator = loader.iterator();
    while (iterator.hasNext()) {
        try (Logger logger = iterator.next()) {
            logger.log(message);
        }
    }
}
```

Простой пример ServiceLoader

ServiceLoader - простой и простой в использовании встроенный механизм динамической загрузки реализаций интерфейса. С помощью сервис-загрузчика - обеспечения средств для создания (но не для проводки) - в Java SE может быть встроен простой механизм впрыска зависимостей. Интерфейс ServiceLoader и разделение реализации становятся естественными, и программы могут быть удобно расширены. На самом деле многие Java API внедрены на основе ServiceLoader

Основные понятия

- Работа на *интерфейсах* услуг
- Получение реализации (-ов) службы через `ServiceLoader`
- Обеспечение внедрения сервисов

Давайте начнем с интерфейса и поместим его в банку, названную, например, `accounting-api.jar`

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Теперь мы предоставляем реализацию этой службы в банке с именем `accounting-impl.jar` , содержащей реализацию услуги

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
```

```
    ...
  }
}
```

далее, `accounting-impl.jar` содержит файл, объявляющий, что эта банка обеспечивает реализацию `AccountingService` . Файл должен иметь путь, начинающийся с `META-INF/services/` и должен иметь то же имя, что и *полное* имя интерфейса:

- `META-INF/services/example.AccountingService`

Содержимое файла является *полностью qualfified* имя реализации:

```
example.impl.DefaultAccountingService
```

Поскольку обе банки находятся в пути к классам программы, которая потребляет `AccountingService` , экземпляр службы может быть получен с помощью ServiceLauncher

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

Поскольку `ServiceLoader` является `Iterable` , он поддерживает несколько поставщиков реализации, в которых программа может выбирать:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
}
```

Обратите внимание, что при вызове `next()` создан новый экземпляр. Если вы хотите повторно использовать экземпляр, вы должны использовать метод `iterator()` для ServiceLoader или для каждого цикла, как показано выше.

Прочитайте ServiceLoader онлайн: https://riptutorial.com/ru/java/topic/5433/serviceloader

глава 34: SortedMap

Вступление

Введение в отсортированную карту.

Examples

Введение в отсортированную карту.

Ключевой момент :-

- Интерфейс SortedMap расширяет карту.
- записи сохраняются в порядке возрастания ключа.

Методы сортированной карты:

- Компаратор компаратора ().
- Объект firstKey ().
- SortedMap headMap (конец объекта).
- Объект lastKey ().
- Подмножество SortedMap (начало объекта, конец объекта).
- SortedMap tailMap (Начало объекта).

пример

```
public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
```

```
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
```

Прочитайте SortedMap онлайн: https://riptutorial.com/ru/java/topic/10748/sortedmap

глава 35: Streams

Вступление

`Stream` представляет последовательность элементов и поддерживает различные виды операций для выполнения вычислений по этим элементам. С Java 8 интерфейс `Collection` имеет два метода для генерации `Stream` : `stream()` и `parallelStream()` . Операции `Stream` являются промежуточными или конечными. Промежуточные операции возвращают `Stream` поэтому несколько промежуточных операций могут быть привязаны до того, как `Stream` будет закрыт. Операции с терминалом либо недействительны, либо возвращают результат без потока.

Синтаксис

- collection.stream ()
- Arrays.stream (массив)
- Stream.iterate (firstValue, currentValue -> nextValue)
- Stream.generate (() -> value)
- Stream.of (elementOfT [, elementOfT, ...])
- Stream.empty ()
- StreamSupport.stream (iterable.spliterator (), false)

Examples

Использование потоков

`Stream` представляет собой последовательность элементов, на которых могут выполняться последовательные и параллельные операции агрегации. Любой данный `Stream` может потенциально иметь неограниченное количество данных, проходящих через него. В результате данные, полученные от `Stream` , обрабатываются индивидуально по мере их поступления, в отличие от пакетной обработки данных в целом. В сочетании с лямбда-выражениями они обеспечивают краткий способ выполнения операций над последовательностями данных с использованием функционального подхода.

Пример: (см. Работу над Ideone)

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .forEach(System.out::println);
```

Выход:

```
ЯБЛОКО
БАНАН
ОРАНЖЕВЫЙ
ГРУША
```

Операции, выполняемые вышеуказанным кодом, можно суммировать следующим образом:

1. Создайте `Stream<String>` содержащий упорядоченный упорядоченный `Stream` элементов `String` для фруктов, используя статический заводский метод `Stream.of(values)` .

2. Операция `filter()` сохраняет только элементы, которые соответствуют заданному предикату (элементы, которые при проверке предикатом возвращают true). В этом случае он сохраняет элементы, содержащие `"a"` . Предикат задается как лямбда-выражение .

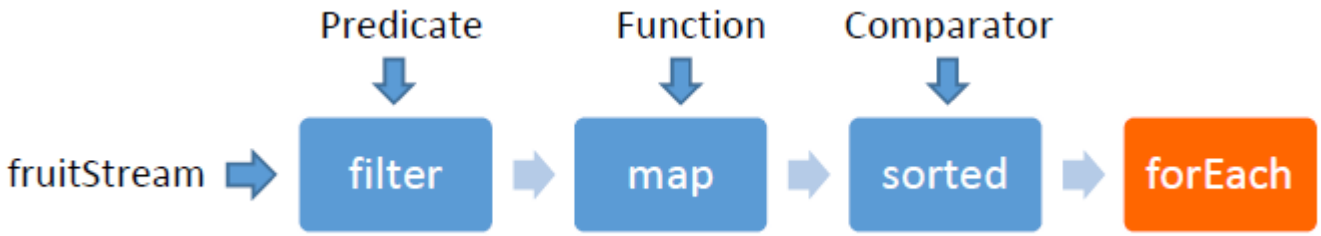
3. Операция `map()` преобразует каждый элемент с помощью заданной функции, называемой mapper. В этом случае каждая `String` фруктов преобразуется в его строчную версию `String` с использованием ссылки на метод `String::toUppercase` .

   > **Обратите внимание,** что операция `map()` вернет поток с другим общим типом, если функция сопоставления возвращает тип, отличный от его входного параметра. Например, в `Stream<String>` вызов `.map(String::isEmpty)` возвращает `Stream<Boolean>`

4. Операция `sorted()` сортирует элементы `Stream` соответствии с их естественным упорядочением (лексикографически, в случае `String`).

5. Наконец, операция `forEach(action)` выполняет действие, действующее на каждый элемент `Stream` , передавая его потребителю . В этом примере каждый элемент просто печатается на консоли. Эта операция является терминальной операцией, что делает невозможным ее повторное использование.

   > **Обратите внимание,** что операции, определенные в `Stream` , выполняются *из-* за операции терминала. Без операции с терминалом поток не обрабатывается. Потоки нельзя использовать повторно. После вызова терминальной операции объект `Stream` становится непригодным.



Операции (как показано выше) соединены вместе, чтобы сформировать то, что можно рассматривать как запрос данных.

Закрытие потоков

> **Обратите внимание, что `Stream` как правило, не нужно закрывать.** Требуется только закрыть потоки, которые работают на каналах ввода-вывода. Большинство типов `Stream` не работают на ресурсах и поэтому не требуют закрытия.

Интерфейс `Stream` расширяет `AutoCloseable` . Потоки могут быть закрыты вызовом метода `close` или с помощью операторов try-with-resource.

Пример использования, когда `Stream` должен быть закрыт, - это когда вы создаете `Stream` строк из файла:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

Интерфейс `Stream` также объявляет метод `Stream.onClose()` который позволяет вам регистрировать обработчики `Runnable` которые будут вызываться, когда поток закрыт. Пример использования - это то, где код, который создает поток, должен знать, когда он потребляется, чтобы выполнить некоторую очистку.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

Обработчик запуска будет выполняться только в том случае, если метод `close()` вызывается, явно или неявно, с помощью инструкции try-with-resources.

Порядок обработки

Обработка объекта `Stream` может быть последовательной или параллельной .

В **последовательном** режиме элементы обрабатываются в порядке источника `Stream` . Если `Stream` упорядочен (например, реализация `SortedMap` или `List`), то гарантируется, что обработка будет соответствовать порядку источника. В других случаях, однако, следует соблюдать осторожность, чтобы не зависеть от порядка (см. `keySet()` порядок итераций Java `HashMap keySet()` ?).

Пример:

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// sequential
long howManyOddNumbers = integerList.stream()
                                    .filter(e -> (e % 2) == 1)
                                    .count();

System.out.println(howManyOddNumbers); // Output: 2
```

Живой на Ideone

Параллельный режим позволяет использовать несколько потоков на нескольких ядрах, но нет гарантии того, в каком порядке обрабатываются элементы.

Если вызывается несколько методов в последовательном `Stream` , не каждый метод должен быть вызван. Например, если `Stream` фильтруется и количество элементов сводится к единице, последующий вызов метода, такого как `sort` , не будет выполняться. Это может увеличить производительность последовательного `Stream` - оптимизация, которая невозможна при параллельном `Stream` .

Пример:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Живой на Ideone

Отличия от контейнеров (или коллекций)

Хотя некоторые действия могут выполняться как в контейнерах, так и в потоках, они в конечном итоге служат различным целям и поддерживают разные операции. Контейнеры больше сосредоточены на том, как хранятся элементы и как эти элементы могут быть доступны эффективно. С другой стороны, `Stream` не обеспечивает прямого доступа и манипулирования его элементами; он больше предназначен для группы объектов как коллективного объекта и выполняет операции над этим объектом в целом. `Stream` и `Collection` - это отдельные абстракции высокого уровня для этих различных целей.

Соберите элементы потока в коллекцию

Собирайте с помощью `toList()` и `toSet()`

Элементы из `Stream` можно легко собрать в контейнер с помощью операции `Stream.collect` :

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

Другие экземпляры коллекции, такие как `Set` , могут быть сделаны с использованием других встроенных методов `Collectors` . Например, `Collectors.toSet()` собирает элементы `Stream` в `Set` .

Явный контроль над реализацией `List` или `Set`

Согласно документации `Collectors#toList()` и `Collectors#toSet()` , нет никаких гарантий по типу, изменчивости, сериализуемости или потокобезопасности возвращаемого `List` или `Set` .

Для явного контроля над возвращаемой реализацией вместо этого можно использовать `Collectors#toCollection(Supplier)` , где данный поставщик возвращает новую и пустую коллекцию.

```
// syntax with method reference
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(ArrayList::new))
);

// syntax with lambda
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(() -> new LinkedHashSet<>()))
);
```

Сбор элементов с помощью toMap

Коллекционер накапливает элементы на карте, где ключ - это идентификатор студента, а значение - значение ученика.

```
  List<Student> students = new ArrayList<Student>();
    students.add(new Student(1,"test1"));
    students.add(new Student(2,"test2"));
    students.add(new Student(3,"test3"));

    Map<Integer, String> IdToName = students.stream()
        .collect(Collectors.toMap(Student::getId, Student::getName));
    System.out.println(IdToName);
```

Выход :

```
{1=test1, 2=test2, 3=test3}
```

У Collectors.toMap есть другая реализация `Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)` . Функция mergeFunction используется в основном для выбора нового значения или сохранения старого значения, если ключ повторяется при добавлении нового члена в Map из списка.

Функция mergeFunction часто выглядит так: `(s1, s2) -> s1` чтобы сохранить значение, соответствующее повторенному ключу, или `(s1, s2) -> s2` чтобы добавить новое значение для повторного ключа.

Сбор элементов на карте коллекций

Пример: от ArrayList до Map <String, List <>>

Часто для этого требуется сделать карту списка из основного списка. Пример: от ученика списка, нам нужно составить карту списка предметов для каждого ученика.

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 12.9));
    list.add(new Student("Davis", SUBJECT.GEOGRAPHY, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 80.0));
    list.add(new Student("Sascha", SUBJECT.SCIENCE, 12.0));
    list.add(new Student("Sascha", SUBJECT.LITERATURE, 50.0));

    list.add(new Student("Robert", SUBJECT.LITERATURE, 12.0));

    Map<String, List<SUBJECT>> map = new HashMap<>();
    list.stream().forEach(s -> {
                map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
            });
    System.out.println(map);
```

Выход:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Пример: от ArrayList до Map <String, Map <>>

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
    list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
```

```
    list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

    list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

    Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

    list.stream().forEach(student -> {
        map.computeIfAbsent(student.getName(), s -> new HashMap<>())
                .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
                .add(student.getMarks());
    });

    System.out.println(map);
```

Выход:

```
{ Robert={ENGLISH=[12.0]},
Sascha={MATH=[80.0, 50.0], ENGLISH=[85.0, 12.0]},
Davis={MATH=[35.0, 37.0], SCIENCE=[12.9, 37.0]} }
```

Чит-лист

Цель	Код
Сбор в `List`	`Collectors.toList()`
Собирать в `ArrayList` с заранее заданным размером	`Collectors.toCollection(() -> new ArrayList<>(size))`
Собрать `Set`	`Collectors.toSet()`
Собрать в `Set` с лучшей итерационной производительностью	`Collectors.toCollection(() -> new LinkedHashSet<>())`
Собрать в нечувствительный к регистру `Set<String>`	`Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))`
Соберите в `EnumSet<AnEnum>` (наилучшая производительность для перечислений)	`Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))`
Соберите `Map<K,V>` с уникальными ключами	`Collectors.toMap(keyFunc,valFunc)`
Карта MyObject.getter () для уникального объекта MyObject	`Collectors.toMap(MyObject::getter, Function.identity())`
Карта MyObject.getter () для	`Collectors.groupingBy(MyObject::getter)`

Цель	Код
нескольких объектов MyObjects	

Бесконечные потоки

Можно создать `Stream` , который не заканчивается. Вызов метода терминала в бесконечном `Stream` приводит к тому, что `Stream` вводит бесконечный цикл. Метод `limit` `Stream` может использоваться для ограничения количества терминов `Stream` которые обрабатывает Java.

В этом примере генерируется `Stream` всех натуральных чисел, начиная с номера 1. Каждый последующий член `Stream` является одним выше предыдущего. Вызывая метод ограничения этого `Stream` , рассматриваются и печатаются только первые пять членов `Stream` .

```
// Generate infinite stream - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// Print out only the first 5 terms
naturalNumbers.limit(5).forEach(System.out::println);
```

Выход:

1
2
3
4
5

Другой способ генерации бесконечного потока - использовать метод Stream.generate . Этот метод принимает лямбда типа Поставщик .

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

Потребительские потоки

`Stream` будет пройден только тогда, когда есть *операция терминала* , например `count()` , `collect()` или `forEach()` . В противном случае операция `Stream` не будет выполнена.

В следующем примере терминальная операция не добавляется в `Stream` , поэтому операция `filter()` не будет вызываться и не будет выводиться вывод, потому что `peek()` НЕ является *терминальной операцией* .

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Живой на Ideone

Это последовательность `Stream` с действительной *работой терминала* , поэтому производится выход.

Вы также можете использовать `forEach` вместо `peek` :

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Живой на Ideone

Выход:

2
4
6
8

После выполнения операции терминала `Stream` потребляется и не может быть повторно использован.

Хотя данный объект потока нельзя использовать повторно, легко создать многоразовый `Iterable` который делегирует потоковый конвейер. Это может быть полезно для возвращения измененного представления живого набора данных без необходимости собирать результаты во временную структуру.

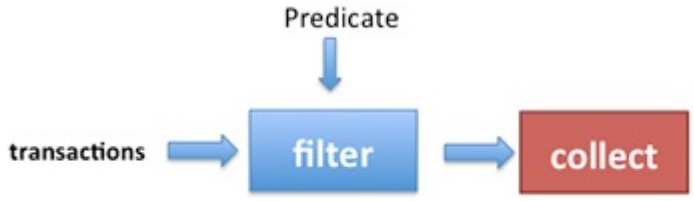
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Выход:

Foo
бар
Foo
бар

Это работает, потому что `Iterable` объявляет один абстрактный метод `Iterator<T> iterator()` . Это делает его эффективным функциональным интерфейсом, реализованным лямбдой, которая создает новый поток для каждого вызова.

В общем, `Stream` работает, как показано на следующем изображении:



ПРИМЕЧАНИЕ . Проверки аргументов всегда выполняются даже без *операции терминала* :

```
try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to
filter()");
}
```

Живой на Ideone

Выход:

> Мы получили исключение NullPointerException, поскольку значение null было передано как аргумент filter ()

Создание карты частоты

Сборщик `groupingBy(classifier, downstream)` позволяет собирать элементы `Stream` в `Map` , классифицируя каждый элемент в группе и выполняя нисходящую операцию над элементами, классифицированными в той же группе.

Классическим примером этого принципа является использование `Map` для подсчета появления элементов в `Stream` . В этом примере классификатор - это просто функция идентификации, которая возвращает элемент as-is. Операция downstream подсчитывает количество равных элементов, используя `counting()` .

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

Операция нисходящего потока сама является сборщиком (`Collectors.counting()`), который работает с элементами типа String и производит результат типа `Long` . Результатом вызова метода `collect` является `Map<String, Long>` .

Это приведет к следующему результату:

> банан = 1

оранжевый = 1
яблоко = 2

Параллельный поток

Примечание. Перед тем, как решить, какой `Stream` использовать, пожалуйста, взгляните на поведение ParallelStream vs Sequential Stream .

Если вы хотите одновременно выполнять операции `Stream` , вы можете использовать любой из этих способов.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

Или же:

```
Stream<String> aParallelStream = data.parallelStream();
```

Чтобы выполнить операции, определенные для параллельного потока, вызовите оператор терминала:

```
aParallelStream.forEach(System.out::println);
```

(Возможен) выход из параллельного `Stream` :

Три
четыре
Один
Два
5

Порядок может измениться, поскольку все элементы обрабатываются параллельно (что *может* ускорить выполнение). При заказе используйте параметр `parallelStream` .

Эффективное воздействие

В случае задействования сети параллельный `Stream` s может ухудшить общую производительность приложения, поскольку все параллельные `Stream` используют общий пул потоков fork-join для сети.

С другой стороны, параллельный `Stream` s может значительно повысить производительность во многих других случаях, в зависимости от количества доступных ядер в текущем CPU на данный момент.

Преобразование потока необязательно в поток значений

Возможно, вам потребуется конвертировать `Stream` излучающий `Optional` для `Stream` значений, испускающий только значения из существующего `Optional` . (т. е. без `null` значения и не имея дело с `Optional.empty()`).

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
                            .filter(Optional::isPresent)
                            .map(Optional::get)
                            .collect(Collectors.toList());

System.out.println(result); //[Hello World]
```

Создание потока

Все java `Collection<E>` имеют методы `stream()` и `parallelStream()` из которых можно построить `Stream<E>` :

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

`Stream<E>` может быть создан из массива с использованием одного из двух методов:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

Разница между `Arrays.stream()` и `Stream.of()` заключается в том, что `Stream.of()` имеет параметр varargs, поэтому его можно использовать как:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

Есть также примитивные `Stream` которые вы можете использовать. Например:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Эти примитивные потоки также могут быть построены с использованием `Arrays.stream()` :

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

Можно создать `Stream` из массива с указанным диапазоном.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Обратите внимание, что любой примитивный поток может быть преобразован в поток с

коротким типом, используя метод `boxed` :

```
Stream<Integer> integerStream = intStream.boxed();
```

Это может быть полезно в некоторых случаях, если вы хотите собирать данные, поскольку примитивный поток не имеет никакого метода `collect` который принимает `Collector` качестве аргумента.

Повторное использование промежуточных операций цепочки потоков

Поток закрывается, когда вызывается операция терминала. Повторное использование потока промежуточных операций, когда только операция терминала изменяется только. мы могли бы создать поставщика потока для создания нового потока со всеми уже созданными промежуточными операциями.

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana","orange",
"grapes", "melon","blueberry","blackberry")
.map(String::toUpperCase).sorted();

  streamSupplier.get().filter(s ->   s.startsWith("A")).forEach(System.out::println);

// APPLE

  streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);

  // BANANA
  // BLACKBERRY
  // BLUEBERRY
```

`int[]` массивы могут быть преобразованы в `List<Integer>` с использованием потоков

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Поиск статистики о числовых потоках

Java 8 предоставляет классы, называемые `IntSummaryStatistics` , `DoubleSummaryStatistics` и `LongSummaryStatistics` которые предоставляют объект состояния для сбора статистики, такой как `count` , `min` , `max` , `sum` и `average` .

Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
IntSummaryStatistics stats = naturalNumbers.stream()
                                           .mapToInt((x) -> x)
                                           .summaryStatistics();
System.out.println(stats);
```

Это приведет к:

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Получить фрагмент потока

Пример. Получите `Stream` из 30 элементов, содержащий от 21 до 50 (включительно) элемент коллекции.

```
final long n = 20L; // the number of elements to skip
final long maxSize = 30L; // the number of elements the stream should be limited to
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

Заметки:

- `IllegalArgumentException` выбрасывается, если `n` отрицательно или `maxSize` отрицательный
- обе `skip(long)` и `limit(long)` промежуточные операции
- если поток содержит менее `n` элементов, то `skip(n)` возвращает пустой поток
- как `skip(long)` и `limit(long)` являются дешевыми операциями на последовательных поточных трубопроводах, но могут быть довольно дорогими на упорядоченных параллельных трубопроводах

Конкатенация потоков

Объявление переменной для примеров:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Пример 1 - Объединение двух `Stream` s

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
// prints: abc123
```

Пример 2 - Конкатенация более двух `Stream` s

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());

System.out.println(concat2.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

В качестве альтернативы , чтобы упростить вложенный `concat()` синтаксис `Stream` ы также могут быть объединены с `flatMap()` :

```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
    // or `.flatMap(Function.identity());` (java.util.function.Function)

System.out.println(concat3.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Будьте осторожны при построении `Stream` из повторной конкатенации, потому что доступ к элементу глубоко конкатенированного `Stream` может привести к глубоким цепочкам вызовов или даже `StackOverflowException` .

IntStream to String

Java не имеет *Char Stream* , поэтому при работе со `String` s и построении `Stream` of `Character` s, опция должна получить `IntStream` кодовых точек с использованием `String.codePoints()` . Таким образом, `IntStream` можно получить следующим образом:

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

Это немного больше, чтобы сделать конверсию другим способом, то есть IntStreamToString. Это можно сделать следующим образом:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Сортировка по потоку

```
List<String> data = new ArrayList<>();
data.add("Sydney");
data.add("London");
data.add("New York");
data.add("Amsterdam");
data.add("Mumbai");
data.add("California");

System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());

System.out.println(sortedData);
```

Выход:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

Также возможно использовать другой механизм сравнения, так как существует перегруженная `sorted` версия, которая принимает в качестве аргумента компаратор.

Кроме того, вы можете использовать выражение лямбда для сортировки:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

Это вывело бы `[Sydney, New York, Mumbai, London, California, Amsterdam]`

Вы можете использовать `Comparator.reverseOrder()` чтобы иметь компаратор, который налагает `reverse` сторону естественного упорядочения.

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Потоки примитивов

Java предоставляет специализированные `Stream` s для трех типов примитивов `IntStream` (для `int` s), `LongStream` (для `long` s) и `DoubleStream` (для `double` s). Помимо оптимизации реализаций их соответствующих примитивов, они также предоставляют несколько конкретных терминальных методов, как правило, для математических операций. Например:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Соберите результаты потока в массив

Аналоговый, чтобы получить коллекцию для `Stream` методом `collect()` , массив может быть получен методом `Stream.toArray()` :

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");

String[] filteredFruits = fruits.stream()
    .filter(s -> s.contains("a"))
    .toArray(String[]::new);

// prints: [apple, banana, pear, orange]
System.out.println(Arrays.toString(filteredFruits));
```

`String[]::new` - это особый вид ссылки на метод: ссылка на конструктор.

Поиск первого элемента, который соответствует предикату

Можно найти первый элемент `Stream` который соответствует условию.

В этом примере мы найдем первое `Integer` , квадрат которого превышает `50000` .

```
IntStream.iterate(1, i -> i + 1) // Generate an infinite stream 1,2,3,4...
    .filter(i -> (i*i) > 50000) // Filter to find elements where the square is >50000
    .findFirst(); // Find the first filtered element
```

Это выражение вернет `OptionalInt` с результатом.

Обратите внимание, что с бесконечным `Stream` Java будет проверять каждый элемент до тех пор, пока не найдет результат. С конечным `Stream` , если Java исчерпывает элементы, но все равно не может найти результат, он возвращает пустой `OptionalInt` .

Использование IntStream для перебора индексов

`Stream` s элементов обычно не позволяет получить доступ к значению индекса текущего элемента. Чтобы перебирать массив или `ArrayList` , имея доступ к индексам, используйте `IntStream.range(start, endExclusive)` .

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };

IntStream.range(0, names.length)
    .mapToObj(i -> String.format("#%d %s", i + 1, names[i]))
    .forEach(System.out::println);
```

Метод `range(start, endExclusive)` возвращает другой `ìntStream` а `mapToObj(mapper)` возвращает поток `String` .

Выход:

1 Jon
2 Дарин
3 Бауке
4 Ханс
5 Марк

Это очень похоже на использование нормального `for` цикла со счетчиком, но с выгодой конвейерных и распараллеливания:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

Сгладить потоки с помощью flatMap ()

`Stream` предметов, которые в свою очередь могут быть потоковыми, может быть сплющен в один непрерывный `Stream` :

Массив списка элементов можно преобразовать в один список.

```
List<String> list1 =  Arrays.asList("one", "two");
      List<String> list2 =  Arrays.asList("three","four","five");
      List<String> list3 =  Arrays.asList("six");
          List<String> finalList = Stream.of(list1, list2,
list3).flatMap(Collection::stream).collect(Collectors.toList());
System.out.println(finalList);

// [one, two, three, four, five, six]
```

Карта, содержащая Список элементов как значений, может быть сглажена в комбинированный список

```
Map<String, List<Integer>> map = new LinkedHashMap<>();
map.put("a", Arrays.asList(1, 2, 3));
map.put("b", Arrays.asList(4, 5, 6));

List<Integer> allValues = map.values() // Collection<List<Integer>>
        .stream()                      // Stream<List<Integer>>
        .flatMap(List::stream)         // Stream<Integer>
        .collect(Collectors.toList());

System.out.println(allValues);
// [1, 2, 3, 4, 5, 6]
```

`List Map` может быть сплющен в один непрерывный `Stream`

```
List<Map<String, String>> list = new ArrayList<>();
Map<String,String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String,String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);


Set<String> output= list.stream()  //  Stream<Map<String, String>>
    .map(Map::values)              // Stream<List<String>>
    .flatMap(Collection::stream)   // Stream<String>
    .collect(Collectors.toSet());  //Set<String>
// [one, two, three,four]
```

Создание карты на основе потока

Простой случай без дубликатов ключей

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

Чтобы сделать вещи более декларативными, мы можем использовать статический метод в

`Function` интерфейсе - `Function.identity()` . Мы можем заменить этот `element -> element` лямбда- `element -> element` `Function.identity()` .

Случай, когда могут быть дубликаты ключей

В javadoc для `Collectors.toMap` указано:

> Если отображаемые ключи содержат дубликаты (в соответствии с `Object.equals(Object)`), при выполнении операции сбора генерируется `IllegalStateException` . Если отображаемые ключи могут иметь дубликаты, `toMap(Function, Function, BinaryOperator)` используйте `toMap(Function, Function, BinaryOperator)` .

```
Stream<String> characters = Stream.of("A", "B", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(
                element -> element.hashCode(),
                element -> element,
                (existingVal, newVal) -> (existingVal + newVal)));

// map = {65=A, 66=BB, 67=C}
```

`BinaryOperator` передается в `Collectors.toMap(...)` генерирует значение, которое нужно сохранить в случае столкновения. Оно может:

- возвращает старое значение, так что первое значение в потоке имеет приоритет,
- вернуть новое значение, так что последнее значение в потоке имеет приоритет или
- объединить старые и новые значения

Группировка по значению

Вы можете использовать `Collectors.groupingBy` когда вам нужно выполнить эквивалент операции каскадирования базы данных «по группам». Чтобы проиллюстрировать это, создается следующая карта, в которой имена людей сопоставляются с фамилиями:

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
        .collect(
                // function mapping input elements to keys
                Collectors.groupingBy(Person::getName,
                // function mapping input elements to values,
                // how to store values
                Collectors.mapping(Person::getSurname, Collectors.toList()))
        );
```

```
// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

Живой на Ideone

Генерация случайных строк с использованием потоков

Иногда бывает полезно создавать случайные `Strings` , возможно, как идентификатор сеанса для веб-службы или начальный пароль после регистрации для приложения. Это может быть легко достигнуто с помощью `Stream` s.

Сначала нам нужно инициализировать генератор случайных чисел. Чтобы повысить безопасность созданных `String` s, рекомендуется использовать `SecureRandom` .

Примечание . Создание `SecureRandom` довольно дорогое, поэтому лучше всего сделать это только один раз и время от времени вызывать один из методов `setSeed()` для его `setSeed()` .

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));
//20 Bytes as a seed is rather arbitrary, it is the number used in the JavaDoc example
```

При создании случайных `String` s мы обычно хотим, чтобы они использовали только определенные символы (например, только буквы и цифры). Поэтому мы можем создать метод, возвращающий `boolean` которое впоследствии может быть использовано для фильтрации `Stream` .

```
//returns true for all chars in 0-9, a-z and A-Z
boolean useThisCharacter(char c){
    //check for range to avoid using all unicode Letter (e.g. some chinese symbols)
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);
}
```

Затем мы сможем использовать RNG для генерации случайной строки определенной длины, содержащей кодировку, которая передает нашу проверку `useThisCharacter` .

```
public String generateRandomString(long length){
    //Since there is no native CharStream, we use an IntStream instead
    //and convert it to a Stream<Character> using mapToObj.
    //We need to specify the boundaries for the int values to ensure they can safely be cast
to char
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c ->
this::useThisCharacter).limit(length);

    //now we can use this Stream to build a String utilizing the collect method.
    String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
    return randomString;
}
```

Использование потоков для реализации математических функций

`Stream` с, и особенно `IntStream` с, является элегантным способом реализации суммирования условий (a). Диапазоны `Stream` могут использоваться как границы суммирования.

Например, приближение Мадхавы к Pi дается формулой (Источник: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Это можно вычислить с произвольной точностью. Например, на 101 срок:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Примечание: с `double` точностью, выбирая верхнюю границу 29, достаточно, чтобы получить результат, который неотличим от `Math.Pi`

Использование потоков и ссылок на методы для записи самодокументирующих процессов

Ссылки на методы делают отличный самодокументирующий код, а использование ссылок на методы с помощью `Stream` делает сложные процессы простыми для чтения и понимания. Рассмотрим следующий код:

```
public interface Ordered {
    default int getOrder(){
        return 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
                    results.add(value);
                }
            }
        }
    }
```

```
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
    });
    return results;
}
```

Этот последний метод, переписанный с использованием ссылок `Stream` и методов, намного читабельнее, и каждый шаг процесса быстро и легко понятен - он не просто короче, он также показывает с первого взгляда, какие интерфейсы и классы отвечают за код на каждом шаге:

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
}
```

Использование потоков Map.Entry для сохранения начальных значений после сопоставления

Когда у вас есть `Stream` вам нужно сопоставить, но вы хотите сохранить начальные значения, вы можете сопоставить `Stream` с `Map.Entry<K,V>` с помощью утилиты, например:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Затем вы можете использовать свой конвертер для обработки `Stream` имеющего доступ к исходным и отображаемым значениям:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

Затем вы можете продолжить обработку этого `Stream` как обычно. Это позволяет избежать накладных расходов на создание промежуточной коллекции.

Категории операций по потоку

Операции потоков делятся на две основные категории: промежуточные и терминальные операции и две подкатегории, без гражданства и состояния.

Промежуточные операции:

Промежуточная операция всегда *ленива* , например простой `Stream.map` . Он не вызывается до тех пор, пока поток фактически не будет потреблен. Это легко проверить:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Промежуточные операции являются общими строительными блоками потока, закодированными после источника, и за ними обычно следует операция терминала, запускающая цепочку потоков.

Терминальные операции

Операции терминала - это то, что вызывает потребление потока. Некоторые из наиболее распространенных - `Stream.forEach` или `Stream.collect` . Они обычно размещаются после цепочки промежуточных операций и почти всегда *стремятся* .

Операции без гражданства

Безгражданство означает, что каждый элемент обрабатывается без контекста других элементов. Операции бездействия позволяют эффективно обрабатывать потоки с использованием памяти. Такие операции, как `Stream.map` и `Stream.filter` которые не требуют информации о других элементах потока, считаются апатридами.

Операции со штатом

Стойкость означает, что операция по каждому элементу зависит от (некоторых) других элементов потока. Для этого требуется сохранение состояния. Операции состояния могут прерываться с длинными или бесконечными потоками. Такие операции, как `Stream.sorted` требуют, чтобы весь поток обрабатывался до того, как `Stream.sorted` какой-либо элемент, который сломается в достаточно длинном потоке элементов. Это может быть продемонстрировано длинным потоком (**выполняется на свой страх и риск**):

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
```

```
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

Это вызовет `Stream.sorted` памяти из-за `Stream.sorted` :

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Преобразование итератора в поток

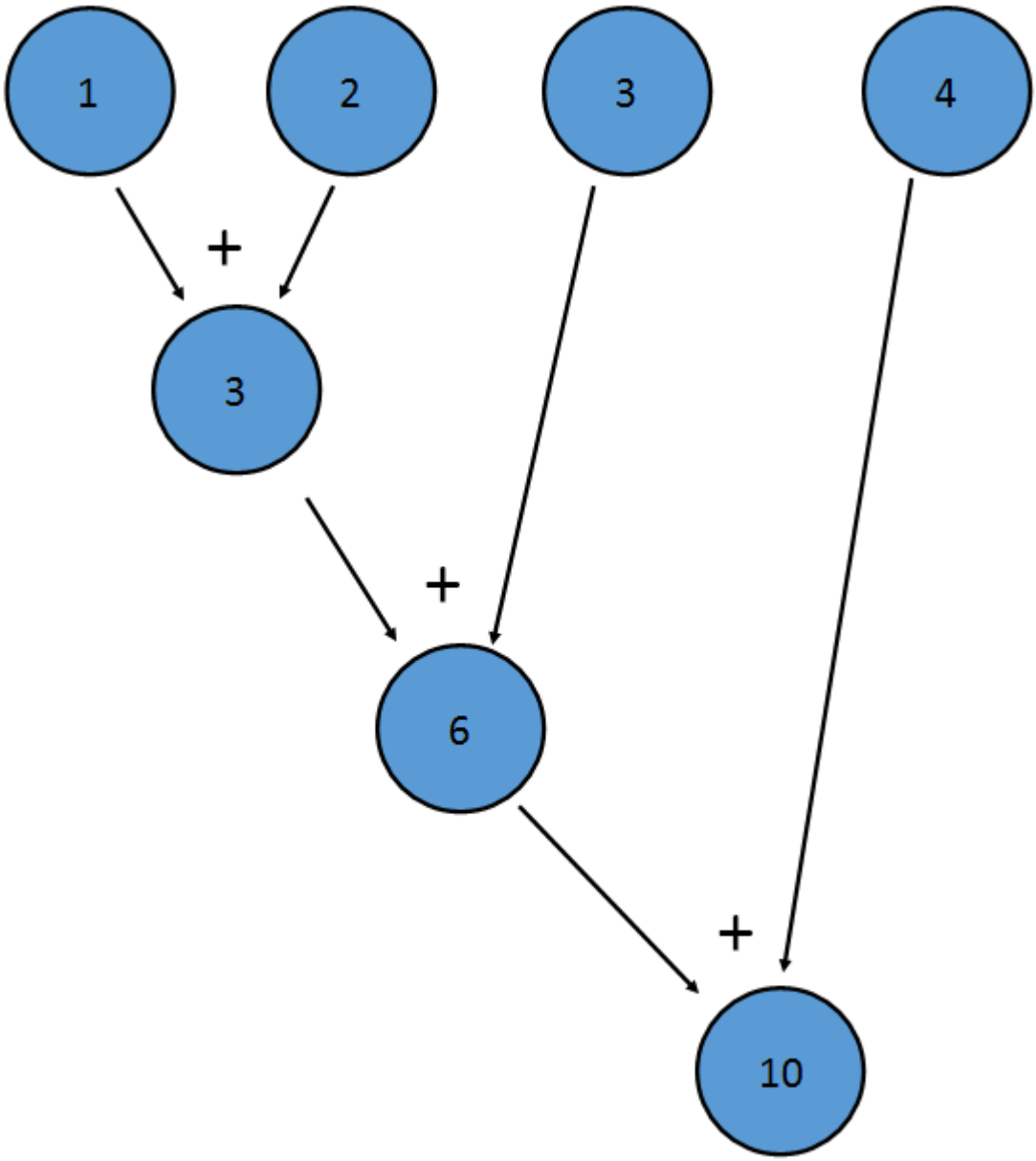
Используйте `Spliterators.spliterator()` или `Spliterators.spliteratorUnknownSize()` для преобразования итератора в поток:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Сокращение потока

Сокращение - это процесс применения бинарного оператора к каждому элементу потока, чтобы привести к одному значению.

Метод `sum()` для `IntStream` является примером сокращения; он применяет дополнение к каждому члену потока, что приводит к одному окончательному значению:



Это эквивалентно `(((1+2)+3)+4)`

Метод `reduce` Stream позволяет создать индивидуальное сокращение. Для реализации метода `sum()` можно использовать метод `reduce` :

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

`Optional` версия возвращается, так что пустые потоки могут обрабатываться соответствующим образом.

Другим примером сокращения является объединение `Stream<LinkedList<T>>` в один `LinkedList<T>` :

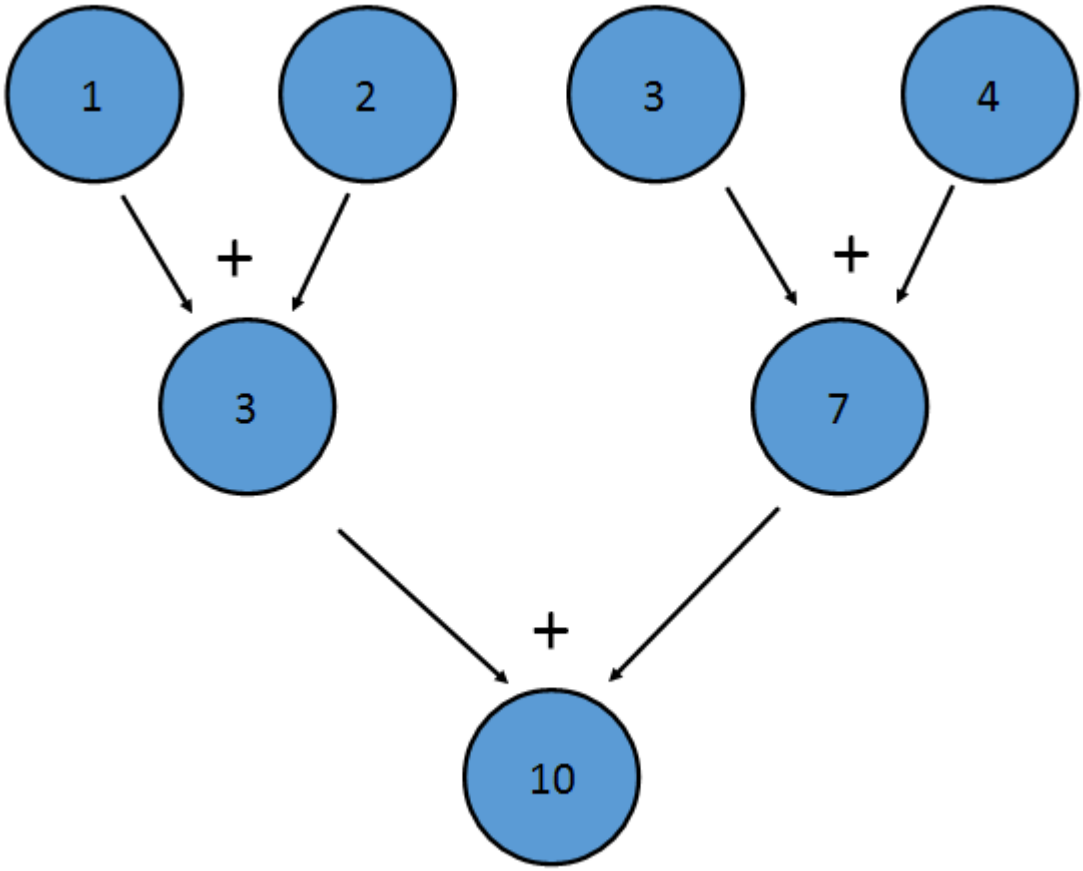
```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T>
list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

Вы также можете предоставить *элемент идентификации* . Например, элемент идентификации для сложения равен 0, как `x+0==x` . Для умножения единичный элемент равен 1, так как `x*1==x` . В вышеприведенном случае элемент идентификации представляет собой пустой `LinkedList<T>` , потому что если вы добавляете пустой список в другой список, список, который вы добавляете, не изменяется:

```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1,
LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

Обратите внимание, что, когда предоставляется элемент идентификации, возвращаемое значение не обернуто в `Optional` -if, вызываемый в пустом потоке, `reduce()` вернет элемент идентификации.

Бинарный оператор также должен быть *ассоциативным* , что означает `(a+b)+c==a+(b+c)` . Это связано с тем, что элементы могут быть уменьшены в любом порядке. Например, приведенное выше сведение может быть выполнено следующим образом:



Это сокращение эквивалентно записи `((1+2)+(3+4))` . Свойство ассоциативности также позволяет Java сокращать `Stream` параллельно - часть потока может быть уменьшена каждым процессором, причем сокращение объединяет результат каждого процессора в конце.

Присоединение потока к одной строке

Часто используемый прецедент, создающий `String` из потока, где элементы потока разделяются определенным символом. Для этого можно использовать метод `Collectors.joining()` , как в следующем примере:

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", "));

System.out.println(result);
```

Выход:

APPLE, BANANA, ORANGE, PEAR

Метод `Collectors.joining()` может также обслуживать pre- и postfixes:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", ", "Fruits: ", "."));

System.out.println(result);
```

Выход:

> Фрукты: APPLE, ORANGE, PEAR.

Живой на Ideone

Прочитайте Streams онлайн: https://riptutorial.com/ru/java/topic/88/streams

глава 36: StringBuffer

Вступление

Введение в класс Java StringBuffer.

Examples

Класс буфера String

Ключевые моменты: -

- используется для создания изменяемой (модифицируемой) строки.
- **Mutable** : - Что можно изменить.
- является потокобезопасным, т. е. несколько потоков не могут получить к нему доступ одновременно.

Методы: -

- публичный синхронизированный StringBuffer append (String s)
- общедоступная синхронизированная вставка StringBuffer (int offset, String s)
- public synchronized StringBuffer replace (int startIndex, int endIndex, String str)
- public synchronized StringBuffer delete (int startIndex, int endIndex)
- открытый синхронизированный реверс StringBuffer ()
- public int capacity ()
- public void secureCapacity (int minimumCapacity)
- public char charAt (индекс int)
- public int length ()
- public String substring (int beginIndex)
- public String substring (int beginIndex, int endIndex)

Пример Отображение различий между реализацией String и String Buffer:

```
class Test {
 public static void main(String args[])
```

```
 {
  String str = "study";
  str.concat("tonight");
  System.out.println(str);      // Output: study

  StringBuffer strB = new StringBuffer("study");
  strB.append("tonight");
  System.out.println(strB);    // Output: studytonight
 }
}
```

Прочитайте StringBuffer онлайн: https://riptutorial.com/ru/java/topic/10757/stringbuffer

глава 37: StringBuilder

Вступление

Класс Java StringBuilder используется для создания изменяемой (модифицируемой) строки. Класс Java StringBuilder аналогичен классу StringBuffer, за исключением того, что он не синхронизирован. Он доступен с JDK 1.5.

Синтаксис

- новый StringBuilder ()
- новый StringBuilder (int capacity)
- новый StringBuilder (CharSequence seq)
- новый StringBuilder (построитель StringBuilder)
- новый StringBuilder (строка строки)
- новый StringJoiner (разделитель CharSequence)
- новый StringJoiner (разделитель CharSequence, префикс CharSequence, суффикс CharSequence)

замечания

Создание нового `StringBuilder` с типом `char` в качестве параметра приведет к вызывая конструктор с аргументом `int capacity` , а не один с аргументом `String string` :

```
StringBuilder v = new StringBuilder('I'); //'I' is a character, "I" is a String.
System.out.println(v.capacity()); --> output 73
System.out.println(v.toString()); --> output nothing
```

Examples

Повторить строку n раз

Задача: создать `String` содержащую `n` повторений `String s` .

Тривиальный подход будет многократно конкатенировать `String`

```
final int n = ...
```

```
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

Это создает `n` новых экземпляров строк, содержащих от 1 до `n` повторений `s` приводит к времени выполнения `O(s.length() * n²) = O(s.length() * (1+2+...+(n-1)+n))` .

Чтобы избежать этого `StringBuilder` следует использовать, что позволяет создавать `String` в `O(s.length() * n)` :

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Сравнение StringBuffer, StringBuilder, Formatter и StringJoiner

В `StringBuffer` , `StringBuilder` , `Formatter` и `StringJoiner` классы классы утилиты Java SE, которые используются в основном для сборки строк из другой информации:

- Класс `StringBuffer` присутствует с Java 1.0 и предоставляет множество методов для создания и модификации «буфера», содержащего последовательность символов.
- Класс `StringBuilder` был добавлен в Java 5 для решения проблем производительности с исходным классом `StringBuffer` . API-интерфейсы для двух кланов по существу одинаковы. Основное различие между `StringBuffer` и `StringBuilder` заключается в том, что первое является потокобезопасным и синхронизированным, а второе - нет.

В этом примере показано, как можно использовать `StringBuilder` :

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();
sb.append("One=").append(one).append(", Color=").append(color).append('\n');
System.out.print(sb);
// Prints "One=1, Colour=red" followed by an ASCII newline.
```

(Класс `StringBuffer` используется одинаково: просто измените `StringBuilder` на `StringBuffer` в приведенном выше)

Классы `StringBuffer` и `StringBuilder` подходят как для сборки, так и для изменения строк; т.е.

они предоставляют методы для замены и удаления символов, а также их добавления в различные. Восстановление двух классов зависит от задачи сборки строк.

- Класс `Formatter` был добавлен в Java 5 и свободно моделируется функцией `sprintf` в стандартной библиотеке C. Он принимает строку *формата* со встроенными *спецификаторами формата* и последовательности других аргументов и генерирует строку путем преобразования аргументов в текст и замены их вместо спецификаторов формата. Подробности спецификаторов формата говорят о том, как аргументы преобразуются в текст.

- Класс `StringJoiner` был добавлен в Java 8. Это специальный форматир специального назначения, который лаконично форматирует последовательность строк с разделителями между ними. Он разработан с *полным* API и может использоваться с потоками Java 8.

Вот некоторые типичные примеры использования `Formatter` :

```
// This does the same thing as the StringBuilder example above
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// Prints "One=1, Colour=red" followed by the platform's line separator

// The same thing using the `String.format` convenience method
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

Класс `StringJoiner` не идеален для вышеуказанной задачи, поэтому здесь приведен пример форматирования массива строк.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

Варианты использования для 4 классов можно суммировать:

- `StringBuilder` подходит для любой задачи сборки строки или изменения строки.
- `StringBuffer` использует (только), когда вам нужна потоковая версия `StringBuilder` .
- `Formatter` предоставляет гораздо более богатые функции форматирования строк, но не так эффективен, как `StringBuilder` . Это связано с тем, что каждый вызов `Formatter.format(...)` влечет за собой:
  - разбор строки `format` ,
  - создание и заполнение массива *varargs* и
  - autoboxing любые примитивные аргументы типа.
- `StringJoiner` обеспечивает `StringJoiner` и эффективное форматирование

последовательности строк с разделителями, но не подходит для других задач форматирования.

Прочитайте StringBuilder онлайн: https://riptutorial.com/ru/java/topic/1037/stringbuilder

глава 38: sun.misc.Unsafe

замечания

Класс `Unsafe` позволяет программе выполнять действия, которые не допускаются компилятором Java. Обычным программам следует избегать использования `Unsafe` .

ПРЕДУПРЕЖДЕНИЯ

1. Если вы ошиблись, используя `Unsafe` API, ваши приложения могут привести к сбою и / или выявлению симптомов, которые трудно диагностировать.

2. `Unsafe` API может быть изменен без предварительного уведомления. Если вы используете его в своем коде, вам может потребоваться переписать код при изменении версий Java.

Examples

Выполнение sun.misc.Unsafe через отражение

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // Handle
    } catch (IllegalArgumentException e) {
        // Handle
    } catch (NoSuchFieldException e) {
        // Handle
    } catch (SecurityException e) {
        // Handle
    }
}
```

`sun.misc.Unsafe` имеет частный конструктор, а статический `getUnsafe()` проверкой загрузчика классов, чтобы обеспечить загрузку кода с помощью загрузчика основного класса. Поэтому одним из способов загрузки экземпляра является использование отражения для получения статического поля.

Выполнение sun.misc.Unsafe через bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
```

```
}
```

Хотя этот пример будет скомпилирован, он, скорее всего, потерпит неудачу во время выполнения, если класс Unsafe не будет загружен с помощью основного загрузчика классов. Чтобы убедиться, что JVM должен быть загружен с соответствующими аргументами, например:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

Затем класс `foo.bar.MyApp` может использовать `UnsafeLoader.loadUnsafe()` .

Получение экземпляра небезопасного

Небезопасно хранится как частное поле, к которому невозможно получить доступ напрямую. Конструктор является закрытым и единственным способом доступа к `public static Unsafe getUnsafe()` имеет привилегированный доступ. Используя рефлексию, можно сделать доступными частные поля:

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
        };

        unsafe = AccessController.doPrivileged(action);
    } catch (final Throwable t) {
        throw new RuntimeException("Exception accessing Unsafe", t);
    }

    UNSAFE = unsafe;
}
```

Использование небезопасных

Ниже перечислены некоторые виды использования небезопасных:

использование	API
Отключение кучи / прямой памяти, перераспределение и освобождение памяти	`allocateMemory(bytes)` , `reallocateMemory(address, bytes)` и `freeMemory(address)`
Запонки памяти	`loadFence()` , `storeFence()` , `fullFence()`

использование	API
Стоянка тока	`park(isAbsolute, time)` , `unpark(thread)`
Прямой доступ к полю и памяти	`get*` и `put*` семейство методов
Выброс исключенных исключений	`throwException(e)`
CAS и атомные операции	`compareAndSwap*` семейство методов
Настройка памяти	`setMemory`
Неустойчивые или параллельные операции	`get*Volatile` , `put*Volatile` , `putOrdered*`

Получаемое семейство методов относится к данному объекту. Если объект имеет значение null, он рассматривается как абсолютный адрес.

```
// Putting a value to a field
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
UNSAFE.putLong(this, fieldOffset , newValue);

// Puting an absolute value
 UNSAFE.putLong(null, address, newValue);
 UNSAFE.putLong(address, newValue);
```

Некоторые методы определены только для int и longs. Вы можете использовать эти методы для поплавков и удвоений, используя `floatToRawIntBits` , `intBitsToFloat`, doubleToRawLongBits , longBitsToDouble`

Прочитайте sun.misc.Unsafe онлайн: https://riptutorial.com/ru/java/topic/6771/sun-misc-unsafe

глава 39: ThreadLocal

замечания

Лучше всего использовать объекты, зависящие от внутренних элементов при вызове вызова, но в противном случае они не имеют аналогов, например `SimpleDateFormat` , `Marshaller`

Для использования `Random` ThreadLocal рассмотрите возможность использования `ThreadLocalRandom`

Examples

Функциональная инициализация ThreadLocal Java 8

```
public static class ThreadLocalExample
{
    private static final ThreadLocal<SimpleDateFormat> format =
        ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

    public String formatDate(Date date)
    {
        return format.get().format(date);
    }
}
```

Основное использование ThreadLocal

Java `ThreadLocal` используется для создания локальных переменных потока. Известно, что потоки объекта разделяют его переменные, поэтому переменная не является потокобезопасной. Мы можем использовать синхронизацию для безопасности потоков, но если мы хотим избежать синхронизации, ThreadLocal позволяет нам создавать переменные, которые являются локальными для потока, то есть только этот поток может читать или записывать эти переменные, поэтому другие потоки, выполняющие один и тот же фрагмент кода не смогут обращаться к другим переменным ThreadLocal.

Это можно использовать, мы можем использовать переменные `ThreadLocal` . в ситуациях, когда у вас есть пул потоков, например, в веб-службе. Например, создание объекта `SimpleDateFormat` каждый раз для каждого запроса занимает много времени, а статический не может быть создан, поскольку `SimpleDateFormat` не является потокобезопасным, поэтому мы можем создать ThreadLocal, чтобы мы могли выполнять поточно-безопасные операции без накладных расходов на создание `SimpleDateFormat` каждый время.

Следующий фрагмент кода показывает, как его можно использовать:

Каждый поток имеет собственную переменную `ThreadLocal` и они могут использовать методы `get()` и `set()` для получения значения по умолчанию или изменения локального значения для Thread.

`ThreadLocal` обычно являются частные статические поля в классах, которые хотят связать состояние с потоком.

Вот небольшой пример, показывающий использование ThreadLocal в java-программе и доказательство того, что каждый поток имеет свою собственную копию переменной `ThreadLocal` .

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
 // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default
Formatter = "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }

}
```

Выход:

```
Thread Name= 0 default Formatter = yyyyMMdd HHmm

Thread Name= 1 default Formatter = yyyyMMdd HHmm

Thread Name= 0 formatter = M/d/yy h:mm a

Thread Name= 2 default Formatter = yyyyMMdd HHmm

Thread Name= 1 formatter = M/d/yy h:mm a

Thread Name= 3 default Formatter = yyyyMMdd HHmm

Thread Name= 4 default Formatter = yyyyMMdd HHmm

Thread Name= 4 formatter = M/d/yy h:mm a

Thread Name= 5 default Formatter = yyyyMMdd HHmm

Thread Name= 2 formatter = M/d/yy h:mm a

Thread Name= 3 formatter = M/d/yy h:mm a

Thread Name= 6 default Formatter = yyyyMMdd HHmm

Thread Name= 5 formatter = M/d/yy h:mm a

Thread Name= 6 formatter = M/d/yy h:mm a

Thread Name= 7 default Formatter = yyyyMMdd HHmm

Thread Name= 8 default Formatter = yyyyMMdd HHmm

Thread Name= 8 formatter = M/d/yy h:mm a

Thread Name= 7 formatter = M/d/yy h:mm a

Thread Name= 9 default Formatter = yyyyMMdd HHmm

Thread Name= 9 formatter = M/d/yy h:mm a
```

Как видно из вывода, Thread-0 изменил значение форматирования, но форматир по умолчанию по-умолчанию-нить-то же, что и инициализированное значение.

Несколько потоков с одним общим объектом

В этом примере у нас есть только один объект, но он разделяется между / выполняется на разных потоках. Обычное использование полей для сохранения состояния было бы невозможно, потому что другой поток тоже увидит это (или, вероятно, не увидит).

```
public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}
```

В Foo мы начинаем отсчет с нуля. Вместо сохранения состояния в поле мы сохраняем наше

текущее число в объекте ThreadLocal, который статически доступен. Обратите внимание, что синхронизация в этом примере не связана с использованием ThreadLocal, а скорее обеспечивает лучший выход на консоль.

```
public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //Although accessing a static field, we get our own (previously saved) value.
                int value = threadLocal.get();
                System.out.println(Thread.currentThread().getName() + ": " + value);

                //Update our own variable
                threadLocal.set(value + 1);

                try {
                    threadLocal.notifyAll();
                    if (i < ITERATIONS - 1) {
                        threadLocal.wait();
                    }
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}
```

Из вывода видно, что каждый поток подсчитывает для себя и не использует значение другого:

```
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
```

```
Thread 2: 9
```

Прочитайте ThreadLocal онлайн: https://riptutorial.com/ru/java/topic/2001/threadlocal

глава 40: TreeMap и TreeSet

Вступление

`TreeMap` и `TreeSet` являются базовыми `TreeSet` Java, добавленными в Java 1.2. `TreeMap` - **измененная** , **упорядоченная** реализация `Map` . Аналогично, `TreeSet` является **изменяемой** , **упорядоченной** реализацией `Set` .

`TreeMap` реализован как дерево Red-Black, которое обеспечивает время доступа `O(log n)` . `TreeSet` реализуется с использованием `TreeMap` с фиктивными значениями.

Обе коллекции **не** являются потокобезопасными.

Examples

TreeMap простого типа Java

Сначала мы создаем пустую карту и вставляем в нее некоторые элементы:

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
```

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
```

```
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

Когда у нас есть несколько элементов на карте, мы можем выполнить некоторые операции:

```
System.out.println(treeMap.firstEntry()); // Prints 1=one
System.out.println(treeMap.lastEntry()); // Prints 12=twelve
System.out.println(treeMap.size()); // Prints 4, since there are 4 elemens in the map
System.out.println(treeMap.get(12)); // Prints twelve
System.out.println(treeMap.get(15)); // Prints null, since the key is not found in the map
```

Мы также можем перебирать элементы карты, используя либо Iterator, либо цикл foreach. Обратите внимание, что записи печатаются в соответствии с их естественным порядком , а не в порядке ввода:

Java SE 7

```
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

```
Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

TreeSet простого типа Java

Сначала мы создаем пустой набор и вставляем в него некоторые элементы:

Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<>();
```

Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

```
treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
treeSet.add(12);
```

Когда у нас есть несколько элементов в наборе, мы можем выполнить некоторые операции:

```
System.out.println(treeSet.first()); // Prints 1
System.out.println(treeSet.last()); // Prints 12
System.out.println(treeSet.size()); // Prints 4, since there are 4 elemens in the set
System.out.println(treeSet.contains(12)); // Prints true
System.out.println(treeSet.contains(15)); // Prints false
```

Мы также можем перебирать элементы карты, используя либо Iterator, либо цикл foreach. Обратите внимание, что записи печатаются в соответствии с их естественным порядком , а не в порядке ввода:

Java SE 7

```
for (Integer i : treeSet) {
    System.out.print(i + " "); //prints 1 4 10 12
}
```

```
Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1 4 10 12
}
```

TreeMap / TreeSet настраиваемого типа Java

Поскольку `TreeMap` s и `TreeSet` сохраняют ключи / элементы в соответствии с их естественным порядком . Для этого ключи `TreeMap` и элементы `TreeSet` должны сопоставляться друг с другом.

Скажем, у нас есть пользовательский класс `Person` :

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

Если мы сохраним его как-есть в `TreeSet` (или ключ в TreeMap):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Тогда мы столкнулись бы с Исключением, таким как этот:

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
    at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

Чтобы исправить это, предположим, что мы хотим заказать экземпляры `Person` на основе порядка их идентификаторов (`private int id`). Мы могли бы сделать это одним из двух способов:

1. Одним из решений является изменение `Person` чтобы он реализовал интерфейс Comparable :

   ```
   public class Person implements Comparable<Person> {
       private int id;
       private String firstName, lastName;
       private Date birthday;

       //... Constuctors, getters, setters and various methods

       @Override
       public int compareTo(Person o) {
           return Integer.compare(this.id, o.id); //Compare by id
       }
   }
   ```

2. Еще одно решение - предоставить `TreeSet` компаратору :

Java SE 8

```
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));
```

```
TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>(){
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});
```

Однако есть два оговорки к обоим подходам:

1. **Очень важно** не изменять какие-либо поля, используемые для упорядочения, как только экземпляр был вставлен в `TreeSet` / `TreeMap` . В приведенном выше примере, если мы изменим `id` человека, который уже вставлен в коллекцию, мы можем столкнуться с неожиданным поведением.

2. Важно правильно и последовательно выполнять сравнение. Согласно Джавадоку :

   Разработчик должен обеспечить `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` для всех x и y. (Это означает, что `x.compareTo(y)` должен генерировать исключение, если `y.compareTo(x)` генерирует исключение).

   Разработчик должен также гарантировать, что отношение транзитивно: `(x.compareTo(y)>0 && y.compareTo(z)>0)` подразумевает `x.compareTo(z)>0` .

   Наконец, разработчик должен убедиться, что `x.compareTo(y)==0` означает, что `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` для всех z.

Безопасность TreeMap и TreeSet

`TreeMap` и `TreeSet` **не** являются потокобезопасными коллекциями, поэтому необходимо соблюдать осторожность при использовании в многопоточных программах.

Оба `TreeMap` и `TreeSet` безопасны при чтении, даже одновременно, несколькими потоками. Поэтому, если они были созданы и заполнены одним потоком (скажем, в начале программы), и только после этого будут прочитаны, но не изменены несколькими потоками, нет причин для синхронизации или блокировки.

Однако, если чтение и изменение одновременно или изменено одновременно более чем одним потоком, сбор может вызывать исключение ConcurrentModificationException или вести себя неожиданно. В этих случаях настоятельно необходимо синхронизировать / заблокировать доступ к коллекции, используя один из следующих подходов:

1. Использование `Collections.synchronizedSorted..` :

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

Это обеспечит реализацию SortedSet / SortedMap, поддерживаемую фактической коллекцией, и синхронизируется с некоторым объектом mutex. Обратите внимание, что это синхронизирует весь доступ на чтение и запись к коллекции на одном замке, поэтому даже одновременные чтения не будут возможны.

2. Ручная синхронизация на каком-либо объекте, как и сама коллекция:

```
 TreeSet<Integer> set = new TreeSet<>();
```

...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. Используя блокировку, такую как ReentrantReadWriteLock :

```
 TreeSet<Integer> set = new TreeSet<>();
 ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

...

```
 //Thread 1
 lock.writeLock().lock();
 set.add(4);
 lock.writeLock().unlock();
```

...

```
 //Thread 2
 lock.readLock().lock();
 set.contains(5);
 lock.readLock().unlock();
```

В отличие от предыдущих методов синхронизации, использование ReadWriteLock позволяет одновременно просматривать несколько потоков с карты.

Прочитайте TreeMap и TreeSet онлайн: https://riptutorial.com/ru/java/topic/9905/treemap-и-treeset

глава 41: Varargs (переменный аргумент)

замечания

Аргумент метода «varargs» позволяет вызывающим сторонам этого метода указывать несколько аргументов назначенного типа, каждый из которых является отдельным аргументом. Он указан в объявлении метода тремя периодами ASCII (. . .) после базового типа.

Сам метод получает эти аргументы как один массив, тип элемента которого является типом аргумента varargs. Массив создается автоматически (хотя вызывающим абонентам по-прежнему разрешено передавать явный массив вместо передачи нескольких значений в виде отдельных аргументов метода).

Правила для варгаров:

1. Варанг должен быть последним аргументом.
2. В методе может быть только один Varargs.

Вы должны следовать выше правилам, иначе программа даст ошибку компиляции.

Examples

Указание параметра varargs

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

Три периода после окончательного параметра указывают, что последний аргумент может быть передан как массив или как последовательность аргументов. Варгары могут использоваться только в конечной позиции аргумента.

Работа с параметрами Varargs

Используя varargs в качестве параметра для определения метода, можно передать либо массив, либо последовательность аргументов. Если последовательность аргументов передана, они автоматически преобразуются в массив.

В этом примере показан как массив, так и последовательность аргументов, передаваемых в метод `printVarArgArray()` , и то, как они обрабатываются одинаково в коде внутри метода:

```
public class VarArgs {

    // this method will print the entire contents of the parameter passed in

    void printVarArgArray(int... x) {
        for (int i = 0; i < x.length; i++) {
            System.out.print(x[i] + ",");
        }
    }

    public static void main(String args[]) {
        VarArgs obj = new VarArgs();

        //Using an array:
        int[] testArray = new int[]{10, 20};
        obj.printVarArgArray(testArray);

        System.out.println(" ");

        //Using a sequence of arguments
        obj.printVarArgArray(5, 6, 5, 8, 6, 31);
    }
}
```

Выход:

```
10,20,
5,6,5,8,6,31
```

Если вы определите метод, подобный этому, он даст ошибки времени компиляции.

```
void method(String... a, int... b , int c){} //Compile time error (multiple varargs )

void method(int... a, String b){} //Compile time error (varargs must be the last argument
```

Прочитайте Varargs (переменный аргумент) онлайн:
https://riptutorial.com/ru/java/topic/1948/varargs--переменный-аргумент-

глава 42: WeakHashMap

Вступление

Концепции слабого хашмапа

Examples

Концепции WeakHashmap

Ключевые моменты: -

- Реализация карты.
- хранит только слабые ссылки на его ключи.

Слабые ссылки : объекты, на которые ссылаются только слабые ссылки, - это мусор, собранный с нетерпением; GC не будет ждать, пока в этом случае ему понадобится память.

Разница между Hashmap и WeakHashMap: -

Если диспетчер памяти Java больше не имеет ссылки на объект, указанный в качестве ключа, то запись на карте будет удалена в WeakHashMap.

Пример :-

```
public class WeakHashMapTest {
    public static void main(String[] args) {
        Map hashMap= new HashMap();

        Map weakHashMap = new WeakHashMap();

        String keyHashMap = new String("keyHashMap");
        String keyWeakHashMap = new String("keyWeakHashMap");

        hashMap.put(keyHashMap, "Ankita");
        weakHashMap.put(keyWeakHashMap, "Atul");
        System.gc();
        System.out.println("Before: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));

        keyHashMap = null;
        keyWeakHashMap = null;

        System.gc();

        System.out.println("After: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));
    }
```

Различия в размерах (HashMap vs WeakHashMap):

Метод calling size () для объекта HashMap возвращает одинаковое количество пар ключ-значение. размер будет уменьшаться только в том случае, если метод remove () явно указан в объекте HashMap.

Поскольку сборщик мусора может отменить ключи в любое время, WeakHashMap может вести себя так, как если бы неизвестный поток молча удалял записи. Таким образом, метод size может возвращать меньшие значения с течением времени. Таким образом, **уменьшение WeakHashMap происходит автоматически** .

Прочитайте WeakHashMap онлайн: https://riptutorial.com/ru/java/topic/10749/weakhashmap

глава 43: XJC

Вступление

XJC - это инструмент Java SE, который компилирует файл схемы XML в полностью аннотированные классы Java.

Он распространяется в пакете JDK и находится по пути `/bin/xjc` .

Синтаксис

- xjc [options] файл схемы / URL / dir / jar ... [-b bindinfo] ...

параметры

параметр	подробности
файл схемы	Файл схемы xsd для преобразования в java

замечания

Инструмент XJC доступен как часть JDK. Он позволяет создавать java-код, аннотированный аннотациями JAXB, подходящими для (un) сортировки.

Examples

Создание Java-кода из простого XSD-файла

Схема XSD (schema.xsd)

Следующая xml-схема (xsd) определяет список пользователей с `name` атрибутов и `reputation` .

```
<?xml version="1.0"?>

<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ns="http://www.stackoverflow.com/users"
           elementFormDefault="qualified"
           targetNamespace="http://www.stackoverflow.com/users">
    <xs:element name="users" type="ns:Users"/>
```

```
    <xs:complexType name="Users">
        <xs:sequence>
            <xs:element type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="User">
        <xs:attribute name="name" use="required" type="xs:string"/>
        <xs:attribute name="reputation" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:int">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:schema>
```

Используя xjc

Для этого требуется, чтобы путь к инструменту xjc (двоичные файлы JDK) находился в переменной пути ОС.

Генерация кода может быть запущена с использованием

```
xjc schema.xsd
```

Это приведет к созданию java-файлов в рабочем каталоге.

Файлы результатов

Будут некоторые дополнительные комментарии, но в основном созданные файлы java выглядят следующим образом:

```
package com.stackoverflow.users;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
```

```
            user = new ArrayList<User>();
        }
        return this.user;
    }

}
```

```
package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}
```

```
package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }
```

```
    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }

}
```

package-info.java

```
@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.stackoverflow.users;
```

Прочитайте XJC онлайн: https://riptutorial.com/ru/java/topic/4538/xjc

глава 44: XOM - Объектная модель XML

Examples

Чтение XML-файла

Чтобы загрузить XML-данные с помощью XOM, вам нужно будет создать `Builder` из которого вы можете создать его в `Document` .

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

Чтобы получить корневой элемент, самый старший родитель в файле xml, вам нужно использовать `getRootElement()` в экземпляре `Document` .

```
Element root = doc.getRootElement();
```

Теперь класс Element имеет множество удобных методов, которые упрощают чтение xml. Ниже перечислены некоторые из наиболее полезных:

- `getChildElements(String name)` - возвращает экземпляр `Elements` который действует как массив элементов
- `getFirstChildElement(String name)` - возвращает первый дочерний элемент с этим тегом.
- `getValue()` - возвращает значение внутри элемента.
- `getAttributeValue(String name)` - возвращает значение атрибута с указанным именем.

Когда вы вызываете `getChildElements()` вы получаете экземпляр `Elements` . Из этого вы можете выполнить цикл и вызвать метод `get(int index)` чтобы получить все элементы внутри.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Пример: Вот пример чтения XML-файла:

Файл XML:

Код для чтения и печати:

```
import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // builder builds xml data
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // get the root element <example>
        Element root = doc.getRootElement();

        // gets all element with tag <person>
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // get the current person element
            Element person = people.get(q);

            // get the name element and its children: first and last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // get the age element
```

```
            Element ageElement = person.getFirstChildElement("age");

            // get the favorite color element
            Element favColorElement = person.getFirstChildElement("fav_color");

            String fName, lName, ageUnit, favColor;
            int age;

            try {
                fName = firstNameElement.getValue();
                lName = lastNameElement.getValue();
                age = Integer.parseInt(ageElement.getValue());
                ageUnit = ageElement.getAttributeValue("unit");
                favColor = favColorElement.getValue();

                System.out.println("Name: " + lName + ", " + fName);
                System.out.println("Age: " + age + " (" + ageUnit + ")");
                System.out.println("Favorite Color: " + favColor);
                System.out.println("----------------");

            } catch (NullPointerException ex){
                ex.printStackTrace();
            } catch (NumberFormatException ex){
                ex.printStackTrace();
            }
        }
    }

}
```

Это выведет на консоль:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

Запись в файл XML

Запись в XML-файл с использованием XOM очень похожа на его чтение, но в этом случае мы делаем экземпляры вместо того, чтобы извлекать их из корня.

Чтобы создать новый элемент, используйте конструктор `Element(String name)` . Вы захотите создать корневой элемент, чтобы его можно было легко добавить в `Document` .

```
Element root = new Element("root");
```

Класс `Element` имеет несколько удобных методов редактирования элементов. Они перечислены ниже:

- `appendChild(String name)` - это будет в основном устанавливать значение имени элемента.
- `appendChild(Node node)` - это сделает `node` элементами parent. (Элементы являются узлами, поэтому вы можете анализировать элементы).
- `addAttribute(Attribute attribute)` - добавит атрибут к элементу.

Класс `Attribute` имеет несколько разных конструкторов. Самый простой - это `Attribute(String name, String value)` .

Когда вы добавите все свои элементы в свой корневой элемент, вы можете превратить его в `Document` . `Document` примет `Element` в качестве аргумента в его конструкторе.

`Serializer` можно использовать для записи XML в файл. Вам нужно будет создать новый выходной поток для анализа в конструкторе `Serializer` .

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

пример

Код:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import nu.xom.Attribute;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;
import nu.xom.Serializer;

public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
            IOException{
        // root element <example>
        Element root = new Element("example");

        // make a array of people to store
        Person[] people = {new Person("Smith", "Dan", "years", "green", 23),
            new Person("Autry", "Bob", "months", "N/A", 3)};

        // add all the people
        for (Person person : people){

            // make the main person element <person>
            Element personElement = new Element("person");
```

```
        // make the name element and it's children: first and last
        Element nameElement = new Element("name");
        Element firstNameElement = new Element("first");
        Element lastNameElement = new Element("last");

        // make age element
        Element ageElement = new Element("age");

        // make favorite color element
        Element favColorElement = new Element("fav_color");

        // add value to names
        firstNameElement.appendChild(person.getFirstName());
        lastNameElement.appendChild(person.getLastName());

        // add names to name
        nameElement.appendChild(firstNameElement);
        nameElement.appendChild(lastNameElement);

        // add value to age
        ageElement.appendChild(String.valueOf(person.getAge()));

        // add unit attribute to age
        ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

        // add value to favColor
        favColorElement.appendChild(person.getFavoriteColor());

        // add all contents to person
        personElement.appendChild(nameElement);
        personElement.appendChild(ageElement);
        personElement.appendChild(favColorElement);

        // add person to root
        root.appendChild(personElement);
    }

    // create doc off of root
    Document doc = new Document(root);

    // the file it will be stored in
    File file = new File("out.xml");
    if (!file.exists()){
        file.createNewFile();
    }

    // get a file output stream ready
    FileOutputStream fileOutputStream = new FileOutputStream(file);

    // use the serializer class to write it all
    Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
    serializer.setIndent(4);
    serializer.write(doc);
}

private static class Person {

    private String lName, fName, ageUnit, favColor;
    private int age;
```

```
        public Person(String lName, String fName, String ageUnit, String favColor, int age){
            this.lName = lName;
            this.fName = fName;
            this.age = age;
            this.ageUnit = ageUnit;
            this.favColor = favColor;
        }

        public String getLastName() { return lName; }
        public String getFirstName() { return fName; }
        public String getAgeUnit() { return ageUnit; }
        public String getFavoriteColor() { return favColor; }
        public int getAge() { return age; }
    }

}
```

Это будет содержимое «out.xml»:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Прочитайте XOM - Объектная модель XML онлайн:
https://riptutorial.com/ru/java/topic/5091/xom---объектная-модель-xml

глава 45: Альтернативные коллекции

замечания

Этот раздел о коллекциях Java из guava, apache, eclipse: Multiset, Bag, Multimap, utils работает из этой библиотеки и так далее.

Examples

Apache HashBag, Guava HashMultiset и Eclipse HashBag

Сумка / ultiset хранит каждый объект в коллекции вместе с количеством вхождений. Дополнительные методы интерфейса позволяют одновременно добавлять или удалять несколько копий объекта. JDK-аналог - это HashMap <T, Integer>, когда значения являются количеством копий этого ключа.

Тип	гуайява	Коллекции коллекционеров Apache	Коллекции GS	JD
Заказ не определен	HashMultiset	HashBag	HashBag	Ha
отсортированный	TreeMultiset	TreeBag	TreeBag	Tr
Вносимые заказ	LinkedHashMultiset	-	-	Li
Параллельный вариант	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	Co
Параллельные и сортированные	-	SynchronizedSortedBag	SynchronizedSortedBag	Co
Неизменяемая коллекция	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	Co
Неизменяемость и сортировка	ImmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	Co)

Примеры :

1. Использование SynchronizedSortedBag из Apache :

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    Bag bag = SynchronizedSortedBag.synchronizedBag(new
TreeBag(Arrays.asList(INPUT_TEXT.split(" "))));

    // Print count words
    System.out.println(bag); // print [1:All!,2:Hello,1:Hi,2:World!]- in natural (alphabet)
order
    // Print all unique words
    System.out.println(bag.uniqueSet());    // print [All!, Hello, Hi, World!]- in natural
(alphabet) order


    // Print count occurrences of words
    System.out.println("Hello = " + bag.getCount("Hello"));    // print 2
    System.out.println("World = " + bag.getCount("World!"));    // print 2
    System.out.println("All = " + bag.getCount("All!"));    // print 1
    System.out.println("Hi = " + bag.getCount("Hi"));    // print 1
    System.out.println("Empty = " + bag.getCount("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.uniqueSet().size());    //print 4
```

2. Использование TreeBag из Eclipse (GC) :

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    MutableSortedBag<String> bag =  TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

    // Print count words
    System.out.println(bag); // print [All!, Hello, Hello, Hi, World!, World!]- in natural
order
    // Print all unique words
    System.out.println(bag.toSortedSet());    // print [All!, Hello, Hi, World!]- in natural
order

    // Print count occurrences of words
    System.out.println("Hello = " + bag.occurrencesOf("Hello"));    // print 2
    System.out.println("World = " + bag.occurrencesOf("World!"));    // print 2
    System.out.println("All = " + bag.occurrencesOf("All!"));    // print 1
    System.out.println("Hi = " + bag.occurrencesOf("Hi"));    // print 1
    System.out.println("Empty = " + bag.occurrencesOf("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.toSet().size());    //print 4
```

3. Использование LinkedHashMultiset из Guava :

```
    // Parse text to separate words
```

```
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split("
")));

    // Print count words
    System.out.println(multiset); // print [Hello x 2, World! x 2, All!, Hi]- in predictable
iteration order
    // Print all unique words
    System.out.println(multiset.elementSet());    // print [Hello, World!, All!, Hi] - in
predictable iteration order

    // Print count occurrences of words
    System.out.println("Hello = " + multiset.count("Hello"));    // print 2
    System.out.println("World = " + multiset.count("World!"));    // print 2
    System.out.println("All = " + multiset.count("All!"));    // print 1
    System.out.println("Hi = " + multiset.count("Hi"));    // print 1
    System.out.println("Empty = " + multiset.count("Empty"));    // print 0

    // Print count all words
    System.out.println(multiset.size());    //print 6

    // Print count unique words
    System.out.println(multiset.elementSet().size());    //print 4
```

Дополнительные примеры:

I. Коллекция Apache:

1. HashBag - заказ не определен
2. SynchronizedBag - одновременный и не заданный порядок
3. SynchronizedSortedBag - - одновременный и отсортированный порядок
4. TreeBag - отсортированный порядок

II. Коллекция GS / Eclipse

5. MutableBag - порядок не определен
6. MutableSortedBag - отсортированный порядок

III. гуайява

7. HashMultiset - порядок не определен
8. TreeMultiset - отсортированный порядок
9. LinkedHashMultiset - порядок вставки
10. ConcurrentHashMultiset - одновременный и не заданный порядок

Мультимап в коллекциях Guava, Apache и Eclipse

Эта многоадресная карта позволяет дублировать пары ключ-значение. Аналогами JDK являются HashMap <K, List>, HashMap <K, Set> и т. Д.

Заказ ключа	Заказ стоимости	дублировать	Аналоговый ключ	Аналоговое значение	гуайява
не определен	Вносимые заказ	да	HashMap	ArrayList	ArrayLis
не определен	не определен	нет	HashMap	HashSet	HashMu
не определен	отсортированный	нет	HashMap	TreeSet	Multimap newMulti HashMap, <TreeSet
Вносимые заказ	Вносимые заказ	да	LinkedHashMap	ArrayList	LinkedLi
Вносимые заказ	Вносимые заказ	нет	LinkedHashMap	LinkedHashSet	LinkedH
отсортированный	отсортированный	нет	TreeMap	TreeSet	TreeMul

Примеры использования Multimap

Задача : проанализировать «Hello World! Hello All! Hi World!» string для разделения слов и печати всех индексов каждого слова с помощью MultiMap (например, Hello = [0, 2], World! = [1, 5] и т. д.),

1. MultiValueMap из Apache

```
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Parse text to words and index
    List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
    // Create Multimap
    MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();


    // Fill Multimap
    int i = 0;
    for(String word: words) {
        multiMap.put(word, i);
        i++;
    }

    // Print all words
    System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
in random orders
    // Print all unique words
    System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
```

```
orders

    // Print all indexes
    System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
    System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
    System.out.println("All = " + multiMap.get("All!"));    // print [3]
    System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
    System.out.println("Empty = " + multiMap.get("Empty"));    // print null

    // Print count unique words
    System.out.println(multiMap.keySet().size());    //print 4
```

2. HashBiMap из коллекции GS / Eclipse

```
    String[] englishWords = {"one", "two", "three","ball","snow"};
    String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

    // Create Multiset
    MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
    // Create English-Polish dictionary
    int i = 0;
    for(String englishWord: englishWords) {
        biMap.put(englishWord, russianWords[i]);
        i++;
    }

    // Print count words
    System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg,
three=trzy} - in random orders
    // Print all unique words
    System.out.println(biMap.keySet());    // print [snow, two, one, three, ball] - in random
orders
    System.out.println(biMap.values());    // print [dwa, kula, jeden, snieg, trzy] - in
random orders

    // Print translate by words
    System.out.println("one = " + biMap.get("one"));    // print one = jeden
    System.out.println("two = " + biMap.get("two"));    // print two = dwa
    System.out.println("kula = " + biMap.inverse().get("kula"));    // print kula = ball
    System.out.println("snieg = " + biMap.inverse().get("snieg"));    // print snieg = snow
    System.out.println("empty = " + biMap.get("empty"));    // print empty = null

    // Print count word's pair
    System.out.println(biMap.size());    //print 5
```

3. HashMultiMap из Гуавы

```
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimap
Multimap<String, Integer> multiMap = HashMultimap.create();

// Fill Multimap
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
```

```
 }

 // Print all words
 System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
keys and values in random orders
 // Print all unique words
 System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
orders

 // Print all indexes
 System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
 System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
 System.out.println("All = " + multiMap.get("All!"));    // print [3]
 System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
 System.out.println("Empty = " + multiMap.get("Empty"));    // print []

 // Print count all words
 System.out.println(multiMap.size());    //print 6

 // Print count unique words
 System.out.println(multiMap.keySet().size());    //print 4
```

Примеры Nore:

I. Коллекция Apache:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. Коллекция GS / Eclipse

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. гуайява

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Сравнить операции с коллекциями - Создание коллекций

Сравнить операции с коллекциями - Создание коллекций

1. Создать список

Описание	JDK	гуайява	GS-коллекции
Создать пустой список	`new ArrayList<> ()`	`Lists.newArrayList()`	`FastList.newList()`
Создать список из значений	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith( "2", "3")`
Создать список с емкостью = 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Создать список из любого собрания	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(coll`
Создать список из любого Итерабельного	-	`Lists.newArrayList(iterable)`	`FastList.newList(iter`
Создать список из Iterator	-	`Lists.newArrayList(iterator)`	-
Создать список из массива	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(`
Создать список, используя заводские	-	-	`FastList.newWithNValu () -> "1")`

Примеры:

```
System.out.println("createArrayList start");
// Create empty list
List<String> emptyGuava = Lists.newArrayList(); // using guava
List<String> emptyJDK = new ArrayList<>(); // using JDK
MutableList<String>  emptyGS = FastList.newList(); // using gs

// Create list with 100 element
List < String > exactly100 = Lists.newArrayListWithCapacity(100); // using guava
List<String> exactly100JDK = new ArrayList<>(100); // using JDK
```

```
    MutableList<String>  empty100GS = FastList.newList(100); // using gs

    // Create list with about 100 element
    List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // using guava
    List<String> approx100JDK = new ArrayList<>(115); // using JDK
    MutableList<String>  approx100GS = FastList.newList(115); // using gs

    // Create list with some elements
    List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // using guava
    List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // using JDK
    MutableList<String>  withElementsGS = FastList.newListWith("alpha", "beta", "gamma"); //
using gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create list from any Iterable interface (any collection)
    Collection<String> collection = new HashSet<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    List<String> fromIterable = Lists.newArrayList(collection); // using guava
    List<String> fromIterableJDK = new ArrayList<>(collection); // using JDK
    MutableList<String>  fromIterableGS = FastList.newList(collection); // using gs

    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create list only from Collection, but guava and gs can create list from
Iterable and Collection */

    // Create list from any Iterator
    Iterator<String> iterator = collection.iterator();
    List<String> fromIterator = Lists.newArrayList(iterator); // using guava
    System.out.println(fromIterator);

    // Create list from any array
    String[] array = {"4", "5", "6"};
    List<String> fromArray = Lists.newArrayList(array); // using guava
    List<String> fromArrayJDK = Arrays.asList(array); // using JDK
    MutableList<String>  fromArrayGS = FastList.newListWith(array); // using gs
    System.out.println(fromArray);
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    // Create list using fabric
    MutableList<String>  fromFabricGS = FastList.newWithNValues(10, () ->
String.valueOf(Math.random())); // using gs
    System.out.println(fromFabricGS);

    System.out.println("createArrayList end");
```

2 Создать набор

Описание	JDK	гуайява	GS-коллекции
Создать пустой набор	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`

Описание	JDK	гуайява	GS-коллекции
Творческий набор из значений	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma") )`	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWit "beta", "gamma")`
Создать набор из любых коллекций	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(co`
Создать набор из любого Итерабельного	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(it`
Создать набор из любого Итератора	-	`Sets.newHashSet(iterator)`	-
Создать набор из массива	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWit`

Примеры:

```
    System.out.println("createHashSet start");
    // Create empty set
    Set<String> emptyGuava = Sets.newHashSet(); // using guava
    Set<String> emptyJDK = new HashSet<>(); // using JDK
    Set<String> emptyGS = UnifiedSet.newSet(); // using gs

    // Create set with 100 element
    Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // using guava
    Set<String> approx100JDK = new HashSet<>(130); // using JDK
    Set<String> approx100GS = UnifiedSet.newSet(130); // using gs

    // Create set from some elements
    Set<String> withElements =  Sets.newHashSet("alpha", "beta", "gamma"); // using guava
    Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); //
using JDK
    Set<String> withElementsGS  = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // using gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create set from any Iterable interface (any collection)
    Collection<String> collection = new ArrayList<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    Set<String> fromIterable = Sets.newHashSet(collection); // using guava
    Set<String> fromIterableJDK = new HashSet<>(collection); // using JDK
    Set<String> fromIterableGS  = UnifiedSet.newSet(collection); // using gs
```

```
    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create set only from Collection, but guava and gs can create set from
Iterable and Collection */

    // Create set from any Iterator
    Iterator<String> iterator = collection.iterator();
    Set<String> fromIterator = Sets.newHashSet(iterator); // using guava
    System.out.println(fromIterator);

    // Create set from any array
    String[] array = {"4", "5", "6"};
    Set<String> fromArray = Sets.newHashSet(array); // using guava
    Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // using JDK
    Set<String> fromArrayGS  = UnifiedSet.newSetWith(array); // using gs
    System.out.println(fromArray);
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    System.out.println("createHashSet end");
```

3 Создать карту

Описание	JDK	гуайява	GS-коллекции
Создать пустую карту	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Создать карту с емкостью = 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Создать карту с другой карты	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`
Создать карту из ключей	-	-	`UnifiedMap.newWithKeysValues("1", "a", "2", "b")`

Примеры:

```
    System.out.println("createHashMap start");
    // Create empty map
    Map<String, String> emptyGuava = Maps.newHashMap(); // using guava
    Map<String, String> emptyJDK = new HashMap<>(); // using JDK
    Map<String, String> emptyGS = UnifiedMap.newMap(); // using gs

    // Create map with about 100 element
```

```
    Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // using guava
    Map<String, String> approx100JDK = new HashMap<>(130); // using JDK
    Map<String, String> approx100GS = UnifiedMap.newMap(130); // using gs

    // Create map from another map
    Map<String, String> map = new HashMap<>(3);
    map.put("k1","v1");
    map.put("k2","v2");
    Map<String, String> withMap =  Maps.newHashMap(map); // using guava
    Map<String, String> withMapJDK = new HashMap<>(map); // using JDK
    Map<String, String> withMapGS = UnifiedMap.newMap(map); // using gs

    System.out.println(withMap);
    System.out.println(withMapJDK);
    System.out.println(withMapGS);

    // Create map from keys
    Map<String, String> withKeys =  UnifiedMap.newWithKeysValues("1", "a", "2", "b");
    System.out.println(withKeys);

    System.out.println("createHashMap end");
```

Дополнительные примеры: CreateCollectionTest

1. CollectionCompare
2. CollectionSearch
3. JavaTransform

Прочитайте Альтернативные коллекции онлайн: https://riptutorial.com/ru/java/topic/2958/альтернативные-коллекции

глава 46: Анализ XML с использованием API JAXP

замечания

XML Parsing - это интерпретация XML-документов, чтобы манипулировать их контентом с помощью разумных конструкций, будь то «узлы», «атрибуты», «документы», «пространства имен» или события, связанные с этими конструкциями.

Java имеет собственный API для обработки документов XML, называемый JAXP или Java API для обработки XML . JAXP и эталонная реализация были включены в каждую версию Java после Java 1.4 (JAXP v1.1) и с тех пор развиваются. Java 8 поставляется с JAXP версии 1.6.

API предоставляет различные способы взаимодействия с XML-документами, которые:

- Интерфейс DOM (Document Object Model)
- Интерфейс SAX (простой API для XML)
- Интерфейс StAX (Streaming API для XML)

Принципы интерфейса DOM

Интерфейс DOM предназначен для предоставления W3C DOM- совместимого способа интерпретации XML. Различные версии JAXP поддерживают различные спецификации DOM (до уровня 3).

В интерфейсе Document Object Model документ XML представлен как дерево, начиная с «Элемента документа». Базовый типом API является `Node` типа, это позволяет перемещаться от `Node` к его родителям, его детям, или его братьям (хотя, не все `Node` с может иметь детей, например, `Text` узлы являются окончательными в дереве, и никогда не имеют детей). Теги XML представлены как `Element` s, которые значительно расширяют `Node` с помощью связанных с атрибутами методов.

Интерфейс DOM очень полезен, поскольку он позволяет «одну строку» анализировать XML-документы как деревья и позволяет легко модифицировать построенное дерево (добавление узла, подавление, копирование и т. Д.) И, наконец, его сериализацию (обратно на диск) после изменений. Это происходит по цене, однако: дерево находится в памяти, поэтому деревья DOM не всегда практичны для огромных XML-документов. Кроме того, построение дерева не всегда является самым быстрым способом работы с XML-контентом, особенно если его не интересуют все части документа XML.

Принципы интерфейса SAX

SAX API - это ориентированный на события API для работы с документами XML. В рамках этой модели компоненты XML-документов интерпретируются как события (например, «открыт тег», «тег закрыт», «встречен текстовый узел», «был встречен комментарий»). ..

API SAX использует подход «синтаксический разбор», где SAX `Parser` отвечает за интерпретацию XML-документа и вызывает методы для делегата (`ContentHandler`) для обработки любого события, которое встречается в документе XML. Обычно один никогда не пишет парсер, но один обеспечивает обработчик для сбора всей необходимой информации из XML-документа.

Интерфейс SAX преодолевает ограничения интерфейса DOM, сохраняя только минимально необходимые данные на уровне анализатора (например, контексты пространств имен, состояние проверки), поэтому только информация, `ContentHandler` для которой вы, разработчик, несет ответственность, - это хранится в памяти. Компромисс заключается в том, что при таком подходе нет возможности «вернуться во времени / XML-документ»: в то время как DOM позволяет `Node` возвращаться к его родительскому объекту, такой возможности нет в SAX.

Принципы интерфейса StAX

API StAX использует аналогичный подход для обработки XML как API SAX (т. Е. Управляемый событиями), единственное очень существенное отличие состоит в том, что StAX является синтаксическим анализатором (где SAX был парсером push). В SAX `Parser` находится под контролем и использует обратные вызовы в `ContentHandler` . В Stax вы вызываете парсер и управляете, когда / если вы хотите получить следующее «событие» XML.

API начинается с XMLStreamReader (или XMLEventReader), которые являются шлюзами, через которые разработчик может спросить `nextEvent()` в стиле итератора.

Examples

Анализ и перемещение документа с использованием DOM API

Учитывая следующий документ:

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
</library>
```

Для построения дерева DOM из `String` можно использовать следующий код:

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
            + "<book id='1'>Effective Java</book>"
            + "<book id='2'>Java Concurrency In Practice</book>"
            + "</library>";

    DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
    // This is useless here, because the XML does not have namespaces, but this option is
usefull to know in cas
    documentBuilderFactory.setNamespaceAware(true);
    DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
    // There are various options here, to read from an InputStream, from a file, ...
    Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));

    // Root of the document
    System.out.println("Root of the XML Document: " +
document.getDocumentElement().getLocalName());

    // Iterate the contents
    NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();
    for (int i = 0; i < firstLevelChildren.getLength(); i++) {
        Node item = firstLevelChildren.item(i);
        System.out.println("First level child found, XML tag name is: " +
item.getLocalName());
        System.out.println("\tid attribute of this tag is : " +
item.getAttributes().getNamedItem("id").getTextContent());
    }

    // Another way would have been
    NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");
}
}
```

Код дает следующее:

```
Root of the XML Document: library
First level child found, XML tag name is: book
id attribute of this tag is : 1
First level child found, XML tag name is: book
id attribute of this tag is : 2
```

Разбор документа с использованием API StAX

Учитывая следующий документ:

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
   <notABook id='3'>This is not a book element</notABook>
</library>
```

Можно использовать следующий код для его анализа и построения карты названий книг по идентификатору книги.

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
                + "<book id='1'>Effective Java</book>"
                + "<book id='2'>Java Concurrency In Practice</book>"
                + "<notABook id='3'>This is not a book element </notABook>"
            + "</library>";

    XMLInputFactory xmlInputFactory = XMLInputFactory.newFactory();
    // Various flavors are possible, e.g. from an InputStream, a Source, ...
    XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

    Map<Integer, String> bookTitlesById = new HashMap<>();

    // We go through each event using a loop
    while (xmlStreamReader.hasNext()) {
        switch (xmlStreamReader.getEventType()) {
            case XMLStreamConstants.START_ELEMENT:
                System.out.println("Found start of element: " +
xmlStreamReader.getLocalName());
                // Check if we are at the start of a <book> element
                if ("book".equals(xmlStreamReader.getLocalName())) {
                    int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("",
"id"));
                    String bookTitle = xmlStreamReader.getElementText();
                    bookTitlesById.put(bookId, bookTitle);
                }
                break;
            // A bunch of other things are possible : comments, processing instructions,
Whitespace...
            default:
                break;
        }
        xmlStreamReader.next();
    }

    System.out.println(bookTitlesById);
```

```
}
```

Эти результаты:

```
Found start of element: library
Found start of element: book
Found start of element: book
Found start of element: notABook
{1=Effective Java, 2=Java Concurrency In Practice}
```

В этом примере нужно позаботиться о нескольких вещах:

1. Использование `xmlStreamReader.getAttributeValue` работает, потому что мы сначала отметили, что парсер находится в состоянии `START_ELEMENT` . В других состояниях (кроме `ATTRIBUTES`) парсеру поручено `IllegalStateException` , поскольку атрибуты могут появляться только в начале элементов.

2. `xmlStreamReader.getTextContent()` же самое касается `xmlStreamReader.getTextContent()` , он работает, потому что мы находимся в `START_ELEMENT` и мы знаем в этом документе, что элемент `<book>` не имеет дочерних узлов, не содержащих текст.

Для более сложных анализов документов (более глубокие, вложенные элементы, ...), хорошая практика «делегировать» парсер под-методам или другим объектам, например, иметь класс или метод `BookParser` и иметь дело с каждым элементом от START_ELEMENT до END_ELEMENT книги XML-тега.

Можно также использовать объект `Stack` для хранения важных данных вверх и вниз по дереву.

Прочитайте Анализ XML с использованием API JAXP онлайн:
https://riptutorial.com/ru/java/topic/3943/анализ-xml-с-использованием-api-jaxp

глава 47: Аннотации

Вступление

В Java аннотация - это форма синтаксических метаданных, которые могут быть добавлены в исходный код Java. Он предоставляет данные о программе, которая не является частью самой программы. Аннотации не оказывают прямого влияния на работу кода, который они комментируют. Классы, методы, переменные, параметры и пакеты могут быть аннотированы.

Синтаксис

- @AnnotationName // «Аннотирование маркера» (без параметров)
- @AnnotationName (someValue) // устанавливает параметр с именем 'value'
- @AnnotationName (param1 = value1) // named parameter
- @AnnotationName (param1 = value1, param2 = value2) // несколько именованных параметров
- @AnnotationName (param1 = {1, 2, 3}) // параметр имени array
- @AnnotationName ({value1}) // массив с одним элементом в качестве параметра с именем 'value'

замечания

Типы параметров

Для параметров могут использоваться только постоянные выражения следующих типов, а также массивы этих типов:

- `String`
- `Class`
- примитивные типы
- Типы перечислений
- Типы аннотаций

Examples

Встроенные аннотации

Стандартная версия Java содержит предопределенные аннотации. Вам не нужно определять их самостоятельно, и вы можете использовать их немедленно. Они позволяют

компилятору разрешить некоторую фундаментальную проверку методов, классов и кода.

@Override

Эта аннотация относится к методу и говорит, что этот метод должен переопределять метод суперкласса или реализовать определение метода абстрактного суперкласса. Если эта аннотация используется с любым другим способом, компилятор выдает ошибку.

Бетонный суперкласс

```
public class Vehicle {
   public void drive() {
        System.out.println("I am driving");
   }
}

class Car extends Vehicle {
    // Fine
    @Override
    public void drive() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Абстрактный класс

```
abstract class Animal  {
   public abstract void makeNoise();
}

class Dog extends Animal {
    // Fine
    @Override
    public void makeNoise() {
        System.out.prinln("Woof");
    }
}
```

Не работает

```
class Logger1 {
    public void log(String logString) {
        System.out.prinln(logString);
    }
}

class Logger2 {
    // This will throw compile-time error. Logger2 is not a subclass of Logger1.
    // log method is not overriding anything
    @Override
    public void log(String logString) {
        System.out.println("Log 2" + logString);
    }
}
```

Основная цель - уловить туманность, где вы думаете, что вы переопределяете метод, но на самом деле определяете новый.

```
class Vehicle {
   public void drive() {
        System.out.println("I am driving");
   }
}

class Car extends Vehicle {
    // Compiler error. "dirve" is not the correct method name to override.
    @Override
    public void dirve() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Обратите внимание, что значение `@Override` со временем изменилось:

- В Java 5 это означало, что аннотированный метод должен был переопределить не абстрактный метод, объявленный в цепочке суперкласса.
- Начиная с Java 6, он *также* выполняется, если аннотированный метод реализует абстрактный метод, объявленный в иерархии классов суперклассов / интерфейсов.

(Иногда это может вызвать проблемы при обратном переносе кода на Java 5.)

@Deprecated

Это означает, что метод устарел. Это может быть несколько причин:

- API является ошибочным и нецелесообразно исправлять,
- использование API, вероятно, приведет к ошибкам,
- API был заменен другим API,
- API устарел,
- API является экспериментальным и подлежит несовместимым изменениям,
- или любую комбинацию вышеуказанного.

Конкретную причину устаревания обычно можно найти в документации API.

Аннотации заставят компилятор испускать ошибку, если вы ее используете. IDE также могут выделить этот метод как-то как устаревший

```
class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // stuff here
```

```
    }

    public void quickThreadSafeMethod() {
        // client code should use this instead
    }
}
```

@SuppressWarnings

Почти во всех случаях, когда компилятор выдает предупреждение, наиболее подходящим действием является устранение причины. В некоторых случаях (например, код Generics, использующий, например, нестандартный код для предварительного генерирования) это может быть невозможно, и лучше запретить эти предупреждения, которые вы ожидаете и не можете исправить, чтобы вы могли более четко видеть неожиданные предупреждения.

Эта аннотация может применяться ко всему классу, методу или строке. В качестве параметра используется категория предупреждения.

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // several methods calling deprecated code here
}

@SuppressWarning("finally")
public boolean checkData() {
    // method calling return from within finally block
}
```

Лучше максимально ограничить объем аннотации, чтобы предотвратить непредвиденные предупреждения. Например, ограничение объема аннотации на одну строку:

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algoritm.slowUnthreadSafeMethod();
// we marked this method deprecated in an example above

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// old library returns, non-generic List containing only integers
```

Предупреждения, поддерживаемые этой аннотацией, могут отличаться от компилятора к компилятору. Только `unchecked` и `deprecation` предупреждения упомянуты в JLS. Непризнанные типы предупреждений будут игнорироваться.

@SafeVarargs

Из-за стирания типа `void method(T... t)` будет преобразован в `void method(Object[] t)` что означает, что компилятор не всегда может проверить, что использование varargs является безопасным по типу. Например:

```
private static <T> void generatesVarargsWarning(T... lists) {
```

Существуют случаи, когда использование безопасно, и в этом случае вы можете аннотировать метод с `SafeVarargs` аннотации `SafeVarargs` для подавления предупреждения. Это явно скрывает предупреждение, если ваше использование также небезопасно.

@FunctionalInterface

Это необязательная аннотация, используемая для обозначения FunctionalInterface. Это заставит компилятор жаловаться, если он не соответствует спецификации FunctionalInterface (имеет один абстрактный метод)

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Проверка аннотации выполнения через отражение

API Reflection Java позволяет программисту выполнять различные проверки и операции над полями, методами и аннотациями классов во время выполнения. Однако для того, чтобы аннотация была видимой во время выполнения, `RetentionPolicy` необходимо изменить на `RUNTIME` , как показано в следующем примере:

```
@interface MyDefaultAnnotation {

}

@Retention(RetentionPolicy.RUNTIME)
@interface MyRuntimeVisibleAnnotation {

}

public class AnnotationAtRuntimeTest {

    @MyDefaultAnnotation
    static class RuntimeCheck1 {
    }

    @MyRuntimeVisibleAnnotation
    static class RuntimeCheck2 {
    }

    public static void main(String[] args) {
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();

        System.out.println("default retention: " + Arrays.toString(annotationsByType));
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));
    }
}
```

Определение типов аннотаций

Типы аннотаций определяются с помощью `@interface` . Параметры определяются аналогично методам регулярного интерфейса.

```
@interface MyAnnotation {
    String param1();
    boolean param2();
    int[] param3();  // array parameter
}
```

Значения по умолчанию

```
@interface MyAnnotation {
    String param1() default "someValue";
    boolean param2() default true;
    int[] param3() default {};
}
```

Мета-аннотаций

Мета-аннотации - это аннотации, которые могут применяться к типам аннотаций. Специальная предопределенная мета-аннотация определяет, как можно использовать типы аннотаций.

@Target

Мета-аннотация `@Target` ограничивает типы, к которым может применяться аннотация.

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
    // this annotation can only be applied to methods
}
```

Несколько значений могут быть добавлены с использованием нотации массива, например `@Target({ElementType.FIELD, ElementType.TYPE})`

Доступные значения

ElementType	цель	пример использования целевого элемента
ANNOTATION_TYPE	типы аннотаций	`@Retention(RetentionPolicy.RUNTIME)` `@interface MyAnnotation`

ElementType	цель	пример использования целевого элемента
КОНСТРУКТОР	конструкторы	`@MyAnnotation public MyClass() {}`
Область	поля, константы перечисления	`@XmlAttribute private int count;`
LOCAL_VARIABLE	объявления переменных внутри методов	`for (@LoopVariable int i = 0; i < 100; i++) { @Unused String resultVariable; }`
ПАКЕТ	пакет (в `package-info.java`)	`@Deprecated package very.old;`
МЕТОД	методы	`@XmlElement public int getCount() {...}`
ПАРАМЕТР	параметры метода / конструктора	`public Rectangle( @NamedArg("width") double width, @NamedArg("height") double height) { ... }`
ТИП	классы, интерфейсы, перечисления	`@XmlRootElement public class Report {}`

Java SE 8

ElementType	цель	пример использования целевого элемента
TYPE_PARAMETER	Объявление параметров типа	`public <@MyAnnotation T> void f(T t) {}`

ElementType	цель	пример использования целевого элемента
TYPE_USE	Использование типа	`Object o = "42";` `String s = (@MyAnnotation String) o;`

@Retention

Мета-аннотация `@Retention` определяет видимость аннотации во время процесса или выполнения компиляции приложений. По умолчанию аннотации включены в `.class` файлы, но не отображаются во время выполнения. Чтобы сделать аннотацию доступной во время выполнения, в этой аннотации необходимо установить `RetentionPolicy.RUNTIME` .

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // this annotation can be accessed with reflections at runtime
}
```

Доступные значения

Политика удержания	эффект
УЧЕБНЫЙ КЛАСС	Аннотации доступны в файле `.class` , но не во время выполнения
RUNTIME	Аннотации доступны во время выполнения и могут быть доступны посредством отражения
ИСТОЧНИК	Аннотации доступны во время компиляции, но не добавляются в `.class` . Аннотацию можно использовать, например, обработчиком аннотации.

@Documented

Мета-аннотация `@Documented` используется для обозначения аннотаций, использование которых должно быть документировано генераторами документации API, такими как javadoc . Он не имеет значений. С помощью `@Documented` все классы, использующие аннотацию, будут перечислены на их сгенерированной странице документации. Без `@Documented` невозможно увидеть, какие классы используют аннотацию в документации.

@Inherited

`@Inherited` метаинформация `@Inherited` имеет отношение к аннотациям, которые применяются к классам. Он не имеет значений. Пометка аннотации как `@Inherited` изменяет способ обработки аннотаций.

- Для не унаследованной аннотации запрос рассматривает только исследуемый класс.
- Для унаследованной аннотации запрос также проверяет цепочку суперкласса (рекурсивно) до тех пор, пока не будет найден экземпляр аннотации.

Обратите внимание, что запрашиваются только суперклассы: любые аннотации, привязанные к интерфейсам в иерархии классов, будут игнорироваться.

@Repeatable

`@Repeatable` мета-аннотация `@Repeatable` в Java 8. Она указывает, что к `@Repeatable` аннотации можно добавить несколько экземпляров аннотации. Эта мета-аннотация не имеет значений.

Получение значений аннотации во время выполнения

Вы можете получить текущие свойства аннотации, используя Reflection, чтобы получить метод или поле или класс, к которым применена аннотация, и затем выбор желаемых свойств.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}


class AnnotationExample {
    // Put the Annotation on the method, but leave the defaults
    @MyAnnotation
    public void testDefaults() throws Exception {
        // Using reflection, get the public method "testDefaults", which is this method with
no args
        Method method = AnnotationExample.class.getMethod("testDefaults", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    //Put the Annotation on the method, but override the settings
    @MyAnnotation(key="baz", value="buzz")
    public void testValues() throws Exception {
        // Using reflection, get the public method "testValues", which is this method with no
args
        Method method = AnnotationExample.class.getMethod("testValues", null);
```

```
        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    public void print(MyAnnotation annotation) {
        // Fetch the MyAnnotation 'key' & 'value' properties, and print them out
        System.out.println(annotation.key() + " = " + annotation.value());
    }

    public static void main(String[] args) {
        AnnotationExample example = new AnnotationExample();
        try {
            example.testDefaults();
            example.testValues();
        } catch( Exception e ) {
            // Shouldn't throw any Exceptions
            System.err.println("Exception [" + e.getClass().getName() + "] - " +
e.getMessage());
            e.printStackTrace(System.err);
        }
    }
}
```

Выход будет

```
foo = bar
baz = buzz
```

Повторяющиеся аннотации

До Java 8 два экземпляра одной аннотации не могли быть применены к одному элементу. Стандартное обходное решение заключалось в использовании аннотации контейнера, содержащей массив некоторой другой аннотации:

```
// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
```

```
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Java SE 8

Java 8 обеспечивает более чистый, более прозрачный способ использования аннотаций контейнеров, используя аннотацию `@Repeatable` . Сначала добавим это в класс `Author` :

```
@Repeatable(Authors.class)
```

Это говорит Java обрабатывать несколько аннотаций `@Author` как если бы они были окружены контейнером `@Authors` . Мы также можем использовать `Class.getAnnotationsByType()` для доступа к массиву `@Author` своим собственным классом, а не через его контейнер:

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Унаследованные аннотации

По умолчанию аннотации классов не применяются к типам, расширяющим их. Это можно изменить, добавив аннотацию `@Inherited` в определение аннотации

пример

Рассмотрим следующие 2 аннотации:

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType {
}
```

а также

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType {
}
```

Если три класса аннотируются следующим образом:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B {
}
```

запуск этого кода

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("____________________________________");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

напечатает результат, подобный этому (в зависимости от пакетов аннотации):

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()
____________________________________
@UninheritedAnnotationType()
null
null
```

Обратите внимание, что аннотации могут наследоваться только от классов, а не от интерфейсов.

Обработка времени компиляции с использованием обработчика аннотаций

В этом примере показано, как выполнить проверку времени компиляции аннотированного элемента.

Аннотации

`@Setter` - это маркер, который можно применить к методам. Аннотации будут отброшены во время компиляции, которые впоследствии не будут доступны.

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

Обработчик аннотации

Класс `SetterProcessor` используется компилятором для обработки аннотаций. Он проверяет, если методы аннотированные с `@Setter` аннотаций являются `public` , неправительственные `static` методы с именем , начинающимся с `set` и имеющий заглавную букву как 4 буквы. Если одно из этих условий не выполняется, в `Messager` записывается ошибка. Компилятор записывает это в stderr, но другие инструменты могут использовать эту информацию по-разному. Например, IDE NetBeans позволяет пользователю задавать обработчики аннотаций, которые используются для отображения сообщений об ошибках в редакторе.

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messager messager;

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
{
        // get elements annotated with the @Setter annotation
```

```
        Set<? extends Element> annotatedElements =
roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // only handle methods as targets
                checkMethod((ExecutableElement) element);
            }
        }

        // don't claim annotations to allow other processors to process them
        return false;
    }

    private void checkMethod(ExecutableElement method) {
        // check for valid name
        String name = method.getSimpleName().toString();
        if (!name.startsWith("set")) {
            printError(method, "setter name must start with \"set\"");
        } else if (name.length() == 3) {
            printError(method, "the method name must contain more than just \"set\"");
        } else if (Character.isLowerCase(name.charAt(3))) {
            if (method.getParameters().size() != 1) {
                printError(method, "character following \"set\" must be upper case");
            }
        }

        // check, if setter is public
        if (!method.getModifiers().contains(Modifier.PUBLIC)) {
            printError(method, "setter must be public");
        }

        // check, if method is static
        if (method.getModifiers().contains(Modifier.STATIC)) {
            printError(method, "setter must not be static");
        }
    }

    private void printError(Element element, String message) {
        messager.printMessage(Diagnostic.Kind.ERROR, message, element);
    }

    @Override
    public void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);

        // get messager for printing errors
        messager = processingEnvironment.getMessager();
    }

}
```

упаковка

Для применения компилятором процессор обработки аннотаций должен быть доступен для SPI (см. ServiceLoader).

Для этого нужно добавить текстовый файл `META-INF/services/javax.annotation.processing.Processor` необходимо добавить в файл jar, содержащий процессор аннотации, и аннотацию в дополнение к другим файлам. В файле должно быть указано полное имя обработчика аннотаций, то есть оно должно выглядеть так:

```
annotation.processor.SetterProcessor
```

Мы предположим, что файл jar называется `AnnotationProcessor.jar` ниже.

Пример аннотированного класса

Следующий класс является примером класса в пакете по умолчанию с аннотациями, которые применяются к правильным элементам в соответствии с политикой хранения. Однако только обработчик аннотации рассматривает второй метод как действительную цель аннотации.

```
import annotation.Setter;

public class AnnotationProcessorTest {

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}
```

Использование обработчика аннотации с javac

Если обработчик аннотации обнаружен с использованием SPI, он автоматически используется для обработки аннотированных элементов. Например, компиляция класса `AnnotationProcessorTest` с использованием

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

дает следующий результат

```
AnnotationProcessorTest.java:6: error: setter must be public
    private void setValue(String value) {}
```

```
                  ^
AnnotationProcessorTest.java:12: error: setter name must start with "set"
    public static void main(String[] args) {}
                       ^
2 errors
```

вместо компиляции в обычном режиме. Файл `.class` не создается.

Этого можно предотвратить, указав параметр `-proc:none` для `javac` . Вы также можете отказаться от обычной компиляции, указав `-proc:only` вместо этого.

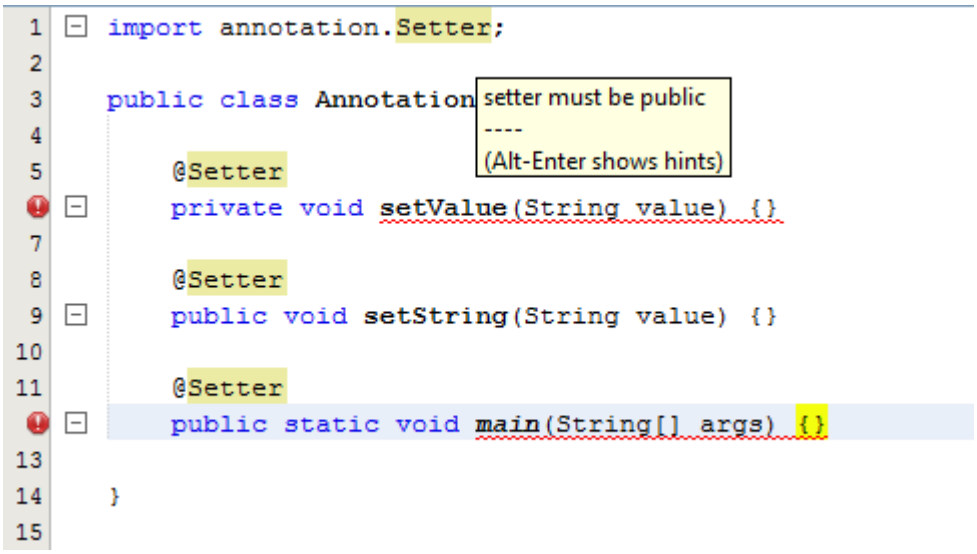
Интеграция IDE

Netbeans

Обработчики аннотаций могут использоваться в редакторе NetBeans. Для этого в настройках проекта необходимо указать процессор аннотации:

1. перейдите в « `Project Properties` > « `Build` > « `Compiling`
2. добавить флажки для `Enable Annotation Processing` и `Enable Annotation Processing in Editor`
3. нажмите « `Add` рядом с списком процессоров аннотаций
4. в появившемся всплывающем окне введите полное имя класса обработчика аннотации и нажмите « `Ok` .

Результат



Идея аннотаций

Спецификация языка Java описывает аннотации следующим образом:

> Аннотирование - это маркер, который связывает информацию с конструкцией программы, но не влияет на время выполнения.

Аннотации могут отображаться перед типами или объявлениями. Они могут появляться в месте, где они могут применяться как к типу, так и к объявлению.
То, к чему относится аннотация, регулируется «мета-аннотацией» `@Target` .
Дополнительную информацию см. В разделе «Определение типов аннотаций» .

Аннотации используются для множества целей. Структуры, такие как Spring и Spring-MVC, используют аннотации для определения того, где должны быть введены зависимости или где должны быть маршрутизированы запросы.

Другие фреймворки используют аннотации для генерации кода. Ломбок и JPA - яркие примеры, которые используют аннотации для генерации кода Java (и SQL).

Цель этой темы - предоставить полный обзор:

- Как определить свои собственные аннотации?
- Какие аннотации предоставляет Java-язык?
- Как используются аннотации на практике?

Аннотации для параметров «этого» и приемника

Когда впервые были введены аннотации Java, не было никаких условий для аннотирования цели метода экземпляра или параметра скрытого конструктора для конструктора внутренних классов. Это было исправлено на Java 8 с добавлением объявлений *параметров приемника* ; см. JLS 8.4.1 .

> Параметр получателя является необязательным синтаксическим устройством для метода экземпляра или конструктора внутреннего класса. Для метода экземпляра параметр приемника представляет объект, для которого вызывается метод. Для конструктора внутреннего класса параметр-приемник представляет собой немедленно включающий экземпляр вновь созданного объекта. В любом случае параметр приемника существует только для того, чтобы разрешить тип отображаемого объекта в исходном коде, чтобы тип мог быть аннотирован. Параметр приемника не является формальным параметром; точнее, это не объявление какой-либо переменной (§4.12.3), оно никогда не связано ни с каким значением, переданным в качестве аргумента в выражении вызова метода или в выражении для создания экземпляра класса, и оно не оказывает никакого влияния на время выполнения.

Следующий пример иллюстрирует синтаксис для обоих типов параметров приемника:

```
public class Outer {
    public class Inner {
        public Inner (Outer this) {
           // ...
        }
        public void doIt(Inner this) {
           // ...
        }
    }
}
```

Единственная цель параметров приемника - дать вам возможность добавлять аннотации. Например, у вас может быть пользовательская аннотация `@IsOpen` , целью которой является утверждение, что объект `Closeable` не был закрыт при вызове метода. Например:

```
public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }

    public void close() {
        // ...
    }
}
```

На одном уровне аннотация `@IsOpen` на `this` может просто служить документацией. Однако мы могли бы сделать больше. Например:

- Обработчик аннотации может вставить проверку времени выполнения, что `this` не в закрытом состоянии при вызове `update` .
- Средство проверки кода может выполнять статический анализ кода, чтобы найти случаи, когда `this` *может быть* закрыто при вызове `update` .

Добавить несколько значений аннотации

Параметр Annotation может принимать несколько значений, если он определен как массив. Например, стандартная аннотация `@SuppressWarnings` определяется следующим образом:

```
public @interface SuppressWarnings {
    String[] value();
}
```

Параметр `value` представляет собой массив строк. Вы можете установить несколько значений, используя нотацию, похожую на инициализаторы массива:

```
@SuppressWarnings({"unused"})
@SuppressWarnings({"unused", "javadoc"})
```

Если вам нужно только установить одно значение, скобки можно опустить:

```
@SuppressWarnings("unused")
```

Прочитайте Аннотации онлайн: https://riptutorial.com/ru/java/topic/157/аннотации

глава 48: Апплеты

Вступление

Апплеты были частью Java с момента его официального выпуска и были использованы для обучения Java и программирования в течение ряда лет.

В последние годы наблюдается активный толчок к удалению от апплетов и других плагинов браузеров, причем некоторые браузеры блокируют их или активно не поддерживают их.

В 2016 году Oracle объявила о своих планах отказаться от плагина, перейдя в плагиновую сеть

Теперь доступны новые и лучшие API-интерфейсы

замечания

Апплет - это приложение Java, которое обычно запускается внутри веб-браузера. Основная идея заключается в том, чтобы взаимодействовать с пользователем без необходимости взаимодействия с сервером и передачи информации. Эта концепция была очень успешной в 2000 году, когда интернет-общение было медленным и дорогостоящим.

Апплет предлагает пять методов контроля жизненного цикла.

имя метода	описание
`init()`	вызывается один раз при загрузке апплета
`destroy()`	вызывается один раз, когда апплет удаляется из памяти
`start()`	вызывается всякий раз, когда апплет становится видимым
`stop()`	вызывается всякий раз, когда апплет накладывается другими окнами
`paint()`	вызывается при необходимости или вручную запускается вызовом `repaint()`

Examples

Минимальный апплет

Очень простой апплет рисует прямоугольник и печатает строку на экране.

```
public class MyApplet extends JApplet{

    private String str = "StackOverflow";

    @Override
    public void init() {
        setBackground(Color.gray);
    }
    @Override
    public void destroy() {}
    @Override
    public void start() {}
    @Override
    public void stop() {}
    @Override
    public void paint(Graphics g) {
        g.setColor(Color.yellow);
        g.fillRect(1,1,300,150);
        g.setColor(Color.red);
        g.setFont(new Font("TimesRoman", Font.PLAIN, 48));
        g.drawString(str, 10, 80);
    }
}
```

Основной класс апплета простирается от `javax.swing.JApplet` .

Java SE 1.2

До появления Java 1.2 и внедрения апплетов swing API были расширены из `java.applet.Applet` .

Апплеты не требуют основного метода. Точка входа контролируется жизненным циклом. Чтобы использовать их, они должны быть встроены в HTML-документ. Это также точка, в которой определяется их размер.

```
<html>
  <head></head>
  <body>
     <applet code="MyApplet.class" width="400" height="200"></applet>
  </body>
</html>
```

Создание графического интерфейса

Апплеры могут быть легко использованы для создания графического интерфейса. Они действуют как `Container` и имеют метод `add()` который принимает любой компонент `awt` или `swing` .

```
public class MyGUIApplet extends JApplet{

    private JPanel panel;
```

```
    private JButton button;
    private JComboBox<String> cmbBox;
    private JTextField textField;

    @Override
    public void init(){
        panel = new JPanel();
        button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                if(((String)cmbBox.getSelectedItem()).equals("greet")) {
                    JOptionPane.showMessageDialog(null,"Hello " + textField.getText());
                } else {
                    JOptionPane.showMessageDialog(null,textField.getText() + " stinks!");
                }
            }
        });
        cmbBox = new JComboBox<>(new String[]{"greet", "offend"});
        textField = new JTextField("John Doe");
        panel.add(cmbBox);
        panel.add(textField);
        panel.add(button);
        add(panel);
    }
}
```

Открытые ссылки из апплета

Вы можете использовать метод `getAppletContext()` чтобы получить объект `AppletContext` который позволяет запросить браузер, чтобы открыть ссылку. Для этого вы используете метод `showDocument()` . Второй параметр указывает браузеру использовать новое окно `_blank` или тот, который показывает апплет `_self` .

```
public class MyLinkApplet extends JApplet{
    @Override
    public void init(){
        JButton button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                AppletContext a = getAppletContext();
                try {
                    URL url = new URL("http://stackoverflow.com/");
                    a.showDocument(url,"_blank");
                } catch (Exception e) { /* omitted for brevity */ }
            }
        });
        add(button);
    }
}
```

Загрузка изображений, аудио и других ресурсов

Java-апплеты могут загружать разные ресурсы. Но поскольку они работают в веб-браузере

клиента, вам необходимо убедиться, что эти ресурсы доступны. Апплеты не могут обращаться к клиентским ресурсам как к локальной файловой системе.

Если вы хотите загружать ресурсы с одного и того же URL-адреса, Applet хранится, вы можете использовать метод `getCodeBase()` для извлечения базового URL-адреса. Для загрузки ресурсов апплеты предлагают методы `getImage()` и `getAudioClip()` для загрузки изображений или аудиофайлов.

Загрузите и покажите изображение

```
public class MyImgApplet extends JApplet{

    private Image img;

    @Override
    public void init(){
        try {
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/so-
logo.png"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
        public void paint(Graphics g) {
            g.drawImage(img, 0, 0, this);
        }
}
```

Загрузка и воспроизведение аудиофайла

```
public class MyAudioApplet extends JApplet{

    private AudioClip audioClip;

    @Override
    public void init(){
        try {
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
    public void start() {
        audioClip.play();
    }
    @Override
    public void stop(){
        audioClip.stop();
    }
}
```

Загрузка и отображение текстового файла

```
public class MyTextApplet extends JApplet{
    @Override
    public void init(){
        JTextArea textArea = new JTextArea();
        JScrollPane sp = new JScrollPane(textArea);
        add(sp);
        // load text
        try {
            URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
            InputStream in = url.openStream();
            BufferedReader bf = new BufferedReader(new InputStreamReader(in));
            String line = "";
            while((line = bf.readLine()) != null) {
                textArea.append(line + "\n");
            }
        } catch(Exception e) { /* omitted for brevity */ }
    }
}
```

Прочитайте Апплеты онлайн: https://riptutorial.com/ru/java/topic/5503/апплеты

глава 49: Атомные типы

Вступление

Java Atomic Types - это простые переменные типы, которые обеспечивают основные операции, которые являются потокобезопасными и атомными, не прибегая к блокировке. Они предназначены для использования в тех случаях, когда блокировка является узким местом параллелизма или существует риск взаимоблокировки или оживления.

параметры

параметр	Описание
задавать	Неустойчивый набор полей
получить	Неустойчивое чтение поля
lazySet	Это упорядоченная операция в полевых условиях
compareAndSet	Если значение представляет собой значение expeed, оно отправляется на новое значение
getAndSet	получить текущее значение и обновить

замечания

Многие из них по существу сочетают волатильные чтения или записи и операции CAS . Лучший способ понять это - посмотреть исходный код напрямую. Например, AtomicInteger , Unsafe.getAndSet

Examples

Создание атомных типов

Для простого многопоточного кода приемлема синхронизация . Однако использование синхронизации имеет сильное влияние, и по мере того, как кодовая база становится более сложной, вероятность возрастает, и в конечном итоге вы столкнетесь с Deadlock , Starvation или Livelock .

В случаях более сложного параллелизма использование Atomic Variables часто является лучшей альтернативой, так как позволяет доступ к отдельной переменной поточно-

безопасным образом без накладных расходов на использование синхронизированных методов или кодовых блоков.

Создание типа `AtomicInteger` :

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Аналогично для других типов экземпляров.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

Аналогично для других типов атомов.

Есть заметное исключение, что нет типов `float` и `double` . Их можно моделировать с помощью `Float.floatToIntBits(float)` и `Float.intBitsToFloat(int)` для `float` а также `Double.doubleToLongBits(double)` и `Double.longBitsToDouble(long)` для удвоений.

Если вы хотите использовать `sun.misc.Unsafe` вы можете использовать любую примитивную переменную в качестве атома, используя атомную операцию в `sun.misc.Unsafe` . Все примитивные типы должны быть преобразованы или закодированы в int или longs, чтобы таким образом использовать его. Подробнее об этом см .: sun.misc.Unsafe .

Мотивация для атомных типов

Простым способом реализации многопоточных приложений является использование встроенных синхронизирующих и блокирующих примитивов Java; например `synchronized` ключевое слово. В следующем примере показано, как мы можем использовать `synchronized` для накопления счетчиков.

```
public class Counters {
    private final int[] counters;

    public Counters(int nosCounters) {
        counters = new int[nosCounters];
    }

    /**
     * Increments the integer at the given index
     */
    public synchronized void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number]++;
        }
    }

    /**
     * Obtains the current count of the number at the given index,
```

```
     * or if there is no number at that index, returns 0.
     */
    public synchronized int getCount(int number) {
        return (number >= 0 && number < counters.length) ? counters[number] : 0;
    }
}
```

Эта реализация будет работать правильно. Однако, если у вас есть большое количество потоков, делающих много одновременных вызовов на одном и том же объекте `Counters` , синхронизация может быть узким местом. В частности:

1. Каждый вызов `synchronized` метода начинается с текущего потока, который получает блокировку для экземпляра `Counters` .
2. Поток будет удерживать блокировку, пока он проверяет значение `number` и обновляет счетчик.
3. Наконец, он освободит блокировку, позволяя другим потокам получить доступ.

Если один поток пытается захватить блокировку, а другой удерживает ее, то попытка попытки будет заблокирована (остановлена) на шаге 1 до тех пор, пока блокировка не будет отпущена. Если несколько потоков ждут, один из них получит его, а остальные будут заблокированы.

Это может привести к возникновению нескольких проблем:

- Если для блокировки много *споров* (т. Е. Много потоков пытаются ее приобрести), то некоторые потоки могут быть заблокированы в течение длительного времени.
- Когда поток блокируется в ожидании блокировки, операционная система, как правило, пытается переключиться на другой поток. Такое *переключение контекста* оказывает относительно большое влияние на производительность процессора.
- Когда есть несколько потоков, заблокированных на одном замке, нет никаких гарантий, что любой из них будет обрабатываться «честно» (т. Е. Каждый поток, как гарантируется, планируется запустить). Это может привести к *голоданию нитей* .

Как реализовать Atomic Types?

Начнем с перезаписи приведенного выше примера с `AtomicInteger` счетчиков `AtomicInteger` :

```
public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
    }
```

```
    /**
     * Increments the integer at the given index
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * Obtains the current count of the object at the given index,
     * or if there is no number at that index, returns 0.
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
                counters[number].get() : 0;
    }
}
```

Мы заменили `int[]` на `AtomicInteger[]` и инициализировали его экземпляром в каждом элементе. Мы также добавили вызовы `incrementAndGet()` и `get()` вместо операций над значениями `int` .

Но самое главное, что мы можем удалить `synchronized` ключевое слово, потому что блокировка больше не требуется. Это работает, потому что операции `incrementAndGet()` и `get()` являются *атомарными* и *потокобезопасными* . В этом контексте это означает, что:

- Каждый счетчик в массиве будет *наблюдаться* только в состоянии «перед» для операции (например, «приращение») или в состоянии «после».
- Предполагая, что операция происходит в момент времени `T` , нить не сможет увидеть состояние «раньше» после времени `T`

Кроме того, хотя два потока могут фактически попытаться обновить один и тот же экземпляр `AtomicInteger` одновременно, реализации операций гарантируют, что только одно приращение происходит одновременно на данном экземпляре. Это делается без блокировки, что часто приводит к повышению производительности.

Как работают Atomic Types?

Атомные типы обычно полагаются на специализированные аппаратные команды в наборе команд целевой машины. Например, наборы инструкций на базе Intel предоставляют инструкцию `CAS` (Compare and Swap), которая будет выполнять определенную последовательность операций с памятью атомарно.

Эти низкоуровневые инструкции используются для реализации операций более высокого уровня в API соответствующих классов `AtomicXxx` . Например, (опять же, в C-подобном псевдокоде):

```
private volatile num;

int increment() {
  while (TRUE) {
    int old = num;
    int new = old + 1;
    if (old == compare_and_swap(&num, old, new)) {
      return new;
    }
  }
}
```

Если на `AtomicXxxx` нет споров, тест `if` будет успешным, и цикл завершится немедленно. Если есть конфликт, то `if` будет терпеть неудачу для всех, кроме одного из потоков, и они будут «вращаться» в цикле для небольшого числа циклов цикла. На практике скорость вращения на несколько порядков (за исключением *нереалистично высоких* уровней конкуренции, когда синхронизация работает лучше, чем атомные классы, потому что, когда операция CAS завершается с ошибкой, повтор будет только увеличивать конкуренцию), чем приостановка потока и переход на другой один.

Кстати, инструкции CAS обычно используются JVM для реализации *незащищенной блокировки* . Если JVM может видеть, что блокировка в настоящий момент не заблокирована, она попытается использовать CAS для получения блокировки. Если CAS преуспевает, тогда нет необходимости выполнять дорогостоящее планирование потоков, переключение контекста и так далее. Дополнительные сведения об используемых методах см. В разделе Блокировка смещения в HotSpot .

Прочитайте Атомные типы онлайн: https://riptutorial.com/ru/java/topic/5963/атомные-типы

глава 50: аудио

замечания

Вместо использования javax.sound.sampled `Clip` вы также можете использовать `AudioClip` который находится из API апплета. Тем не менее рекомендуется использовать `Clip` поскольку `AudioClip` только старше и представляет собой ограниченную функциональность.

Examples

Воспроизведение аудиофайла

Необходимый импорт:

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

Этот код будет создавать клип и воспроизводить его непрерывно после запуска:

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

Получите массив со всеми поддерживаемыми типами файлов:

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

Воспроизведение MIDI-файла

Файлы MIDI можно воспроизводить с помощью нескольких классов из пакета `javax.sound.midi` . `Sequencer` выполняет воспроизведение MIDI-файла, и многие его методы могут использоваться для установки элементов управления воспроизведением, таких как подсчет циклов, темп, отключение звука и другие.

Общее воспроизведение MIDI-данных может быть выполнено следующим образом:

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
```

```
    public static void main(String[] args) {
        try {
            Sequencer sequencer = MidiSystem.getSequencer(); // Get the default Sequencer
            if (sequencer==null) {
                System.err.println("Sequencer device not supported");
                return;
            }
            sequencer.open(); // Open device
            // Create sequence, the File must contain MIDI file data.
            Sequence sequence = MidiSystem.getSequence(new File(args[0]));
            sequencer.setSequence(sequence); // load it into sequencer
            sequencer.start();  // start the playback
        } catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Чтобы остановить воспроизведение, используйте:

```
sequencer.stop(); // Stop the playback
```

Секвенсер может быть настроен на отключение одной или нескольких дорожек последовательности во время воспроизведения, поэтому ни один из инструментов в указанных играх не воспроизводится. Следующий пример устанавливает первый трек в последовательности, которая должна быть отключена:

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

Секвенсор может воспроизводить последовательность несколько раз, если задано количество циклов. Следующее устанавливает секвенсер для воспроизведения последовательности четыре раза и бесконечно:

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

Секвенсер не всегда должен играть последовательность с самого начала и не должен играть последовательность до конца. Он может начинаться и заканчиваться в любой момент, указав *галочку* в последовательности, чтобы начать и завершить. Также можно указать вручную, какой тик в последовательности, которую должен играть секвенсор:

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

Секвенсоры также могут играть MIDI-файл с определенным темпом, который можно контролировать, указав темп в битах в минуту (BPM) или микросекундах на четвертную

ноту (MPQ). Можно также скорректировать коэффициент, в котором воспроизводится последовательность.

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);
```

Когда вы закончите использовать `Sequencer` , помните, чтобы закрыть его

```
sequencer.close();
```

Звук из чистого металла

Вы также можете пойти почти голыми металлами при создании звука с помощью java. Этот код будет записывать необработанные двоичные данные в звуковой буфер OS для генерации звука. Чрезвычайно важно понимать ограничения и необходимые вычисления для генерации звука, подобного этому. Поскольку воспроизведение в основном мгновенное, расчеты должны выполняться почти в режиме реального времени.

Таким образом, этот метод непригоден для более сложной выборки звука. Для таких целей использование специализированных инструментов - лучший подход.

Следующий метод генерирует и непосредственно выводит прямоугольную волну заданной частоты в заданном объеме для заданной продолжительности.

```
public void rectangleWave(byte volume, int hertz, int msecs) {
    final SourceDataLine dataLine;
    // 24 kHz x 8bit, single-channel, signed little endian AudioFormat
    AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
    try {
        dataLine = AudioSystem.getSourceDataLine(af);
        dataLine.open(af, 10_000); // audio buffer size: 10k samples
    } catch (LineUnavailableException e) {
        throw new RuntimeException(e);
    }

    int waveHalf = 24_000 / hertz; // samples for half a period
    byte[] buffer = new byte[waveHalf * 20];
    int samples = msecs * (24_000 / 1000); // 24k (samples / sec) / 1000 (ms/sec) * time(ms)

    dataLine.start(); // starts playback
    int sign = 1;

    for (int i = 0; i < samples; i += buffer.length) {
        for (int j = 0; j < 20; j++) { // generate 10 waves into buffer
            sign *= -1;
            // fill from the jth wave-half to the j+1th wave-half with volume
            Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
        }
        dataLine.write(buffer, 0, buffer.length); //
    }
    dataLine.drain(); // forces buffer drain to hardware
    dataLine.stop();  // ends playback
```

```
}
```

Для более дифференцированного способа генерации различных звуковых волн необходимы синусные расчеты и, возможно, больший размер выборки. Это приводит к значительно более сложному коду и соответственно опущено.

Базовый аудиовыход

Привет, аудио! Java, воспроизводящий звуковой файл из локального или интернет-хранилища, выглядит следующим образом. Он работает с несжатыми файлами .wav и не должен использоваться для воспроизведения mp3 или сжатых файлов.

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;

public class SoundClipTest {

   // Constructor
   public SoundClipTest() {
      try {
         // Open an audio input stream.
          File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //you could
also get the sound file with an URL
          AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
          AudioFormat format = audioIn.getFormat();
         // Get a sound clip resource.
         DataLine.Info info = new DataLine.Info(Clip.class, format);
         Clip clip = (Clip)AudioSystem.getLine(info);
         // Open audio clip and load samples from the audio input stream.
         clip.open(audioIn);
         clip.start();
      } catch (UnsupportedAudioFileException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      } catch (LineUnavailableException e) {
         e.printStackTrace();
      }
   }

   public static void main(String[] args) {
      new SoundClipTest();
   }
}
```

Прочитайте аудио онлайн: https://riptutorial.com/ru/java/topic/160/аудио

глава 51: Безопасность и криптография

Examples

Вычислить криптографические хэши

Для вычисления хэшей относительно небольших блоков данных с использованием разных алгоритмов:

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5    hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1   hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256 hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

Производит этот вывод:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Дополнительные алгоритмы могут быть доступны в зависимости от вашей реализации платформы Java.

Создание криптографически случайных данных

Для генерации выборок криптографически случайных данных:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Производит вывод, аналогичный:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Обратите внимание, что вызов `nextBytes()` может блокироваться при `nextBytes()` энтропии в зависимости от используемого алгоритма.

Чтобы указать алгоритм и поставщик:

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");

randomness.nextBytes(sample);

System.out.println("Provider: " + randomness.getProvider());
System.out.println("Algorithm: " + randomness.getAlgorithm());
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Производит вывод, аналогичный:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Создание пар общих / закрытых ключей

Для генерации пар ключей с использованием разных алгоритмов и размеров ключей:

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

Дополнительные алгоритмы и размеры ключей могут быть доступны для вашей реализации платформы Java.

Чтобы указать источник случайности для использования при создании ключей:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Вычислить и проверить цифровые подписи

Чтобы вычислить подпись:

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);
```

```
final byte[] signature = signer.sign();
```

Обратите внимание, что алгоритм подписи должен быть совместим с алгоритмом, используемым для генерации пары ключей.

Чтобы проверить подпись:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

Производит этот вывод:

```
Signature: true
```

Шифрование и расшифровка данных с помощью открытых / закрытых ключей

Для шифрования данных с помощью открытого ключа:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

Производит вывод, аналогичный:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Обратите внимание, что при создании объекта `Cipher` вам нужно указать преобразование, совместимое с типом используемого ключа. (См. Стандартные имена алгоритмов JCA для списка поддерживаемых преобразований.). Для данных шифрования RSA длина `message.getBytes()` должна быть меньше размера ключа. См. Этот SO-ответ для подробностей.

Чтобы расшифровать данные:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
```

```
rsa.update(cipherText);
final String result = new String(rsa.doFinal());

System.out.println("Decrypted: " + result);
```

Производит следующий вывод:

```
Decrypted: Hello
```

Прочитайте Безопасность и криптография онлайн: https://riptutorial.com/ru/java/topic/7529/безопасность-и-криптография

глава 52: Безопасность и криптография

Вступление

Практики безопасности на Java можно разделить на две широкие, неопределенно определенные категории; Безопасность платформы Java и защищенное программирование на Java.

Практики безопасности платформы Java имеют дело с управлением безопасностью и целостностью JVM. Он включает такие темы, как управление поставщиками JCE и политиками безопасности.

Безопасные методы программирования Java касаются лучших способов написания защищенных программ Java. Он включает такие темы, как использование случайных чисел и криптография, а также предотвращение уязвимостей.

замечания

В то время как примеры должны быть четко сформулированы, некоторые темы, которые необходимо охватить, следующие:

1. Концепция / структура поставщика JCE
2. Элемент списка

Examples

JCE

Расширение Java Cryptography Extension (JCE) - это структура, встроенная в JVM, позволяющая разработчикам легко и безопасно использовать криптографию в своих программах. Он делает это, предоставляя простой, портативный интерфейс для программистов, используя систему JCE Providers для безопасного осуществления основных криптографических операций.

Ключи и управление ключами

В то время как JCE обеспечивает криптографические операции и генерирование ключей, разработчик фактически может управлять своими ключами. Здесь необходимо предоставить дополнительную информацию.

Одной из общепринятых рекомендаций по управлению ключами во время выполнения является сохранение их только в `byte` массивов `byte` и никогда не как строки. Это связано с

тем, что строки Java неизменяемы и не могут быть вручную «очищены» или «обнулены» в памяти; в то время как ссылка на строку может быть удалена, точная строка останется в памяти, пока ее сегмент памяти не будет собран и повторно использован для сбора мусора. У злоумышленника будет большое окно, в котором они могут сбросить память программы и легко найти ключ. Напротив, массивы `byte` изменяемы, и их содержимое может быть перезаписано на месте; это хорошая идея «обнулить» ваши ключи, как только они вам больше не понадобятся.

Общие уязвимости Java

Нужен контент

Проблемы с сетью

Нужен контент

Случайность и вы

Нужен контент

Для большинства приложений класс `java.utils.Random` является прекрасным источником «случайных» данных. Если вам нужно выбрать случайный элемент из массива или создать случайную строку или создать временный «уникальный» идентификатор, вероятно, вы должны использовать `Random` .

Однако многие криптографические системы полагаются на случайность для своей безопасности, а случайность, предоставляемая `Random` не имеет достаточно высокого качества. Для любой криптографической операции, требующей случайного ввода, вы должны использовать `SecureRandom` .

Хеширование и проверка

Необходима дополнительная информация.

Криптографическая хеш-функция является членом класса функций с тремя важными свойствами; последовательности, уникальности и необратимости.

Согласованность: при одинаковых данных хеш-функция всегда возвращает одно и то же значение. То есть, если X = Y, f (x) всегда будет равно f (y) для хэш-функции f.

Уникальность: никакие два входа хеш-функции никогда не приведут к одному и тому же выводу. То есть, если X! = Y, f (x)! = F (y), для любых значений X и Y.

Необратимость. Неправдоподобно, если не невозможно, «перевернуть» хэш-функцию. То есть, учитывая только f (X), не должно быть способа найти исходный X, чтобы не

поставить все возможные значения X через функцию f (грубая сила). Не должно быть функции f1, для которой f1 (f (X)) = X.

Во многих функциях отсутствует хотя бы один из этих атрибутов. Например, известно, что MD5 и SHA1 имеют коллизии, т. Е. Два входа, которые имеют одинаковый выход, поэтому им не хватает уникальности. Некоторые функции, которые в настоящее время считаются безопасными, это SHA-256 и SHA-512.

Прочитайте Безопасность и криптография онлайн: https://riptutorial.com/ru/java/topic/9371/безопасность-и-криптография

глава 53: Бит-манипуляция

замечания

- В отличие от C / C ++, Java полностью нейтральна по отношению к базовому аппарату. По умолчанию вы не становитесь большим или маленьким поведением; вы должны явно указать, какое поведение вы хотите.
- Тип `byte` подписан, диапазон от -128 до +127. Чтобы преобразовать значение байта в его беззнаковый эквивалент, замаскируйте его с помощью 0xFF следующим образом: `(b & 0xFF)` .

Examples

Упаковка / распаковка значений в виде фрагментов

Обычно производительность памяти сводит несколько значений в одно примитивное значение. Это может быть полезно для передачи различной информации в одну переменную.

Например, можно упаковать 3 байта - например, цветовой код в RGB - в один int.

Упаковка значений

```
// Raw bytes as input
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// Packed in big endian: x == 0x65FF31
int x = (b[0] & 0xFF) << 16  // Red
      | (b[1] & 0xFF) <<  8  // Green
      | (b[2] & 0xFF) <<  0; // Blue

// Packed in little endian: y == 0x31FF65
int y = (b[0] & 0xFF) <<  0
      | (b[1] & 0xFF) <<  8
      | (b[2] & 0xFF) << 16;
```

Распаковка значений

```
// Raw int32 as input
int x = 0x31FF65;

// Unpacked in big endian: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >>  8),
    (byte)(x & 0xFF)
};
```

```
// Unpacked in little endian: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >>  8),
    (byte)(x >> 16)
};
```

Проверка, настройка, очистка и переключение отдельных битов. Использование длинной битовой маски

Предполагая, что мы хотим изменить бит `n` целочисленного примитива, `i` (байт, короткий, char, int или long):

```
(i & 1 << n) != 0 // checks bit 'n'
i |= 1 << n;      // sets bit 'n' to 1
i &= ~(1 << n);   // sets bit 'n' to 0
i ^= 1 << n;      // toggles the value of bit 'n'
```

Использование long / int / short / byte в качестве битовой маски:

```
public class BitMaskExample {
    private static final long FIRST_BIT = 1L << 0;
    private static final long SECOND_BIT = 1L << 1;
    private static final long THIRD_BIT = 1L << 2;
    private static final long FOURTH_BIT = 1L << 3;
    private static final long FIFTH_BIT = 1L << 4;
    private static final long BIT_55 = 1L << 54;

    public static void main(String[] args) {
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);
    }

    private static void checkBitMask(long bitmask) {
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));
        System.out.println("FOURTh_BIT: " + ((bitmask & FOURTH_BIT) != 0));
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));
    }
}
```

Печать

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTh_BIT: false
FIFTH_BIT: true
BIT_55: true
```

который соответствует этой маске мы прошли в качестве `checkBitMask` параметра: `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` .

Выражая силу 2

Для выражения степени 2 (2 ^ n) целых чисел можно использовать операцию бит-сдвига, которая позволяет явно указать `n` .

Синтаксис в основном:

```
int pow2 = 1<<n;
```

Примеры:

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

Это особенно полезно при определении постоянных значений, которые должны сделать очевидным, что используется сила 2, вместо использования шестнадцатеричных или десятичных значений.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

Простым методом вычисления мощности int 2 будет

```
int pow2(int exp){
    return 1<<exp;
}
```

Проверка того, является ли число мощностью 2

Если целое число `x` равно 2, устанавливается только один бит, тогда как `x-1` имеет все биты, установленные после этого. Например: `4` равно `100` и `3` равно `011` как двоичное число, которое удовлетворяет вышеупомянутому условию. Ноль не равен 2 и должен быть проверен явно.

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

Использование левого и правого сдвига

Предположим, у нас есть три вида разрешений: **READ** , **WRITE** и **EXECUTE** . Каждое

разрешение может варьироваться от 0 до 7. (Предположим, что система с четырьмя битами)

RESOURCE = READ WRITE EXECUTE (12-разрядное число)

RESOURCE = 0100 0110 0101 = 4 6 5 (12-разрядное число)

Как мы можем получить разрешения (12-разрядного номера), установленные выше (12-разрядное число)?

0100 0110 0101

0000 0000 0111 (&)

0000 0000 0101 = 5

Таким образом, мы можем получить разрешения **EXECUTE RESOURCE** . Теперь, что, если мы хотим получить **READ-** разрешения **RESOURCE** ?

0100 0110 0101

0111 0000 0000 (&)

0100 0000 0000 = 1024

Правильно? Вероятно, вы это принимаете? Но разрешения приведены в 1024. Мы хотим получить только разрешения READ для ресурса. Не волнуйтесь, поэтому у нас были операторы смены. Если мы увидим, разрешения READ на 8 бит превысят фактический результат, поэтому, если применить некоторый оператор сдвига, который приведет к разрешению READ до самого правильного результата? Что делать, если мы это сделаем:

0100 0000 0000 >> 8 => 0000 0000 0100 (потому что это положительное число, замененное на 0, если вы не заботитесь о знаке, просто используйте беззнаковый оператор сдвига вправо)

Теперь у нас есть разрешения **READ,** которые **равны** 4.

Теперь, например, нам предоставлены разрешения **READ** , **WRITE** , **EXECUTE** для **RESOURCE** , что мы можем сделать, чтобы сделать разрешения для этого **РЕСУРСА** ?

Давайте сначала рассмотрим пример двоичных разрешений. (Все еще предполагая систему с 4-разрядными номерами)

READ = 0001

WRITE = 0100

EXECUTE = 0110

Если вы думаете, что мы просто сделаем это:

`READ | WRITE | EXECUTE` , вы несколько правы, но не совсем. Смотрите, что произойдет, если мы будем выполнять READ | НАПИСАТЬ | ВЫПОЛНИТЬ

0001 | 0100 | 0110 => 0111

Но разрешения фактически представлены (в нашем примере) как 0001 0100 0110

Итак, чтобы сделать это, мы знаем, что **READ** размещен на 8 бит позади, **WRITE** помещается на 4 бита, а **PERMISSIONS** помещается последним. Система номеров, используемая для разрешений **RESOURCE, на** самом деле составляет 12 бит (в нашем примере). Он может (будет) отличаться в разных системах.

(READ << 8) | (WRITE << 4) | (EXECUTE)

0000 0000 0001 << 8 (READ)

0001 0000 0000 (сдвиг влево на 8 бит)

0000 0000 0100 << 4 (WRITE)

0000 0100 0000 (сдвиг влево на 4 бита)

0000 0000 0001 (ВЫПОЛНИТЬ)

Теперь, если мы добавим результаты вышеперечисленного, это будет нечто подобное;

0001 0000 0000 (READ)

0000 0100 0000 (ЗАПИСЬ)

0000 0000 0001 (ВЫПОЛНИТЬ)

0001 0100 0001 (РАЗРЕШЕНИЯ)

Класс java.util.BitSet

Начиная с версии 1.7 существует класс java.util.BitSet, который обеспечивает простой и удобный интерфейс хранения и манипулирования битами:

```
final BitSet bitSet = new BitSet(8); // by default all bits are unset

IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}

bitSet.set(3); // {0, 2, 3, 4, 6}

bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false
```

```
bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - expands automatically
```

`BitSet` реализует `Clonable` и `Serializable` , а под капотом все значения бит хранятся в `long[] words` поле `long[] words` , которое автоматически расширяется.

Он также поддерживает целые логические операции `and` , `or` , `xor` , `andNot` :

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));
```

Подписанный беззнаковый сдвиг

В Java все примитивы числа подписаны. Например, int всегда представляет значения из [-2 ^ 31 - 1, 2 ^ 31], сохраняя первый бит для подписи значения - 1 для отрицательного значения, 0 для положительного.

Операторы основного сдвига `>>` и `<<` являются операторами-операторами. Они сохранят знак ценности.

Но программисты часто используют номера для хранения *значений без знака* . Для int это означает смещение диапазона до [0, 2 ^ 32 - 1], чтобы иметь в два раза большее значение, чем с подписанным int.

Для тех опытных пользователей бит для знака не имеет смысла. Вот почему Java добавила `>>>` , оператор с левым сдвигом, не считая этого бита знака.

```
                initial value:                4 (                                100)
     signed left-shift: 4 << 1                8 (                               1000)
    signed right-shift: 4 >> 1                2 (                                 10)
 unsigned right-shift: 4 >>> 1                2 (                                 10)
                initial value:               -4 (   11111111111111111111111111111100)
    signed left-shift: -4 << 1                -8 (   11111111111111111111111111111000)
   signed right-shift: -4 >> 1                -2 (   11111111111111111111111111111110)
unsigned right-shift: -4 >>> 1      2147483646 (    1111111111111111111111111111110)
```

Почему нет `<<<` ?

Это исходит из предполагаемого определения сдвига вправо. Когда он заполняет опустошенные места слева, нет никакого решения принять бит за знак. Как следствие, нет необходимости в двух разных операторах.

См. Этот вопрос для более детального ответа.

Прочитайте Бит-манипуляция онлайн: https://riptutorial.com/ru/java/topic/1177/бит-

манипуляция

глава 54: Валюта и деньги

Examples

Добавить пользовательскую валюту

Обязательные JAR-адреса в пути к классам:

- javax.money :money-api:1.0 (JSR354 деньги и валюта api)
- org.javamoney: moneta: 1.0 (Реализация ссылок)
- javax: аннотация-апи: 1.2. (Общие аннотации, используемые для эталонной реализации)

```
// Let's create non-ISO currency, such as bitcoin

// At first, this will throw UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// This happens because bitcoin is unknown to default currency
// providers
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// We will build new currency using CurrencyUnitBuilder provided by org.javamoney.moneta
CurrencyUnit bitcoin = CurrencyUnitBuilder
    .of("BTC", "BtcCurrencyProvider") // Set currency code and currency provider name
    .setDefaultFractionDigits(2)      // Set default fraction digits
    .build(true);                     // Build new currency unit. Here 'true' means
                                      // currency unit is to be registered and
                                      // accessible within default monetary context

// Now BTC is available
System.out.println(Monetary.isCurrencyAvailable("BTC")); // True
```

Прочитайте Валюта и деньги онлайн: https://riptutorial.com/ru/java/topic/8359/валюта-и-деньги

глава 55: Ведение журнала (java.util.logging)

Examples

Использование регистратора по умолчанию

В этом примере показано, как использовать атрибут api по умолчанию.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyClass {

    // retrieve the logger for the current class
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());

    public void foo() {
        LOG.info("A log message");
        LOG.log(Level.INFO, "Another log message");

        LOG.fine("A fine message");

        // logging an exception
        try {
            // code might throw an exception
        } catch (SomeException ex) {
            // log a warning printing "Something went wrong"
            // together with the exception message and stacktrace
            LOG.log(Level.WARNING, "Something went wrong", ex);
        }

        String s = "Hello World!";

        // logging an object
        LOG.log(Level.FINER, "String s: {0}", s);

        // logging several objects
        LOG.log(Level.FINEST, "String s: {0} has length {1}", new Object[]{s, s.length()});
    }

}
```

Уровни регистрации

Java Logging Api имеет 7 уровней . Уровни в порядке убывания:

- `SEVERE` (наибольшее значение)
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`

- `FINER`
- `FINEST` (самое низкое значение)

Уровень по умолчанию - `INFO` (но это зависит от системы и используется виртуальной машиной).

Примечание . Существуют также уровни `OFF` (можно использовать для отключения регистрации) и `ALL` (oposite `OFF`).

Пример кода для этого:

```
import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

        logger.severe("Message logged by SEVERE");
        logger.warning("Message logged by WARNING");
        logger.info("Message logged by INFO");
        logger.config("Message logged by CONFIG");
        logger.fine("Message logged by FINE");
        logger.finer("Message logged by FINER");
        logger.finest("Message logged by FINEST");

        // All of above methods are really just shortcut for
        // public void log(Level level, String msg):
        logger.log(Level.FINEST, "Message logged by FINEST");
    }
}
```

По умолчанию запуск этого класса будет выводить только сообщения с уровнем выше, чем `CONFIG` :

```
Jul 23, 2016 9:16:11 PM LevelsExample main
SEVERE: Message logged by SEVERE
Jul 23, 2016 9:16:11 PM LevelsExample main
WARNING: Message logged by WARNING
Jul 23, 2016 9:16:11 PM LevelsExample main
INFO: Message logged by INFO
```

Регистрация сложных сообщений (эффективно)

Давайте рассмотрим пример регистрации, который вы можете увидеть во многих программах:

```
public class LoggingComplex {

    private static final Logger logger =
        Logger.getLogger(LoggingComplex.class.getName());

    private int total = 50, orders = 20;
    private String username = "Bob";
```

```
    public void takeOrder() {
        // (...) making some stuff
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
        // (...) some other stuff
    }

    // some other methods and calculations
}
```

Вышеприведенный пример выглядит отлично, но многие программисты забывают, что Java VM является стековой машиной. Это означает, что все параметры метода вычисляются **перед** выполнением метода.

Этот факт имеет решающее значение для ведения журнала на Java, особенно для регистрации чего-то на низких уровнях, таких как `FINE` , `FINER` , `FINEST` которые по умолчанию отключены. Давайте рассмотрим байт-код Java для `takeOrder()` .

Результат для `javap -c LoggingComplex.class` выглядит примерно так:

```
public void takeOrder();
    Code:
       0: getstatic     #27 // Field logger:Ljava/util/logging/Logger;
       3: ldc           #45 // String User %s ordered %d things (%d in total)
       5: iconst_3
       6: anewarray     #3  // class java/lang/Object
       9: dup
      10: iconst_0
      11: aload_0
      12: getfield      #40 // Field username:Ljava/lang/String;
      15: aastore
      16: dup
      17: iconst_1
      18: aload_0
      19: getfield      #36 // Field orders:I
      22: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      25: aastore
      26: dup
      27: iconst_2
      28: aload_0
      29: getfield      #34 // Field total:I
      32: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      35: aastore
      36: invokestatic  #53 // Method
java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
      39: invokevirtual #59 // Method java/util/logging/Logger.fine:(Ljava/lang/String;)V
      42: return
```

В строке 39 выполняется фактическое ведение журнала. Вся предыдущая работа (загрузка переменных, создание новых объектов, объединение строк в `format`) может быть зря, если уровень ведения журнала установлен выше, чем `FINE` (и по умолчанию он есть). Такое ведение журнала может быть очень неэффективным, потребляя ненужные ресурсы памяти и процессора.

Вот почему вы должны спросить, включен ли уровень, который вы хотите использовать.

Правильный путь должен быть:

```
public void takeOrder() {
    // making some stuff
    if (logger.isLoggable(Level.FINE)) {
        // no action taken when there's no need for it
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
    }
    // some other stuff
}
```

Поскольку Java 8:

Класс Logger имеет дополнительные методы, которые используют параметр `Supplier<String>` , который может быть просто предоставлен лямбдой:

```
public void takeOrder() {
    // making some stuff
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",
            username, orders, total));
    // some other stuff
}
```

Метод `get()` поставщиков `get()` в этом случае лямбда - вызывается только тогда, когда соответствующий уровень включен, и поэтому конструкция `if` больше не нужна.

Прочитайте Ведение журнала (java.util.logging) онлайн:
https://riptutorial.com/ru/java/topic/2010/ведение-журнала--java-util-logging-

глава 56: Видимость (контроль доступа к членам класса)

Синтаксис

- public type name [= value];
- имя частного типа [= значение];
- имя защищенного типа [= значение];
- имя типа [= значение];
- public class name {
- имя класса {

замечания

Из учебника Java :

Модификаторы уровня доступа определяют, могут ли другие классы использовать конкретное поле или вызвать конкретный метод. Существует два уровня контроля доступа:

- На верхнем уровне - `public` или *package-private* (без явного модификатора).
- На уровне участника - `public` , `private` , `protected` или *пакетно-закрытый* (без явного модификатора).

Класс может быть объявлен `public` модификатором, и в этом случае этот класс будет виден всем классам. Если класс не имеет модификатора (по умолчанию, также известного как *private-package*), он отображается только в пределах его собственного пакета.

На уровне члена вы также можете использовать `public` модификатор или модификатор (*private-package*), как и классы верхнего уровня, и с тем же значением. Для участников есть два дополнительных модификатора доступа: `private` и `protected` . `private` модификатор указывает, что к члену можно получить доступ только в своем классе. `protected` модификатор указывает, что к члену можно получить доступ только в своем собственном пакете (как и в *пакете private*) и, кроме того, подклассе его класса в другом пакете.

В следующей таблице показан доступ к членам, разрешенным каждым модификатором.

Уровни доступа:

Модификатор	Учебный класс	пакет	Подкласс	Мир
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N

Модификатор	Учебный класс	пакет	Подкласс	Мир
нет модификатора	Y	Y	N	N
`private`	Y	N	N	N

Examples

Элементы интерфейса

```
public interface MyInterface {
    public void foo();
    int bar();

    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

Члены интерфейса всегда имеют общедоступную видимость, даже если ключевое слово `public` опущено. Таким образом, как `foo()` , `bar()` , `TEXT` , `ANSWER` , `X` , так и `Y` имеют общую видимость. Тем не менее, доступ по-прежнему может ограничиваться содержащимся интерфейсом - поскольку `MyInterface` имеет общедоступную видимость, к его членам можно получить доступ из любого места, но если у `MyInterface` была видимость пакета, его члены были бы доступны только из одного и того же пакета.

Общественная видимость

Видно для класса, пакета и подкласса.

Давайте посмотрим пример с классом Test.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Теперь попробуем создать экземпляр класса. В этом примере **мы можем** получить доступ к `number` потому что он является `public` .

```
public class Other{
```

```
    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}
```

Частная видимость

`private` видимость позволяет доступ к переменной только для своего класса. Они часто используются **вместе** с `public` геттерами и сеттерами.

```
class SomeClass {
    private int variable;

    public int getVariable() {
        return variable;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}

public class SomeOtherClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();

        // These statement won't compile because SomeClass#variable is private:
        sc.variable = 7;
        System.out.println(sc.variable);

        // Instead, you should use the public getter and setter:
        sc.setVariable(7);
        System.out.println(sc.getVariable());
    }
}
```

Видимость пакета

Без **модификатора** значение по умолчанию - видимость пакета. *Из документации Java* «[видимость пакета] указывает, имеют ли классы в том же пакете, что и класс (независимо от их происхождения), доступ к члену». В этом примере из `javax.swing` ,

```
package javax.swing;
public abstract class JComponent extends Container … {
    …
    static boolean DEBUG_GRAPHICS_LOADED;
    …
}
```

`DebugGraphics` находится в одном пакете, поэтому `DEBUG_GRAPHICS_LOADED` доступен.

```
package javax.swing;
```

```
public class DebugGraphics extends Graphics {
    …
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    …
}
```

В этой статье приводятся некоторые сведения об этой теме.

Защищенная видимость

Защищенная видимость приводит к тому, что этот элемент видим для своего пакета вместе с любым из его подклассов.

В качестве примера:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
}
```

Теперь мы расширим этот класс и попытаемся получить доступ к одному из его `protected` членов.

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

Вы также сможете получить доступ к `protected` члену без его расширения, если вы получаете доступ к нему из одного пакета.

Обратите внимание, что этот модификатор работает только с членами класса, а не с самим классом.

Резюме модификаторов доступа к члену класса

Модификатор доступа	видимость	наследование
Частный	Только класс	Не может быть унаследован
Нет модификатора / пакета	В пакете	Доступно, если подкласс в пакете

Модификатор доступа	видимость	наследование
защищенный	В пакете	Доступно в подклассе
общественного	Везде	Доступно в подклассе

Когда-то был `private protected` (оба ключевых слова сразу), который можно было применить к методам или переменным, чтобы сделать их доступными из подкласса вне пакета, но сделать их закрытыми для классов в этом пакете. Однако это было удалено в выпуске Java 1.0 .

Прочитайте Видимость (контроль доступа к членам класса) онлайн: https://riptutorial.com/ru/java/topic/134/видимость--контроль-доступа-к-членам-класса-

глава 57: Виртуальная машина Java (JVM)

Examples

Это основы.

JVM - это **абстрактная вычислительная машина** или **виртуальная машина,** которая находится в вашей ОЗУ. Он имеет независимую от платформы среду исполнения, которая интерпретирует байт-код Java в собственный машинный код. (Javac - компилятор Java, который компилирует ваш Java-код в Bytecode)

Java-программа будет запущена внутри JVM, которая затем отображается на базовую физическую машину. Это один из инструментов программирования в JDK.

(`Byte code` - это независимый от платформы код, который запускается на каждой платформе, а `Machine code` - это код, специфичный для платформы, который запускается только на определенной платформе, такой как windows или linux, и зависит от исполнения.)

Некоторые из компонентов: -

- Класс Loder - загрузить файл .class в оперативную память.
- Верификатор байтов - проверьте, есть ли в вашем коде какие-либо нарушения ограничения доступа.
- Механизм выполнения - преобразование байтового кода в исполняемый машинный код.
- JIT (как раз вовремя) - JIT является частью JVM, которая используется для повышения производительности JVM. Она будет динамически компилировать или транслировать java-байт-код в собственный машинный код во время выполнения.

(Edited)

Прочитайте Виртуальная машина Java (JVM) онлайн: https://riptutorial.com/ru/java/topic/8110/виртуальная-машина-java--jvm-

глава 58: Виртуальный доступ Java

Examples

Введение в JNA

Что такое JNA?

Java Native Access (JNA) - это разработанная сообществом библиотека, предоставляющая Java-программам легкий доступ к родным общим библиотекам (`.dll` файлы в Windows, файлы `.so` в Unix ...)

Как я могу использовать его?

- Во-первых, загрузите последнюю версию JNA и укажите ее jna.jar в CLASSPATH вашего проекта.
- Во-вторых, скопируйте, скомпилируйте и запустите Java-код ниже

  Для целей этого введения мы предполагаем, что используемая нативная платформа - это Windows. Если вы работаете на другой платформе, просто замените строку `"msvcrt"` на строку `"c"` в приведенном ниже коде.

Маленькая Java-программа ниже выведет сообщение на консоль, вызвав функцию C `printf` .

CRuntimeLibrary.java

```
package jna.introduction;

import com.sun.jna.Library;
import com.sun.jna.Native;

// We declare the printf function we need and the library containing it (msvcrt)...
public interface CRuntimeLibrary extends Library {

   CRuntimeLibrary INSTANCE =
       (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

   void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;
```

```
// Now we call the printf function...
public class MyFirstJNAProgram {
    public static void main(String args[]) {
         CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

Куда пойти сейчас?

Перейдите в другую тему или перейдите на официальный сайт .

Прочитайте Виртуальный доступ Java онлайн: https://riptutorial.com/ru/java/topic/5244/виртуальный-доступ-java

глава 59: Вложенные и внутренние классы

Вступление

Используя Java, разработчики могут определять класс в другом классе. Такой класс называется вложенным классом . Вложенные классы называются внутренними классами, если они были объявлены как нестатические, если нет, их просто называют статическими вложенными классами. Эта страница предназначена для документирования и предоставления подробных сведений о том, как использовать Java Nested и Inner Classes.

Синтаксис

- public class OuterClass {public class InnerClass {}} // Внутренние классы также могут быть приватными
- public class OuterClass {public static class StaticNestedClass {}} // Статические вложенные классы также могут быть приватными
- public void method () {private class LocalClass {}} // Локальные классы всегда приватные
- SomeClass anonymousClassInstance = new SomeClass () {}; // Анонимные внутренние классы нельзя назвать, следовательно, доступ является спорным. Если «SomeClass ()» является абстрактным, тело должно реализовать все абстрактные методы.
- SomeInterface anonymousClassInstance = new SomeInterface () {}; // Тело должно реализовать все методы интерфейса.

замечания

Терминология и классификация

Спецификация языка Java (JLS) классифицирует различные типы Java-классов следующим образом:

> *Класс верхнего уровня* - это класс, который не является вложенным классом.
>
> *Вложенным классом* является любой класс, чье объявление происходит внутри тела другого класса или интерфейса.
>
> *Внутренний класс* представляет собой вложенный класс, который явно или неявно не объявлен статическим.
>
> Внутренний класс может быть *не статическим классом-членом* , *локальным классом* или *анонимным классом* . Класс-член интерфейса неявно статичен, поэтому он никогда не считается внутренним классом.

На практике программисты ссылаются на класс верхнего уровня, который содержит внутренний класс как «внешний класс». Кроме того, существует тенденция использовать «вложенный класс» для обозначения только (явно или неявно) статических вложенных классов.

Обратите внимание, что между анонимными внутренними классами и лямбдами существует тесная связь, но лямбды - это классы.

Семантические различия

- Классы верхнего уровня - это «базовый случай». Они видны другим частям программы, подчиненным нормальным правилам видимости, основанным на семантике модификатора доступа. Если они не являются абстрактными, они могут быть созданы каким-либо кодом, где соответствующие конструкторы видны на основе модификаторов доступа.

- Статические вложенные классы следуют тем же правилам доступа и создания экземпляров, что и классы верхнего уровня, за двумя исключениями:

    - Вложенный класс может быть объявлен как `private` , что делает его недоступным вне его класса верхнего уровня.
    - Вложенный класс имеет доступ к `private` членам охватывающего класса верхнего уровня и всего его тестируемого класса.

    Это делает статические вложенные классы полезными, когда вам нужно представлять несколько «типов сущностей» в пределах границы жесткой абстракции; например, когда вложенные классы используются для скрытия «деталей реализации».

- Внутренние классы добавляют возможность доступа к нестационарным переменным, объявленным в охватываемых областях:

    - Нестатический класс-член может ссылаться на переменные экземпляра.
    - Локальный класс (объявленный внутри метода) также может ссылаться на локальные переменные метода, при условии, что они являются `final` . (Для Java 8 и более поздних версий они могут быть *фактически окончательными* .)
    - Анонимный внутренний класс может быть объявлен как в классе, так и в методе и может обращаться к переменным в соответствии с теми же правилами.

    Тот факт, что экземпляр внутреннего класса может ссылаться на переменные в экземпляре охватывающего класса, имеет последствия для экземпляра. В частности, экземпляр-экземпляр должен быть предоставлен, как неявно, так и явно, при создании экземпляра внутреннего класса.

Examples

Простой стек с использованием вложенного класса

```
public class IntStack {

    private IntStackNode head;

    // IntStackNode is the inner class of the class IntStack
    // Each instance of this inner class functions as one link in the
    // Overall stack that it helps to represent
    private static class IntStackNode {

        private int val;
        private IntStackNode next;

        private IntStackNode(int v, IntStackNode n) {
            val = v;
            next = n;
        }
    }

    public IntStack push(int v) {
        head = new IntStackNode(v, head);
        return this;
    }

    public int pop() {
        int x = head.val;
        head = head.next;
        return x;
    }
}
```

И их использование, которое (в частности) вовсе не признает существование вложенного класса.

```
public class Main {
    public static void main(String[] args) {

        IntStack s = new IntStack();
        s.push(4).push(3).push(2).push(1).push(0);

        //prints: 0, 1, 2, 3, 4,
        for(int i = 0; i < 5; i++) {
            System.out.print(s.pop() + ", ");
        }
    }
}
```

Статические и нестатические вложенные классы

При создании вложенного класса вы сталкиваетесь с выбором наличия вложенного класса static:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

Или нестатический:

```
public class OuterClass2 {

    private class NestedClass {

    }

}
```

По своей сути, статические вложенные классы *не имеют окружающего* ***экземпляра*** внешнего класса, тогда как нестатические вложенные классы делают. Это влияет как на то, где / когда разрешено создавать экземпляр вложенного класса, так и к каким экземплярам этих вложенных классов разрешен доступ. Добавление к приведенному выше примеру:

```
public class OuterClass1 {

    private int aField;
    public void aMethod(){}

    private static class StaticNestedClass {
        private int innerField;

        private StaticNestedClass() {
             innerField = aField; //Illegal, can't access aField from static context
             aMethod();           //Illegal, can't call aMethod from static context
        }

        private StaticNestedClass(OuterClass1 instance) {
             innerField = instance.aField; //Legal
        }

    }

    public static void aStaticMethod() {
        StaticNestedClass s = new StaticNestedClass(); //Legal, able to construct in static
context
        //Do stuff involving s...
    }

}

public class OuterClass2 {

    private int aField;

    public void aMethod() {}
```

```
    private class NestedClass {
        private int innerField;

        private NestedClass() {
             innerField = aField; //Legal
             aMethod(); //Legal
        }
    }

    public void aNonStaticMethod() {
        NestedClass s = new NestedClass(); //Legal
    }

    public static void aStaticMethod() {
        NestedClass s = new NestedClass(); //Illegal. Can't construct without surrounding
OuterClass2 instance.
                                           //As this is a static context, there is no
surrounding OuterClass2 instance
    }
}
```

Таким образом, ваше решение статического и нестатического в основном зависит от того, нужно ли вам напрямую обращаться к полям и методам внешнего класса, хотя это также имеет последствия для того, когда и где вы можете построить вложенный класс.

Как правило, сделайте ваши вложенные классы статичными, если вам не нужно обращаться к полям и методам внешнего класса. Подобно тому, как ваши поля являются закрытыми, если они вам не нужны, это уменьшает видимость, доступную для вложенного класса (не разрешая доступ к внешнему экземпляру), уменьшая вероятность ошибки.

Модификаторы доступа для внутренних классов

Полное описание модификаторов доступа в Java можно найти здесь . Но как они взаимодействуют с внутренними классами?

`public` , как обычно, предоставляет неограниченный доступ к любой области, доступной для доступа к типу.

```
public class OuterClass {

    public class InnerClass {

        public int x = 5;

    }

    public InnerClass createInner() {
        return new InnerClass();
    }
}

public class SomeOtherClass {

    public static void main(String[] args) {
```

```
        int x = new OuterClass().createInner().x; //Direct field access is legal
    }
}
```

оба `protected` а модификатор по умолчанию (ничего) ведет себя так же, как ожидалось, так же, как и для не-вложенных классов.

`private` , интересно, не ограничивает класс, к которому он принадлежит. Скорее, он ограничивает блок компиляции - .java-файл. Это означает, что внешние классы имеют полный доступ к полям и методам Inner, даже если они выделены как `private` .

```
public class OuterClass {

    public class InnerClass {

        private int x;
        private void anInnerMethod() {}
    }

    public InnerClass aMethod() {
        InnerClass a = new InnerClass();
        a.x = 5; //Legal
        a.anInnerMethod(); //Legal
        return a;
    }
}
```

Сам внутренний класс может иметь видимость, отличную от `public` . Пометив его `private` или другим модификатором ограниченного доступа, другим (внешним) классам не разрешается импортировать и присваивать тип. Однако они все равно могут получать ссылки на объекты этого типа.

```
public class OuterClass {

    private class InnerClass{}

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //Illegal, can't find type
        OuterClass.InnerClass x = o.makeInnerClass(); //Illegal, InnerClass has visibility
private
        Object x = o.makeInnerClass(); //Legal
    }
}
```

Анонимные внутренние классы

Анонимный внутренний класс является формой внутреннего класса, который объявляется и создается с помощью одного утверждения. Как следствие, нет названия для класса, который можно использовать в другом месте программы; т.е. анонимно.

Анонимные классы обычно используются в ситуациях, когда вам нужно создать легкий класс для передачи в качестве параметра. Обычно это делается с помощью интерфейса. Например:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

Этот анонимный класс определяет объект `Comparator<String>` (`CASE_INSENSITIVE`), который сравнивает две строки, игнорируя различия в случае.

Другие интерфейсы, которые часто реализуются и создаются с использованием анонимных классов, являются `Runnable` и `Callable` . Например:

```
// An anonymous Runnable class is used to provide an instance that the Thread
// will run when started.
Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
              System.out.println("Hello world");
        }
    });
t.start();  // Prints "Hello world"
```

Анонимные внутренние классы также могут основываться на классах. В этом случае анонимный класс неявно `extends` существующий класс. Если расширяемый класс является абстрактным, то анонимный класс должен реализовать все абстрактные методы. Он также может переопределять не абстрактные методы.

Конструкторы

Анонимный класс не может иметь явный конструктор. Вместо этого определяется неявный конструктор, который использует `super(...)` для передачи любых параметров конструктору в расширяемом классе. Например:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

`SomeClass` конструктор для нашего анонимного подкласса `SomeClass` вызовет конструктор `SomeClass` который соответствует сигнатуре вызова `SomeClass(int, String)` . Если конструктор недоступен, вы получите ошибку компиляции. Любые исключения, создаваемые согласованным конструктором, также выдаются неявным конструктором.

Естественно, это не работает при расширении интерфейса. Когда вы создаете анонимный класс из интерфейса, суперкласс класса представляет собой `java.lang.Object` которого есть только конструктор no-args.

Метод Местные Внутренние Классы

Класс, написанный внутри метода, называемого **локальным внутренним классом метода** . В этом случае объем внутреннего класса ограничен в рамках метода.

Локальный внутренний класс метода может быть создан только внутри метода, где определяется внутренний класс.

Пример использования локального внутреннего класса метода:

```
public class OuterClass {
    private void outerMethod() {
       final int outerInt = 1;
        // Method Local Inner Class
        class MethodLocalInnerClass {
            private void print() {
                System.out.println("Method local inner class " + outerInt);
            }
        }
        // Accessing the inner class
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

Выполнение даст результат: `Method local inner class 1` .

Доступ к внешнему классу из нестатического внутреннего класса

Ссылка на внешний класс использует имя класса, и `this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

Вы можете напрямую обращаться к полям и методам внешнего класса.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

Но в случае столкновения имен вы можете использовать ссылку на внешний класс.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        private int counter;

        public void method() {
            System.out.println("My counter: " + counter);
            System.out.println("Outer counter: " + OuterClass.this.counter);

            // updating my counter
            counter = OuterClass.this.counter;
        }
    }
}
```

Создать экземпляр нестатического внутреннего класса извне

Внутренний класс, который видим любому внешнему классу, также может быть создан из этого класса.

Внутренний класс зависит от внешнего класса и требует ссылки на его экземпляр. Чтобы создать экземпляр внутреннего класса, `new` оператор нужно вызвать только в экземпляре внешнего класса.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Обратите внимание на использование как `outer.new` .

Прочитайте Вложенные и внутренние классы онлайн: https://riptutorial.com/ru/java/topic/3317/вложенные-и-внутренние-классы

глава 60: Возможности Java SE 7

Вступление

В этом разделе вы найдете краткое описание новых функций, добавленных в язык программирования Java в Java SE 7. В других областях, таких как JDBC и Java Virtual Machine (JVM), есть много других новых функций, которые не будут охвачены в этой теме.

замечания

Усовершенствования в Java SE 7

Examples

Новые возможности языка программирования Java SE 7

- Бинарные литералы : интегральные типы (байт, короткий, int и long) также могут быть выражены с использованием системы двоичных чисел. Чтобы указать бинарный литерал, добавьте префикс 0b или 0B в число.
- Строки в операторах switch : вы можете использовать объект String в выражении оператора switch
- Заявление try-with-resources: оператор try-with-resources представляет собой оператор try, который объявляет один или несколько ресурсов. Ресурс - это объект, который должен быть закрыт после завершения программы. Оператор try-with-resources гарантирует, что каждый ресурс будет закрыт в конце инструкции. В качестве ресурса может использоваться любой объект, реализующий java.lang.AutoCloseable, который включает в себя все объекты, которые реализуют java.io.Closeable.
- Улавливание множественных типов исключений и исключение исключений с улучшенным контролем типов : один блок catch может обрабатывать более одного типа исключения. Эта функция может уменьшить дублирование кода и уменьшить соблазн, чтобы поймать чрезмерно широкое исключение.
- Подчеркивания в числовых литералах : любое число символов подчеркивания (_) может отображаться где угодно между цифрами в числовом литерале. Эта функция позволяет вам, например, разделять группы цифр в числовых литералах, что может улучшить читаемость вашего кода.
- Вывод типа для создания Generic Instance : вы можете заменить аргументы типа, необходимые для вызова конструктора родового класса с пустым набором параметров типа (<>), если компилятор может вывести аргументы типа из контекста. Эта пара угловых скобок неофициально называется алмазом.
- Улучшенные предупреждения и ошибки компилятора при использовании невосстанавливаемых формальных параметров с помощью методов Varargs

Бинарные литералы

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;
```

Оператор try-with-resources

В примере читается первая строка из файла. Он использует экземпляр `BufferedReader` для чтения данных из файла. `BufferedReader` - это ресурс, который должен быть закрыт после завершения программы:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

В этом примере ресурс, объявленный в инструкции try-with-resources, представляет собой `BufferedReader` . Оператор объявления появляется в круглых скобках сразу после ключевого слова try. Класс `BufferedReader` , в Java SE 7 и более поздних версиях, реализует интерфейс `java.lang.AutoCloseable` . Поскольку экземпляр `BufferedReader` объявлен в инструкции try-with-resource, он будет закрыт независимо от того, завершает ли оператор try нормально или внезапно (в результате метода `BufferedReader.readLine` генерирует `IOException`).

Подчеркивает числовые литералы

В следующем примере показаны другие способы использования подчеркивания в числовых литералах:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Вы можете разместить символы подчеркивания только между цифрами; вы не можете

размещать символы подчеркивания в следующих местах:

- В начале или конце номера
- Рядом с десятичной точкой в литерале с плавающей запятой
- До суффикса F или L
- В позициях, где ожидается строка цифр

Тип вывода для создания общего экземпляра

Ты можешь использовать

```
Map<String, List<String>> myMap = new HashMap<>();
```

вместо

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Однако вы не можете использовать

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

потому что он не может скомпилировать. Обратите внимание, что бриллиант часто работает в методах; однако предлагается использовать алмаз в основном для объявлений переменных.

Строки в операторах

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
     String typeOfDay;
     switch (dayOfWeekArg) {
         case "Monday":
             typeOfDay = "Start of work week";
             break;
         case "Tuesday":
         case "Wednesday":
         case "Thursday":
             typeOfDay = "Midweek";
             break;
         case "Friday":
             typeOfDay = "End of work week";
             break;
         case "Saturday":
         case "Sunday":
             typeOfDay = "Weekend";
             break;
```

```
            default:
                throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
        }
        return typeOfDay;
    }
```

Прочитайте Возможности Java SE 7 онлайн: https://riptutorial.com/ru/java/topic/8272/возможности-java-se-7

глава 61: Возможности Java SE 8

Вступление

В этом разделе вы найдете резюме новых функций, добавленных в язык программирования Java в Java SE 8. В других областях, таких как JDBC и Java Virtual Machine (JVM), есть много других новых функций, которые не будут охвачены в этой теме.

замечания

Ссылка: Усовершенствования в Java SE 8

Examples

Новые возможности языка программирования Java SE 8

- В этом выпуске была представлена новая функция языка Lambda Expressions . Они позволяют вам рассматривать функциональность как аргумент метода или код как данные. Лямбда-выражения позволяют более компактно выражать экземпляры интерфейсов с одним методом (называемые функциональными интерфейсами).
    - Ссылки на методы предоставляют легко читаемые лямбда-выражения для методов, которые уже имеют имя.
    - Способы по умолчанию позволяют добавлять новые функциональные возможности в интерфейсы библиотек и обеспечивать двоичную совместимость с кодом, написанным для более старых версий этих интерфейсов.
    - Новые и усовершенствованные API-интерфейсы, которые используют выражения лямбда-выражения и потоки в Java SE 8, описывают новые и расширенные классы, которые используют лямбда-выражения и потоки.
- Улучшенный вывод типа. Компилятор Java использует целевую типизацию для вывода параметров типа общего вызова метода. Целевой тип выражения - это тип данных, который ожидает компилятор Java в зависимости от того, где выражается выражение. Например, вы можете использовать целевой тип задания назначения для вывода типа в Java SE 7. Однако в Java SE 8 вы можете использовать целевой тип для вывода типа в других контекстах.
    - Целевой ввод в выражениях лямбда
    - Вывод типа
- Повторяющиеся аннотации предоставляют возможность применять один и тот же тип аннотации более одного раза к одному и тому же объявлению или типу использования.
- Типовые аннотации предоставляют возможность применять аннотацию везде, где используется тип, а не только для объявления. Эта функция, используемая с

подключаемой системой типа, позволяет улучшить проверку вашего кода.

- Отражение параметра метода. Вы можете получить имена формальных параметров любого метода или конструктора с помощью метода `java.lang.reflect.Executable.getParameters` . (Класс Method и Constructor расширяет класс Executable и поэтому наследует метод `Executable.getParameters`). Однако `.class` не сохраняют формальные имена параметров по умолчанию. Чтобы сохранить формальные имена параметров в конкретном `.class` файл, и , таким образом , позволяет Reflection API , чтобы получить формальные имена параметров, компилировать исходный файл с - `-parameters` выбором JAVAC компилятора.
- Date-time-api - добавлено новое время api в `java.time` . Если это используется, вам не нужно указывать часовой пояс.

Прочитайте Возможности Java SE 8 онлайн: https://riptutorial.com/ru/java/topic/8267/возможности-java-se-8

глава 62: Выбор коллекций

Вступление

Java предлагает широкий выбор коллекций. Выбор какой коллекции для использования может быть сложным. См. Раздел «Примеры» для простой в использовании блок-схемы, чтобы выбрать нужную коллекцию для задания.

Examples

Блок-схема Java Collections

Используйте следующую блок-схему, чтобы выбрать нужную коллекцию для задания.

Эта блок-схема была основана на [http://i.stack.imgur.com/aSDsG.png) .

Прочитайте Выбор коллекций онлайн: https://riptutorial.com/ru/java/topic/10846/выбор-коллекций

глава 63: Выражения

Вступление

Выражения в Java являются основной конструкцией для выполнения вычислений.

замечания

Для справки о операторах, которые можно использовать в выражениях, см. Операторы .

Examples

Приоритет оператора

Когда выражение содержит несколько операторов, его потенциально можно читать по-разному. Например, математическое выражение `1 + 2 x 3` можно читать двумя способами:

1. Добавьте `1` и `2` и умножьте результат на `3` . Это дает ответ `9` . Если мы добавим круглые скобки, это будет выглядеть как `( 1 + 2 ) x 3` .
2. Добавьте `1` к результату умножения `2` и `3` . Это дает ответ `7` . Если мы добавим круглые скобки, это будет выглядеть как `1 + ( 2 x 3 )` .

В математике соглашение состоит в том, чтобы читать выражение вторым способом. Общее правило заключается в том, что умножение и деление выполняются до сложения и вычитания. Когда используется более продвинутая математическая нотация, либо значение либо «самоочевидно» (для обученного математика!), Либо круглые скобки добавляются для устранения неоднозначности. В любом случае эффективность обозначений для передачи смысла зависит от интеллекта и общих знаний математиков.

В Java есть четкие правила о том, как читать выражение, основанное на *приоритете* используемых операторов.

В общем случае каждому оператору приписывается значение *приоритета* ; см. таблицу ниже.

Например:

```
1 + 2 * 3
```

Приоритет `+` меньше приоритета `*` , поэтому результат выражения равен 7, а не 9.

Описание	Операторы / конструкции (первичные)	старшинство	Ассоциативность
спецификатор Скобки Создание экземпляра Доступ к полям Доступ к массиву Вызов метода Ссылка на метод	имя . название (expr) new первичный . название primary [expr] primary (expr, ...) primary :: name	15	Слева направо
Пошаговый прирост	expr ++ , expr --	14	-
Предварительное увеличение Одинарный В ролях [1]	++ expr, -- expr, + expr, - expr, ~ expr ! выраж, (тип) expr	13	- Справа налево Справа налево
Multiplicative	* /%	12	Слева направо
присадка	+ -	11	Слева направо
сдвиг	<< >> >>>	10	Слева направо
реляционный	<> <=> = instanceof	9	Слева направо
равенство	==! =	8	Слева направо
Побитовое И	&	7	Слева направо
Побитовое исключение ИЛИ	^	6	Слева направо
Побитовое включение ИЛИ	|	5	Слева направо
Логические И	&&	4	Слева направо
Логический ИЛИ	||	3	Слева направо
Условный [1]	? :	2	Справа налево
присваивание Лямбда [1]	= * = / =% = + = - = << = >> = >>> = & = ^ = | = ->	1	Справа налево

[1] Приоритет экспрессии Lambda является сложным, так как он также может возникать после литья или в качестве третьей части условного тернарного оператора.

Константные выражения

Постоянное выражение является выражением, которое дает примитивный тип или String, и значение которого может быть оценено во время компиляции до литерала. Выражение должно оцениваться без исключения исключения и должно состоять только из следующего:

- Примитивные и строковые литералы.
- Набирает типы для примитивных типов или `String` .
- Следующие унарные операторы: `+` , `-` , `~` и `!` ,
- Следующие бинарные операторы: `*` , `/` , `%` , `+` , `-` , `<<` , `>>` , `>>>` , `<` , `<=` , `>` , `>=` , `==` `!=` , `&` , `^` , `|` , `&&` и `||` ,
- Тернарный условный оператор `?` `:` .
- Ориентированные на скобки выражения.
- Простые имена, относящиеся к постоянным переменным. (Постоянная переменная - это переменная, объявленная как `final` где выражение инициализатора само является константным выражением.)
- Квалифицированные имена формы `<TypeName>` `.` `<Identifier>` которые относятся к постоянным переменным.

Обратите внимание, что в приведенном выше списке *исключаются* `++` и `--` , операторы присваивания, `class` и `instanceof` , вызовы методов и ссылки на общие переменные или поля.

Константные выражения типа `String` приводят к «интернированной» `String` , а операции с плавающей запятой в константных выражениях оцениваются с помощью FP-строгой семантики.

Использование для константных выражений

Константные выражения могут использоваться (примерно) в любом месте, где может использоваться нормальное выражение. Однако они имеют особое значение в следующих контекстах.

Для операторов case в операторах switch требуются константные выражения. Например:

```
switch (someValue) {
```

```
case 1 + 1:             // OK
case Math.min(2, 3):    // Error - not a constant expression
    doSomething();
}
```

Когда выражение в правой части присваивания является постоянным выражением, тогда назначение может выполнять примитивное сужение преобразования. Это разрешено при условии, что значение константного выражения находится в пределах диапазона типа с левой стороны. (См. JLS 5.1.3 и 5.2) Например:

```
byte b1 = 1 + 1;              // OK - primitive narrowing conversion.
byte b2 = 127 + 1;            // Error - out of range
byte b3 = b1 + 1;             // Error - not a constant expession
byte b4 = (byte) (b1 + 1);    // OK
```

Когда константное выражение используется как условие в `do` , `while` или `for` , то оно влияет на анализ читаемости. Например:

```
while (false) {
    doSomething();            // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();            // OK
}
```

(Обратите внимание , что это не относится , `if` заявления. Компилятор Java позволяет `then` или `else` блок , `if` оператор будет недоступен. Это аналог Java условной компиляции в C и C ++.)

Наконец, `static final` поля в классе или интерфейсе с постоянными инициализаторами выражения инициализируются с нетерпением. Таким образом, гарантируется, что эти константы будут наблюдаться в инициализированном состоянии, даже если в графике зависимостей инициализации класса есть цикл.

Для получения дополнительной информации см. JLS 15.28. Константные выражения .

Порядок оценки выражений

Выражения Java оцениваются по следующим правилам:

- Операнды оцениваются слева направо.
- Операторы оператора оцениваются перед оператором.
- Операторы оцениваются в соответствии с приоритетом оператора
- Списки аргументов оцениваются слева направо.

Простой пример

В следующем примере:

```
int i = method1() + method2();
```

порядок оценки:

1. Левый операнд оператора `=` вычисляется по адресу `i` .
2. Вычисляется левый операнд оператора `+` (`method1()`).
3. `method2()` правый операнд оператора `+` (`method2()`).
4. `+` операция `+` .
5. Операция `=` вычисляется, присваивая результат добавления к `i` .

Обратите внимание, что если эффекты вызовов наблюдаемы, вы сможете заметить, что вызов `method1` происходит до вызова `method2` .

Пример с оператором, который имеет побочный эффект

В следующем примере:

```
int i = 1;
intArray[i] = ++i + 1;
```

порядок оценки:

1. Вычисляется левый операнд оператора `=` . Это дает адрес `intArray[1]` .
2. Оценивается предварительный прирост. Это добавляет `1` к `i` и оценивает до `2` .
3. Проверяется правый операнд `+` .
4. Операция `+` оценивается как: `2 + 1` -> `3` .
5. Операция `=` вычисляется, присваивая значение `3` `intArray[1]` .

Обратите внимание, что, поскольку левый операнд `=` сначала оценивается, на него не влияет побочный эффект подвыражения `++i` .

Ссылка:

- JLS 15.7 - Порядок оценки

Основы экспрессии

Выражения в Java являются основной конструкцией для выполнения вычислений. Вот некоторые примеры:

```
1                 // A simple literal is an expression
1 + 2             // A simple expression that adds two numbers
(i + j) / k       // An expression with multiple operations
(flag) ? c : d    // An expression using the "conditional" operator
(String) s        // A type-cast is an expression
```

```
obj.test()        // A method call is an expression
new Object()      // Creation of an object is an expression
new int[]         // Creation of an object is an expression
```

В общем случае выражение состоит из следующих форм:

- Имена выражений, которые состоят из:
  - Простые идентификаторы; например, `someIdentifier`
  - Квалифицированные идентификаторы; например `MyClass.someField`
- Первичные, состоящие из:
  - литералы; например, `1` , `1.0` , `'X'` , `"hello"` , `false` и `null`
  - Литеральные выражения класса; например, `MyClass.class`
  - `this` и `<TypeName> . this`
  - Семантические выражения; например `( a + b )`
  - Выражения создания экземпляра класса; например, `new MyClass(1, 2, 3)`
  - Выражения создания экземпляра массива; например, `new int[3]`
  - Выражения доступа к полю; например `obj.someField` или `this.someField`
  - Выражения доступа к массиву; например, `vector[21]`
  - Вызов метода; например `obj.doIt(1, 2, 3)`
  - Ссылки на методы (Java 8 и более поздние версии); например `MyClass::doIt`
- Унарные выражения оператора; eg `!a` или `i++`
- Бинарные выражения оператора; например `a + b` или `obj == null`
- Тернарные выражения оператора; например `(obj == null) ? 1 : obj.getCount()`
- Лямбда-выражения (Java 8 и более поздние версии); например `obj -> obj.getCount()`

Подробности различных форм выражений можно найти в других разделах.

- В разделе « Операторы» рассматриваются унарные, двоичные и тернарные выражения операторов.
- В выражении лямбда-выражений рассматриваются лямбда-выражения и ссылочные выражения методов.
- Тема « Классы и объекты» охватывает выражения создания экземпляра класса.
- В разделе « Массивы» рассматриваются выражения доступа к массиву и выражения создания экземпляра массива.
- В литературной теме рассматриваются различные виды литературных выражений.

Тип выражения

В большинстве случаев выражение имеет статический тип, который можно определить во время компиляции путем изучения и его подвыражений. Они называются *автономными* выражениями.

Однако (в Java 8 и более поздних версиях) следующие выражения могут быть *поли выражениями* :

- Семантические выражения
- Выражения создания экземпляра класса
- Выражения вызова метода
- Справочные выражения метода
- Условные выражения
- Лямбда-выражения

Когда выражение является поли-выражением, на его тип может влиять *целевой тип* выражения; т.е. для чего он используется.

Значение выражения

Значение выражения - это присвоение, совместимое с его типом. Исключением является то, когда произошло *загрязнение кучи* ; например, потому что предупреждения «небезопасного преобразования» были (ненадлежащим образом) подавлены или проигнорированы.

Выражения выражений

В отличие от многих других языков, Java обычно не позволяет выражениям использоваться в качестве операторов. Например:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Поскольку результат оценки такого выражения не может быть использован, и поскольку он не может повлиять на выполнение программы каким-либо другим способом, разработчики Java приняли позицию, что такое использование является либо ошибкой, либо ошибочным.

Однако это не относится ко всем выражениям. Подмножество выражений (фактически) является законным как утверждения. Набор содержит:

- Выражение присваивания, включая *операции-и-* назначения.
- Предварительные и последующие выражения приращения и уменьшения.
- Вызов метода (`void` или `void`).
- Выражения экземпляра класса.

Прочитайте Выражения онлайн: https://riptutorial.com/ru/java/topic/8167/выражения

глава 64: Генерация случайных чисел

замечания

Ничто не является случайным, и поэтому javadoc называет эти числа псевдослучайными. Эти числа создаются с помощью генератора псевдослучайных чисел .

Examples

Псевдо-случайные числа

Java предоставляет в составе пакета `utils` базовый генератор псевдослучайных чисел, который называется « `Random` . Этот объект может использоваться для генерации псевдослучайного значения как любого из встроенных числовых типов данных (`int` , `float` и т. Д.). Вы также можете использовать его для генерации случайного логического значения или случайного массива байтов. Пример использования:

```
import java.util.Random;

...

Random random = new Random();
int randInt = random.nextInt();
long randLong = random.nextLong();

double randDouble = random.nextDouble(); //This returns a value between 0.0 and 1.0
float randFloat = random.nextFloat(); //Same as nextDouble

byte[] randBytes = new byte[16];
random.nextBytes(randBytes); //nextBytes takes a user-supplied byte array, and fills it with
random bytes. It returns nothing.
```

ПРИМЕЧАНИЕ. Этот класс производит только низкокачественные псевдослучайные числа и никогда не должен использоваться для генерации случайных чисел для криптографических операций или в других ситуациях, где важна более качественная случайность (для этого вы хотели бы использовать класс `SecureRandom` , как указано ниже). Объяснение различия между «безопасным» и «неуверенным» случайным образом выходит за рамки этого примера.

Псевдо случайные числа в определенном диапазоне

Метод `nextInt(int bound) Random` принимает верхнюю эксклюзивную границу, то есть число, которое возвращаемое случайное значение должно быть меньше. Однако только метод `nextInt` принимает `nextInt` ; `nextLong` , `nextDouble` и т.п. нет.

```
Random random = new Random();
random.nextInt(1000); // 0 - 999

int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

Начиная с Java 1.7, вы также можете использовать `ThreadLocalRandom` (источник). Этот класс обеспечивает поточно-безопасный PRNG (генератор псевдослучайных чисел). Обратите внимание, что метод `nextInt` этого класса принимает как верхнюю, так и нижнюю границу.

```
import java.util.concurrent.ThreadLocalRandom;

// nextInt is normally exclusive of the top value,
// so add 1 to make it inclusive
ThreadLocalRandom.current().nextInt(min, max + 1);
```

Обратите внимание, что в официальной документации указано, что `nextInt(int bound)` может делать странные вещи, когда `bound` около $2^{30}+1$ (выделено курсивом):

> Алгоритм немного сложный. **Он отклоняет значения, которые приведут к неравномерному распределению** (из-за того, что 2 ^ 31 не делится на n). Вероятность отклонения значения зависит от n. **Наихудший случай равен n = 2 ^ 30 + 1, для которого вероятность отклонения равна 1/2, а ожидаемое число итераций до окончания цикла равно 2.**

Другими словами, указание границы (немного) уменьшит производительность метода `nextInt` , и это снижение производительности станет более выраженным, так как `bound` приближается к половине максимального значения int.

Создание криптографически безопасных псевдослучайных чисел

`Random` и `ThreadLocalRandom` достаточно хороши для повседневного использования, но у них есть большая проблема: они основаны на линейном конгруэнтном генераторе , алгоритме, выход которого можно предсказать довольно легко. Таким образом, эти два класса **не** подходят для криптографических целей (таких как генерация ключей).

Можно использовать `java.security.SecureRandom` в ситуациях, когда требуется PRNG с `java.security.SecureRandom` который очень трудно предсказать. Предсказание случайных чисел, созданных экземплярами этого класса, достаточно сложно, чтобы обозначить класс как **криптографически безопасный** .

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
```

```
        rng.nextBytes(randomBytes); // Fills randomBytes with random bytes (duh)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

Помимо криптографической защиты `SecureRandom` имеет гигантский период в 2^{160}, по сравнению с периодом `Random` s 2^{48}. У этого есть один недостаток быть значительно медленнее, чем `Random` и другие линейные PRNG, такие как Mersenne Twister и Xorshift.

Обратите внимание, что реализация SecureRandom зависит от платформы и поставщика. По умолчанию `SecureRandom` (данный поставщиком `SUN` в `sun.security.provider.SecureRandom`):

- на Unix-подобных системах, засеянных данными из `/dev/random` и / или `/dev/urandom`.
- в Windows, засеянных вызовами `CryptGenRandom()` в CryptoAPI.

Выбор случайных чисел без дубликатов

```
/**
 * returns a array of random numbers with no duplicates
 * @param range the range of possible numbers for ex. if 100 then it can be anywhere from 1-
100
 * @param length the length of the array of random numbers
 * @return array of random numbers with no duplicates.
 */
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length<range){
        // this is where all the random numbers
        int[] randomNumbers = new int[length];

        // loop through all the random numbers to set them
        for (int q = 0; q < randomNumbers.length; q++){

            // get the remaining possible numbers
            int remainingNumbers = range-q;

            // get a new random number from the remainingNumbers
            int newRandSpot = (int) (Math.random()*remainingNumbers);

            newRandSpot++;

            // loop through all the possible numbers
            for (int t = 1; t < range+1; t++){

                // check to see if this number has already been taken
                boolean taken = false;
                for (int number : randomNumbers){
                    if (t==number){
                        taken = true;
                        break;
                    }
                }

                // if it hasnt been taken then remove one from the spots
                if (!taken){
                    newRandSpot--;
```

```
                    // if we have gone though all the spots then set the value
                    if (newRandSpot==0){
                        randomNumbers[q] = t;
                    }
                }
            }
        }
        return randomNumbers;
    } else {
        // invalid can't have a length larger then the range of possible numbers
    }
    return null;
}
```

Метод работает путем циклизации, хотя массив, который имеет размер запрашиваемой длины и находит оставшуюся длину возможных чисел. Он устанавливает случайное число этих возможных номеров `newRandSpot` и находит, что число внутри оставшегося числа осталось. Он делает это, перебирая диапазон и проверяя, было ли это число уже выполнено.

Например, если диапазон равен 5, а длина равна 3, и мы уже выбрали число 2. Тогда у нас есть 4 оставшихся числа, поэтому мы получаем случайное число от 1 до 4 и прокручиваем диапазон (5), пропуская любые числа что мы уже использовали (2).

Теперь предположим, что следующее число, выбранное между 1 и 4, равно 3. В первом цикле мы получаем 1, который еще не был взят, поэтому мы можем удалить 1 из 3, сделав его 2. Теперь во втором цикле мы получим 2, который был взят поэтому мы ничего не делаем. Мы следуем этой схеме до тех пор, пока не дойдем до 4, где, как только мы удалим 1, она станет 0, поэтому мы устанавливаем новый randomNumber равным 4.

Создание случайных чисел с заданным семенем

```
//Creates a Random instance with a seed of 12345.
Random random = new Random(12345L);

//Gets a ThreadLocalRandom instance
ThreadLocalRandom tlr = ThreadLocalRandom.current();

//Set the instance's seed.
tlr.setSeed(12345L);
```

Использование одного и того же семени для генерации случайных чисел будет возвращать одинаковые числа каждый раз, поэтому установка другого семени для каждого `Random` экземпляра является хорошей идеей, если вы не хотите в итоге дублировать числа.

Хорошим методом получения `Long` который отличается для каждого вызова, является `System.currentTimeMillis()` :

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Создание случайного числа с использованием apache-common lang3

Мы можем использовать `org.apache.commons.lang3.RandomUtils` для генерации случайных чисел с использованием одной строки.

```
int x = RandomUtils.nextInt(1, 1000);
```

Метод `nextInt(int startInclusive, int endExclusive)` принимает диапазон.

Помимо int, мы можем генерировать случайные `long` , `double` , `float` и `bytes` используя этот класс.

Класс `RandomUtils` содержит следующие методы:

```
static byte[] nextBytes(int count) //Creates an array of random bytes.
static double nextDouble() //Returns a random double within 0 - Double.MAX_VALUE
static double nextDouble(double startInclusive, double endInclusive) //Returns a random double
within the specified range.
static float nextFloat() //Returns a random float within 0 - Float.MAX_VALUE
static float nextFloat(float startInclusive, float endInclusive) //Returns a random float
within the specified range.
static int nextInt() //Returns a random int within 0 - Integer.MAX_VALUE
static int nextInt(int startInclusive, int endExclusive) //Returns a random integer within the
specified range.
static long nextLong() //Returns a random long within 0 - Long.MAX_VALUE
static long nextLong(long startInclusive, long endExclusive) //Returns a random long within
the specified range.
```

Прочитайте Генерация случайных чисел онлайн: https://riptutorial.com/ru/java/topic/890/генерация-случайных-чисел

глава 65: Геттеры и сеттеры

Вступление

В этой статье обсуждаются геттеры и сеттеры; стандартный способ обеспечения доступа к данным в Java-классах.

Examples

Добавление геттеров и сеттеров

Инкапсуляция является базовой концепцией в ООП. Речь идет об обертывании данных и кода в виде единого блока. В этом случае рекомендуется объявлять переменные как `private` а затем обращаться к ним через `Getters` и `Setters` для просмотра и / или изменения их.

```
public class Sample {
  private String  name;
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Эти частные переменные не могут быть доступны непосредственно извне класса. Следовательно, они защищены от несанкционированного доступа. Но если вы хотите просмотреть или изменить их, вы можете использовать Getters и Setters.

`getXxx()` вернет текущее значение переменной `xxx` , в то время как вы можете установить значение переменной `xxx` с помощью `setXxx()` .

Соглашение об именах методов (в переменной example называется `variableName`):

- Все `boolean` переменные

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- `boolean` переменные

```
 isVariableName()     //Getter, The variable name should start with uppercase
 setVariableName(...) //Setter, The variable name should start with uppercase
```

Публичные Getters и Setters являются частью определения свойства Java Bean.

Использование сеттера или геттера для реализации ограничения

Setters и Getters позволяют объекту содержать частные переменные, к которым можно получить доступ и изменить с ограничениями. Например,

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
           this.name = name;
    }
}
```

В этом классе `Person` есть единственная переменная: `name` . К этой переменной можно обратиться с помощью `getName()` и изменить с помощью `setName(String)` , однако для установки имени требуется, чтобы новое имя имеет длину более 2 символов и не должно быть нулевым. Использование метода setter вместо того, чтобы публиковать `name` переменной, позволяет другим устанавливать значение `name` с определенными ограничениями. То же самое можно применить к методу геттера:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

В модифицированном `getName()` выше `name` возвращается, только если его длина меньше или равна 16. В противном случае возвращается `"Name is too large"` . Это позволяет программисту создавать переменные, которые достижимы и могут быть изменены, но, тем не менее, они препятствуют нежелательным редактированию переменных.

Зачем использовать Getters и Setters?

Рассмотрим базовый класс, содержащий объект с геттерами и сеттерами в Java:

```
public class CountHolder {
  private int count = 0;

  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

Мы не можем получить доступ к переменной `count` потому что она закрыта. Но мы можем получить доступ к методам `getCount()` и `setCount(int)` потому что они общедоступны. Для некоторых это может поставить вопрос; зачем вводить посредника? Почему бы просто просто не считать их общедоступными?

```
public class CountHolder {
  public int count = 0;
}
```

Для всех целей и задач эти два являются точно такими же, функционально. Разница между ними - расширяемость. Подумайте, что говорит каждый класс:

- **Во-первых** : «У меня есть метод, который даст вам значение `int` и метод, который установит это значение для другого `int` ».
- **Во-вторых** : «У меня есть `int` который вы можете установить и получить, как пожелаете».

Они могут казаться похожими, но первый на самом деле гораздо более защищен по своей природе; он только позволяет вам взаимодействовать с его внутренней природой, как **она** диктует. Это оставляет мяч в его суде; он выбирает, как происходят внутренние взаимодействия. Вторая сторона внесла свою внутреннюю реализацию извне и теперь не только подвержена внешним пользователям, но, в случае API, **привержена** поддержке этой реализации (или иным образом освобождает API, не поддерживающий обратную совместимость).

Давайте рассмотрим, хотим ли мы синхронизировать доступ к изменению и доступу к счету. Во-первых, это просто:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

но во втором примере это становится практически невозможным без прохождения и изменения каждого места, на которое ссылается переменная `count` . Что еще хуже, если это элемент, который вы предоставляете в библиотеке, которую вы будете потреблять другими, у вас **нет** способа выполнить эту модификацию и вынуждены сделать жесткий

выбор, упомянутый выше.

Поэтому он задает вопрос; являются ли общедоступные переменные когда-либо хорошей (или, по крайней мере, не злой)?

Я не уверен. С одной стороны, вы можете увидеть примеры открытых переменных, которые выдержали проверку времени (IE: переменная `out` ссылается `System.out`). С другой стороны, предоставление публичной переменной не приносит пользы за пределы чрезвычайно минимальных накладных расходов и потенциального сокращения словесности. Мое руководство здесь будет состоять в том, что если вы планируете делать переменную публику, вы должны судить об этом против этих критериев с **крайним** предрассудком:

1. Переменные не должны иметь никаких мыслимых оснований **когда** - **либо** изменений в его реализации. Это очень легко повредить (и, даже если вы все исправите, требования могут измениться), поэтому геттеры / сеттеры - общий подход. Если у вас будет общедоступная переменная, это действительно нужно продумать, особенно если она будет выпущена в библиотеке / framework / API.
2. На переменную нужно ссылаться достаточно часто, чтобы гарантировать минимальный выигрыш от сокращения подробностей. Я даже не думаю, что накладные расходы на использование метода в сравнении с прямой ссылкой должны рассматриваться здесь. Это слишком незначительно для того, что я с консервативной оценкой должен составлять 99,9% приложений.

Там, наверное, больше, чем я не рассматривал с головы. Если вы когда-либо сомневаетесь, всегда используйте геттеры / сеттеры.

Прочитайте Геттеры и сеттеры онлайн: https://riptutorial.com/ru/java/topic/3560/геттеры-и-сеттеры

глава 66: Даты и время (java.time. *)

Examples

Простые мануалы даты

Получить текущую дату.

```
LocalDate.now()
```

Получите вчерашнюю дату.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Получить дату завтра

```
LocalDate t = LocalDate.now().plusDays(1);
```

Получите конкретную дату.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

В дополнение к методам « `plus` и « `minus` существует набор методов «с», которые можно использовать для установки определенного поля в экземпляре `LocalDate` .

```
LocalDate.now().withMonth(6);
```

В приведенном выше примере возвращается новый экземпляр с месяцем, установленным в июне (это отличается от `java.util.Date` где `setMonth` был проиндексирован 0, `setMonth` с 5 июня).

Поскольку манипуляции с LocalDate возвращают неизменяемые экземпляры LocalDate, эти методы также могут быть соединены вместе.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

Это даст нам завтрашний день через год.

Дата и время

Дата и время без информации о часовом поясе

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
```

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Дата и время с информацией о часовом поясе

```
ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(localDate, localTime, zoneId);
ZonedDateTime now = ZonedDateTime.now(); // Default time zone
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");
```

Дата и время со смещенной информацией (т. Е. Не учитываются изменения ДСТ)

```
ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(localDate, localTime, zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // Offset taken from the default ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");
```

Операции по датам и времени

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDateTime anHourFromNow = LocalDateTime.now().plusHours(1);
Long daysBetween = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration duration = Duration.between(Instant.now(), ZonedDateTime.parse("2016-07-
27T07:00:00+01:00[Europe/Stockholm]"))
```

Мгновенный

Представляет мгновенное время. Можно рассматривать как оболочку вокруг метки времени Unix.

```
Instant now = Instant.now();
Instant epoch1 = Instant.ofEpochMilli(0);
Instant epoch2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(epoch1, epoch2); // 0
```

Использование различных классов API Date Time

Следующий пример также содержит объяснение, необходимое для понимания примера внутри него.

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
```

```
import java.util.TimeZone;
public class SomeMethodsExamples {

/**
 * Has the methods of the class {@link LocalDateTime}
 */
public static void checkLocalDateTime() {
    LocalDateTime localDateTime = LocalDateTime.now();
    System.out.println("Local Date time using static now() method ::: >>> "
            + localDateTime);

    LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("AET")));
    System.out
            .println("LOCAL TIME USING now(ZoneId zoneId) method ::: >>>>"
                    + ldt1);

    LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
            .of(ZoneId.SHORT_IDS.get("PST"))));
    System.out
            .println("Local TIME USING now(Clock.system(ZoneId.of())) ::: >>>> "
                    + ldt2);

    System.out
            .println("Following is a static map in ZoneId class which has mapping of short
timezone names to their Actual timezone names");
    System.out.println(ZoneId.SHORT_IDS);

}

/**
 * This has the methods of the class {@link LocalDate}
 */
public static void checkLocalDate() {
    LocalDate localDate = LocalDate.now();
    System.out.println("Gives date without Time using now() method. >> "
            + localDate);
    LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
    System.out
            .println("now() is overridden to take ZoneID as parametere using this we can get
the same date under different timezones. >> "
                    + localDate2);
}

/**
 * This has the methods of abstract class {@link Clock}. Clock can be used
 * for time which has time with {@link TimeZone}.
 */
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // Represents time according to ISO 8601
    System.out.println("Time using Clock class : " + clock.instant());
}

/**
 * This has the {@link Instant} class methods.
 */
public static void checkInstant() {
    Instant instant = Instant.now();
```

```
    System.out.println("Instant using now() method :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());

    System.out.println("Instants using now(Clock clock) :: " + ins1);

}

/**
 * This class checks the methods of the {@link Duration} class.
 */
public static void checkDuration() {
    // toString() converts the duration to PTnHnMnS format according to ISO
    // 8601 standard. If a field is zero its ignored.

    // P is the duration designator (historically called "period") placed at
    // the start of the duration representation.
    // Y is the year designator that follows the value for the number of
    // years.
    // M is the month designator that follows the value for the number of
    // months.
    // W is the week designator that follows the value for the number of
    // weeks.
    // D is the day designator that follows the value for the number of
    // days.
    // T is the time designator that precedes the time components of the
    // representation.
    // H is the hour designator that follows the value for the number of
    // hours.
    // M is the minute designator that follows the value for the number of
    // minutes.
    // S is the second designator that follows the value for the number of
    // seconds.

    System.out.println(Duration.ofDays(2));
}

/**
 * Shows Local time without date. It doesn't store or represenet a date and
 * time. Instead its a representation of Time like clock on the wall.
 */
public static void checkLocalTime() {
    LocalTime localTime = LocalTime.now();
    System.out.println("LocalTime :: " + localTime);
}

/**
 * A date time with Time zone details in ISO-8601 standards.
 */
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
            .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);

}
}
```

Форматирование даты

До Java 8 в пакете `java.text` классы `DateFormat` и `SimpleDateFormat` и этот устаревший код

будет продолжать использоваться некоторое время.

Но Java 8 предлагает современный подход к обработке форматирования и анализа.

При форматировании и синтаксическом анализе сначала вы передаете объект `String` в `DateTimeFormatter` и, в свою очередь, используете его для форматирования или синтаксического анализа.

```
import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //Parsing
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //Default format
                      ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //Custom format

        System.out.println(ldp1 + "\n" + ldp2); //Will be printed in Default format

        //Formatting
        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) +"\n"+ldtf1.format(dtF3));
    }
}
```

Важное замечание, вместо использования пользовательских шаблонов, является хорошей практикой для использования предопределенных форматировщиков. Ваш код выглядит более понятным, и использование ISO8061 определенно поможет вам в долгосрочной перспективе.

Вычислить разницу между 2 локальными датами

Используйте `LocalDate` и `ChronoUnit` :

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

теперь, поскольку метод `between` перечислением `ChronoUnit` принимает 2 `Temporal` параметра как параметры, поэтому вы можете без проблем `LocalDate` экземплярами `LocalDate`

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Прочитайте Даты и время (java.time. *) онлайн: https://riptutorial.com/ru/java/topic/4813/даты-и-время--java-time----

глава 67: Двигатель JavaScript Nashorn

Вступление

Nashorn - это движок JavaScript, разработанный на Java Oracle и выпущенный с помощью Java 8. Nashorn позволяет внедрять Javascript в Java-приложения через JSR-223 и позволяет разрабатывать автономные приложения Javascript и обеспечивает лучшую производительность во время выполнения и лучшее соответствие ECMA нормализованная спецификация Javascript.

Синтаксис

- ScriptEngineManager // Обеспечивает механизм обнаружения и установки для классов ScriptEngine; использует SPI (интерфейс поставщика услуг)
- ScriptEngineManager.ScriptEngineManager () // Рекомендуемый конструктор
- ScriptEngine // Обеспечивает интерфейс для языка сценариев
- ScriptEngine ScriptEngineManager.getEngineByName (String shortName) // Заводский метод для данной реализации
- Object ScriptEngine.eval (String script) // Выполняет указанный скрипт
- Object ScriptEngine.eval (Reader reader) // Загружает и затем выполняет скрипт из указанного источника
- ScriptContext ScriptEngine.getContext () // Возвращает привязки по умолчанию, читателей и авторов
- void ScriptContext.setWriter (писатель писателя) // Устанавливает адресата для отправки вывода сценария в

замечания

Nashorn - это движок JavaScript, написанный на Java и включенный в Java 8. Все, что вам нужно, связано с пакетом `javax.script` .

Обратите внимание, что `ScriptEngineManager` предоставляет общий API, позволяющий получать скриптовые механизмы для различных языков сценариев (т.е. не только Nashorn, а не только JavaScript).

Examples

Установить глобальные переменные

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
```

```
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");

// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// 'Data defined in Java.' printed on standard output
```

Привет, Насорн

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Execute an hardcoded script
try {
    engine.eval("print('Hello Nashorn!');");
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Выполнить файл JavaScript

```
// Required imports
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// Obtain an instance of the JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Load and execute a script from the file 'demo.js'
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Script from file!' printed on standard output
```

demo.js :

```
print('Script from file!');
```

Выход перехвата скрипта

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Setup a custom writer
StringWriter stringWriter = new StringWriter();
// Modify the engine context so that the custom writer is now the default
// output writer of the engine
engine.getContext().setWriter(stringWriter);

// Execute some script
try {
    engine.eval("print('Redirected text!');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Nothing printed on standard output, but
// stringWriter.toString() contains 'Redirected text!'
```

Оценка арифметических строк

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

//String to be evaluated
String str = "3+2*4+5";
//Value after doing Arithmetic operation with operator precedence will be 16

//Printing the value
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

//Outcome:
//Value of the string after arithmetic evaluation is printed on standard output.
//In this case '16.0' will be printed on standard output.
```

Использование объектов Java в JavaScript в Нашорне

Можно передавать объекты Java в механизм Nashorn для обработки в Java-коде. В то же время существуют некоторые конструкции JavaScript (и Nashorn), и не всегда ясно, как они работают с объектами Java.

Ниже приведена таблица, описывающая поведение собственных объектов Java внутри конструкций JavaScript.

Протестированные конструкции:

1. Выражение в условии if. В выражении JS выражение if не должно быть логическим, в отличие от Java. Он оценивается как false для так называемых значений ложности (null, undefined, 0, пустые строки и т. Д.),
2. для каждого утверждения Nashorn имеет особый тип цикла - для каждого - который может перебирать разные объекты JS и Java.
3. Получение размера объекта. В объектах JS есть длина свойства, которая возвращает размер массива или строки.

Результаты:

Тип	Если	для каждого	.length
Java null	ложный	Нет итераций	**исключение**
Пустая строка Java	ложный	Нет итераций	0
Строка Java	правда	Итерации по строковым символам	Длина строки
Java Integer / Long	значение! = 0	Нет итераций	не определено
Java ArrayList	правда	Итерации по элементам	Длина списка
Java HashMap	правда	Итерирует значения	ноль
Java HashSet	правда	Итерирует элементы	не определено

Recommendatons:

- Целесообразно использовать `if (some_string)` чтобы проверить, не является ли строка пустой и не пустой
- `for each` можно безопасно использовать для итерации по любой коллекции, и она не вызывает исключений, если коллекция не является итерируемой, нулевой или неопределенной
- Перед тем, как получить длину объекта, он должен быть проверен как null или undefined (то же самое верно для любой попытки вызова метода или получения свойства объекта Java)

Внедрение интерфейса из скрипта

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
    }

    public static void main(String[] args) throws IOException {
        // Obtain an instance of JavaScript engine
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            //evaluate a script
            /* pet.js */
            /*
                var Pet = Java.type("InterfaceImplementationExample.Pet");

                new Pet() {
                    eat: function() { print("eat"); }
                }
            */

            Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

            pet.eat();
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }

        // Outcome:
        // 'eat' printed on standard output
    }
}
```

Установить и получить глобальные переменные

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name","Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello '+name+'!';");
    // Get value
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
```

```
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Прочитайте Двигатель JavaScript Nashorn онлайн: https://riptutorial.com/ru/java/topic/166/двигатель-javascript-nashorn

глава 68: Дженерики

Вступление

Generics - это средство общего программирования, которое расширяет систему типов Java, чтобы позволить типу или методу работать на объектах разных типов, обеспечивая при этом безопасность типа времени компиляции. В частности, среда коллекций Java поддерживает дженерики, чтобы указать тип объектов, хранящихся в экземпляре коллекции.

Синтаксис

- class ArrayList <E> {} // общий класс с параметром типа E
- класс HashMap <K, V> {} // общий класс с двумя параметрами типа K и V
- <E> void print (элемент E) {} // общий метод с параметром типа E
- ArrayList <String> имена; // объявление общего класса
- ArrayList <?> Объекты; // объявление общего класса с неизвестным параметром типа
- new ArrayList <String> () // создание экземпляра универсального класса
- new ArrayList <> () // экземпляр с типом вывода "diamond" (Java 7 или новее)

замечания

Генерики реализуются в Java через стирание типа, что означает, что во время выполнения информация типа, указанная в создании экземпляра универсального класса, недоступна. Например, оператор `List<String> names = new ArrayList<>();` создает объект списка, из которого тип элемента `String` не может быть восстановлен во время выполнения. Однако, если список хранится в поле типа `List<String>` или передается параметру method / constructor этого же типа или возвращается из метода этого типа возврата, то полная информация о типе *может* быть восстановлена во время выполнения через API Java Reflection.

Это также означает, что при выдаче родовому типу (например: `(List<String>) list`), литой является *непроверенный бросок* . Поскольку параметр `<String>` стирается, JVM не может проверить правильность приведения из `List<?>` `List<String>` ; JVM видит только показ `List` для `List` во время выполнения.

Examples

Создание общего класса

Generics позволяет классам, интерфейсам и методам брать другие классы и интерфейсы в

качестве параметров типа.

В этом примере используется общий класс `Param` для принятия одного **параметра типа** `T` , ограниченного угловыми скобками (`<>`):

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Чтобы создать экземпляр этого класса, укажите **аргумент типа** вместо `T` Например, `Integer` :

```
Param<Integer> integerParam = new Param<Integer>();
```

Аргумент типа может быть любым ссылочным типом, включая массивы и другие общие типы:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

В Java SE 7 и более поздних версиях аргумент типа может быть заменен пустым набором аргументов типа (`<>`), называемым *алмазом* :

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

В отличие от других идентификаторов, параметры типа не имеют ограничений именования. Однако их имена обычно являются первой буквой их цели в верхнем регистре. (Это верно даже во всех официальных JavaDocs.)
Примеры включают `T` для «type» , `E` для «element» и `K` / `V` для «key» / «value» .

Расширение общего класса

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
```

```
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

`AbstractParam` - абстрактный класс, объявленный с параметром типа `T` При расширении этого класса этот параметр типа может быть заменен аргументом типа, записанным внутри `<>` , или параметр типа может оставаться неизменным. В первом и втором примерах ниже `String` и `Integer` заменяют параметр типа. В третьем примере параметр типа остается неизменным. В четвертом примере вообще не используются обобщения, поэтому он аналогичен классу `Object` . Компилятор будет предупреждать о том, что `AbstractParam` является сырым типом, но он скомпилирует класс `ObjectParam` . Пятый пример имеет 2 типа параметров (см. «Параметры нескольких типов» ниже), выбирая второй параметр как параметр типа, переданный суперклассу.

```
public class Email extends AbstractParam<String> {
    // ...
}

public class Age extends AbstractParam<Integer> {
    // ...
}

public class Height<T> extends AbstractParam<T> {
    // ...
}

public class ObjectParam extends AbstractParam {
    // ...
}

public class MultiParam<T, E> extends AbstractParam<E> {
    // ...
}
```

Ниже приведено использование:

```
Email email = new Email();
email.setValue("test@example.com");
String retrievedEmail = email.getValue();

Age age = new Age();
age.setValue(25);
Integer retrievedAge = age.getValue();
int autounboxedAge = age.getValue();

Height<Integer> heightInInt = new Height<>();
heightInInt.setValue(125);

Height<Float> heightInFloat = new Height<>();
heightInFloat.setValue(120.3f);
```

```
MultiParam<String, Double> multiParam = new MultiParam<>();
multiParam.setValue(3.3);
```

Обратите внимание, что в классе `Email` метод `T getValue()` действует так, как будто он имеет подпись `String getValue()` , а метод `void setValue(T)` действует так, как если бы он был объявлен `void setValue(String)` .

Также возможно создать экземпляр с анонимным внутренним классом с пустыми фигурными фигурными скобками (`{}`):

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

Обратите внимание, что использование алмаза с анонимными внутренними классами не допускается.

Множественные параметры типа

Java предоставляет возможность использовать более одного параметра типа в универсальном классе или интерфейсе. Параметры нескольких типов могут использоваться в классе или интерфейсе, помещая **список разделенных запятыми** типов между угловыми скобками. Пример:

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

Использование может быть выполнено следующим образом:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

Объявление общего метода

Методы также могут иметь общие параметры типа.

```
public class Example {

    // The type parameter T is scoped to the method
    // and is independent of type parameters of other methods.
    public <T> List<T> makeList(T t1, T t2) {
        List<T> result = new ArrayList<T>();
        result.add(t1);
        result.add(t2);
        return result;
    }

    public void usage() {
        List<String> listString = makeList("Jeff", "Atwood");
        List<Integer> listInteger = makeList(1, 2);
    }
}
```

Обратите внимание, что нам не нужно передавать фактический аргумент типа в общий метод. Компилятор вводит аргумент типа для нас на основе целевого типа (например, переменную, для которой присваивается результат), или типы фактических аргументов. Обычно он выдает наиболее специфический аргумент типа, который сделает правильный тип вызова.

Иногда, хотя и редко, может быть необходимо переопределить этот тип вывода с явными аргументами типа:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

Это необходимо в этом примере, потому что компилятор не может «смотреть вперед», чтобы увидеть, что `Object` нужен для `T` после вызова `stream()` и в противном случае он `makeList` бы `String` на `makeList` аргументов `makeList` . Обратите внимание, что язык Java не поддерживает исключение класса или объекта, на который вызывается метод (`this` в приведенном выше примере), когда аргументы типа явно указаны.

Бриллиант

Java SE 7

Java 7 представила *Diamond* [1], чтобы удалить некоторую котельную плиту вокруг создания экземпляра универсального класса. С Java 7+ вы можете написать:

```
List<String> list = new LinkedList<>();
```

Там, где вам приходилось писать в предыдущих версиях, это:

```
List<String> list = new LinkedList<String>();
```

Одно ограничение - для анонимных классов , где вы все равно должны указывать параметр типа в экземпляре:

```
// This will compile:

Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};

// But this will not:

Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

Java SE 8

Хотя использование алмаза с анонимными внутренними классами не поддерживается в Java 7 и 8, **оно будет включено как новая функция в Java 9** .

Сноска:

1 - Некоторые люди называют `<>` использование «алмазного *оператора* ». Это неверно. Алмаз не ведет себя как оператор и не описан или не указан нигде в JLS или (официальном) Java-учебном пособии как оператор. Действительно, `<>` не является даже явным символом Java. Скорее это `<` токен, за которым следует токен `>` , и это законный (хотя и плохой стиль), чтобы иметь пробелы или комментарии между ними. JLS и Tutorials последовательно ссылаются на `<>` как на «алмаз», и поэтому это правильный термин для него.

Требование нескольких верхних границ («расширяет A & B»)

Вы можете потребовать, чтобы общий тип расширил несколько верхних границ.

Пример: мы хотим отсортировать список чисел, но `Number` не реализует `Comparable` .

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
```

```
  Collections.sort( n );
}
```

В этом примере `T` должен расширять `Number` *и* реализовывать `Comparable<T>` который должен соответствовать всем «нормальным» реализациям встроенных чисел, таких как `Integer` или `BigDecimal` но не подходит для более экзотических, таких как `Striped64` .

Поскольку множественное наследование не разрешено, вы можете использовать не более одного класса в качестве привязки и должны быть первыми перечислены. Например, `<T extends Comparable<T> & Number>` не допускается, потому что Comparable является интерфейсом, а не классом.

Создание ограниченного общего класса

Вы можете ограничить допустимые типы, используемые в **общем классе,** путем ограничения этого типа в определении класса. Учитывая следующую простую иерархию типов:

```
public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}
```

Без **ограниченных дженериков** мы не можем создать класс-контейнер, который является общим и знает, что каждый элемент является животным:

```
public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
```

```
            System.out.println(t.makeSound());
        }
    }
}
```

С общим ограничением в определении класса это теперь возможно.

```
public class BoundedAnimalContainer<T extends Animal> { // Note bound here.

    private Collection<T> col;

    public BoundedAnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Now works because T is extending Animal
            System.out.println(t.makeSound());
        }
    }
}
```

Это также ограничивает действительные экземпляры родового типа:

```
// Legal
AnimalContainer<Cat> a = new AnimalContainer<Cat>();

// Legal
AnimalContainer<String> a = new AnimalContainer<String>();
```

```
// Legal because Cat extends Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// Illegal because String doesn't extends Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

Решаем между `T`,`? супер T` и `? расширяет T`

Синтаксис Java генерирует ограниченные подстановочные знаки, представляя неизвестный тип `?` является:

- `? extends T` представляет собой верхний ограниченный шаблон. Неизвестный тип представляет тип, который должен быть подтипом T или сам тип T.
- `? super T` представляет собой нижний ограниченный шаблон. Неизвестный тип представляет тип, который должен быть супертипом T или самим типом T.

Как правило, вы должны использовать

- `? extends T` если вам нужен только «чтение» доступа («ввод»)
- `? super T` если вам нужно «написать» доступ («выход»)
- `T` если вам нужны оба («изменить»)

Использование `extends` или `super` обычно *лучше,* потому что это делает ваш код более гибким (как в: разрешении использования подтипов и супертипов), как вы увидите ниже.

```
class Shoe {}
class IPhone {}
interface Fruit {}
class Apple implements Fruit {}
class Banana implements Fruit {}
class GrannySmith extends Apple {}

   public class FruitHelper {

         public void eatAll(Collection<? extends Fruit> fruits) {}

         public void addApple(Collection<? super Apple> apples) {}
}
```

Теперь компилятор сможет обнаружить определенное плохое использование:

```
 public class GenericsTest {
      public static void main(String[] args){
  FruitHelper fruitHelper = new FruitHelper() ;
    List<Fruit> fruits = new ArrayList<Fruit>();
    fruits.add(new Apple()); // Allowed, as Apple is a Fruit
    fruits.add(new Banana()); // Allowed, as Banana is a Fruit
    fruitHelper.addApple(fruits); // Allowed, as "Fruit super Apple"
    fruitHelper.eatAll(fruits); // Allowed

    Collection<Banana> bananas = new ArrayList<>();
    bananas.add(new Banana()); // Allowed
    //fruitHelper.addApple(bananas); // Compile error: may only contain Bananas!
    fruitHelper.eatAll(bananas); // Allowed, as all Bananas are Fruits

    Collection<Apple> apples = new ArrayList<>();
    fruitHelper.addApple(apples); // Allowed
    apples.add(new GrannySmith()); // Allowed, as this is an Apple
    fruitHelper.eatAll(apples); // Allowed, as all Apples are Fruits.

    Collection<GrannySmith> grannySmithApples = new ArrayList<>();
    fruitHelper.addApple(grannySmithApples); //Compile error: Not allowed.
                                  // GrannySmith is not a supertype of Apple
    apples.add(new GrannySmith()); //Still allowed, GrannySmith is an Apple
    fruitHelper.eatAll(grannySmithApples);//Still allowed, GrannySmith is a Fruit

    Collection<Object> objects = new ArrayList<>();
    fruitHelper.addApple(objects); // Allowed, as Object super Apple
    objects.add(new Shoe()); // Not a fruit
    objects.add(new IPhone()); // Not a fruit
    //fruitHelper.eatAll(objects); // Compile error: may contain a Shoe, too!
}
```

Выбрав правильный `T ? super T` или `? extends T` *необходимо* для использования с подтипами.

Затем компилятор может обеспечить безопасность типа; вам не нужно бросать (который не является безопасным по типу и может вызвать ошибки программирования), если вы используете их правильно.

Если это непросто понять, пожалуйста, помните правило **PECS** :

> **P** roducer использует « **E** xtends», а **C** onsumer использует « **S** uper».

(У производителя есть только доступ на запись, и у Потребителя есть только доступ для чтения)

Преимущества универсального класса и интерфейса

Код, который использует generics, имеет много преимуществ по сравнению с нестандартным кодом. Ниже приведены основные преимущества

Более строгие проверки во время компиляции

Компилятор Java применяет сильную проверку типов к универсальному коду и выдает ошибки, если код нарушает безопасность типов. Исправить ошибки времени компиляции проще, чем исправлять ошибки времени выполнения, которые могут быть трудно найти.

Устранение отливок

Следующий фрагмент кода без дженериков требует кастинга:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

При повторном написании для *использования дженериков* код не требует кастинга:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Включение программистов для реализации общих алгоритмов

Используя generics, программисты могут реализовывать общие алгоритмы, которые работают с коллекциями разных типов, могут быть настроены и безопасны в типе и их легче читать.

Связывание общего параметра с более чем 1 типом

Общие параметры также могут быть привязаны к нескольким типам с использованием синтаксиса `T extends Type1 & Type2 & ...`

Предположим, вы хотите создать класс, универсальный тип которого должен реализовывать как `Flushable` и `Closeable` , вы можете написать

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Теперь `ExampleClass` принимает только общие параметры, типы, которые реализуют как `Flushable` **и** `Closeable` .

```
ExampleClass<BufferedWriter> arg1; // Works because BufferedWriter implements both Flushable
and Closeable

ExampleClass<Console> arg4; // Does NOT work because Console only implements Flushable
ExampleClass<ZipFile> arg5; // Does NOT work because ZipFile only implements Closeable

ExampleClass<Flushable> arg2; // Does NOT work because Closeable bound is not satisfied.
ExampleClass<Closeable> arg3; // Does NOT work because Flushable bound is not satisfied.
```

Методы класса могут выбирать вывод общих аргументов типа как `Closeable` и `Flushable` .

```
class ExampleClass<T extends Flushable & Closeable> {
    /* Assign it to a valid type as you want. */
    public void test (T param) {
        Flushable arg1 = param; // Works
        Closeable arg2 = param; // Works too.
    }

    /* You can even invoke the methods of any valid type directly. */
    public void test2 (T param) {
        param.flush(); // Method of Flushable called on T and works fine.
        param.close(); // Method of Closeable called on T and works fine too.
    }
}
```

Замечания:

Вы не можете привязать общий параметр к любому типу, используя предложение *OR* (`|`). Поддерживается только предложение *AND* (`&`). Общий тип может расширять только один класс и многие интерфейсы. Класс должен быть помещен в начало списка.

Создание экземпляра родового типа

Из-за стирания типа следующее не будет работать:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

Тип `T` стирается. Поскольку во время выполнения JVM не знает, что изначально было `T` , он не знает, какой конструктор должен вызывать.

обходные

1. Передача класса `T` при вызове `genericMethod` :

   ```
   public <T> void genericMethod(Class<T> cls) {
       try {
           T t = cls.newInstance();
       } catch (InstantiationException | IllegalAccessException e) {
            System.err.println("Could not instantiate: " + cls.getName());
       }
   }
   ```

   ```
   genericMethod(String.class);
   ```

   Который выдает исключения, поскольку нет способа узнать, имеет ли переданный класс доступный конструктор по умолчанию.

Java SE 8

2. Передача ссылки на конструктор `T` :

   ```
   public <T> void genericMethod(Supplier<T> cons) {
       T t = cons.get();
   }
   ```

   ```
   genericMethod(String::new);
   ```

Обращаясь к объявленному типовому типу в пределах его собственного объявления

Как вы собираетесь использовать экземпляр (возможно, далее) унаследованного

типичного типа в объявлении метода в объявлении общего типа? Это одна из проблем, с которыми вам придется столкнуться, когда вы копаете немного глубже в дженериках, но все же довольно распространенный.

Предположим, что у нас есть `DataSeries<T>` (здесь), который определяет общий ряд данных, содержащий значения типа `T` Очень сложно работать с этим типом непосредственно, когда мы хотим выполнить много операций, например, с двойными значениями, поэтому мы определяем `DoubleSeries extends DataSeries<Double>` . Предположим, что исходный `DataSeries<T>` имеет метод `add(values)` который добавляет другую серию одинаковой длины и возвращает новую. Как мы применяем тип `values` и тип возврата как `DoubleSeries` а не `DataSeries<Double>` в нашем производном классе?

Проблема может быть решена путем добавления параметра универсального типа, возвращающего назад и расширяющего объявляемый тип (применяется к интерфейсу здесь, но то же самое относится к классам):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Здесь `T` представляет собой тип данных, который имеет ряд, например `Double` и `DS` . Унаследовал тип (или типы) теперь могут быть легко реализованы путем замены упомянутого выше параметра, соответствующего производного типа, таким образом, получая конкретное `Double` основанное определение формы:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

В настоящий момент даже IDE будет реализовывать вышеупомянутый интерфейс с правильными типами на месте, что после того, как бит наполнения содержимого может выглядеть так:

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> data;

    DoubleSeriesImpl(Collection<Double> data) {
        this.data = new ArrayList<>(data);
    }

    @Override
    public DoubleSeries add(DoubleSeries values) {
        List<Double> incoming = values != null ? values.data() : null;
        if (incoming == null || incoming.size() != data.size()) {
            throw new IllegalArgumentException("bad series");
        }
        List<Double> newdata = new ArrayList<>(data.size());
```

```
        for (int i = 0; i < data.size(); i++) {
            newdata.add(this.data.get(i) + incoming.get(i)); // beware autoboxing
        }
        return DoubleSeries.instance(newdata);
    }

    @Override
    public List<Double> data() {
        return Collections.unmodifiableList(data);
    }
}
```

Как вы можете видеть, метод `add` объявлен как `DoubleSeries add(DoubleSeries values)` и компилятор счастлив.

При необходимости шаблон может быть дополнительно вложен.

Использование экземпляра с Generics

Использование дженериков для определения типа в instanceof

Рассмотрим следующий общий класс `Example` объявленный с формальным параметром `<T>` :

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

Это всегда даст ошибку компиляции, поскольку, как только компилятор компилирует *исходный код Java в байт-код Java,* он применяет процесс, известный как *стирание типа* , который преобразует весь общий код в не общий код, что делает невозможным отличать T-типы во время выполнения. Тип, используемый с `instanceof` должен быть *повторно подкрепляемым* , что означает, что вся информация о типе должна быть доступна во время выполнения, и обычно это не относится к родовым типам.

Следующий класс представляет, что два разных класса `Example` , `Example<String>` и `Example<Number>` , выглядят так, что после того, как дженерики разделились *стиранием типа* :

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Поскольку типы исчезли, JVM не знает, какой тип `T`

Исключение из предыдущего правила

Вы всегда можете использовать *неограниченный шаблон* (?) Для указания типа в `instanceof`

следующим образом:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

Это может быть полезно для оценки того, является ли экземпляр `obj List` или нет:

```
System.out.println(isAList("foo")); // prints false
System.out.println(isAList(new ArrayList<String>()); // prints true
System.out.println(isAList(new ArrayList<Float>()); // prints true
```

На самом деле неограниченный подстановочный знак считается воспроизводимым типом.

Использование общего экземпляра с instanceof

Другая сторона монеты заключается в том, что использование экземпляра `t` из `T` с `instanceof` является законным, как показано в следующем примере:

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

потому что после стирания типа класс будет выглядеть следующим образом:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

Так как даже если стирание типа происходит в любом случае, теперь JVM может различать разные типы в памяти, даже если они используют один и тот же ссылочный тип (`Object`), как показано в следующем фрагменте:

```
Object obj1 = new String("foo"); // reference type Object, object type String
Object obj2 = new Integer(11); // reference type Object, object type Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, it's an Integer, not a String
```

Различные способы реализации универсального интерфейса (или расширения общего класса)

Предположим, что был указан следующий общий интерфейс:

```
public interface MyGenericInterface<T> {
```

```
    public void foo(T t);
}
```

Ниже перечислены возможные способы его реализации.

Внеродная реализация класса с определенным типом

Выберите определенный тип, чтобы заменить параметр формального типа `<T>` `MyGenericClass` и реализовать его, как `MyGenericClass` в следующем примере:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

Этот класс относится только к `String` , и это означает, что использование `MyGenericInterface` с различными параметрами (например, `Integer` , `Object` и т. Д.) Не будет компилироваться, как показано в следующем фрагменте:

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // OK, legal
myClass.foo(11); // NOT OK, does not compile
myClass.foo(new Object()); // NOT OK, does not compile
```

Общая реализация класса

Объявите другой общий интерфейс с формальным параметром типа `<T>` который реализует `MyGenericInterface` следующим образом:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Обратите внимание, что может использоваться другой формальный параметр типа:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
    // other methods...
}
```

Внедрение класса Raw

Объявите не-общий класс, который реализует `MyGenericInteface` как *необработанный тип* (вообще не используя общий):

```
public class MyGenericSubclass implements MyGenericInterface {
```

```
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

Этот способ **не** рекомендуется, так как он не на 100% безопасен во время выполнения, потому что он смешивает *сырой тип* (подкласса) с *дженериками* (интерфейса), и это также запутывает. Современные компиляторы Java поднимут предупреждение с такой реализацией, однако код - по соображениям совместимости с более старым JVM (1.4 или более ранним) - скомпилируется.

Все перечисленные выше способы также разрешены при использовании универсального класса в качестве супертипа вместо общего интерфейса.

Использование Generics для автоматического создания

С помощью дженериков можно вернуть все, что ожидает абонент:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

Метод будет скомпилирован с предупреждением. Код на самом деле более безопасен, чем выглядит, потому что время выполнения Java будет выполняться при его использовании:

```
Bar bar = foo.get("bar");
```

Это менее безопасно, если вы используете общие типы:

```
List<Bar> bars = foo.get("bars");
```

Здесь приведение будет выполняться, когда возвращаемый тип представляет собой какой-либо `List` (т. `ClassCastException` Возвращаемый `List<String>` не приведет к `ClassCastException` , в конечном итоге вы получите его, когда `ClassCastException` элементы из списка).

Чтобы обойти эту проблему, вы можете создать API, который использует типизированные ключи:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

наряду с этим методом `put()` :

```
public <T> T put(Key<T> key, T value);
```

При таком подходе вы не можете поместить неправильный тип в карту, поэтому результат

всегда будет правильным (если вы случайно не создадите два ключа с тем же именем, но с разными типами).

Связанные с:

- Тип-безопасная карта

Получить класс, который удовлетворяет общему параметру во время выполнения

Многие несвязанные общие параметры, такие как те, которые используются в статическом методе, не могут быть восстановлены во время выполнения (см. *Другие темы* на *Erasure*). Однако существует общая стратегия, используемая для доступа к типу, удовлетворяющему параметру generic для класса во время выполнения. Это позволяет использовать общий код, который зависит от доступа к типу, *без* необходимости передавать информацию типа типа через каждый вызов.

Фон

Общая параметризация в классе может быть проверена путем создания анонимного внутреннего класса. Этот класс будет захватывать информацию о типе. В общем, этот механизм упоминается как **токены супертекста** , которые подробно описаны в блоге Neal Gafter .

Реализации

Три общие реализации на Java:

- Гуава TypeToken
- Spring ParameterizedTypeReference
- Тип отзыва Джексона

Пример использования

```
public class DataService<MODEL_TYPE> {
     private final DataDao dataDao = new DataDao();
     private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>
                                                                (getClass()){}.getRawType();
     public List<MODEL_TYPE> getAll() {
         return dataDao.getAllOfType(type);
    }
}

// the subclass definitively binds the parameterization to User
// for all instances of this class, so that information can be
// recovered at runtime
public class UserService extends DataService<User> {}

public class Main {
    public static void main(String[] args) {
```

```
            UserService service = new UserService();
            List<User> users = service.getAll();
    }
}
```

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/java/topic/92/дженерики

глава 69: Документирование кода Java

Вступление

Документация для Java-кода часто создается с помощью javadoc . Javadoc был создан Sun Microsystems с целью генерации документации API в формате HTML из исходного кода Java. Использование формата HTML дает удобство гиперссылки связанных документов вместе.

Синтаксис

- / ** - запуск JavaDoc в классе, поле, методе или пакете
- @author // Чтобы назвать автора класса, интерфейса или перечисления. Требуется.
- @version // Версия этого класса, интерфейса или перечисления. Требуется. Вы можете использовать макросы, такие как% I% или% G%, для программного обеспечения для управления исходным кодом, которое необходимо заполнить при оформлении заказа.
- @param // Чтобы показать аргументы (параметры) метода или конструктора. Укажите один тег @param для каждого параметра.
- @return // Чтобы показать типы возвращаемых значений для непустых методов.
- @exception // Показывает, какие исключения могут быть выбраны из метода или конструктора. Исключения, которые **ДОЛЖНЫ** быть пойманы, должны быть перечислены здесь. Если вы хотите, вы можете также включить те, которые не нужно захватывать, например ArrayIndexOutOfBoundsException. Укажите одно исключение @ для каждого исключения, которое может быть выбрано.
- @throws // То же, что @exception.
- @see // Ссылки на метод, поле, класс или пакет. Используйте в виде package.Class # что-то.
- @since // Когда этот метод, поле или класс были добавлены. Например, JDK-8 для класса, такого как java.util.Optional <T> .
- @serial, @serialField, @serialData // Используется для отображения serialVersionUID.
- @deprecated // Чтобы пометить класс, метод или поле как устаревшие. Например, будет java.io.StringBufferInputStream . Смотрите полный список существующих устаревших классов здесь .
- {@link} // Подобно @see, но может использоваться с пользовательским текстом: {@ link #setDefaultCloseOperation (int closeOperation) см. JFrame # setDefaultCloseOperation для получения дополнительной информации}.
- {@linkplain} // Похож на {@link}, но без шрифта кода.
- {@code} // Для буквенного кода, например, HTML-тегов. Например: {@code <html> </ html>}. Однако это будет использовать моноширинный шрифт. Чтобы получить тот же результат без моноширинного шрифта, используйте {@literal}.

- {@literal} // То же, что и {@code}, но без моноширинного шрифта.
- {@value} // Показывает значение статического поля: значение JFrame # EXIT_ON_CLOSE равно {@value}. Или вы можете ссылаться на определенное поле: Использует имя приложения {@value AppConstants # APP_NAME}.
- {@docRoot} // Корневая папка JavaDoc HTML относительно текущего файла. Пример: Кредиты .
- HTML разрешен: <code> «Привет cookies» .substring (3) </ code>.
- * / - конец объявления JavaDoc

замечания

Javadoc - это инструмент, входящий в состав JDK, который позволяет конвертировать комментарии в коде в документацию HTML. Спецификация Java API была сгенерирована с использованием Javadoc. То же самое можно сказать и о большей части документации сторонних библиотек.

Examples

Документация по классу

Все комментарии Javadoc начинаются с комментария блока, за которым следует звездочка (`/**`) и заканчивается, когда комментарий блока (`*/`). При желании каждая строка может начинаться с произвольного пробела и одной звездочки; они игнорируются при создании файлов документации.

```
/**
 * Brief summary of this class, ending with a period.
 *
 * It is common to leave a blank line between the summary and further details.
 * The summary (everything before the first period) is used in the class or package
 * overview section.
 *
 * The following inline tags can be used (not an exhaustive list):
 * {@link some.other.class.Documentation} for linking to other docs or symbols
 * {@link some.other.class.Documentation Some Display Name} the link's appearance can be
 * customized by adding a display name after the doc or symbol locator
 * {@code code goes here} for formatting as code
 * {@literal <>[]()foo} for interpreting literal text without converting to HTML markup
 * or other tags.
 *
 * Optionally, the following tags may be used at the end of class documentation
 * (not an exhaustive list):
 *
 * @author John Doe
 * @version 1.0
 * @since 5/10/15
 * @see some.other.class.Documentation
 * @deprecated This class has been replaced by some.other.package.BetterFileReader
 *
 * You can also have custom tags for displaying additional information.
```

```
 * Using the @custom.<NAME> tag and the -tag custom.<NAME>:htmltag:"context"
 * command line option, you can create a custom tag.
 *
 * Example custom tag and generation:
 * @custom.updated 2.0
 * Javadoc flag: -tag custom.updated:a:"Updated in version:"
 * The above flag will display the value of @custom.updated under "Updated in version:"
 *
 */
public class FileReader {
}
```

Те же теги и формат, используемые для `Classes` могут также использоваться для `Enums` и `Interfaces` .

Методическая документация

Все комментарии Javadoc начинаются с комментария блока, за которым следует звездочка (`/**`) и заканчивается, когда комментарий блока (`*/`). При желании каждая строка может начинаться с произвольного пробела и одной звездочки; они игнорируются при создании файлов документации.

```
/**
 * Brief summary of method, ending with a period.
 *
 * Further description of method and what it does, including as much detail as is
 * appropriate.  Inline tags such as
 * {@code code here}, {@link some.other.Docs}, and {@literal text here} can be used.
 *
 * If a method overrides a superclass method, {@inheritDoc} can be used to copy the
 * documentation
 * from the superclass method
 *
 * @param stream Describe this parameter.  Include as much detail as is appropriate
 *               Parameter docs are commonly aligned as here, but this is optional.
 *               As with other docs, the documentation before the first period is
 *               used as a summary.
 *
 * @return Describe the return values.  Include as much detail as is appropriate
 *         Return type docs are commonly aligned as here, but this is optional.
 *         As with other docs, the documentation before the first period is used as a
 *         summary.
 *
 * @throws IOException Describe when and why this exception can be thrown.
 *                     Exception docs are commonly aligned as here, but this is
 *                     optional.
 *                     As with other docs, the documentation before the first period
 *                     is used as a summary.
 *                     Instead of @throws, @exception can also be used.
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated  Describe why this method is outdated. A replacement can also be specified.
 */
public String[] read(InputStream stream) throws IOException {
    return null;
}
```

Полевая документация

Все комментарии Javadoc начинаются с комментария блока, за которым следует звездочка (`/**`) и заканчивается, когда комментарий блока (`*/`). При желании каждая строка может начинаться с произвольного пробела и одной звездочки; они игнорируются при создании файлов документации.

```
/**
 * Fields can be documented as well.
 *
 * As with other javadocs, the documentation before the first period is used as a
 * summary, and is usually separated from the rest of the documentation by a blank
 * line.
 *
 * Documentation for fields can use inline tags, such as:
 * {@code code here}
 * {@literal text here}
 * {@link other.docs.Here}
 *
 * Field documentation can also make use of the following tags:
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated Describe why this field is outdated
 */
public static final String CONSTANT_STRING = "foo";
```

Пакетная документация

Java SE 5

В Javadocs можно создать документацию на уровне пакета, используя файл `package-info.java` . Этот файл должен быть отформатирован, как показано ниже. Ведущие пробелы и звездочки необязательные, обычно присутствующие в каждой строке для форматирования причины

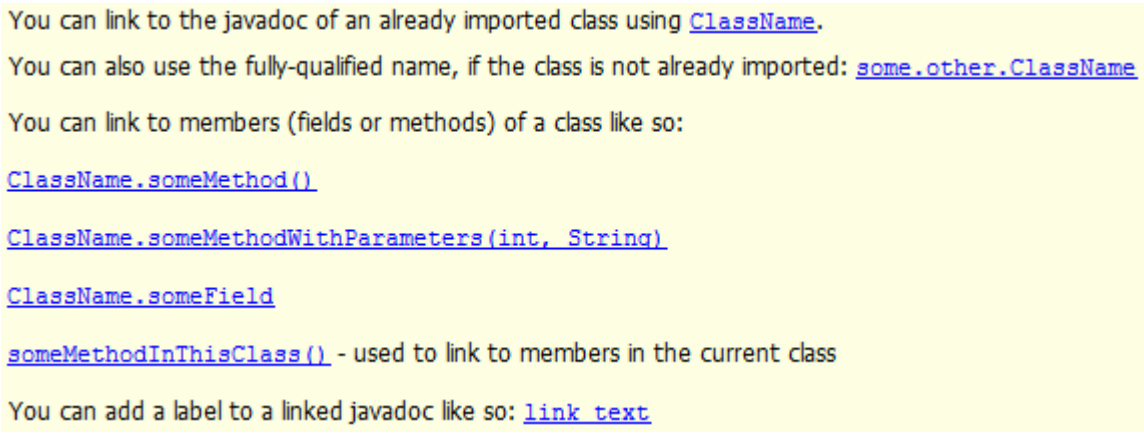
```
/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * Inline tags such as {@code code here}, {@link reference.to.other.Documentation},
 * and {@literal text here} can be used in this documentation.
 */
package com.example.foo;

// The rest of the file must be empty.
```

В приведенном выше случае вы должны поместить этот файл `package-info.java` в папку пакета Java `com.example.foo` .

связи

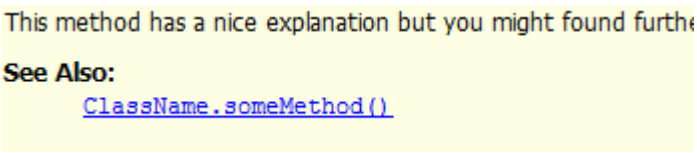
Связывание с другими Javadocs выполняется с помощью тега `@link` :

```
/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 *  {@link some.other.ClassName}
 *
 * You can link to members (fields or methods) of a class like so:
 *  {@link ClassName#someMethod()}
 *  {@link ClassName#someMethodWithParameters(int, String)}
 *  {@link ClassName#someField}
 *  {@link #someMethodInThisClass()} - used to link to members in the current class
 *
 * You can add a label to a linked javadoc like so:
 *  {@link ClassName#someMethod() link text}
 */
```



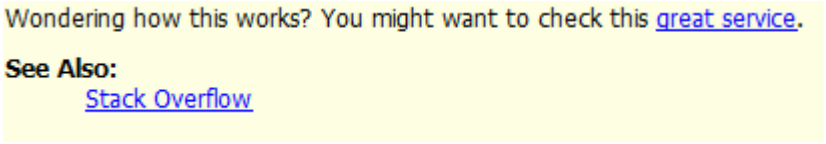
С тегом `@see` вы можете добавлять элементы в раздел « *См. Также* ». Подобно `@param` или `@return` место, где они появляются, не имеет значения. Спектр говорит, что вы должны написать его после `@return` .

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```



Если вы хотите добавить **ссылки на внешние ресурсы,** вы можете просто использовать тег HTML `<a>` . Вы можете использовать его в любом месте или внутри обоих тегов `@link` и `@see` .

```
/**
 * Wondering how this works? You might want
 * to check this <a href="http://stackoverflow.com/">great service</a>.
 *
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>
 */
```



Создание Javadocs из командной строки

Многие IDE обеспечивают поддержку для генерации HTML из Javadocs автоматически; некоторые средства сборки (например, Maven и Gradle) также имеют плагины, которые могут обрабатывать создание HTML.

Однако для создания Javadoc HTML эти инструменты не требуются; это можно сделать с помощью инструмента командной строки `javadoc` .

Самое основное использование инструмента:

```
javadoc JavaFile.java
```

Что будет генерировать HTML из комментариев Javadoc в `JavaFile.java` .

Более практичное использование инструмента командной строки, который будет рекурсивно читать все java-файлы в `[source-directory]` , создавать документацию для `[package.name]` и всех подпакетов и размещать сгенерированный HTML в `[docs-directory]` является:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Документация по встроенному коду

Помимо кода документации Javadoc можно документировать встроенный.

Комментарии Single Line начинаются с `//` и могут быть размещены после оператора в той же строке, но не раньше.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

Многострочные комментарии определяются между `/*` и `*/` . Они могут охватывать

несколько строк и могут быть помещены между операторами.

```
public void method(Object object) {

  /*
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
}
```

JavaDocs - это специальная форма многострочных комментариев, начиная с `/**` .

Поскольку слишком много встроенных комментариев могут уменьшить читаемость кода, их следует использовать редко, если код недостаточно понятен или дизайнерское решение не является очевидным.

Дополнительным вариантом использования однострочных комментариев является использование TAG, которые являются короткими ключевыми словами, основанными на соглашениях. Некоторые среды разработки признают определенные соглашения для таких комментариев. Обычными примерами являются

- `//TODO`
- `//FIXME`

Или укажите ссылки, то есть для Jira

- `//PRJ-1234`

Фрагменты кода внутри документации

Канонический способ написания кода внутри документации - с конструкцией `{@code }` . Если у вас есть многострочный код, завершающий внутри `<pre></pre>` .

```
/**
 * The Class TestUtils.
 * <p>
 * This is an {@code inline("code example")}.
 * <p>
 * You should wrap it in pre tags when writing multiline code.
 * <pre>{@code
 *  Example example1 = new FirstLineExample();
 *  example1.butYouCanHaveMoreThanOneLine();
 * }</pre>
 * <p>
 * Thanks for reading.
 */
class TestUtils {
```

Иногда вам может потребоваться ввести сложный код внутри комментария javadoc. Знак `@` является особенно проблематичным. Использование старого `<code>` рядом с конструкцией

`{@literal }` решает проблему.

```
/**
 * Usage:
 * <pre><code>
 * class SomethingTest {
 * {@literal @}Rule
 *  public SingleTestRule singleTestRule = new SingleTestRule("test1");
 *
 * {@literal @}Test
 *  public void test1() {
 *      // only this test will be executed
 *  }
 *
 *  ...
 * }
 * </code></pre>
 */
class SingleTestRule implements TestRule { }
```

Прочитайте Документирование кода Java онлайн: https://riptutorial.com/ru/java/topic/140/документирование-кода-java

глава 70: Загрузчики классов

замечания

Класс loader - это класс, основной целью которого является опосредование местоположения и загрузки классов, используемых приложением. Погрузчик классов также может находить и загружать *ресурсы* .

Стандартные классы classloader могут загружать классы и ресурсы из каталогов в файловой системе и из JAR и ZIP-файлов. Они также могут загружать и кэшировать файлы JAR и ZIP с удаленного сервера.

Классовые загрузчики обычно закодированы, так что JVM будет пытаться загружать классы из стандартных библиотек классов, предпочитая источники, предоставленные приложением. Пользовательские загрузчики классов позволяют программисту изменять это. Также можно делать такие вещи, как дешифрование файлов байт-кода и модификация байт-кода.

Examples

Создание и использование загрузчика классов

Этот базовый пример показывает, как приложение может создавать экземпляр класса loader и использовать его для динамической загрузки класса.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
Classloader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

Созданный в этом примере загрузчик классов будет иметь загрузчик по умолчанию как родительский, и сначала попытается найти любой класс в родительском загрузчике классов, прежде чем смотреть в «extra.jar». Если запрошенный класс уже загружен, вызов `findClass` вернет ссылку на ранее загруженный класс.

`findClass` может завершиться неудачей различными способами. Наиболее распространенными являются:

- Если названный класс не может быть найден, вызов с throw `ClassNotFoundException` .
- Если названный класс зависит от какого-либо другого класса, который не может быть найден, вызов вызовет `NoClassDefFoundError` .

Внедрение пользовательского классаLoader

Каждый пользовательский загрузчик должен прямо или косвенно расширить класс `java.lang.ClassLoader` . Основными *точками расширения* являются следующие методы:

- `findClass(String)` - перегружает этот метод, если ваш загрузчик классов следует стандартной модели делегирования для загрузки классов.
- `loadClass(String, boolean)` - перегружает этот метод для реализации альтернативной модели делегирования.
- `findResource` и `findResources` - перегружать эти методы для настройки загрузки ресурсов.

Методы `defineClass` которые отвечают за фактическую загрузку класса из массива байтов, являются `final` чтобы предотвратить перегрузку. Перед вызовом `defineClass` .

Вот простой, который загружает определенный класс из массива байтов:

```
public class ByteArrayClassLoader extends ClassLoader {
    private String classname;
    private byte[] classfile;

    public ByteArrayClassLoader(String classname, byte[] classfile) {
        this.classname = classname;
        this.classfile = classfile.clone();
    }

    @Override
    protected Class findClass(String classname) throws ClassNotFoundException {
        if (classname.equals(this.classname)) {
            return defineClass(classname, classfile, 0, classfile.length);
        } else {
            throw new ClassNotFoundException(classname);
        }
    }
}
```

Поскольку мы только переопределили метод `findClass` , этот пользовательский загрузчик классов будет вести себя следующим образом при вызове `loadClass` .

1. Метод `loadClass` `findLoadedClass` вызывает `findLoadedClass` чтобы узнать, был ли класс с этим именем уже загружен этим загрузчиком классов. Если это удастся, результирующий объект `Class` возвращается запрашивающему.
2. Метод `loadClass` затем передает родительскому загрузчику классов, вызывая его вызов `loadClass` . Если родитель может обработать запрос, он вернет объект `Class` который затем возвращается запрашивающему.
3. Если родительский загрузчик классов не может загрузить класс, `findClass` затем вызывает наш метод override `findClass` , передавая имя загружаемого класса.
4. Если запрашиваемое имя соответствует `this.classname` , мы вызываем `defineClass` для загрузки фактического класса из `this.classfile` байтов `this.classfile` . Затем возвращается возвращаемый объект `Class` .
5. Если имя не `ClassNotFoundException` , мы бросаем `ClassNotFoundException` .

Загрузка внешнего файла .class

Чтобы загрузить класс, мы сначала должны его определить. Класс определяется `ClassLoader` . Есть только одна проблема, Oracle не написала код `ClassLoader` с этой доступной функцией. Чтобы определить класс, нам нужно получить доступ к методу с именем `defineClass()` который является приватным методом класса `ClassLoader` .

Чтобы получить доступ к нему, мы создадим новый класс `ByteClassLoader` и распространим его на `ClassLoader` . Теперь, когда мы расширили наш класс до `ClassLoader` , мы можем получить доступ к приватным методам `ClassLoader` . Чтобы сделать `defineClass()` доступным, мы создадим новый метод, который будет действовать как зеркало для частного `defineClass()` . Для вызова частного метода нам потребуется имя класса, `name` , класс байт, `classBytes` , смещенные первый байт, который будет `0` , поскольку `classBytes` данные 'начинается в `classBytes[0]` , и смещение последнего байта, который будет `classBytes.lenght` потому что он представляет собой размер данных, который будет последним смещением.

```
public class ByteClassLoader extends ClassLoader {

    public Class<?> defineClass(String name, byte[] classBytes) {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

}
```

Теперь у нас есть общедоступный `defineClass()` . Его можно вызвать, передав имя класса и байты класса в качестве аргументов.

Допустим, у нас есть класс с именем `MyClass` в пакете `stackoverflow` ...

Для вызова метода нам нужны байты класса, поэтому мы создаем объект `Path` представляющий `Path` нашего класса, используя метод `Paths.get()` и передавая путь двоичного класса в качестве аргумента. Теперь мы можем получить байты класса с `Files.readAllBytes(path)` . Поэтому мы создаем экземпляр `ByteClassLoader` и используем метод, который мы создали, `defineClass()` . У нас уже есть байты класса, но для вызова нашего метода нам также нужно имя класса, которое дается именем пакета (точка) канонического имени класса, в данном случае `stackoverflow.MyClass` .

```
Path path = Paths.get("MyClass.class");

ByteClassLoader loader = new ByteClassLoader();
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path);
```

Примечание . Метод `defineClass()` возвращает объект `Class<?>` . Вы можете сохранить его, если хотите.

Чтобы загрузить класс, мы просто вызываем `loadClass()` и передаем имя класса. Этот метод может генерировать `ClassNotFoundException` поэтому нам нужно использовать блок cath try

```
try{
    loader.loadClass("stackoverflow.MyClass");
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
```

Прочитайте Загрузчики классов онлайн: https://riptutorial.com/ru/java/topic/5443/загрузчики-классов

глава 71: Защищенные объекты

Синтаксис

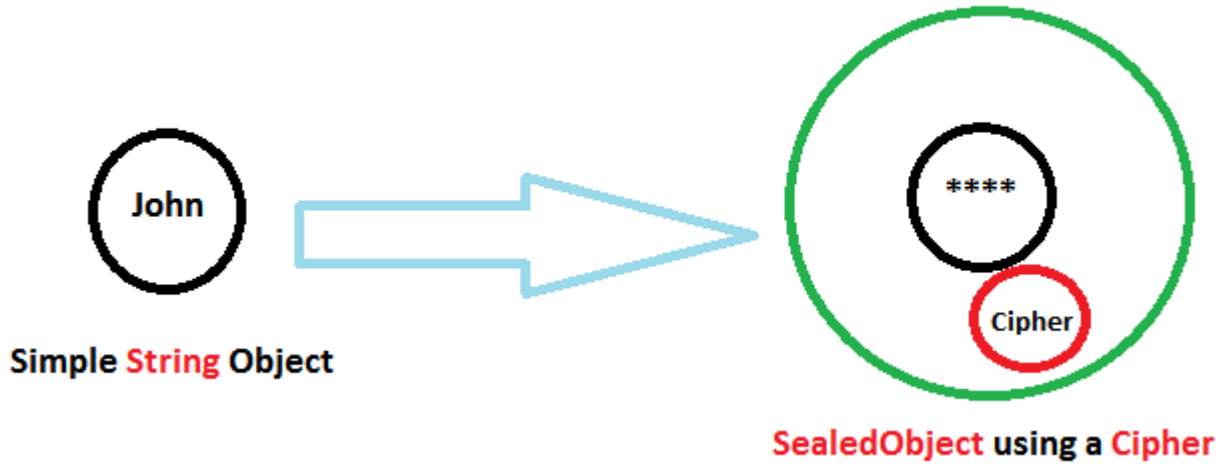
- SealedObject sealedObject = новый SealedObject (obj, cipher);
- SignedObject signedObject = новый SignedObject (obj, signedKey, signedEngine);

Examples

SealedObject (javax.crypto.SealedObject)

Этот класс позволяет программисту создать объект и защитить его конфиденциальность криптографическим алгоритмом.

Для любого объекта Serializable можно создать объект **SealedObject,** который инкапсулирует исходный объект в последовательном формате (т. **E.** «Глубокую копию») и печатает (шифрует) его сериализованное содержимое, используя криптографический алгоритм, такой как AES, DES, для защиты его конфиденциальность. Зашифрованный контент впоследствии может быть дешифрован (с соответствующим алгоритмом с использованием правильного ключа дешифрования) и де-сериализован, что даст исходный объект.

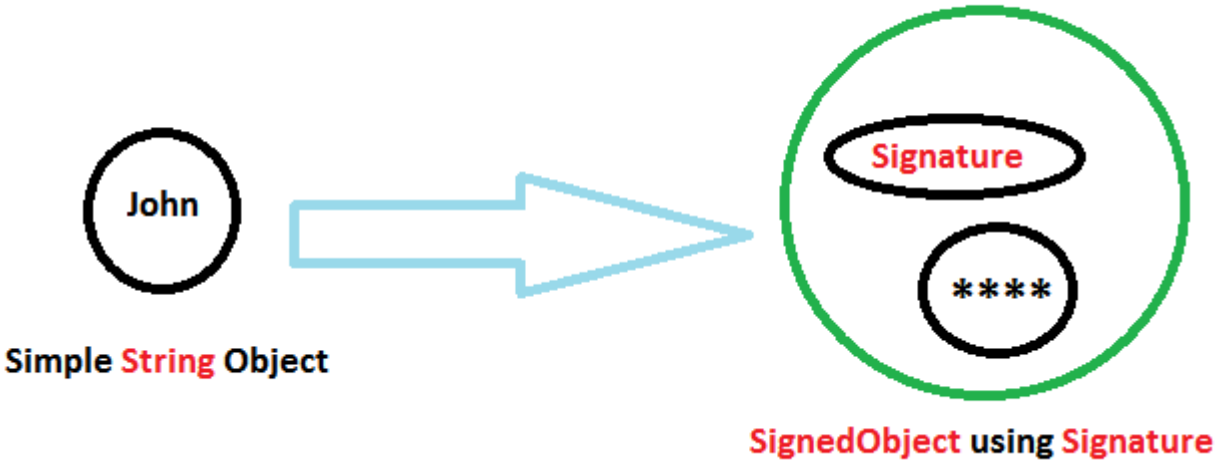



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject (java.security.SignedObject)

SignedObject - это класс с целью создания аутентичных объектов времени исполнения, целостность которых не может быть скомпрометирована без обнаружения.

Более конкретно, SignedObject содержит другой объект Serializable, объект (to-be-), подписанный и его подпись.




```
//Create a key
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// create a private key
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// create a Signature
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// create a simple object
Serializable obj = new String("John");
// sign our object
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```

Прочитайте Защищенные объекты онлайн: https://riptutorial.com/ru/java/topic/5528/защищенные-объекты

глава 72: Изменение байтового кода

Examples

Что такое Bytecode?

Bytecode - это набор инструкций, используемых JVM. Чтобы проиллюстрировать это, возьмем эту программу Hello World.

```
public static void main(String[] args){
    System.out.println("Hello World");
}
```

Это то, чем он превращается в компиляцию в байт-код.

```
public static main([Ljava/lang/String; args)V
    getstatic java/lang/System out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

Какова логика этого?

getstatic - возвращает значение статического поля класса. В этом случае *PrintStream «Out»* of *System* .

ldc - Вставьте константу в стек. В этом случае строка «Hello World»

invokevirtual - **вызывает** метод на загруженной ссылке в стеке и помещает результат в стек. Параметры метода также берутся из стека.

Ну, должно быть, правильнее?

Есть 255 опкодов, но не все они реализованы. Таблицу со всеми текущими кодами операций можно найти здесь: списки инструкций для байт-кода Java .

Как я могу писать / редактировать байт-код?

Существует несколько способов записи и редактирования байт-кода. Вы можете использовать компилятор, использовать библиотеку или использовать программу.

Для записи:

- жасмин
- Кракатау

Для редактирования:

- Библиотеки
  - KAK M
  - Javassist
  - BCEL - *не поддерживает Java 8+*
- инструменты
  - Bytecode-Viewer
  - JBytedit
  - reJ - *не поддерживает Java 8+*
  - JBE - *не поддерживает Java 8+*

Я хотел бы узнать больше о байтекоде!

Вероятно, определенная страница документации специально для байт-кода. Эта страница посвящена модификации байт-кода с использованием разных библиотек и инструментов.

Как редактировать файлы jar с помощью ASM

Сначала нужно загружать классы из банки. Мы будем использовать три метода для этого процесса:

- loadClasses (File)
- readJar (JarFile, JarEntry, Карта)
- getNode (байт [])

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2],
bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // This class doesn't have a valid magic
```

```
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}
```

С помощью этих методов загрузка и изменение файла jar становится простым делом изменения ClassNodes на карте. В этом примере мы заменим все строки в банке заглавными, используя Tree API.

```
File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
// Iterate ClassNodes
for (ClassNode cn : nodes.values()){
    // Iterate methods in class
    for (MethodNode mn : cn.methods){
        // Iterate instructions in method
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // If the instruction is loading a constant value
            if (ain.getOpcode() == Opcodes.LDC){
                // Cast current instruction to Ldc
                // If the constant is a string then capitalize it.
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}
```

Теперь, когда все строки ClassNode были изменены, нам нужно сохранить изменения. Чтобы сохранить изменения и иметь рабочий выход, нужно сделать несколько вещей:

- Экспорт ClassNodes в байты
- Загружать записи неклассического jar *(пример: Manifest.mf / другие двоичные*

ресурсы в банке) в виде байтов
- Сохранить все байты в новой банке

Из последней части выше мы создадим три метода.

- processNodes (Map <String, ClassNode> узлы)
- loadNonClasses (Файл jarFile)
- saveAsJar (Карта <String, byte []> outBytes, String fileName)

Использование:

```
Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");
```

Используемые методы:

```
static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass>
mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // Iterate nodes and add them to the map of <Class names , Class bytes>
    // Using Compute_Frames ensures that stack-frames will be re-calculated automatically
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // Iterate all entries
    while ((entry = jis.getNextEntry()) != null) {
        try {
            String name = entry.getName();
            if (!name.endsWith(".class") && !entry.isDirectory()) {
                // Apache Commons - byte[] toByteArray(InputStream input)
                //
                // Add each entry to the map <Entry name , Entry bytes>
                byte[] bytes = IOUtils.toByteArray(jis);
                entries.put(name, bytes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            jis.closeEntry();
        }
    }
    jis.close();
    return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
```

```
    try {
        // Create jar output stream
        JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
        // For each entry in the map, save the bytes
        for (String entry : outBytes.keySet()) {
            // Appent class names to class entries
            String ext = entry.contains(".") ? "" : ".class";
            out.putNextEntry(new ZipEntry(entry + ext));
            out.write(outBytes.get(entry));
            out.closeEntry();
        }
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Вот и все. Все изменения будут сохранены в «sample-edit.jar».

Как загрузить ClassNode в качестве класса

```
/**
 * Load a class by from a ClassNode
 *
 * @param cn
 *            ClassNode to load
 * @return
 */
public static Class<?> load(ClassNode cn) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
cw.toByteArray());
}

/**
 * Classloader that loads a class from bytes.
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }

    public Class<?> get(String name, byte[] bytes) {
        Class<?> c = defineClass(name, bytes, 0, bytes.length);
        resolveClass(c);
        return c;
    }
}
```

Как переименовать классы в файле jar

```
public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
```

```
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings)
{
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name,
cw.toByteArray());
    }
    return out;
}
```

SimpleRemapper - это существующий класс в библиотеке ASM. Однако он позволяет изменять имена классов. Если вы хотите переименовать поля и методы, вы должны создать свою собственную реализацию класса Remapper.

Javassist Basic

Javassist - это инструментальная библиотека байт-кода, которая позволяет вам модифицировать байт-код, вводящий Java-код, который будет преобразован в байт-код Javassist и добавлен к инструментальному классу / методу во время выполнения.

Давайте напишем первый трансформатор, который фактически возьмет гипотетический класс «com.my.to.be.instrumented.MyClass» и добавит к инструкциям каждого метода вызов журнала.

```
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
        ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;

        // into the transformer will arrive every class loaded so we filter
        // to match only what we need
        if (className.equals("com/my/to/be/instrumented/MyClass")) {

            try {
                // retrive default Javassist class pool
                ClassPool cp = ClassPool.getDefault();
```

```
                // get from the class pool our class with this qualified name
                CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
                // get all the methods of the retrieved class
                CtMethod[] methods = cc.getDeclaredMethods()
                for(CtMethod meth : methods) {
                    // The instrumentation code to be returned and injected
                    final StringBuffer buffer = new StringBuffer();
                    String name = meth.getName();
                    // just print into the buffer a log for example
                    buffer.append("System.out.println(\"Method " + name + " executed\" );");
                    meth.insertBefore(buffer.toString())
                }
                // create the byteclode of the class
                byteCode = cc.toBytecode();
                // remove the CtClass from the ClassPool
                cc.detach();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        return byteCode;
    }
}
```

Теперь, чтобы использовать этот трансформатор (чтобы наша JVM вызывала преобразование метода для каждого класса во время загрузки), нам нужно добавить этот инструмент с помощью агента:

```
import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}
```

Последний шаг, чтобы начать наш первый эксперимент с инструментами, - это фактически зарегистрировать этот класс агента для запуска JVM-машины. Самый простой способ сделать это - зарегистрировать его с помощью опции в команде оболочки:

```
java -javaagent:myAgent.jar MyJavaApplication
```

Поскольку мы видим, что проект agent / transformer добавлен как банка к исполнению любого приложения с именем MyJavaApplication, которое должно содержать класс с именем «com.my.to.be.instrumented.MyClass», чтобы фактически выполнить наш введенный код.

Прочитайте Изменение байтового кода онлайн: https://riptutorial.com/ru/java/topic/3747/изменение-байтового-кода

глава 73: Инкапсуляция

Вступление

Представьте, что у вас был класс с довольно важными переменными, и они были установлены (другими программистами из их кода) на неприемлемые значения. Их код привел ошибки в ваш код. В качестве решения, в ООП, вы позволяете изменять состояние объекта (хранящегося в его переменных) только с помощью методов. Скрытие состояния объекта и обеспечение всего взаимодействия с помощью методов объектов известно как инкапсуляция данных.

замечания

Гораздо проще начать с маркировки переменной `private` и разоблачить ее, если необходимо, чтобы скрыть уже `public` переменную.

Существует одно исключение, когда инкапсуляция может оказаться нецелесообразной: «немые» структуры данных (классы, единственной целью которых является сохранение переменных).

```
public class DumbData {
    public String name;
    public int timeStamp;
    public int value;
}
```

В этом случае интерфейс класса - это данные, которые он хранит.

Обратите внимание, что переменные, помеченные как `final` могут быть помечены как `public` не нарушая инкапсуляцию, поскольку они не могут быть изменены после установки.

Examples

Инкапсуляция для сохранения инвариантов

Есть две части класса: интерфейс и реализация.

Интерфейс представляет собой открытую функциональность класса. Его общедоступные методы и переменные являются частью интерфейса.

Реализация - это внутренние действия класса. Другие классы не должны знать о реализации класса.

Инкапсуляция относится к практике скрытия реализации класса от всех пользователей

этого класса. Это позволяет классу делать предположения о его внутреннем состоянии.

Например, возьмите этот класс, представляющий угол:

```
public class Angle {

    private double angleInDegrees;
    private double angleInRadians;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        a.angleInRadians = Math.PI*degrees/180;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInRadians = radians;
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInRadians;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
        this.angleInRadians = Math.PI*degrees/180;
    }

    public void setRadians(double radians){
        this.angleInRadians = radians;
        this.angleInDegrees = radians*180/Math.PI;
    }
    private Angle(){}
}
```

Этот класс основан на базовом предположении (или *инварианте*): **angleInDegrees и angleInRadians всегда синхронизированы** . Если бы члены класса были общедоступными, не было бы никаких гарантий того, что оба представления углов будут скоррелированы.

Инкапсуляция для уменьшения сцепления

Инкапсуляция позволяет делать внутренние изменения в классе, не затрагивая какой-либо код, вызывающий класс. Это уменьшает *связь* , или какой-либо данный класс полагается на реализацию другого класса.

Например, давайте изменим реализацию класса Angle из предыдущего примера:

```
public class Angle {

    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians){
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}
```

Реализация этого класса изменилась так, что он сохраняет только одно представление угла и вычисляет другой угол, когда это необходимо.

Однако **реализация изменилась, но интерфейс этого не сделал** . Если вызывающий класс полагался на доступ к методу angleInRadians, его нужно было бы изменить, чтобы использовать новую версию `Angle` . Вызывающие классы не должны заботиться о внутреннем представлении класса.

Прочитайте Инкапсуляция онлайн: https://riptutorial.com/ru/java/topic/1295/инкапсуляция

глава 74: Интерфейс Dequeue

Вступление

Deque - линейный набор, который поддерживает вставку и удаление элементов на обоих концах.

Имя deque не подходит для «двойной очереди» и обычно произносится как «колода».

В большинстве реализаций Deque не установлены фиксированные ограничения на количество элементов, которые они могут содержать, но этот интерфейс поддерживает ограничения, ограниченные пропускной способностью, а также те, у которых нет ограничения по фиксированному размеру.

Интерфейс Deque является более богатым абстрактным типом данных, чем стек и очередь, поскольку он одновременно выполняет как стеки, так и очереди

замечания

Дженерики могут использоваться с Deque.

```
Deque<Object> deque = new LinkedList<Object>();
```

Когда deque используется как очередь, результаты FIFO (First-In-First-Out) приводят к результату.

Deques также может использоваться как стеки LIFO (Last-In-First-Out).

Дополнительные сведения о методах см. В этой документации.

Examples

Добавление элементов в Deque

```
Deque deque = new LinkedList();

//Adding element at tail
deque.add("Item1");

//Adding element at head
deque.addFirst("Item2");

//Adding element at tail
deque.addLast("Item3");
```

Удаление элементов из Deque

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();

//Retrieves and removes the last element of this deque.
Object lastItem = deque.removeLast();
```

Извлечение элемента без удаления

```
//Retrieves, but does not remove, the head of the queue represented by this deque
Object headItem = deque.element();

//Retrieves, but does not remove, the first element of this deque.
Object firstItem = deque.getFirst();

//Retrieves, but does not remove, the last element of this deque.
Object lastItem  = deque.getLast();
```

Итерация через Deque

```
//Using Iterator
Iterator iterator = deque.iterator();
while(iterator.hasNext(){
  String Item = (String) iterator.next();
}

//Using For Loop
for(Object object : deque) {
    String Item = (String) object;
}
```

Прочитайте Интерфейс Dequeue онлайн: https://riptutorial.com/ru/java/topic/10156/интерфейс-dequeue

глава 75: Интерфейс Java Native

параметры

параметр	подробности
JNIEnv	Указатель на среду JNI
jobject	Объект , который ссылается на не- `static native` метода
JClass	Класс, который вызвал `static native` метод

замечания

Для настройки JNI требуется как Java, так и собственный компилятор. В зависимости от IDE и ОС требуется некоторая настройка. Руководство для Eclipse можно найти здесь . Полный учебник можно найти здесь .

Это шаги для настройки связи Java-C ++ в окнах:

- Скомпилируйте исходные файлы Java (`.java`) в классы (`.class`) с помощью `javac` .
- Создайте файлы заголовка (`.h`) из классов Java, содержащих `native` методы, используя `javah` . Эти файлы «инструктируют» собственный код, какие методы он отвечает за реализацию.
- Включите файлы заголовков (`#include`) в исходные файлы C ++ (`.cpp`), реализующие `native` методы.
- Скомпилируйте исходные файлы C ++ и создайте библиотеку (`.dll`). Эта библиотека содержит реализацию собственного кода.
- Укажите путь к библиотеке (`-Djava.library.path`) и загрузите его в исходный файл Java (`System.loadLibrary(...)`).

Обратные вызовы (вызов методов Java из собственного кода) требуют указания дескриптора метода. Если дескриптор неверен, возникает ошибка времени выполнения. Из-за этого полезно иметь дескрипторы для нас, это можно сделать с помощью `javap -s` .

Examples

Вызов методов C ++ из Java

Статические и членские методы в Java могут быть помечены как *родные,* чтобы указать, что их реализация должна быть найдена в файле общей библиотеки. После выполнения собственного метода JVM ищет соответствующую функцию в загружаемых библиотеках

(см. Загрузка собственных библиотек), используя простую схему переключения имен, выполняет преобразование аргументов и установку стека, а затем передает управление собственному коду.

Код Java

```
/*** com/example/jni/JNIJava.java **/

package com.example.jni;

public class JNIJava {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    // Obviously, native methods may not have a body defined in Java
    public native void printString(String name);
    public static native double average(int[] nums);

    public static void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("Invoked C++ 'printString' from Java");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("Got result from C++ 'average': " + d);
    }
}
```

Код C ++

Заголовочные файлы, содержащие декларации `javah` функций, должны быть сгенерированы с использованием инструмента `javah` для целевых классов. Выполнение следующей команды в каталоге сборки:

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... создает следующий заголовочный файл (*комментарии для краткости*):

```
// com_example_jni_JNIJava.hpp

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h> // The JNI API declarations

#ifndef _Included_com_example_jni_JNIJava
#define _Included_com_example_jni_JNIJava
#ifdef __cplusplus
extern "C" { // This is absolutely required if using a C++ compiler
#endif

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
  (JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
```

```
  (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
}
#endif
#endif
```

Вот пример реализации:

```
// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis,
jstring string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis,
jintArray intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}
```

Выход

Запуск класса example выше дает следующий результат:

> Вызывается C ++ 'printString' из Java
> Получил результат от C ++ «средний»: 3.4

Вызов методов Java из C ++ (callback)

Вызов метода Java из собственного кода - это двухэтапный процесс:

1. получить указатель метода с `GetMethodID` , используя имя и дескриптор метода;
2. вызовите одну из функций `Call*Method` перечисленных здесь .

Код Java

```
/*** com.example.jni.JNIJavaCallback.java ***/

package com.example.jni;

public class JNIJavaCallback {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    public static void main(String[] args) {
        new JNIJavaCallback().callback();
    }

    public native void callback();

    public static void printNum(int i) {
        System.out.println("Got int from C++: " + i);
    }

    public void printFloat(float i) {
        System.out.println("Got float from C++: " + i);
    }
}
```

Код C ++

```
// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject
jthis) {
    jclass thisClass = env->GetObjectClass(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}
```

Выход

Поплавок с C ++: 5.221
Получил int из C ++: 17

Получение дескриптора

Дескрипторы (или *сигнатуры внутреннего типа*) получают с помощью **javap-** программы в скомпилированном файле `.class` . Вот результат работы `javap -p -s com.example.jni.JNIJavaCallback` :

```
Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
  static {};
    descriptor: ()V

  public com.example.jni.JNIJavaCallback();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

  public native void callback();
    descriptor: ()V

  public static void printNum(int);
    descriptor: (I)V  // <---- Needed

  public void printFloat(float);
    descriptor: (F)V  // <---- Needed
}
```

Загрузка собственных библиотек

Общая идиома для загрузки файлов общей библиотеки в Java:

```
public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Вызовы в `System.loadLibrary` почти всегда статичны, чтобы происходить во время загрузки класса, гарантируя, что собственный метод не может быть запущен до того, как загружена общая библиотека. Однако возможно следующее:

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

Это позволяет отложить загрузку общей библиотеки до тех пор, пока это не будет

необходимо, но требует дополнительной осторожности, чтобы избежать `java.lang.UnsatisfiedLinkError` s.

Поиск целевого файла

Поиск общих файлов библиотеки осуществляется в путях, определенных системным свойством `java.library.path` , которые могут быть переопределены с помощью аргумента `-Djava.library.path=` JVM во время выполнения:

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Следите за разделителями системных путей: например, Windows использует ; а не :

Обратите внимание, что `System.loadLibrary` разрешает имена файлов библиотек зависимым от платформы образом: фрагмент кода выше ожидает файл с именем `libExample.so` в Linux и `Example.dll` в Windows.

Альтернативой `System.loadLibrary` является `System.load(String)` , которая переносит полный путь к файлу общей библиотеки, обходя поиск в `java.library.path` :

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Прочитайте Интерфейс Java Native онлайн: https://riptutorial.com/ru/java/topic/168/интерфейс-java-native

глава 76: Интерфейс инструмента JVM

замечания

Интерфейс инструмента JVM TM

Версия 1.2

http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

Examples

Итерация над объектами, доступными из объекта (Heap 1.0)

```
#include <vector>
#include <string>


#include "agent_util.hpp"
//this file can be found in Java SE Development Kit 8u101 Demos and Samples
//see http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-
demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
*   Struct used for jvmti->SetTag(object, <pointer to tag>);
*   http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
*
*/
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
* Utility function: jlong -> Tag*
*/
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
* Utility function: Tag* -> jlong
*/
static jlong tagToPointer(Tag* tag)
```

```
{
    return (jlong)(ptrdiff_t)(void*)tag;
}

/*
*  Heap 1.0 Callback
*  http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
*/
static jvmtiIterationControl JNICALL heabObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
    //iterate only over reference field
    if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
    {
        return JVMTI_ITERATION_IGNORE;
    }
    auto tag_ptr_list = (std::vector<jlong>*)(ptrdiff_t)(void*)user_data;
    //create and assign tag
    auto t = pointerToTag(*tag_ptr);
    t->referrer_tag = referrer_tag;
    t->size = size;
    *tag_ptr = tagToPointer(t);
    //collect tag
    (*tag_ptr_list).push_back(*tag_ptr);

    return JVMTI_ITERATION_CONTINUE;
}

/*
*  Main function for demonstration of Iterate Over Objects Reachable From Object
*
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject

*
*/
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
    std::vector<jlong> tag_ptr_list;

    auto t = new Tag();
    jvmti->SetTag(object, tagToPointer(t));
    tag_ptr_list.push_back(tagToPointer(t));

    stdout_message("tag list size before call callback:  %d\n", tag_ptr_list.size());
    /*
    *    Call Callback for every reachable object reference
    *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject

    */
    jvmti->IterateOverObjectsReachableFromObject(object, &heabObjectReferencesCallback,
(void*)&tag_ptr_list);
    stdout_message("tag list size after call callback:  %d\n", tag_ptr_list.size());

    if (tag_ptr_list.size() > 0)
```

```
    {
        jint found_count = 0;
        jlong* tags = &tag_ptr_list[0];
        jobject* found_objects;
        jlong* found_tags;

        /*
        *    collect all tagged object (via *tag_ptr = pointer to tag )
        *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
        */
        jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
&found_tags);
        stdout_message("found %d objects\n", found_count);

        for (auto i = 0; i < found_count; ++i)
        {
            jobject found_object = found_objects[i];

            char* classSignature;
            jclass found_object_class = env->GetObjectClass(found_object);
            /*
            *    Get string representation of found_object_class
            *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
            */
            jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

            jint hashCode;
            /*
            *    Getting hash code for found_object
            *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
            */
            jvmti->GetObjectHashCode(found_object, &hashCode);

            //save all it in Tag
            Tag* t = pointerToTag(found_tags[i]);
            t->classSignature = classSignature;
            t->hashCode = hashCode;
        }

        //print all saved information
        for (auto i = 0; i < found_count; ++i)
        {
            auto t = pointerToTag(found_tags[i]);
            auto rt = pointerToTag(t->referrer_tag);

            if (t->referrer_tag != 0)
            {
                stdout_message("referrer object %s#%d --> object %s#%d (size: %2d)\n",
                    rt->classSignature, rt->hashCode, t->classSignature, t->hashCode, t-
>size);
            }
        }
    }
}
```

Получить среду JVMTI

Внутри метода Agent_OnLoad:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Пример инициализации внутри метода Agent_OnLoad

```
/* Callback for JVMTI_EVENT_VM_INIT */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
    jint runtime_version;
    jvmti->GetVersionNumber(&runtime_version);
    stdout_message("JVMTI Version: %d\n", runtime_verision);
}

/* Agent_OnLoad() is called first, we prepare for a VM_INIT event here. */
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM* vm, char* options, void* reserved)
{
    jint rc;
    jvmtiEventCallbacks callbacks;
    jvmtiCapabilities capabilities;
    jvmtiEnv* jvmti;

    /* Get JVMTI environment */
    rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
    if (rc != JNI_OK)
    {
        return -1;
    }

    /* Immediately after getting the jvmtiEnv* we need to ask for the
    *  capabilities this agent will need.
    */
    jvmti->GetCapabilities(&capabilities);
    capabilities.can_tag_objects = 1;
    jvmti->AddCapabilities(&capabilities);

    /* Set callbacks and enable event notifications */
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.VMInit = &vm_init;

    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
    jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

    return JNI_OK;
}
```

Прочитайте Интерфейс инструмента JVM онлайн: https://riptutorial.com/ru/java/topic/3316/интерфейс-инструмента-jvm

глава 77: Интерфейсы

Вступление

Интерфейс является ссылочным типом, похожим на класс, который может быть объявлен с помощью ключевого слова `interface` . Интерфейсы могут содержать только константы, сигнатуры методов, методы по умолчанию, статические методы и вложенные типы. Органы метода существуют только для методов по умолчанию и статических методов. Подобно абстрактным классам, интерфейсы не могут быть созданы - они могут быть реализованы только классами или расширены другими интерфейсами. Интерфейс является распространенным способом достижения полной абстракции в Java.

Синтаксис

- открытый интерфейс Foo {void foo (); / * любые другие методы * /}
- открытый интерфейс Foo1 расширяет Foo {void bar (); / * любые другие методы * /}
- public class Foo2 реализует Foo, Foo1 {/ * реализацию Foo и Foo1 * /}

Examples

Объявление и реализация интерфейса

Объявление интерфейса с использованием ключевого слова `interface` :

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Переопределить аннотацию

```
@Override
public String getSound() {
    // Code goes here...
}
```

Это заставляет компилятор проверить, что мы переопределяем и не позволяет программе определять новый метод или испортить подпись метода.

Интерфейсы реализуются с помощью `implements` ключевого слова.

```
public class Cat implements Animal {

    @Override
    public String getSound() {
```

```
        return "meow";
    }
}

public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

В этом примере классы `Cat` и `Dog` **должны** определять метод `getSound()` поскольку методы интерфейса являются абстрактно абстрактными (за исключением методов по умолчанию).

Использование интерфейсов

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Реализация нескольких интерфейсов

Класс Java может реализовывать несколько интерфейсов.

```
public interface NoiseMaker {
    String noise = "Making Noise"; // interface variables are public static final by default

    String makeNoise(); //interface methods are public abstract by default
}

public interface FoodEater {
    void eat(Food food);
}

public class Cat implements NoiseMaker, FoodEater {
    @Override
    public String makeNoise() {
        return "meow";
    }

    @Override
    public void eat(Food food) {
        System.out.println("meows appreciatively");
    }
}
```

Обратите внимание, как класс `Cat` **должен** реализовывать наследованные `abstract` методы в обоих интерфейсах. Кроме того, обратите внимание, как класс может практически реализовать столько интерфейсов, сколько необходимо (предел **65535** из-за ограничения JVM).

```
NoiseMaker noiseMaker = new Cat(); // Valid
FoodEater foodEater = new Cat(); // Valid
Cat cat = new Cat(); // valid

Cat invalid1 = new NoiseMaker(); // Invalid
Cat invalid2 = new FoodEater(); // Invalid
```

Замечания:

1. Все переменные, объявленные в интерфейсе, являются `public static final`
2. Все методы, объявленные в методах интерфейса, являются `public abstract` (этот оператор действителен только через Java 7. Из Java 8 вам разрешено иметь методы в интерфейсе, которые не обязательно должны быть абстрактными, такие методы известны как методы по умолчанию)
3. Интерфейсы не могут быть объявлены `final`
4. Если несколько интерфейсов объявляют метод, имеющий идентичную подпись, то он эффективно обрабатывается только как один метод, и вы не можете отличить, от какого метода интерфейса
5. Соответствующий файл **InterfaceName.class** будет создан для каждого интерфейса, после компиляции

Расширение интерфейса

Интерфейс может расширять другой интерфейс с помощью ключевого слова `extends` .

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

Теперь класс, реализующий `ExtendedResourceService` , должен будет реализовать как `getResource()` и `updateResource()` .

Расширение нескольких интерфейсов

В отличие от классов ключевое слово `extends` может использоваться для расширения нескольких интерфейсов (разделенных запятыми), что позволяет сочетать интерфейсы с новым интерфейсом

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}
```

```
public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

В этом случае для класса, реализующего `ExtendedResourceService` , потребуется реализовать `getResource()` , `getAlternateResource()` и `updateResource()` .

Использование интерфейсов с универсалами

Предположим, вы хотите определить интерфейс, который позволяет публиковать / потреблять данные для разных типов каналов (например, AMQP, JMS и т. Д.), Но вы хотите, чтобы их можно было отключить ...

Давайте определим базовый интерфейс ввода-вывода, который можно повторно использовать в нескольких реализациях:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Теперь я могу создать экземпляр этого интерфейса, но поскольку у нас нет стандартных реализаций для этих методов, ему понадобится реализация, когда мы его создадим:

```
    IO<String, String> mockIO = new IO<String, String>() {

        private String channel = "somechannel";

        @Override
        public void publish(String data) {
            System.out.println("Publishing " + data + " to " + channel);
        }

        @Override
        public String consume() {
            System.out.println("Consuming from " + channel);
            return "some useful data";
        }

        @Override
        public String RPCSubmit(String data) {
            return "received " + data + " just now ";
        }

    };

    mockIO.consume(); // prints: Consuming from somechannel
    mockIO.publish("TestData"); // Publishing TestData to somechannel
    System.out.println(mockIO.RPCSubmit("TestData")); // received TestData just now
```

Мы также можем сделать что-то более полезное с этим интерфейсом, допустим, мы хотим использовать его для переноса некоторых основных функций RabbitMQ:

```
public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;

    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }

    @Override
    public String RPCSubmit(String data) {
        return rabbit.rpcPublish(exchange, queue, data);
    }

}
```

Предположим, я хочу использовать этот интерфейс ввода-вывода в качестве способа подсчета посещений моего веб-сайта с момента последнего перезапуска системы, а затем с возможностью отображать общее количество посещений - вы можете сделать что-то вроде этого:

```
import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }

}
```

Теперь давайте использовать VisitCounter:

```
    VisitCounter counter = new VisitCounter();

    // just had 4 visits, yay
    counter.publish(4);
    // just had another visit, yay
    counter.publish(1);

    // get data for stats counter
    System.out.println(counter.consume()); // prints 5

    // show data for stats counter page, but include that as a page view
    System.out.println(counter.RPCSubmit(1)); // prints 6
```

При реализации нескольких интерфейсов вы не можете реализовать один и тот же интерфейс дважды. Это также относится к общим интерфейсам. Таким образом, следующий код недействителен и приведет к ошибке компиляции:

```
interface Printer<T> {
    void print(T value);
}

// Invalid!
class SystemPrinter implements Printer<Double>, Printer<Integer> {
    @Override public void print(Double d){ System.out.println("Decimal: " + d); }
    @Override public void print(Integer i){ System.out.println("Discrete: " + i); }
}
```

Полезность интерфейсов

Во многих случаях интерфейсы могут быть чрезвычайно полезными. Например, скажем, что у вас есть список животных, и вы хотели пройти через список, каждый из которых печатает звук, который они создают.

```
{cat, dog, bird}
```

Один из способов сделать это - использовать интерфейсы. Это позволило бы вызвать тот же метод для всех классов

```
public interface Animal {
    public String getSound();
}
```

Любой класс, который `implements Animal` также должен иметь метод `getSound()` , но все они могут иметь разные реализации

```
public class Dog implements Animal {
    public String getSound() {
        return "Woof";
```

```
    }
}

public class Cat implements Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Bird implements Animal{
    public String getSound() {
        return "Chirp";
    }
}
```

Теперь у нас есть три разных класса, каждый из которых имеет метод `getSound()` . Поскольку все эти классы `implement` интерфейс `Animal` , который объявляет метод `getSound()` , любой экземпляр `Animal` может иметь на нем `getSound()`

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Поскольку каждый из них является `Animal` , мы могли бы даже поместить животных в список, пропустить их и распечатать их звуки

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Поскольку порядок массива - `Dog` , `Cat` , а затем `Bird` , *«Woof Meow Chirp»* будет напечатан на консоли.

Интерфейсы также могут использоваться как возвращаемое значение для функций. Например, возвращение `Dog` если вход *«собака»* , `Cat` если вход *«cat»* , и « `Bird` если это *« птица»* , а затем печать звука этого животного может быть выполнена с использованием

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new Dog();
        case "cat":
            return new Cat();
        case "bird":
            return new Bird();
        default:
            return null;
    }
}
```

```
public String getAnimalSoundByName(String name){
    Animal animal = getAnimalByName(name);
    if (animal == null) {
        return null;
    } else {
        return animal.getSound();
    }
}

String dogSound = getAnimalSoundByName("dog"); // "Woof"
String catSound = getAnimalSoundByName("cat"); // "Meow"
String birdSound = getAnimalSoundByName("bird"); // "Chirp"
String lightbulbSound = getAnimalSoundByName("lightbulb"); // null
```

Интерфейсы также полезны для расширяемости, поскольку, если вы хотите добавить новый тип `Animal` , вам не нужно ничего менять с помощью операций, которые вы выполняете на них.

Внедрение интерфейсов в абстрактном классе

Метод, определенный в `interface` , по умолчанию является `public abstract` . Когда `abstract class` реализует `interface` , любые методы, которые определены в `interface` , не должны быть реализованы `abstract class` . Это связано с тем, что `class` , объявленный `abstract` может содержать объявления абстрактных методов. Поэтому первым конкретным подклассом является реализация любых `abstract` методов, унаследованных от любых интерфейсов и / или `abstract class` .

```
public interface NoiseMaker {
    void makeNoise();
}

public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}
```

Начиная с Java 8, `interface` может объявлять по `default` реализации методов, что означает, что метод не будет `abstract` , поэтому любые конкретные подклассы не будут вынуждены реализовать этот метод, но наследуют реализацию по `default` если не переопределены.

Методы по умолчанию

Представленные в Java 8 методы по умолчанию - это способ указания реализации внутри интерфейса. Это можно использовать, чтобы избежать типичного класса «Base» или « Abstract», предоставляя частичную реализацию интерфейса и ограничивая иерархию подклассов.

Реализация шаблона наблюдателя

Например, можно реализовать шаблон Observer-Listener непосредственно в интерфейсе, предоставляя большую гибкость реализующим классам.

```
interface Observer {
    void onAction(String a);
}

interface Observable{
    public abstract List<Observer> getObservers();

    public default void addObserver(Observer o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}
```

Теперь любой класс может быть выполнен «Наблюдаемым», просто реализуя интерфейс Observable, будучи свободным, чтобы быть частью другой иерархии классов.

```
abstract class Worker{
    public abstract void work();
}

public class MyWorker extends Worker implements Observable {

    private List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    public List<Observer> getObservers() {
        return myObservers;
    }

    @Override
    public void work(){
        notify("Started work");

        // Code goes here...

        notify("Completed work");
```

```
    }

    public static void main(String[] args) {
        MyWorker w = new MyWorker();

        w.addListener(new Observer() {
            @Override
            public void onAction(String a) {
                System.out.println(a + " (" + new Date() + ")");
            }
        });

        w.work();
    }
}
```

Проблема с алмазами

Компилятор в Java 8 знает о проблеме с алмазом, которая возникает, когда класс реализует интерфейсы, содержащие метод с той же сигнатурой.

Для его решения класс реализации должен переопределить совместно используемый метод и обеспечить его собственную реализацию.

```
interface InterfaceA {
    public default String getName(){
        return "a";
    }
}

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //Must provide its own implementation
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );                  // Prints "ab"
        System.out.println( ((InterfaceA)c).getName() );    // Prints "ab"
        System.out.println( ((InterfaceB)c).getName() );    // Prints "ab"
    }
}
```

По-прежнему существует проблема с методами с тем же именем и параметрами с разными типами возвращаемых данных, которые не будут компилироваться.

Использовать методы по умолчанию для решения проблем совместимости

Реализация метода по умолчанию очень удобна, если метод добавлен к интерфейсу в существующей системе, где интерфейсы используются несколькими классами.

Чтобы не разбивать всю систему, вы можете предоставить реализацию метода по умолчанию при добавлении метода к интерфейсу. Таким образом, система все еще будет компилироваться, и фактические реализации могут быть сделаны шаг за шагом.

Дополнительные сведения см. В разделе « Способы по умолчанию» .

Модификаторы в интерфейсах

Руководство по стилю Java Java:

> Модификаторы не должны выписываться, если они неявные.

(См. Модификаторы в стандартном стандартном коде Oracle для контекста и ссылку на фактический документ Oracle.)

Это руководство по стилю применяется, в частности, к интерфейсам. Рассмотрим следующий фрагмент кода:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

переменные

Все переменные интерфейса являются неявно *константами* с неявным `public` (доступным для всех), `static` (доступны по имени интерфейса) и `final` (должны быть инициализированы во время объявления) модификаторами:

```
public static final int VARIABLE = 0;
```

методы

1. Все методы, которые *не обеспечивают реализацию,* являются неявно `public` и `abstract` .

```
~~public abstract~~ void method();
```

Java SE 8

2. Все методы со `static` или модификатором по `default` *должны обеспечивать реализацию* и неявно `public` .

```
~~public~~ static void staticMethod() { ... }
```

После того, как все вышеуказанные изменения будут применены, мы получим следующее:

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Усиление параметров ограниченного типа

Параметры ограниченного типа позволяют вам устанавливать ограничения на аргументы общего типа:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

Но параметр типа может привязываться только к одному типу класса.

Тип интерфейса может быть привязан к типу, который уже имел привязку. Это достигается с помощью символа `&` :

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {
```

```
}
```

Это усиливает привязку, что потенциально требует аргументов типа для получения нескольких типов.

Несколько типов интерфейсов могут быть привязаны к параметру типа:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

Но следует использовать с осторожностью. Связывание нескольких интерфейсов обычно является признаком запаха кода , предполагая, что должен быть создан новый тип, который действует как адаптер для других типов:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {

}
```

Прочитайте Интерфейсы онлайн: https://riptutorial.com/ru/java/topic/102/интерфейсы

глава 78: Исключения и обработка исключений

Вступление

Объекты типа `Throwable` и его подтипы могут быть отправлены в стек с ключевым словом `throw` и пойманы с помощью `try…catch` statement.

Синтаксис

- void someMethod () throws SomeException {} // объявляет метод, заставляет вызывающие вызовы метода catch, если SomeException - это проверенный тип исключения
- пытаться {

```
someMethod(); //code that might throw an exception
```

  }
- catch (SomeException e) {

```
 System.out.println("SomeException was thrown!"); //code that will run if certain
exception (SomeException) is thrown
```

  }
- в конце концов {

```
 //code that will always run, whether try block finishes or not
```

  }

Examples

Захват исключения с помощью try-catch

Исключение можно поймать и обработать с помощью инструкции `try...catch` . (На самом деле утверждения `try` принимают другие формы, как описано в других примерах о `try...catch...finally` и `try-with-resources` .)

Уловка с одним блоком catch

Самая простая форма выглядит так:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

Поведение простого `try...catch` выглядит следующим образом:

- Выполняются операторы в блоке `try` .
- Если исключение не вызывается операторами в блоке `try` , тогда управление переходит к следующему утверждению после `try...catch` .
- Если в блоке `try` выбрано исключение.
    - Объект исключения проверяется, является ли он экземпляром `SomeException` или подтипом.
    - Если это так, то `catch` блок будет *поймать* исключение:
        - Переменная `e` привязана к объекту исключения.
        - Выполняется код в блоке `catch` .
        - Если этот код генерирует исключение, то вновь созданное исключение распространяется вместо исходного.
        - В противном случае управление переходит к следующему утверждению после `try...catch` .
    - Если это не так, исходное исключение продолжает распространяться.

Попытка с несколькими уловами

У `try...catch` также может быть несколько блоков `catch` . Например:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

Если существует несколько блоков `catch` , они проверяются по одному за раз, начиная с первого, пока не будет найдено совпадение для исключения. Соответствующий обработчик выполняется (как указано выше), а затем управление передается следующему оператору после инструкции `try...catch` . Блоки `catch` после того, который совпадает, всегда пропускаются, *даже если код обработчика генерирует исключение* .

Стратегия совпадения «сверху вниз» имеет последствия для случаев, когда исключения в блоках `catch` не пересекаются. Например:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

Этот фрагмент кода выводит «Исключение», а не «Исключение RuntimeException». Поскольку `RuntimeException` является подтипом `Exception` , первый (более общий) `catch` будет сопоставлен. Второй (более конкретный) `catch` никогда не будет выполнен.

Урок, который следует извлечь из этого, состоит в том, что сначала должны появиться наиболее специфические блоки `catch` (с точки зрения типов исключений), а самые общие из них должны быть последними. (Некоторые компиляторы Java предупреждают вас, если `catch` никогда не будет выполнена, но это не ошибка компиляции.)

Блоки с несколькими исключениями

Java SE 7

Начиная с Java SE 7, один блок `catch` может обрабатывать список несвязанных исключений. Отображается тип исключения, разделенный символом вертикальной полосы (`|`). Например:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

Поведение множественного исключения - это простое расширение для случая с одним исключением. `catch` совпадает, если выбранное исключение совпадает (хотя бы) с одним из перечисленных исключений.

В спецификации есть некоторые дополнительные тонкости. Тип `e` является синтетическим *объединением* типов исключений в списке. Когда используется значение `e` , его статический тип является наименее распространенным супертипом объединения типов. Однако, если `e` забрасывается внутри блока `catch` , типы исключений, которые выбрасываются, являются типами объединения. Например:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
```

```
        report(e);
        throw e;
    }
```

В приведенном выше `IOException` и `SQLException` проверяются исключениями, наименее распространенным супертипом которых является `Exception` . Это означает, что метод `report` должен соответствовать `report(Exception)` . Однако компилятор знает, что `throw` может вызывать только `IOException` или `SQLException` . Таким образом, `method` может быть объявлен как `throws Exception` `throws IOException, SQLException` а не `throws Exception` . (Что хорошо: см. Pitfall - Throwing Throwable, Exception, Error или RuntimeException .)

Выброс исключения

В следующем примере показаны основы исключения исключения:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

Исключение составляет 3-я строка. Это утверждение можно разбить на две части:

- `new IllegalArgumentException(...)` создает экземпляр класса `IllegalArgumentException` с сообщением, описывающим ошибку, о которой сообщается исключение.
- `throw ...` затем бросает объект исключения.

Когда исключение выбрасывается, это приводит к тому, что закрывающие операторы *прерываются аномально* до тех пор, пока не будет *обработано* исключение. Это описано в других примерах.

Хорошей практикой является как создание, так и создание объекта исключения в одном выражении, как показано выше. Также хорошей практикой является включение значимого сообщения об ошибке в исключение, чтобы помочь программисту понять причину проблемы. Однако это не обязательно сообщение, которое вы должны показывать конечному пользователю. (Для начала Java не имеет прямой поддержки для интернационализации сообщений об исключениях.)

Есть еще несколько моментов:

- Мы объявили `checkNumber` как `checkNumber throws IllegalArgumentException` . Это не было строго необходимо, поскольку исключение `IllegalArgumentException` является проверенным исключением; см . Иерархию исключений Java - Непроверенные и проверенные исключения . Тем не менее, это хорошая практика, чтобы сделать это, а также включить исключения, вызванные комментариями javadoc метода.

- Код сразу после того, как заявление о `throw` *недостижима* . Следовательно, если бы мы написали это:

```
throw new IllegalArgumentException("it is bad");
return;
```

компилятор сообщит об ошибке компиляции для оператора `return` .

Цепочка исключений

Многие стандартные исключения имеют конструктор со вторым аргументом `cause` в дополнение к обычному аргументу `message` . `cause` позволяет вам перехватывать исключения. Вот пример.

Сначала мы определяем неконтролируемое исключение, которое наше приложение бросает, когда оно встречает непоправимую ошибку. Обратите внимание, что мы включили конструктор, который принимает аргумент `cause` .

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Далее приведен код, иллюстрирующий цепочку исключений.

```
public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("File is empty: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("Cannot read file: " + file, ex);
    }
}
```

`throw` внутри блока `try` обнаруживает проблему и сообщает об этом через исключение простым сообщением. Напротив, `throw` внутри блока `catch` обрабатывает `IOException` , обертывая его в новое (отмеченное) исключение. Однако это не исключает первоначальное исключение. `IOException` в качестве `cause` , мы записываем его так, чтобы

его можно было напечатать в стеке, как описано в разделе «Создание и чтение стеков» .

Пользовательские исключения

В большинстве случаев проще с точки зрения кодового дизайна использовать существующие общие классы `Exception` при бросании исключений. Это особенно верно, если вам нужно только исключение, чтобы нести простое сообщение об ошибке. В этом случае исключение RuntimeException обычно предпочтительнее, поскольку это не проверенное исключение. Другие классы исключений существуют для общих классов ошибок:

- UnsupportedOperationException - некоторая операция не поддерживается
- IllegalArgumentException - недопустимое значение параметра передавалось методу
- IllegalStateException - ваш API имеет внутреннее состояние, которое никогда не должно происходить, или которое происходит в результате неправильного использования вашего API

Случаи , когда вы хотите использовать пользовательский класс исключений включают в себя следующее:

- Вы пишете API или библиотеку для использования другими пользователями, и вы хотите, чтобы пользователи вашего API могли специально улавливать и обрабатывать исключения из вашего API и *иметь возможность отличать эти исключения от других, более общих исключений* .
- Вы бросаете исключения для **определенной ошибки** в одной части вашей программы, которую вы хотите поймать и обработать в другой части вашей программы, и хотите отличать эти ошибки от других, более общих ошибок.

Вы можете создавать свои собственные исключения, расширяя `RuntimeException` для неконтролируемого исключения или проверяя исключение, расширяя любое `Exception` *которое также не является подклассом RuntimeException* , потому что:

> Подклассы исключения, которые также не являются подклассами RuntimeException, проверяются исключениями

```
public class StringTooLongException extends RuntimeException {
    // Exceptions can have methods and fields like other classes
    // those can be useful to communicate information to pieces of code catching
    // such an exception
    public final String value;
    public final int maximumLength;

    public StringTooLongException(String value, int maximumLength){
        super(String.format("String exceeds maximum Length of %s: %s", maximumLength, value));
        this.value = value;
        this.maximumLength = maximumLength;
    }
}
```

Они могут использоваться как предопределенные исключения:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

И поля могут использоваться там, где исключение поймано и обработано:

```
void anotherMethod(String value){
    try {
        validateString(value);
    } catch(StringTooLongException e){
        System.out.println("The string '" + e.value +
                "' was longer than the max of " + e.maximumLength );
    }
}
```

Имейте в виду, что, согласно Oracle Java Documentation :

> [...] Если клиент может разумно ожидать восстановления от исключения, сделайте это проверенным исключением. Если клиент не может ничего сделать для восстановления из исключения, сделайте его незафиксированным исключением.

Больше:

- Почему RuntimeException не требует явной обработки исключений?

Оператор try-with-resources

Java SE 7

Как иллюстрирует пример утверждения try-catch-final , очистка ресурсов с использованием предложения `finally` требует значительного количества кода «котельной пластины» для правильной реализации краев. Java 7 предоставляет гораздо более простой способ решения этой проблемы в форме инструкции *try-with-resources* .

Что такое ресурс?

Java 7 представила интерфейс `java.lang.AutoCloseable` позволяющий управлять классами с помощью инструкции *try-with-resources* . Экземпляры классов, которые реализуют `AutoCloseable` , называются *ресурсами* . Обычно их нужно утилизировать своевременно, а не полагаться на сборщик мусора, чтобы распоряжаться ими.

Интерфейс `AutoCloseable` определяет один метод:

```
public void close() throws Exception
```

Метод `close()` должен распоряжаться ресурсом соответствующим образом. В спецификации указано, что безопасно вызывать метод на ресурсе, который уже был удален. Кроме того, для классов, реализующих `Autocloseable` , *настоятельно рекомендуется* объявить метод `close()` для исключения более конкретного исключения, чем `Exception` , или вообще никакого исключения.

Широкий спектр стандартных классов Java и интерфейсов реализует `AutoCloseable` . Они включают:

- `InputStream` , `OutputStream` и их подклассы
- `Reader` , `Writer` и их подклассы
- `Socket` и `ServerSocket` и их подклассы
- `Channel` и его подклассы и
- JDBC интерфейсы `Connection` , `Statement` и `ResultSet` и их подклассы.

Приложения и сторонние классы могут это сделать.

Основная инструкция try-with-resource

Синтаксис *try-with-resources* основан на классических формах *try-catch* , *try-finally* и *try-catch-finally* . Вот пример «базовой» формы; т.е. форму без `catch` или, `finally` .

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

Ресурсы для управления объявляются как переменные в разделе `(...)` после предложения `try` . В приведенном выше примере мы объявляем `stream` переменной ресурса и инициализируем его для вновь созданного `PrintStream` .

После инициализации переменных ресурса выполняется блок `try` . Когда это будет завершено, `stream.close()` будет вызываться автоматически, чтобы гарантировать, что ресурс не протекает. Обратите внимание, что вызов `close()` происходит независимо от того, как выполняется блок.

Расширенные инструкции try-with-resource

Оператор *try-with-resources* может быть расширен с помощью блоков `catch` и `finally` , как и в синтаксисе pre-Java 7 *try-catch-finally* . Следующий фрагмент кода добавляет `catch` блок к нашему предыдущему , чтобы иметь дело с `FileNotFoundException` что `PrintStream` конструктор может бросить:

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
} catch (FileNotFoundException ex) {
    System.err.println("Cannot open the file");
} finally {
    System.err.println("All done");
}
```

Если либо инициализация ресурса, либо блок try генерирует исключение, тогда будет выполняться блок `catch` . Блок `finally` всегда будет выполнен, как и в обычном заявлении *try-catch-finally* .

Следует отметить еще пару вещей:

- Переменная ресурса *выходит из области действия* в блоках `catch` и `finally` .
- Очистка ресурсов произойдет до того, как оператор попытается сопоставить блок `catch` .
- Если автоматическая очистка ресурсов породила исключение, то это *можно было* бы поймать в одном из блоков `catch` .

Управление несколькими ресурсами

В приведенных выше фрагментах кода отображается один управляемый ресурс. На самом деле, *try-with-resources* могут управлять несколькими ресурсами в одном заявлении. Например:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

Это ведет себя так, как вы ожидали. Оба `is` и `os` автоматически закрываются в конце блока `try` . Есть несколько замечаний:

- Инициализация происходит в порядке кода, а позже инициализаторы переменных ресурсов могут использовать значения предыдущих.
- Все переменные ресурсов, которые были успешно инициализированы, будут очищены.
- Переменные ресурсов очищаются в *обратном порядке* их деклараций.

Таким образом, в приведенном выше примере, `is` инициализируется перед `os` и очищена после него, и `is` будет очищена , если есть исключение при инициализации `os` .

Эквивалентность try-with-resource и классического try-catch-finally

Спецификация языка Java определяет поведение форм *try-with-resource* в терминах

классического заявления *try-catch-finally* . (Подробную информацию см. В JLS.)

Например, этот базовый *try-with-resource* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

определяется как эквивалент этого *try-catch-finally* :

```
// Note that the constructor is not part of the try-catch statement
PrintStream stream = new PrintStream("hello.txt");

// This variable is used to keep track of the primary exception thrown
// in the try statement. If an exception is thrown in the try block,
// any exception thrown by AutoCloseable.close() will be suppressed.
Throwable primaryException = null;

// The actual try block
try {
    stream.println("Hello world!");
} catch (Throwable t) {
    // If an exception is thrown, remember it for the finally block
    primaryException = t;
    throw t;
} finally {
    if (primaryException == null) {
        // If no exception was thrown so far, exceptions thrown in close() will
        // not be caught and therefore be passed on to the enclosing code.
        stream.close();
    } else {
        // If an exception has already been thrown, any exception thrown in
        // close() will be suppressed as it is likely to be related to the
        // previous exception. The suppressed exception can be retrieved
        // using primaryException.getSuppressed().
        try {
            stream.close();
        } catch (Throwable suppressedException) {
            primaryException.addSuppressed(suppressedException);
        }
    }
}
```

(JLS указывает, что фактические переменные `t` и `primaryException` будут невидимы для обычного Java-кода.)

Расширенная форма *try-with-resources* определяется как эквивалентность базовой форме. Например:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

эквивалентно:

```
try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

Создание и чтение стоп-кадров

Когда создается объект исключения (т. `Throwable` Когда вы его `new`), конструктор `Throwable` захватывает информацию о контексте, в котором было создано исключение. Позже эта информация может выводиться в виде stacktrace, которая может использоваться для диагностики проблемы, вызвавшей исключение в первую очередь.

Печать стоп-кадра

Печать stacktrace - это просто вызов метода printStackTrace `printStackTrace()` . Например:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

Метод `printStackTrace()` без аргументов будет печатать на стандартный вывод приложения; т.е. текущий `System.out` . Существуют также `printStackTrace(PrintStream)` и `printStackTrace(PrintStream)` `printStackTrace(PrintWriter)` перегружают эту печать в указанный `Stream` или `Writer` .

Заметки:

1. В стеке нет сведений о самом исключении. Вы можете использовать метод `toString()` для получения этих деталей; например

   ```
   // Print exception and stacktrace
   System.out.println(ex);
   ex.printStackTrace();
   ```

2. Печать Stacktrace следует использовать экономно; см. Pitfall - чрезмерные или неуместные стеки . Часто лучше использовать фреймворк протоколирования и передавать объект исключения для регистрации.

Понимание stacktrace

Рассмотрим следующую простую программу, состоящую из двух классов в двух файлах. (Мы показали имена файлов и добавили номера строк для иллюстрации.)

```
File: "Main.java"
1   public class Main {
2       public static void main(String[] args) {
3           new Test().foo();
4       }
5   }

File: "Test.java"
1   class Test {
2       public void foo() {
3           bar();
4       }
5
6       public int bar() {
7           int a = 1;
8           int b = 0;
9           return a / b;
10      }
```

Когда эти файлы будут скомпилированы и запущены, мы получим следующий результат.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.bar(Test.java:9)
        at Test.foo(Test.java:3)
        at Main.main(Main.java:3)
```

Прочитайте эту строку за раз, чтобы понять, что она нам говорит.

Строка №1 говорит нам, что поток, называемый «main», завершился из-за неперехваченного исключения. Полное имя исключения - `java.lang.ArithmeticException` , а сообщение об ошибке - «/ нолем».

Если мы рассмотрим javadocs для этого исключения, в нем говорится:

> Брошено, когда произошло исключительное арифметическое условие.
> Например, целое число «делить на ноль» выдает экземпляр этого класса.

В самом деле, сообщение «/ by zero» является сильным намеком на то, что причиной исключения является то, что какой-то код попытался что-то делить на ноль. Но что?

Остальные 3 строки - это трассировка стека. Каждая строка представляет вызов метода (или конструктора) в стеке вызовов, и каждый из них сообщает нам три вещи:

- имя выполняемого класса и метода,
- имя файла исходного кода,
- номер строки исходного кода выполняемого оператора

Эти строки stacktrace перечислены с фреймом для текущего вызова вверху. Верхний кадр в нашем примере выше находится в методе `Test.bar` и в строке 9 файла Test.java. Это следующая строка:

```
    return a / b;
```

Если мы посмотрим пару строк ранее в файле, где `b` инициализируется, очевидно, что `b` будет иметь нулевое значение. Мы можем без всякого сомнения сказать, что это причина исключения.

Если нам нужно идти дальше, мы можем видеть из stacktrace, что `bar()` вызывается из `foo()` в строке 3 Test.java и что `foo()` в свою очередь, вызывался из `Main.main()` .

Примечание. Названия классов и методов в кадрах стека являются внутренними именами для классов и методов. Вам нужно будет признать следующие необычные случаи:

- Вложенный или внутренний класс будет выглядеть как «OuterClass $ InnerClass».
- Анонимный внутренний класс будет выглядеть как «OuterClass $ 1», «OuterClass $ 2» и т. Д.
- Когда выполняется код в конструкторе, инициализатор поля экземпляра или блок инициализатора экземпляра, имя метода будет «".
- Когда выполняется код в инициализаторе статического поля или статическом инициализаторе, имя метода будет «".

(В некоторых версиях Java код форматирования stacktrace будет обнаруживать и повторять повторяющиеся последовательности стека, как это может произойти, когда приложение выходит из строя из-за чрезмерной рекурсии.)

Цепочка исключений и вложенные стеки

Java SE 1.4

Цепочка исключений происходит, когда кусок кода ловит исключение, а затем создает и генерирует новый, передавая первое исключение в качестве причины. Вот пример:

```
File: Test,java
1   public class Test {
2      int foo() {
3           return 0 / 0;
4      }
5
6       public Test() {
7           try {
8               foo();
9           } catch (ArithmeticException ex) {
10              throw new RuntimeException("A bad thing happened", ex);
11          }
12      }
```

```
13
14      public static void main(String[] args) {
15          new Test();
16      }
17  }
```

Когда вышеуказанный класс скомпилирован и запущен, мы получаем следующую стек:

```
Exception in thread "main" java.lang.RuntimeException: A bad thing happened
        at Test.<init>(Test.java:10)
        at Test.main(Test.java:15)
Caused by: java.lang.ArithmeticException: / by zero
        at Test.foo(Test.java:3)
        at Test.<init>(Test.java:8)
        ... 1 more
```

Стек strace начинается с имени класса, метода и стека вызовов для исключения, которое (в данном случае) вызвало сбой приложения. За ним следует строка «Caused by:», которая сообщает об исключении `cause` . Сообщается имя класса и сообщение, за которым следуют стеки кадров исключения. Трассировка заканчивается символом «... N больше», который указывает, что последние N кадров такие же, как и для предыдущего исключения.

«Причиненный:» включается только в вывод, если `cause` первичного исключения не равна `null`). Исключения могут быть цепочки неограниченно, и в этом случае stacktrace может иметь несколько следов «Caused by:».

Примечание: механизм `cause` был обнаружен только в `Throwable` API в Java 1.4.0. До этого цепочка исключений должна была быть реализована приложением с использованием настраиваемого поля исключения для представления причины и пользовательского метода `printStackTrace` .

Захват stacktrace как строки

Иногда приложение должно иметь возможность захватывать стек в качестве `String` Java, чтобы его можно было использовать для других целей. Общий подход для этого заключается в создании временного `OutputStream` или `Writer` который записывает в буфер в памяти и передает его в `printStackTrace(...)` .

Библиотеки Apache Commons и Guava предоставляют утилиты для захвата stacktrace как строки:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

Если вы не можете использовать сторонние библиотеки в базе кода, то выполните следующий метод:

```
   /**
     * Returns the string representation of the stack trace.
     *
     * @param throwable the throwable
     * @return the string.
     */
    public static String stackTraceToString(Throwable throwable) {
        StringWriter stringWriter = new StringWriter();
        throwable.printStackTrace(new PrintWriter(stringWriter));
        return stringWriter.toString();
    }
```

Обратите внимание, что если вы намерены проанализировать stacktrace, проще использовать `getStackTrace()` и `getCause()` чем пытаться проанализировать stacktrace.

Обработка InterruptedException

`InterruptedException` - запутанный зверь - он проявляется в кажущихся безобидными методами, таких как `Thread.sleep()` , но некорректная обработка его приводит к жесткому управлению кодом, который плохо работает в параллельных средах.

В самом `Thread.interrupt()` случае, если `InterruptedException` пойман, это означает, что кто-то, где-то, вызвал `Thread.interrupt()` в потоке, в котором работает ваш код. Возможно, вы склонны сказать: «Это мой код! Я никогда его не прерываю! " и поэтому сделать что-то вроде этого:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

Но это точно неправильный способ справиться с «невозможным» событием. Если вы знаете, что ваше приложение никогда не столкнется с `InterruptedException` вы должны рассматривать такое событие как серьезное нарушение предположений вашей программы и выходить как можно быстрее.

Правильный способ обработки «невозможного» прерывания выглядит так:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new AssertionError(e);
}
```

Это делает две вещи; он сначала восстанавливает статус прерывания потока (как будто `InterruptedException` не было выбрано в первую очередь), а затем оно выдает `AssertionError`

указывающее, что основные инварианты вашего приложения были нарушены. Если вы точно знаете, что никогда не будете прерывать поток, этот код работает в этом, это безопасно, поскольку блок `catch` никогда не должен быть достигнут.

Использование класса Guava `Uninterruptibles` помогает упростить этот шаблон; вызов `Uninterruptibles.sleepUninterruptibly()` игнорирует прерванное состояние потока до истечения времени ожидания (после чего он восстанавливается для последующих вызовов, чтобы проверить и выбросить собственное `InterruptedException`). Если вы знаете, что никогда не будете прерывать такой код, это безопасно избегает необходимости обертывания вызовов сна в блоке try-catch.

Чаще всего, однако, вы не можете гарантировать, что ваш поток никогда не будет прерван. В частности, если вы пишете код, который будет исполнен `Executor` или каким-либо другим управлением потоками, важно, чтобы ваш код быстро реагировал на прерывания, иначе ваше приложение остановится или даже закроется.

В таких случаях лучше всего разрешить `InterruptedException` распространять стек вызовов, добавляя к каждому методу `throws InterruptedException` . Это может показаться kludgy, но это на самом деле желательное свойство - подписи вашего метода теперь указывают абонентам, что он будет оперативно реагировать на прерывания.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

В ограниченных случаях (например, при переопределении метода, который не `throw` никаких проверенных исключений) вы можете сбросить прерванный статус без повышения исключения, ожидая, что какой-либо код будет выполнен рядом с обработкой прерывания. Это задерживает обработку прерывания, но не полностью подавляет его.

```
// Suppresses the exception but resets the interrupted state letting later code
// detect the interrupt and handle it properly.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  return ...; // your expectations are still broken at this point - try not to do more work.
}
```

Иерархия исключений Java - неконтролируемые и проверенные исключения

Все исключения Java являются экземплярами классов в иерархии классов Exception. Это можно представить следующим образом:

- `java.lang.Throwable` - это базовый класс для всех классов исключений. Его методы и

конструкторы реализуют целый ряд функций, общих для всех исключений.
    - `java.lang.Exception` - это суперкласс всех нормальных исключений.
        - различные стандартные и настраиваемые классы исключений.
        - `java.lang.RuntimeException` - это суперкласс всех нормальных исключений, которые являются *неконтролируемыми исключениями* .
            - различные стандартные и настраиваемые классы исключений во время выполнения.
    - `java.lang.Error` - это суперкласс всех исключений «фатальной ошибки».

Заметки:

1. Различие между *проверенными* и *непроверенными* исключениями описано ниже.
2. Класс `Throwable` , `Exception` и `RuntimeException` следует рассматривать как `abstract` ; см. Pitfall - Throwing Throwable, Exception, Error или RuntimeException .
3. Исключения `Error` выбрасываются JVM в ситуациях, когда было бы опасно или неразумно для приложения пытаться восстановить.
4. Было бы неразумно объявлять пользовательские подтипы `Throwable` . Java-инструменты и библиотеки могут предполагать, что `Error` и `Exception` являются единственными прямыми подтипами `Throwable` и неверно `Throwable` , если это предположение неверно.

Проверено против исключенных исключений

Одна из критических замечаний в отношении поддержки исключений на некоторых языках программирования заключается в том, что трудно узнать, какие исключения может дать данный метод или процедура. Учитывая, что необработанное исключение может привести к сбою программы, это может сделать исключения источником хрупкости.

Язык Java решает эту проблему с помощью механизма проверенных исключений. Во-первых, Java классифицирует исключения на две категории:

- Проверенные исключения обычно представляют собой ожидаемые события, с которыми приложение должно иметь дело. Например, `IOException` и его подтипы представляют собой условия ошибки, которые могут возникать в операциях ввода-вывода. Например, файл открывается с ошибкой, потому что файл или каталог не существует, чтение и запись в сети происходит неудачно, поскольку сетевое соединение было повреждено и так далее.

- Исключенные исключения обычно представляют собой непредвиденные события, с которыми приложение не справляется. Как правило, это результат ошибки в приложении.

(В дальнейшем « `throw` » ссылается на любое исключение, явно выраженное (с помощью оператора `throw`) или неявно (в случае неудачной разыменования, типа cast и т. Д.). Аналогично, «распространяемый» относится к исключению, которое было выбрано в вложенный вызов и не попадает в этот вызов. Пример кода

ниже иллюстрирует это.)

Вторая часть проверенного механизма исключений заключается в том, что существуют ограничения на методы, в которых может произойти проверочное исключение:

- Когда проверяемое исключение выбрано или распространено в методе, оно *должно* быть либо поймано методом, либо указано в предложении `throws` метода. (Значение примера `throws` описано в этом примере .)
- Когда проверенное исключение генерируется или распространяется в блоке инициализатора, оно должно быть уловлено блоком.
- Проверенное исключение не может быть передано вызовом метода в выражении инициализации поля. (Невозможно поймать такое исключение).

Короче говоря, проверенное исключение должно быть обработано или объявлено.

Эти ограничения не применяются к исключенным исключениям. Сюда относятся все случаи, когда исключение выбрано неявно, поскольку все такие случаи вызывают неконтролируемые исключения.

Проверенные примеры исключений

Эти фрагменты кода предназначены для иллюстрации проверенных ограничений исключений. В каждом случае мы показываем версию кода с ошибкой компиляции, а вторую версию с исправленной ошибкой.

```
// This declares a custom checked exception.
public class MyException extends Exception {
    // constructors omitted.
}

// This declares a custom unchecked exception.
public class MyException2 extends RuntimeException {
    // constructors omitted.
}
```

В первом примере показано, как явно заброшенные проверенные исключения могут быть объявлены как «брошенные», если они не должны обрабатываться в методе.

```
// INCORRECT
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0;                // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

// CORRECTED
public void methodThrowingCheckedException(boolean flag) throws MyException {
```

```
    int i = 1 / 0;                // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}
```

Во втором примере показано, как можно обработать распространенное проверенное исключение.

```
// INCORRECT
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt");  // Compilation error
    // FileInputStream throws IOException or a subclass if the file cannot
    // be opened.  IOException is a checked exception.
    ...
}

// CORRECTED (Version A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// CORRECTED (Version B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("Cannot open file: " + ex.getMessage());
    }
}
```

В последнем примере показано, как обрабатывать проверенное исключение в инициализаторе статического поля.

```
// INCORRECT
public class Test {
    private static final InputStream is =
            new FileInputStream("someFile.txt");  // Compilation error
}

// CORRECTED
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
            tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("Cannot open file: " + ex.getMessage());
        }
        is = tmp;
    }
}
```

Обратите внимание, что в этом последнем случае нам также приходится иметь дело с проблемами, которые `is` могут быть назначены более одного раза, и при этом также необходимо назначать даже в случае исключения.

Вступление

Исключения - это ошибки, возникающие при выполнении программы. Рассмотрим программу Java, ниже которой делятся два целых числа.

```
class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Теперь мы компилируем и выполняем приведенный выше код и видим результат для попытки деления на ноль:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Disivion.java:14)
```

Деление на ноль - это недопустимая операция, которая приведет к значению, которое не может быть представлено как целое число. Java справляется с этим, *вызывая **исключение*** . В этом случае исключение является экземпляром класса *ArithmeticException* .

Примечание . Пример создания и чтения стеков стека объясняет, что означает результат после двух чисел.

Утилита *исключения* - это контроль потока, который он позволяет. Без использования исключений типичным решением этой проблемы может быть, прежде всего, проверить, если `b == 0` :

```
class Division {
    public static void main(String[] args) {

        int a, b, result;
```

```
        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Это выводит сообщение. `You cannot divide by zero.` на консоль и изящно выходит из программы, когда пользователь пытается делить на ноль. Эквивалентным способом справиться с этой проблемой с помощью *обработки исключений* будет замена управления `if` с помощью блока `try-catch` :

```
...

a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("An ArithmeticException occurred. Perhaps you tried to divide by
zero.");
    return;
}

...
```

Блок catch try выполняется следующим образом:

1. Начните выполнение кода в блоке `try` .
2. Если в блоке try возникает *исключение* , немедленно прервите его и проверьте, не *обнаружено* ли это исключение блоком `catch` (в этом случае, когда Exception является экземпляром исключения `ArithmeticException`).
3. Если исключение *поймано* , оно назначается переменной `e` и выполняется блок `catch` .
4. Если завершен блок `try` или `catch` (т.е. во время выполнения кода не выполняются неперехваченные исключения), продолжайте выполнение кода ниже блока `try-catch` .

Обычно считается хорошей практикой использовать *обработку исключений* как часть нормального управления потоком приложения, где поведение в противном случае было бы неопределенным или неожиданным. Например, вместо того, чтобы возвращать значение `null` когда метод терпит неудачу, обычно лучше *использовать исключение,* чтобы

приложение, использующее метод, могло определить свой собственный контроль потока для ситуации посредством *обработки исключений* вида, показанного выше. В некотором смысле это оборачивается проблемой возврата определенного *типа* , поскольку любой из нескольких видов *исключений* может быть *брошен,* чтобы указать конкретную проблему, которая произошла.

Дополнительные советы о том, как и как не использовать исключения, см. В разделе «Явные ошибки» - использование исключений

Операторы возврата в блоке try catch

Хотя это плохая практика, в блок обработки исключений можно добавить несколько операторов return:

```
 public static int returnTest(int number){
    try{
        if(number%2 == 0) throw new Exception("Exception thrown");
        else return x;
    }
    catch(Exception e){
        return 3;
    }
    finally{
        return 7;
    }
}
```

Этот метод всегда будет возвращать 7, поскольку блок finally, связанный с блоком try / catch, выполняется прежде, чем что-либо будет возвращено. Теперь, как наконец, `return 7;` , это значение заменяет значения возврата try / catch.

Если блок catch возвращает примитивное значение и это примитивное значение впоследствии изменяется в блоке finally, возвращается значение, возвращаемое в блоке catch, и изменения из блока finally будут проигнорированы.

В приведенном ниже примере будет напечатан «0», а не «1».

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
```

```
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
        }
    }
}
```

Расширенные возможности исключений

В этом примере описаны некоторые дополнительные функции и прецеденты для исключений.

Проверка программной таблицы программно

Java SE 1.4

Основное использование стека стека исключений заключается в предоставлении информации об ошибке приложения и его контексте, чтобы программист мог диагностировать и исправлять проблему. Иногда его можно использовать для других вещей. Например, для класса `SecurityManager` может потребоваться проверка стека вызовов, чтобы решить, следует ли доверять коду, вызывающему вызов.

Вы можете использовать исключения для проверки стека вызовов программно следующим образом:

```
    Exception ex = new Exception();   // this captures the call stack
    StackTraceElement[] frames = ex.getStackTrace();
    System.out.println("This method is " + frames[0].getMethodName());
    System.out.println("Called from method " + frames[1].getMethodName());
```

Есть несколько важных предостережений по этому поводу:

1. Информация, доступная в `StackTraceElement` , ограничена. Существует больше информации, чем отображается `printStackTrace` . (Значения локальных переменных в кадре недоступны.)

2. В javadocs для `getStackTrace()` что JVM разрешено оставлять рамки:

   > Некоторые виртуальные машины могут при некоторых обстоятельствах опустить один или несколько кадров стека из трассировки стека. В крайнем случае виртуальная машина, которая не имеет информации о трассировке стека, относящейся к этому методу, разрешает возвращать массив нулевой длины из этого метода.

Оптимизация конструкции исключения

Как упоминалось в другом месте, построение исключения является довольно дорогостоящим, поскольку оно влечет за собой захват и запись информации обо всех кадрах стека в текущем потоке. Иногда мы знаем, что эта информация никогда не будет использоваться для данного исключения; например, stacktrace никогда не будет напечатана. В этом случае существует трюк реализации, который мы можем использовать в пользовательском исключении, чтобы не захватывать информацию.

Информация о кадре стека, необходимая для stacktraces, записывается, когда конструкторы `Throwable` называют метод `Throwable.fillInStackTrace()` . Этот метод является `public` , что означает, что подкласс может его переопределить. Хитрость заключается в том, чтобы переопределить метод, унаследованный от `Throwable` тем, который ничего не делает; например

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

Проблема с этим подходом заключается в том, что исключение, которое переопределяет `fillInStackTrace()` никогда не может захватить стек, и бесполезно в сценариях, где вам это нужно.

Стирание или замена стека

Java SE 1.4

В некоторых ситуациях stacktrace для исключения, созданного обычным способом, содержит либо неверную информацию, либо информацию, которую разработчик не хочет показывать пользователю. Для этих сценариев `Throwable.setStackTrace` может использоваться для замены массива объектов `StackTraceElement` который содержит информацию.

Например, для удаления информации об стеках исключений можно использовать следующее:

```
exception.setStackTrace(new StackTraceElement[0]);
```

Подавленные исключения

Java SE 7

Java 7 представила конструкцию *try-with-resources* и связанную с ней концепцию исключения исключений. Рассмотрим следующий фрагмент:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

Когда выбрано исключение, `try` вызовет функцию `close()` на `w` которая будет очищать любой буферный вывод, а затем закрыть `FileWriter` . Но что произойдет, если `IOException` будет `IOException` при очистке вывода?

Случается, что любое исключение, которое бросается при очистке ресурса, *подавляется* . Исключение поймано и добавлено в список исключенных исключений первичного исключения. Затем *try-with-resources* продолжат очистку других ресурсов. Наконец, основное исключение будет восстановлено.

Аналогичная картина возникает, если исключение было выбрано во время инициализации ресурса или если блок `try` завершился нормально. Исключение составляет первое исключение, которое является основным исключением, а последующие, возникающие из-за очистки, подавляются.

Подавленные исключения могут быть извлечены из основного объекта исключения, вызвав `getSuppressedExceptions` .

Утверждения try-finally и try-catch-finally

Команда `try...catch...finally` объединяет обработку исключений с кодом очистки. Блок `finally` содержит код, который будет выполняться при любых обстоятельствах. Это делает их подходящими для управления ресурсами и других видов очистки.

Примерка наконец

Вот пример более простой (`try...finally`):

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

Поведение `try...finally` выглядит следующим образом:

- Выполняется код в блоке `try` .
- Если исключение не было выбрано в блоке `try` :
    - Выполняется код в блоке `finally` .
    - Если блок `finally` генерирует исключение, это исключение распространяется.

    - В противном случае управление переходит к следующему утверждению после `try...finally` .
- Если в блоке try было выбрано исключение:
    - Выполняется код в блоке `finally` .
    - Если блок `finally` генерирует исключение, это исключение распространяется.
    - В противном случае исходное исключение продолжает распространяться.

Код внутри блока `finally` всегда будет выполнен. (Единственные исключения - если вызывается `System.exit(int)` или паника JVM.) Таким образом, `finally` блок - это правильный код места, который всегда необходимо выполнить; например, закрытие файлов и других ресурсов или освобождение блокировок.

попробуй поймать, наконец,

Наш второй пример показывает, как `catch` и, `finally` можно использовать вместе. Это также иллюстрирует, что очистка ресурсов не является простой.

```
// This code snippet writes the first line of a file to a string
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger.warn("Unexpected IO error", ex);  // logging the exception
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // ignore / discard this exception
        }
    }
}
```

Полный набор (гипотетических) поведений `try...catch...finally` в этом примере слишком сложно описать здесь. Простая версия заключается в том, что код в блоке `finally` всегда будет выполняться.

Рассматривая это с точки зрения управления ресурсами:

- Мы объявляем «ресурс» (т. `reader` Переменную `reader`) перед блоком `try` чтобы он был доступен для блока `finally` .
- Помещая `new FileReader(...)` , `catch` может обрабатывать любое исключение `IOError` из броска при открытии файла.
- Нам нужен `reader.close()` в блоке `finally` потому что есть некоторые пути исключения, которые мы не можем перехватывать ни в блоке `try` ни в блоке `catch` .
- Однако, поскольку исключение *могло* быть выброшено до того, как был инициализирован `reader` , нам также понадобится явный критерий `null` .

- Наконец, `reader.close()` может (гипотетически) вызвать исключение. Нас это не волнует, но если мы не поймаем исключение в источнике, нам нужно будет разобраться с ним в стеке вызовов.

Java SE 7

Java 7 и более поздние версии предоставляют альтернативный синтаксис try-with-resources, который значительно упрощает очистку ресурсов.

Предложение 'throws' в объявлении метода

Механизм *исключенных исключений* Java требует, чтобы программист объявил, что определенные методы *могут* вызывать указанные проверенные исключения. Это делается с использованием предложения `throws` . Например:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

`throws OddNumberException` объявляют, что вызов `checkEven` *может* генерировать исключение, которое имеет тип `OddNumberException` .

Предложение `throws` может объявлять список типов и может включать в себя неконтролируемые исключения, а также проверенные исключения.

```
public void checkEven(Double number)
        throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

В чем смысл объявления исключенных исключений?

Предложение `throws` в объявлении метода служит для двух целей:

1. Он сообщает компилятору, какие исключения выбрасываются таким образом, чтобы компилятор мог сообщать об исключенных (отмеченных) исключениях как ошибки.
2. Это говорит программисту, который пишет код, который вызывает метод, какие исключения ожидать. Для этой цели часто возникает вопрос о включении исключенных исключений в список `throws` .

Обратите внимание: список `throws` также используется инструментом javadoc при создании документации API, а также типичными подсказками метода типичного IDE.

Броски и метод переопределения

Предложение `throws` является частью сигнатуры метода для переопределения метода. Метод переопределения может быть объявлен с тем же набором проверенных исключений, что и метод переопределения, или подмножество. Однако метод переопределения не может добавлять дополнительные проверенные исключения. Например:

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException is an
unchecked exception
    ...

@Override
public void checkEven(Double number) throws OddNumberException // OK—identical to the
superclass
    ...

class PrimeNumberException extends OddNumberException {}
class NonEvenNumberException extends OddNumberException {}

@Override
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException //
OK—these are both subclasses

@Override
public void checkEven(Double number) throws IOExcepion         // ERROR
```

Причиной этого правила является то, что если метод overriden может выставить проверенное исключение, которое переопределенный метод не может выбрасывать, это приведет к разрыву подстановки.

Прочитайте Исключения и обработка исключений онлайн:
https://riptutorial.com/ru/java/topic/89/исключения-и-обработка-исключений

глава 79: Исполнители, Исполнительные службы и пулы потоков

Вступление

Интерфейс Executor в Java обеспечивает способ расцепления заданий от механики того, как будет выполняться каждая задача, включая сведения о потреблении потоков, планировании и т. Д. Обычно используется Executor вместо явного создания потоков. С Executors разработчикам не придется значительно переписывать свой код, чтобы иметь возможность легко настраивать политику выполнения своих задач.

замечания

Ловушки

- Когда вы планируете задачу для повторного выполнения, в зависимости от используемого ScheduledExecutorService, ваша задача может быть приостановлена с любого последующего исполнения, если выполнение вашей задачи вызывает исключение, которое не обрабатывается. См. Mother F ** k ScheduledExecutorService!

Examples

Огонь и Забыть - Управляемые задачи

Исполнители принимают `java.lang.Runnable` который содержит (потенциально вычислительный или иной долговременный или тяжелый) код для запуска в другом потоке.

Использование:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

Обратите внимание, что с этим исполнителем у вас нет средств для возврата какого-либо вычисленного значения.
С Java 8 можно использовать lambdas, чтобы сократить пример кода.

Java SE 8

```
Executor exec = anExecutor;
```

```
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

Обычным Исполнителем является `ThreadPoolExecutor` , который занимается обработкой потоков. Вы можете настроить минимальное количество потоков, которые исполнитель всегда должен поддерживать, когда не так много делать (это называется размером ядра) и максимальный размер потока, к которому может увеличиваться пул, если есть больше работы. Как только рабочая нагрузка снижается, пул медленно уменьшает количество потоков, пока не достигнет минимального размера.

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    1,                                      // keep at least one thread ready,
                                            // even if no Runnables are executed
    5,                                      // at most five Runnables/Threads
                                            // executed in parallel
    1, TimeUnit.MINUTES,                    // idle Threads terminated after one
                                            // minute, when min Pool size exceeded
    new ArrayBlockingQueue<Runnable>(10)); // outstanding Runnables are kept here

pool.execute(new Runnable() {
    @Override public void run() {
        //code to run
    }
});
```

Примечание. Если вы сконфигурируете `ThreadPoolExecutor` с *неограниченной* очередью, то количество потоков не будет превышать `corePoolSize` поскольку новые потоки создаются только в том случае, если очередь заполнена:

ThreadPoolExecutor со всеми параметрами:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

от JavaDoc

> Если существует больше, чем corePoolSize, но меньше потоков MaximumPoolSize, новый поток будет создан только в том случае, если очередь заполнена.

Преимущества:

1. BlockingQueue можно контролировать, а сценарии отсутствия памяти можно избежать. Производительность приложения не будет снижена с ограниченным ограниченным размером очереди.

2. Вы можете использовать существующие или создавать новые политики Отклонения

Обработчика.

1. В по умолчанию ThreadPoolExecutor.AbortPolicy обработчик выдает исключение ExjectExceptionException после отклонения.

2. В `ThreadPoolExecutor.CallerRunsPolicy` поток, запускающий выполнение, запускает задачу. Это обеспечивает простой механизм управления с обратной связью, который замедляет скорость подачи новых задач.

3. В `ThreadPoolExecutor.DiscardPolicy` задача, которая не может быть выполнена, просто удаляется.

4. В `ThreadPoolExecutor.DiscardOldestPolicy` , если исполнитель не закрыт, задача во главе рабочей очереди отбрасывается, а затем выполняется повторное выполнение (что может снова потерпеть неудачу, в результате чего это будет повторяться).

3. Пользовательский `ThreadFactory` может быть настроен, что полезно:

   1. Чтобы задать более описательное имя потока
   2. Чтобы установить статус демона нити
   3. Чтобы установить приоритет потока

Вот пример использования ThreadPoolExecutor

Извлечение значения из вычисления - Callable

Если ваше вычисление дает некоторое возвращаемое значение, которое требуется позже, простая задача Runnable недостаточна. Для таких случаев вы можете использовать `ExecutorService.submit( Callable <T>)` который возвращает значение после завершения выполнения.

Служба вернет `Future` которое вы можете использовать для получения результата выполнения задачи.

```
// Submit a callable for execution
ExecutorService pool = anExecutorService;
Future<Integer> future = pool.submit(new Callable<Integer>() {
    @Override public Integer call() {
        //do some computation
        return new Random().nextInt();
    }
});
// ... perform other tasks while future is executed in a different thread
```

Когда вам нужно получить результат в будущем, вызовите `future.get()`

- Подождите бесконечно для будущего, чтобы закончить результат.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Подождите, пока закончится будущее, но не больше указанного времени.

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
    // otherwise TimeoutException is thrown.
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
catch (InterruptedException || ExecutionException || TimeoutException e) {
    // handle appropriately
}
```

Если результат запланированной или запущенной задачи больше не требуется, вы можете вызвать `Future.cancel(boolean)` чтобы отменить его.

- Вызов `cancel(false)` просто удалит задачу из очереди задач для запуска.
- Вызов `cancel(true)` *также* прервет задачу, если она в данный момент запущена.

Планирование задач для запуска в определенное время, после задержки или многократного

Класс `ScheduledExecutorService` предоставляет методы для планирования отдельных или повторяющихся задач несколькими способами. В следующем примере кода предполагается, что `pool` был объявлен и инициализирован следующим образом:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

В дополнение к обычным методам `ExecutorService` API `ScheduledExecutorService` добавляет 4 метода, которые планируют задачи и возвращают объекты `ScheduledFuture` . Последний может использоваться для получения результатов (в некоторых случаях) и отмены задач.

Запуск задачи после фиксированной задержки

В следующем примере планируется запуск задачи через десять минут.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Запуск заданий с фиксированной скоростью

В следующем примере планируется запуск задачи через десять минут, а затем несколько раз со скоростью один раз в минуту.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Выполнение задачи будет продолжаться в соответствии с графиком до тех пор, пока `pool` будет закрыт, `future` будет отменено или одна из задач встретит исключение.

Гарантируется, что задачи, запланированные по заданному вызову `scheduledAtFixedRate` , не будут перекрываться во времени. Если задача занимает больше времени, чем заданный период, то последующие и последующие задания могут начинаться с опоздания.

Запуск задач с фиксированной задержкой

В следующем примере планируется запуск задачи через десять минут, а затем несколько раз с задержкой в одну минуту между завершением одной задачи и последующим запуском.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Выполнение задачи будет продолжаться в соответствии с графиком до тех пор, пока `pool` будет закрыт, `future` будет отменено или одна из задач встретит исключение.

Отказ от отказа

Если

1. вы пытаетесь отправить задания на выключение Executor или
2. очередь является насыщенной (возможно только с ограниченными) и достигается максимальное количество потоков,

`RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` .

Поведение по умолчанию заключается в том, что вы получите исключение RejectedExecutionException, которое вызывается вызывающим. Но есть более

предопределенное поведение:

- **ThreadPoolExecutor.AbortPolicy** (по умолчанию выбрасывает REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (выполняет задачу по потоку вызывающего абонента - *блокирует ее*)
- **ThreadPoolExecutor.DiscardPolicy** (тихо отмените задачу)
- **ThreadPoolExecutor.DiscardOldestPolicy** (тихо отбрасывает **старую** задачу в очереди и повторяет выполнение новой задачи)

Вы можете установить их с помощью одного из конструкторов ThreadPool:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler) // <--

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) // <--
```

Вы можете также реализовать свое собственное поведение, расширив интерфейс RejectedExecutionHandler :

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

Отправить () vs execute () отличия обработки исключений

Обычно команда execute () используется для вызовов огня и забвения (без необходимости анализа результата), а команда submit () используется для анализа результата объекта Future.

Мы должны знать о ключевой разнице механизмов обработки исключений между этими двумя командами.

Исключения из submit () проглатываются каркасом, если вы их не поймали.

Пример кода, чтобы понять разницу:

Случай 1: подайте команду Runnable with execute (), которая сообщает об исключении.

```
import java.util.concurrent.*;
import java.util.*;
```

```
public class ExecuteSubmitDemo {
    public ExecuteSubmitDemo() {
        System.out.println("creating service");
        ExecutorService service = Executors.newFixedThreadPool(2);
        //ExtendedExecutor service = new ExtendedExecutor();
        for (int i = 0; i < 2; i++){
            service.execute(new Runnable(){
                 public void run(){
                    int a = 4, b = 0;
                    System.out.println("a and b=" + a + ":" + b);
                    System.out.println("a/b:" + (a / b));
                    System.out.println("Thread Name in Runnable after divide by
zero:"+Thread.currentThread().getName());
                 }
            });
        }
        service.shutdown();
    }
    public static void main(String args[]){
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();
    }
}

class ExtendedExecutor extends ThreadPoolExecutor {

   public ExtendedExecutor() {
       super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));
   }
   // ...
   protected void afterExecute(Runnable r, Throwable t) {
     super.afterExecute(r, t);
     if (t == null && r instanceof Future<?>) {
       try {
         Object result = ((Future<?>) r).get();
       } catch (CancellationException ce) {
           t = ce;
       } catch (ExecutionException ee) {
           t = ee.getCause();
       } catch (InterruptedException ie) {
           Thread.currentThread().interrupt(); // ignore/reset
       }
     }
     if (t != null)
       System.out.println(t);
   }
 }
```

выход:

```
creating service
a and b=4:0
a and b=4:0
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-2"
java.lang.ArithmeticException: / by zero
        at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmeticException: / by zero
        at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
```

```
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:744)
```

Случай 2: Заменить execute () с помощью submit (): `service.submit(new Runnable(){` В этом случае Исключения проглатываются инфраструктурой, так как метод run () не обнаружил их явно.

выход:

```
creating service
a and b=4:0
a and b=4:0
```

Случай 3: измените newFixedThreadPool на ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

выход:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

Я продемонстрировал этот пример, чтобы охватить две темы: используйте свой собственный ThreadPoolExecutor и обработайте Exectpion с помощью пользовательского ThreadPoolExecutor.

Другое простое решение вышеуказанной проблемы: когда вы используете обычную команду ExecutorService & submit, получите объект Future из submit () команды call get () API on Future. Вызовите три исключения, которые были указаны в реализации метода afterExecute. Преимущество пользовательского ThreadPoolExecutor по этому подходу: вы должны обрабатывать механизм обработки исключений только в одном месте - Custom ThreadPoolExecutor.

Примеры использования для различных типов конструкций параллелизма

1. ExecutorService

   ```
   ExecutorService executor = Executors.newFixedThreadPool(50);
   ```

   Он прост и удобен в использовании. Он скрывает детали уровня `ThreadPoolExecutor` .

   Я предпочитаю это, когда количество задач `Callable/Runnable` малочисленно, а

нагромождение задач в неограниченной очереди не увеличивает память и ухудшает производительность системы. Если у вас есть ограничения на `CPU/Memory` , я предпочитаю использовать `ThreadPoolExecutor` с ограничениями емкости и `RejectedExecutionHandler` для обработки отказа от задач.

2. CountDownLatch

   `CountDownLatch` будет инициализирован с заданным счетом. Этот счет уменьшается с помощью вызовов метода `countDown()` . Темы, ожидающие, что этот счет достигнет нуля, могут вызвать один из методов `await()` . Вызов `await()` блокирует поток до тех пор, пока счетчик не достигнет нуля. *Этот класс позволяет потоку java ждать, пока другой набор потоков завершит выполнение своих задач.*

   Случаи применения:

   1. Достижение максимального параллелизма. Иногда мы хотим начать несколько потоков одновременно для достижения максимального параллелизма
   2. Подождите N потоков до завершения перед запуском
   3. Обнаружение взаимоблокировки.

3. ThreadPoolExecutor : он обеспечивает больше контроля. Если приложение ограничено количеством ожидающих задач Runnable / Callable, вы можете использовать ограниченную очередь, установив максимальную емкость. Когда очередь достигает максимальной емкости, вы можете определить RejectionHandler. Java предоставляет четыре типа политик `RejectedExecutionHandler` .

   1. `ThreadPoolExecutor.AbortPolicy` , обработчик выдает исключение ExjectExceptionException после отклонения.
   2. ThreadPoolExecutor.CallerRunsPolicy`, поток, который вызывает выполнение выполнения, запускает задачу. Это обеспечивает простой механизм управления с обратной связью, который замедляет скорость подачи новых задач.
   3. В `ThreadPoolExecutor.DiscardPolicy` задача, которая не может быть выполнена, просто удаляется.
   4. `ThreadPoolExecutor.DiscardOldestPolicy` , если исполнитель не закрыт, задача во главе рабочей очереди отбрасывается, а затем выполняется повторное выполнение (что может снова потерпеть неудачу, в результате чего это будет повторяться).

Если вы хотите имитировать поведение `CountDownLatch` , вы можете использовать `invokeAll()` .

4. Еще один механизм, который вы не указали, - ForkJoinPool

`ForkJoinPool` был добавлен в Java на Java 7. `ForkJoinPool` похож на Java `ExecutorService` но с одним отличием. `ForkJoinPool` упрощает задачу разделения работы на более мелкие задачи, которые затем отправляются в `ForkJoinPool` . `ForkJoinPool` задачи происходит в `ForkJoinPool` когда потоки рабочего потока крадут задачи из очереди занятых рабочих потоков.

Java 8 представила еще один API в ExecutorService для создания пула кражи работы. Вам не нужно создавать `RecursiveTask` и `RecursiveAction` но все равно использовать `ForkJoinPool` .

```
public static ExecutorService newWorkStealingPool()
```

> Создает пул потоков, обрабатывающих работу, используя все доступные процессоры в качестве целевого уровня параллелизма.

По умолчанию в качестве параметра потребуется количество ядер процессора.

Все эти четыре механизма дополняют друг друга. В зависимости от уровня детализации, который вы хотите контролировать, вы должны выбрать правильные.

Подождите завершения всех задач в ExecutorService

Давайте рассмотрим различные варианты ожидания выполнения задач, представленных Исполнителю

1. ExecutorService `invokeAll()`

   > Выполняет заданные задачи, возвращая список фьючерсов, подтверждающих их статус и результаты, когда все будет завершено.

Пример:

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("creating service");
        ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        List<MyCallable> futureList = new ArrayList<MyCallable>();
        for (int i = 0; i < 10; i++){
            MyCallable myCallable = new MyCallable((long)i);
            futureList.add(myCallable);
        }
        System.out.println("Start");
        try{
            List<Future<Long>> futures = service.invokeAll(futureList);
        } catch(Exception err){
            err.printStackTrace();
```

```
        }
        System.out.println("Completed");
        service.shutdown();
    }
    public static void main(String args[]){
        InvokeAllDemo demo = new InvokeAllDemo();
    }
    class MyCallable implements Callable<Long>{
        Long id = 0L;
        public MyCallable(Long val){
            this.id = val;
        }
        public Long call(){
            // Add your business logic
            return id;
        }
    }
}
```

2. CountDownLatch

   Вспомогательное средство синхронизации, которое позволяет одному или нескольким потокам дождаться завершения набора операций в других потоках.

   CountDownLatch инициализируется с заданным подсчетом. Методы ожидания выполняются до тех пор, пока текущий счетчик не достигнет нуля из-за вызовов метода `countDown()` , после чего все ожидающие потоки освобождаются, и любые последующие вызовы ожидания возвращаются немедленно. Это одноразовый феномен - счетчик не может быть сброшен. Если вам нужна версия, которая сбрасывает счетчик, рассмотрите возможность использования **CyclicBarrier** .

3. ForkJoinPool или `newWorkStealingPool()` для исполнителей

4. Итерация через все объекты `Future` созданные после отправки в `ExecutorService`

5. Рекомендуемый способ закрытия страницы документации Oracle по адресу : ExecutorService :

```
void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
      // Wait a while for existing tasks to terminate
      if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        pool.shutdownNow(); // Cancel currently executing tasks
        // Wait a while for tasks to respond to being cancelled
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
      }
    } catch (InterruptedException ie) {
      // (Re-)Cancel if current thread also interrupted
      pool.shutdownNow();
      // Preserve interrupt status
```

```
      Thread.currentThread().interrupt();
    }
```

`shutdown()`: инициирует упорядоченное завершение работы, в котором выполняются ранее поставленные задачи, но новые задачи не будут приняты.

`shutdownNow()`: пытается остановить все активное выполнение задач, останавливает обработку ожидающих задач и возвращает список задач, ожидающих выполнения.

В приведенном выше примере, если ваши задачи занимают больше времени для завершения, вы можете изменить, если условие на условие

замещать

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

с

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Случаи использования различных типов ExecutorService

Исполнители возвращают разный тип ThreadPools, удовлетворяющий конкретным потребностям.

1. `public static ExecutorService newSingleThreadExecutor()`

   > Создает Исполнителя, который использует один рабочий поток, работающий с неограниченной очередью

   Существует разница между `newFixedThreadPool(1)` и `newSingleThreadExecutor()` как сообщает java-документ для последнего:

   > В отличие от эквивалентного newFixedThreadPool (1), возвращенный исполнитель гарантированно не может быть перенастроен для использования дополнительных потоков.

   Это означает, что `newFixedThreadPool` можно переконфигурировать позже в программе: `((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` Это невозможно для `newSingleThreadExecutor`

   Случаи применения:

   1. Вы хотите выполнить поставленные задачи в последовательности.
   2. Вам нужен только один поток для обработки всего вашего запроса

Минусы:

1. Неограниченная очередь вредна

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

   Создает пул потоков, который повторно использует фиксированное количество потоков, работающих с общей неограниченной очередью. В любой момент, в большинстве случаев nThreads будут активными задачами обработки. Если дополнительные задачи передаются, когда все потоки активны, они будут ждать в очереди до тех пор, пока поток не будет доступен

   Случаи применения:

   1. Эффективное использование доступных ядер. Настроить `nThreads` как `Runtime.getRuntime().availableProcessors()`
   2. Когда вы решите, что количество потоков не должно превышать число в пуле потоков

   Минусы:

   1. Неограниченная очередь является вредной.

3. `public static ExecutorService newCachedThreadPool()`

   Создает пул потоков, который при необходимости создает новые потоки, но будет повторно использовать ранее созданные потоки, когда они будут доступны

   Случаи применения:

   1. Для недолговечных асинхронных задач

   Минусы:

   1. Неограниченная очередь является вредной.
   2. Каждая новая задача создаст новый поток, если все существующие потоки заняты. Если задача занимает много времени, будет создано больше потоков, что ухудшит производительность системы. Альтернатива в этом случае: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

   Создает пул потоков, который может планировать выполнение команд после заданной задержки или выполнять их периодически.

   Случаи применения:

   1. Обработка повторяющихся событий с задержками, которые будут происходить в

будущем в определенный промежуток времени

Минусы:

1. Неограниченная очередь является вредной.

5. `public static ExecutorService newWorkStealingPool()`

   Создает пул потоков, обрабатывающих работу, используя все доступные процессоры в качестве целевого уровня параллелизма

Случаи применения:

1. Для деления и преодоления типа проблем.
2. Эффективное использование простаивающих потоков. Холостые потоки крадут задачи из занятых потоков.

Минусы:

1. Неограниченный размер очереди вреден.

Вы можете увидеть один из общих недостатков во всех этих ExecutorService: неограниченная очередь. Это будет рассмотрено с помощью ThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

С `ThreadPoolExecutor` вы можете

1. Размер пула потоков управления динамически
2. Установите емкость для `BlockingQueue`
3. Определить `RejectionExecutionHander` когда очередь заполнена
4. `CustomThreadFactory` для добавления дополнительных функций во время создания темы `(public Thread newThread(Runnable r)`

Использование пулов потоков

В пулах потоков в основном используются методы вызова в `ExecutorService` .

Следующие способы могут быть использованы для отправки работ для выполнения:

метод	Описание
`submit`	Выполняет представленную работу и возвращает будущее, которое может быть использовано для получения результата
`execute`	Выполнять задачу в будущем, не получая никакого возвращаемого значения

метод	Описание
`invokeAll`	Выполните список задач и верните список фьючерсов
`invokeAny`	Выполняет все, но возвращает только результат успешной (без исключений)

Как только вы закончите с пулом потоков, вы можете вызвать `shutdown()` чтобы завершить пул потоков. Это выполняет все ожидающие задачи. Чтобы дождаться выполнения всех задач, вы можете выполнить цикл вокруг `awaitTermination` или `isShutdown()` .

Прочитайте Исполнители, Исполнительные службы и пулы потоков онлайн:
https://riptutorial.com/ru/java/topic/143/исполнители--исполнительные-службы-и-пулы-потоков

глава 80: Использование ThreadPoolExecutor в приложениях MultiThreaded.

Вступление

При создании приложения, работающего с данными и данными, может быть очень полезно для выполнения задач, требующих много времени, в асинхронном режиме и одновременного выполнения нескольких задач. В этом разделе будет представлена концепция использования ThreadPoolExecutors для одновременного выполнения нескольких ансинхронных задач.

Examples

Выполнение асинхронных задач, где не требуется возвращаемое значение с использованием экземпляра Runnable Class

Некоторые приложения могут захотеть создать так называемые задачи «Fire & Forget», которые могут периодически запускаться, и не нужно возвращать какой-либо тип значения, возвращаемый после завершения назначенной задачи (например, очистка старых файлов temp, вращающихся журналов, автосохранение государство).

В этом примере мы создадим два класса: один, который реализует интерфейс Runnable, и тот, который содержит метод main ().

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber,
timeout));
```

```
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
        executorService.shutdown();
    }
}
```

Запуск AsyncExample1.main () привел к следующему выводу:

```
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is sleeping for 18 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is sleeping for 6 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is sleeping for 14 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is sleeping for 10 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is sleeping for 7 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is sleeping for 8 seconds
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is done sleeping
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is done sleeping
Dec 28, 2016 2:21:09 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is done sleeping
Dec 28, 2016 2:21:10 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is done sleeping
Dec 28, 2016 2:21:11 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
```

```
INFO: Task 1 is done sleeping
Dec 28, 2016 2:21:13 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is done sleeping
Dec 28, 2016 2:21:17 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is done sleeping
Dec 28, 2016 2:21:21 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is done sleeping

Process finished with exit code 0
```

Замечания Примечание. В приведенном выше выводе есть несколько вещей,

1. Задачи не выполнялись в предсказуемом порядке.
2. Поскольку каждая задача спала за (псевдо) случайное количество времени, они не обязательно заполнялись в том порядке, в котором они были вызваны.

Выполнение асинхронных задач, в которых требуется возвращаемое значение Использование экземпляра класса вызываемого класса

Часто необходимо выполнить долговременную задачу и использовать результат этой задачи после ее завершения.

В этом примере мы создадим два класса: один, который реализует интерфейс Callable <T> (где T - тип, который мы хотим вернуть), и тот, который содержит метод main ().

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++){
            Future<Integer> submittedFuture = executorService.submit(new
AsyncValueTypeTaskCompleter(i));
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }
}
```

Запуск AsyncExample2.main () привел к следующему выводу:

```
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
```

```
INFO: Task 5 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is sleeping
Dec 28, 2016 3:07:16 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is done sleeping
Dec 28, 2016 3:07:16 PM AsyncExample2 main
INFO: A task just completed after sleeping for 1 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:19 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is done sleeping
Dec 28, 2016 3:07:19 PM AsyncExample2 main
INFO: A task just completed after sleeping for 4 seconds
Dec 28, 2016 3:07:20 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is done sleeping
Dec 28, 2016 3:07:20 PM AsyncExample2 main
INFO: A task just completed after sleeping for 5 seconds
Dec 28, 2016 3:07:21 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is done sleeping
Dec 28, 2016 3:07:21 PM AsyncExample2 main
INFO: A task just completed after sleeping for 6 seconds
Dec 28, 2016 3:07:25 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is done sleeping
Dec 28, 2016 3:07:25 PM AsyncExample2 main
INFO: A task just completed after sleeping for 10 seconds
Dec 28, 2016 3:07:27 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is done sleeping
Dec 28, 2016 3:07:27 PM AsyncExample2 main
INFO: A task just completed after sleeping for 12 seconds
Dec 28, 2016 3:07:29 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is done sleeping
Dec 28, 2016 3:07:29 PM AsyncExample2 main
INFO: A task just completed after sleeping for 14 seconds
Dec 28, 2016 3:07:31 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is done sleeping
Dec 28, 2016 3:07:31 PM AsyncExample2 main
INFO: A task just completed after sleeping for 16 seconds
```

Замечания:

В приведенном выше выводе есть несколько вещей,

1. Каждый вызов ExecutorService.submit () возвращал экземпляр Future, который был сохранен в списке для последующего использования
2. Будущее содержит метод с именем isDone (), который можно использовать для проверки того, была ли наша задача выполнена, прежде чем пытаться проверить ее возвращаемое значение. Вызов метода Future.get () в будущем, который еще не выполнен, блокирует текущий поток до завершения задачи, что потенциально отрицает многие преимущества, получаемые при выполнении задачи асинхронно.
3. Метод executeorService.shutdown () был вызван до проверки возвращаемых значений

объектов Future. Это не требуется, но было сделано таким образом, чтобы показать, что это возможно. Метод executorService.shutdown () не препятствует завершению задач, которые уже были отправлены в ExecutorService, а скорее предотвращает добавление новых задач в очередь.

Определение асинхронных задач Inline с использованием Lambdas

Хотя хороший дизайн программного обеспечения часто максимизирует повторное использование кода, иногда бывает полезно определить асинхронные задачи, встроенные в ваш код, с помощью выражений лямбда, чтобы максимизировать читаемость кода.

В этом примере мы создадим один класс, который содержит метод main (). Внутри этого метода мы будем использовать выражения Lambda для создания и выполнения экземпляров Callable и Runnable <T>.

AsyncExample3.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
                log.info(String.format("Runnable %d has been submitted and will sleep for %d
seconds", index, timeout));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Runnable %d has finished sleeping", index));
            });
            Future<Integer> submittedFuture = executorService.submit(() -> {
                int timeout = getTimeout();
                log.info(String.format("Callable %d will begin sleeping", index));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Callable %d is done sleeping", index));
                return timeout;
            });
            futures.add(submittedFuture);
        }
        executorService.shutdown();
```

```
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }

    public static int getTimeout(){
        return ThreadLocalRandom.current().nextInt(1, 20);
    }
}
```

Замечания:

В приведенном выше выводе есть несколько вещей,

1. Лямбда-выражения имеют доступ к переменным и методам, доступным для области, в которой они определены, но все переменные должны быть окончательными (или фактически окончательными) для использования внутри лямбда-выражения.
2. Нам не нужно указывать, является ли наше Lambda выражение Callable или Runnable <T> явно, возвращаемый тип автоматически выводится по типу возвращаемого значения.

Прочитайте Использование ThreadPoolExecutor в приложениях MultiThreaded. онлайн: https://riptutorial.com/ru/java/topic/8646/использование-threadpoolexecutor-в-приложениях-multithreaded-

глава 81: Использование других языков сценариев в Java

Вступление

Java сама по себе является чрезвычайно мощным языком, но ее мощность может быть расширена благодаря JSR223 (Java Specification Request 223), представляющему механизм сценария

замечания

API Java Scripting позволяет внешним скриптам взаимодействовать с Java

API-интерфейс Scripting API позволяет взаимодействовать между скриптом и java. Языки сценариев должны иметь реализацию Script Engine на пути к классам.

По умолчанию JavaScript (также известный как ECMAScript) предоставляется по умолчанию по умолчанию. Каждый скриптовый движок имеет контекст сценария, в котором все переменные, функции, методы хранятся в привязках. Иногда вы можете использовать несколько контекстов, поскольку они поддерживают перенаправление вывода на буферный Writer и ошибку на другую.

Есть много других библиотек сценариев, таких как Jython и JRuby. Пока они находятся на пути к классу, вы можете использовать код eval.

Мы можем использовать привязки для отображения переменных в скрипте. В некоторых случаях нам нужно несколько привязок, поскольку подверженность переменных движку в основном заключается в том, чтобы подвергать переменные только этому движку, иногда нам нужно выставлять определенные переменные, такие как системная среда и путь, одинаковый для всех двигателей того же типа. В этом случае нам требуется привязка, которая является глобальной областью. Отображение переменных, которые выставляют его всем движкам сценариев, созданным тем же EngineFactory

Examples

Оценка файла javascript в режиме -scripting nashorn

```
public class JSEngine {

    /*
    * Note Nashorn is only available for Java-8 onwards
    * You can use rhino from ScriptEngineManager.getEngineByName("js");
```

```
    */

    ScriptEngine engine;
    ScriptContext context;
    public Bindings scope;

    // Initialize the Engine from its factory in scripting mode
    public JSEngine(){
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");
        // Script context is an interface so we need an implementation of it
        context = new SimpleScriptContext();
        // Create bindings to expose variables into
        scope = engine.createBindings();
    }

    // Clear the bindings to remove the previous variables
    public void newBatch(){
        scope.clear();
    }

    public void execute(String file){
        try {
            // Get a buffered reader for input
            BufferedReader br = new BufferedReader(new FileReader(file));
            // Evaluate code, with input as bufferdReader
            engine.eval(br);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public void eval(String code){
        try {
            // Engine.eval basically treats any string as a line of code and evaluates it,
executes it
            engine.eval(code);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Apply the bindings to the context and set the engine's default context
    public void startBatch(int SCP){
        context.setBindings(scope, SCP);
        engine.setContext(context);
    }

    // We use the invocable interface to access methods from the script
    // Invocable is an optional interface, please check if your engine implements it
    public Invocable invocable(){
        return (Invocable)engine;
    }

}
```

Теперь основной метод

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.eval("print(x);");
}
```

Ваш результат должен быть похож на этот

```
hello world
```

Как вы видите, открытая переменная x была напечатана. Теперь тестирование с помощью файла.

> Здесь у нас есть test.js

```
print(x);
function test(){
    print("hello test.js:test");
}
test();
```

И обновленный основной метод

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.execute("./test.js");
}
```

Предполагая, что test.js находится в том же каталоге, что и ваше приложение, вы должны иметь аналогичный результат

```
hello world
hello test.js:test
```

Прочитайте Использование других языков сценариев в Java онлайн:
https://riptutorial.com/ru/java/topic/9926/использование-других-языков-сценариев-в-java

глава 82: Использование ключевого слова static

Синтаксис

- public static int myVariable; // Объявление статической переменной
- public static myMethod () {} // Объявление статического метода
- публичный статический окончательный двойной MY_CONSTANT; // Объявление константной переменной, которая разделяется между всеми экземплярами класса
- публичный финальный двойной MY_CONSTANT; // Объявление константной переменной, специфичной для этого экземпляра класса (наилучшим образом используемого в конструкторе, который генерирует другую константу для каждого экземпляра)

Examples

Использование static для объявления констант

Поскольку ключевое слово `static` используется для доступа к полям и методам без инстанцированного класса, его можно использовать для объявления констант для использования в других классах. Эти переменные останутся постоянными во всех экземплярах класса. По соглашению, `static` переменные всегда `ALL_CAPS` и используют `ALL_CAPS` подчеркивания, а не случай верблюда. например:

```
 static E STATIC_VARIABLE_NAME
```

Поскольку константы не могут измениться, `static` могут также использоваться с `final` модификатором:

Например, чтобы определить математическую константу pi:

```
public class MathUtilities {

    static final double PI = 3.14159265358

}
```

Который может использоваться в любом классе как константа, например:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
```

```
        return (2 * radius * MathUtilities.PI);
    }

}
```

Использование static с этим

Static предоставляет метод или переменную память, которая *не* выделяется для каждого экземпляра класса. Скорее, статическая переменная распределяется между всеми членами класса. Кстати, попытка рассматривать статическую переменную как члена экземпляра класса приведет к предупреждению:

```
public class Apple {
    public static int test;
    public int test2;
}

Apple a = new Apple();
a.test = 1; // Warning
Apple.test = 1; // OK
Apple.test2 = 1; // Illegal: test2 is not static
a.test2 = 1; // OK
```

Методы, объявленные статическими, ведут себя одинаково, но с дополнительным ограничением:

> Вы не можете использовать `this` ключевое слово в них!

```
public class Pineapple {

    private static int numberOfSpikes;
    private int age;

    public static getNumberOfSpikes() {
        return this.numberOfSpikes; // This doesn't compile
    }


    public static getNumberOfSpikes() {
        return numberOfSpikes; // This compiles
    }

}
```

В общем, лучше всего объявить общие методы, которые применяются к различным экземплярам класса (например, клонов) `static` , сохраняя при этом методы, подобные `equals()` как нестатические. `main` метод Java-программы всегда статичен, что означает, что ключевое слово `this` не может использоваться внутри `main()` .

Ссылка на нестатический элемент из статического контекста

Статические переменные и методы не являются частью экземпляра. Всегда будет одна

копия этой переменной независимо от того, сколько объектов вы создадите для определенного класса.

Например, вы можете иметь неизменный список констант, было бы неплохо сохранить его статическим и инициализировать его только один раз внутри статического метода. Это даст вам значительное увеличение производительности, если вы регулярно создаете несколько экземпляров определенного класса.

Кроме того, вы также можете иметь статический блок в классе. Вы можете использовать его для назначения значения по умолчанию статической переменной. Они выполняются только один раз, когда класс загружается в память.

Переменная экземпляра, как подсказывает название, зависит от экземпляра конкретного объекта, и они живут, чтобы служить прихотям. Вы можете играть с ними в течение определенного жизненного цикла объекта.

Все поля и методы класса, используемые внутри статического метода этого класса, должны быть статическими или локальными. Если вы попытаетесь использовать переменные или методы экземпляра (нестатические), ваш код не будет компилироваться.

```
public class Week {
    static int daysOfTheWeek = 7; // static variable
    int dayOfTheWeek; // instance variable

    public static int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this will cause errors
    }

    public int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this is valid
    }

    public static int getDaysLeftInTheWeek(int today){
        return Week.daysOfTheWeek-today; // this is valid
    }

}
```

Прочитайте Использование ключевого слова static онлайн:
https://riptutorial.com/ru/java/topic/2253/использование-ключевого-слова-static

глава 83: Итератор и Итерабель

Вступление

`java.util.Iterator` является стандартным интерфейсом Java SE для объекта, реализующего шаблон проектирования Iterator. Интерфейс `java.lang.Iterable` предназначен для объектов, которые могут *предоставить* итератор.

замечания

Можно выполнить итерацию по массиву с использованием цикла `for` -each, хотя массивы java не реализуют Iterable; итерация выполняется с помощью JVM с использованием недопустимого индекса в фоновом режиме.

Examples

Использование цикла Iterable in for

Классы, реализующие интерфейс `Iterable<>` могут использоваться `for` циклов. Это на самом деле только синтаксический сахар для получения итератора от объекта и использования его для получения всех элементов последовательно; он делает код более понятным, быстрее записывать конец, менее подверженный ошибкам.

```
public class UsingIterable {

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);

        // List extends Collection, Collection extends Iterable
        Iterable<Integer> iterable = intList;

        // foreach-like loop
        for (Integer i: iterable) {
            System.out.println(i);
        }

        // pre java 5 way of iterating loops
        for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
            Integer item = i.next();
            System.out.println(item);
        }
    }
}
```

Использование исходного итератора

Хотя использование цикла foreach (или «расширенный для цикла») прост, иногда полезно

использовать итератор напрямую. Например, если вы хотите вывести кучу разделенных запятыми значений, но не хотите, чтобы последний элемент имел запятую:

```
List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() "moves" the iterator to the next entry and returns it's value.
    String entry = iterator.next();
    System.out.print(entry);
    if (iterator.hasNext()){
        // If the iterator has another element after the current one:
        System.out.print(",");
    }
}
```

Это намного проще и понятнее, чем наличие переменной `isLastEntry` или выполнение вычислений с индексом цикла.

Создание собственного Iterable.

Чтобы создать свой собственный Iterable, как с любым интерфейсом, вы просто реализуете абстрактные методы в интерфейсе. Для `Iterable` существует только один, который называется `iterator()` . Но его возвращаемый тип `Iterator` сам по себе является интерфейсом с тремя абстрактными методами. Вы можете вернуть итератор, связанный с какой-либо коллекцией, или создать собственную собственную реализацию:

```
public static class Alphabet implements Iterable<Character> {

    @Override
    public Iterator<Character> iterator() {
        return new Iterator<Character>() {
            char letter = 'a';

            @Override
            public boolean hasNext() {
                return letter <= 'z';
            }

            @Override
            public Character next() {
                return letter++;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException("Doesn't make sense to remove a
letter");
            }
        };
    }
}
```

Использовать:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
    }
}
```

Новый `Iterator` должен иметь состояние, указывающее на первый элемент, каждый вызов следующего обновляет его состояние, чтобы указать на следующий. Функция `hasNext()` проверяет, находится ли итератор в конце. Если итератор был связан с изменяемой коллекцией, то может быть реализован необязательный метод `remove()` итератора, чтобы удалить элемент, указанный в настоящее время из базовой коллекции.

Удаление элементов с помощью итератора

Метод `Iterator.remove()` является необязательным методом, который удаляет элемент, возвращенный предыдущим вызовом, в `Iterator.next()` . Например, следующий код заполняет список строк, а затем удаляет все пустые строки.

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String el = it.next();
  if (el.equals("")) {
    it.remove();
  }
}
System.out.println("New Size : " + names.size());
```

Выход :

```
Old Size : 5
New Size : 3
```

Обратите внимание, что приведенный выше код является безопасным способом удаления элементов при повторении типичной коллекции. Если вместо этого вы пытаетесь удалить элементы из коллекции следующим образом:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

типичная коллекция (такая как `ArrayList`), которая предоставляет итераторам с *быстрой* семантикой итератора с *ошибкой,* выдает исключение `ConcurrentModificationException` .

Метод `remove()` может вызываться только один раз после `next()` вызова `next()` . Если он вызывается перед вызовом `next()` или если он вызывается дважды после вызова `next()` вызов `remove()` будет вызывать `IllegalStateException` .

Операция `remove` описывается как *необязательная* операция; т.е. не все итераторы это позволят. Примеры, в которых он не поддерживается, включают в себя итераторы для неизменных коллекций, представления только для чтения коллекций или коллекции фиксированного размера. Если `remove()` вызывается, когда итератор не поддерживает удаление, он `UnsupportedOperationException` исключение `UnsupportedOperationException` .

Прочитайте Итератор и Итерабель онлайн: https://riptutorial.com/ru/java/topic/172/итератор-и-итерабель

глава 84: Календарь и его подклассы

замечания

Начиная с Java 8, `Calendar` и его подклассы были заменены пакетом java.time и его подпакетами. Они должны быть предпочтительными, если для устаревшего API не требуется Календарь.

Examples

Создание объектов календаря

Объекты `Calendar` могут быть созданы с помощью `getInstance()` или с помощью конструктора `GregorianCalendar` .

Важно отметить, что месяцы в `Calendar` основаны на нулевом значении, что означает, что JANUARY представлен значением `int` 0. Для обеспечения лучшего кода всегда используйте константы `Calendar` , такие как `Calendar.JANUARY` чтобы избежать недоразумений.

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY,
1, 6, 55, 10);
```

Примечание . Всегда используйте константы месяца: числовое представление вводит в заблуждение , например `Calendar.JANUARY` имеет значение `0`

Увеличение / уменьшение полей календаря

`add()` и `roll()` могут использоваться для увеличения / уменьшения полей `Calendar` .

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

Метод `add()` влияет на все поля и ведет себя эффективно, если нужно добавить или вычесть фактические даты из календаря

```
calendar.add(Calendar.MONTH, -6);
```

Вышеуказанная операция удаляет шесть месяцев из календаря, возвращая нас к 30 сентября 2015 года.

Чтобы изменить конкретное поле, не влияя на другие поля, используйте `roll()` .

```
calendar.roll(Calendar.MONTH, -6);
```

Вышеупомянутая операция удаляет шесть месяцев с текущего *месяца* , поэтому месяц идентифицируется как сентябрь. Никакие другие поля не были скорректированы; год не изменился с этой операцией.

Поиск AM / PM

С классом Calendar легко найти AM или PM.

```
  Calendar cal = Calendar.getInstance();
  cal.setTime(new Date());
  if (cal.get(Calendar.AM_PM) == Calendar.PM)
      System.out.println("It is PM");
```

Выделение календарей

Чтобы получить разницу между двумя `Calendar` , используйте `getTimeInMillis()` :

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //outputs 86400000 (24 * 60 *
60 * 1000)
```

Прочитайте Календарь и его подклассы онлайн: https://riptutorial.com/ru/java/topic/165/календарь-и-его-подклассы

глава 85: Карта Enum

Вступление

Класс Java EnumMap - это специализированная реализация Map для ключей перечисления. Он наследует классы Enum и AbstractMap.

Параметры для класса java.util.EnumMap.

K: Это тип ключей, поддерживаемых этой картой. V: Это тип отображаемых значений.

Examples

Пример карты карты Enum

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class EnumMapExample {
// Creating enum
    public enum Key{
           One, Two, Three
           };
public static void main(String[] args) {
    EnumMap<Key, Book> map = new EnumMap<Key, Book>(Key.class);
    // Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    // Adding Books to Map
       map.put(Key.One, b1);
       map.put(Key.Two, b2);
       map.put(Key.Three, b3);
    // Traversing EnumMap
       for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

        }
}
}
```

Прочитайте Карта Enum онлайн: https://riptutorial.com/ru/java/topic/10158/карта-enum

глава 86: Карты

Вступление

Интерфейс java.util.Map представляет собой сопоставление между ключами и их значениями. Карта не может содержать дубликаты ключей; и каждый ключ может отображать не более одного значения.

Поскольку `Map` является интерфейсом, вам необходимо создать конкретную реализацию этого интерфейса, чтобы использовать его; существует несколько реализаций `Map` , и в основном используются `java.util.HashMap` и `java.util.TreeMap`

замечания

Карта - это объект, который хранит *ключи* со связанным *значением* для каждого ключа. Ключ и его значение иногда называют *парой ключ / значение* или *записью* . Карты обычно предоставляют следующие возможности:

- Данные сохраняются на карте в парах ключ / значение.
- Карта может содержать только одну запись для определенного ключа. Если карта содержит запись с определенным ключом, и вы пытаетесь сохранить вторую запись с тем же ключом, вторая запись заменит первую. Другими словами, это изменит значение, связанное с ключом.
- Карты обеспечивают быструю работу, чтобы проверить, существует ли ключ на карте, чтобы получить значение, связанное с ключом, и удалить пару ключ / значение.

Наиболее часто используемой реализацией карты является HashMap . Он хорошо работает с ключами, которые являются строками или цифрами.

Обычные карты, такие как HashMap, неупорядочены. Итерация через пары ключ / значение может возвращать отдельные записи в любом порядке. Если вам необходимо выполнять итерацию через записи карты контролируемым образом, вы должны посмотреть на следующее:

- Сортированные карты, такие как TreeMap, будут перебирать ключи в натуральном порядке (или в порядке, который вы можете указать, предоставив Comparator). Например, сортированная карта, использующая числа как ключи, должна была бы перебирать свои записи в числовом порядке.

- LinkedHashMap позволяет выполнять итерацию через записи в том же порядке, в котором они были вставлены в карту, или по порядку последнего доступа.

Examples

Добавить элемент

1. прибавление

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Выход: `First element.`

2. Override

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Выход: `New element.`

`HashMap` качестве примера используется `HashMap` . Могут использоваться и другие реализации, реализующие интерфейс `Map` .

Добавить несколько элементов

Мы можем использовать `V put(K key,V value)` :

> Связывает указанное значение с указанным ключом на этой карте (дополнительная операция). Если в карте ранее содержалось сопоставление для ключа, старое значение заменяется указанным значением.

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal);// Will print null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // Will print null yet again
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // will print Second element.
System.out.println(map.size()); // Will print 2 as key having
// value 2 was replaced.

Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

Выход:

```
3
```

Чтобы добавить много элементов, вы можете использовать внутренние классы следующим образом:

```
Map<Integer, String> map = new HashMap<>() {{
    // This is now an anonymous inner class with an unnamed instance constructor
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}};
```

Имейте в виду, что создание анонимного внутреннего класса не всегда эффективно и может привести к утечке памяти, поэтому, когда это возможно, вместо этого используйте блок инициализатора:

```
static Map<Integer, String> map = new HashMap<>();

static {
    // Now no inner classes are created so we can avoid memory leaks
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}
```

Приведенный выше пример делает карту статической. Его также можно использовать в нестационарном контексте, удалив все вхождения `static` .

В дополнение к тому, что большинство реализаций поддерживают `putAll` , которые могут добавлять все записи в одну карту другому:

```
another.putAll(one);
```

Использование методов сопоставления по умолчанию из Java 8

Примеры использования методов по умолчанию, введенных в Java 8 в интерфейсе карты

1. Использование **getOrDefault**

Возвращает значение, отображаемое на ключ, или если ключ отсутствует, возвращает значение по умолчанию

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element");
map.get(1);                                 // => First element
map.get(2);                                 // => null
map.getOrDefault(2, "Default element");     // => Default element
```

2. Использование **forEach**

Позволяет выполнять операцию, указанную в «действии» для каждой записи карты

```
 Map<Integer, String> map = new HashMap<Integer, String>();
 map.put(1, "one");
 map.put(2, "two");
 map.put(3, "three");
 map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));

  // Key: 1 :: Value: one
  // Key: 2 :: Value: two
  // Key: 3 :: Value: three
```

3. Использование **replaceAll**

Будет заменено новым значением, только если присутствует ключ

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replaceAll((key,value)->value+10);   //{john=30, paul=40, peter=50}
```

4. Использование **putIfAbsent**

Пара ключей-значений добавляется к карте, если ключ отсутствует или отображается на нуль

```
   Map<String, Integer> map = new HashMap<String, Integer>();
   map.put("john", 20);
   map.put("paul", 30);
   map.put("peter", 40);
   map.putIfAbsent("kelly", 50);     //{john=20, paul=30, peter=40, kelly=50}
```

5. Использование **remove**

   Удаляет ключ только в том случае, если он связан с заданным значением

   ```
   Map<String, Integer> map = new HashMap<String, Integer>();
   map.put("john", 20);
   map.put("paul", 30);
   map.put("peter", 40);
   map.remove("peter",40); //{john=30, paul=40}
   ```

6. Использование **замены**

   Если ключ присутствует, значение заменяется новым значением. Если ключ отсутствует, ничего не делает.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
```

```
map.put("paul", 30);
map.put("peter", 40);
map.replace("peter",50); //{john=20, paul=30, peter=50}
map.replace("jack",60); //{john=20, paul=30, peter=50}
```

7. Использование **computeIfAbsent**

Этот метод добавляет запись на Карту. ключ указан в функции, и значение является результатом применения функции сопоставления

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.computeIfAbsent("kelly", k->map.get("john")+10); //{john=20, paul=30, peter=40,
kelly=30}
    map.computeIfAbsent("peter", k->map.get("john")+10); //{john=20, paul=30, peter=40,
kelly=30} //peter already present
```

8. Использование **computeIfPresent**

Этот метод добавляет запись или изменяет существующую запись на Карте. Ничего не делает, если запись с этим ключом отсутствует

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.computeIfPresent("kelly", (k,v)->v+10); //{john=20, paul=30, peter=40} //kelly not
present
    map.computeIfPresent("peter", (k,v)->v+10); //{john=20, paul=30, peter=50} // peter
present, so increase the value
```

9. Использование **вычисления**

Этот метод заменяет значение ключа на новое вычисляемое значение

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.compute("peter", (k,v)->v+50); //{john=20, paul=30, peter=90} //Increase the value
```

10. Использование **слияния**

Добавляет пару ключ-значение к карте, если ключ отсутствует, или значение для ключа равно null Заменяет значение на вновь вычисленное значение, если ключ присутствует. Ключ удаляется с карты, если новое вычисляемое значение равно null

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
```

```
    map.put("peter", 40);

    //Adds the key-value pair to the map, if key is not present or value for the key is null
    map.merge("kelly", 50 , (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40,
kelly=50}

    //Replaces the value with the newly computed value, if the key is present
    map.merge("peter", 50 , (k,v)->map.get("john")+10); //{john=20, paul=30, peter=30,
kelly=50}

    //Key is removed from the map , if new value computed is null
    map.merge("peter", 30 , (k,v)->map.get("nancy")); //{john=20, paul=30, kelly=50}
```

Очистить карту

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "First element.");
map.put(2, "Second element.");
map.put(3, "Third element.");

map.clear();

System.out.println(map.size());   // => 0
```

Итерация через содержимое Карты

Карты предоставляют методы, позволяющие вам получить доступ к ключам, значениям или парам ключ-значение карты в виде коллекций. Вы можете проходить через эти коллекции. Например, на следующей карте:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Итерация с помощью клавиш карты:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

Печать:

> Дарин Димитров
> Джон Скит
> BalusC

`keySet()` предоставляет ключи карты в виде `Set` . `Set` используется, поскольку ключи не могут содержать повторяющиеся значения. Итерация через набор дает каждый ключ по очереди. HashMaps не заказываются, поэтому в этом примере ключи могут быть

возвращены в любом порядке.

Итерирование по значениям карты:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

Печать:

```
715567
927654
708826
```

`values()` возвращает значения карты как `Collection` . Итерация через коллекцию дает каждое значение по очереди. Опять же, значения могут быть возвращены в любом порядке.

Итерация через ключи и значения вместе

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

Печать:

```
Дарин Димитров = 715567
Jon Skeet = 927654
BalusC = 708826
```

`entrySet()` возвращает коллекцию объектов `Map.Entry` . Map.Entry предоставляет доступ к ключу и значению для каждой записи.

Объединение, объединение и составление карт

Используйте `putAll` чтобы поместить каждого члена одной карты в другую. Ключи, уже присутствующие на карте, будут перезаписаны соответствующими значениями.

```
Map<String, Integer> numbers = new HashMap<>();
numbers.put("One", 1)
numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)
```

Это дает следующее отображение в `numbers` :

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

Если вы хотите комбинировать значения вместо их перезаписи, вы можете использовать `Map.merge` , добавленный в Java 8, который использует предоставленную пользователем `BiFunction` для объединения значений для дубликатов ключей. `merge` работает с отдельными ключами и значениями, поэтому вам нужно будет использовать цикл или `Map.forEach` . Здесь мы объединяем строки для дубликатов ключей:

```
for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));
```

Если вы хотите принудительно использовать ограничение, нет дубликатов ключей, вы можете использовать функцию слияния, которая генерирует `AssertionError` :

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Составление карты <X, Y> и карты <Y, Z> для получения карты <X, Z>

Если вы хотите составить два сопоставления, вы можете сделать это следующим образом

```
    Map<String, Integer> map1 = new HashMap<String, Integer>();
    map1.put("key1", 1);
    map1.put("key2", 2);
    map1.put("key3", 3);

    Map<Integer, Double> map2 = new HashMap<Integer, Double>();
    map2.put(1, 1.0);
    map2.put(2, 2.0);
    map2.put(3, 3.0);

    Map<String, Double> map3 = new new HashMap<String, Double>();
    map1.forEach((key,value)->map3.put(key,map2.get(value)));
```

Это дает следующее отображение

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Проверьте, существует ли ключ

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Карты могут содержать нулевые значения

Для карт нужно быть уверенным, чтобы не путать «содержащий ключ» с «имеющим ценность». Например, `HashMap` s может содержать нуль, что означает, что следующее абсолютно нормальное поведение:

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

Более формально, нет гарантии, что `map.contains(key) <=> map.get(key)!=null`

Итерирование картографических записей Эффективно

В этом разделе приведены коды и контрольные показатели для десяти уникальных реализаций примеров, которые перебирают записи `Map<Integer, Integer>` и генерируют сумму значений `Integer` . Все примеры имеют алгоритмическую сложность `Θ(n)` , однако критерии все еще полезны для того, чтобы дать представление о том, какие реализации более эффективны в среде «реального мира».

1. Реализация с использованием **`Iterator`** с **`Map.Entry`**

```
    Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry<Integer, Integer> pair = it.next();
        sum += pair.getKey() + pair.getValue();
    }
```

2. Реализация с использованием **`for`** с **`Map.Entry`**

```
    for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
        sum += pair.getKey() + pair.getValue();
    }
```

3. Реализация с использованием **`Map.forEach`** (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Реализация с использованием `Map.keySet for`

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Реализация с использованием `Map.keySet` с `Iterator`

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Реализация с использованием `for` с `Iterator` и `Map.Entry`

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
         map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. Реализация с использованием `Stream.forEach` (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Реализация с использованием `Stream.forEach` с `Stream.parallel` (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
   .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Внедрение с использованием `IterableMap` из коллекций Apache

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Реализация с использованием `MutableMap` из коллекций Eclipse

```
 mutableMap.forEachKeyValue((key, value) -> {
     sum += key + value;
 });
```

Тесты производительности (*код доступен в Github*)

Условия тестирования: Windows 8.1 64-бит, Intel i7-4790 3,60 ГГц, 16 ГБ

1. Средняя производительность 10 испытаний (100 элементов) Лучшее: 308 ± 21 нс / оп

```
Benchmark                            Score     Error  Units
test3_UsingForEachAndJava8             308  ±     21  ns/op
test10_UsingEclipseMutableMap          309  ±      9  ns/op
test1_UsingWhileAndMapEntry            380  ±     14  ns/op
test6_UsingForAndIterator              387  ±     16  ns/op
test2_UsingForEachAndMapEntry          391  ±     23  ns/op
test7_UsingJava8StreamAPI              510  ±     14  ns/op
test9_UsingApacheIterableMap           524  ±      8  ns/op
test4_UsingKeySetAndForEach            816  ±     26  ns/op
test5_UsingKeySetAndIterator           863  ±     25  ns/op
test8_UsingJava8StreamAPIParallel     5552  ±    185  ns/op
```

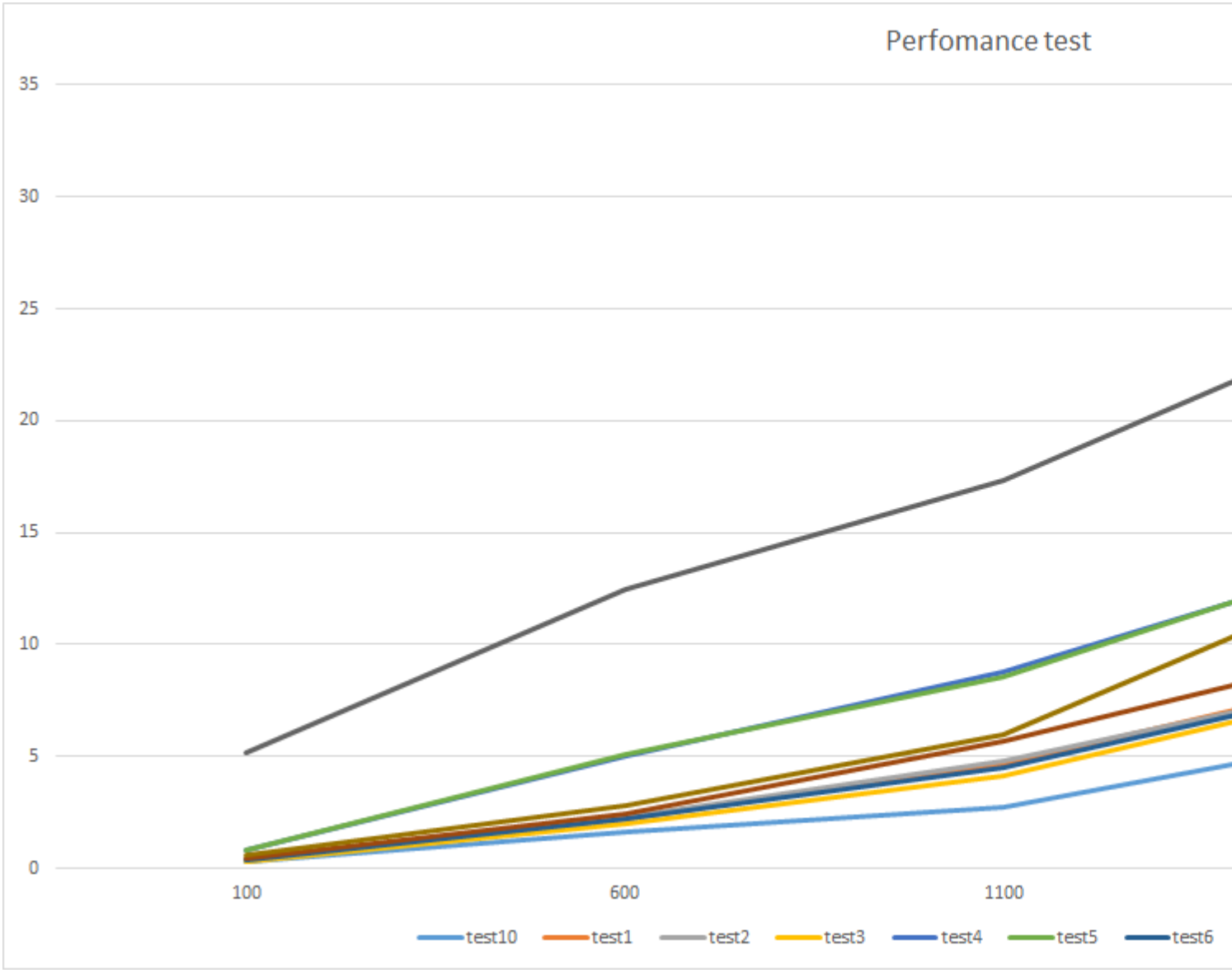
2. Средняя производительность 10 испытаний (10000 элементов) Лучшее: 37,606 ± 0,790 мкс / оп

```
Benchmark                            Score     Error  Units
test10_UsingEclipseMutableMap        37606  ±    790  ns/op
test3_UsingForEachAndJava8           50368  ±    887  ns/op
test6_UsingForAndIterator            50332  ±    507  ns/op
test2_UsingForEachAndMapEntry        51406  ±   1032  ns/op
test1_UsingWhileAndMapEntry          52538  ±   2431  ns/op
test7_UsingJava8StreamAPI            54464  ±    712  ns/op
test4_UsingKeySetAndForEach          79016  ±  25345  ns/op
test5_UsingKeySetAndIterator         91105  ±  10220  ns/op
test8_UsingJava8StreamAPIParallel   112511  ±    365  ns/op
test9_UsingApacheIterableMap        125714  ±   1935  ns/op
```

3. Средняя производительность 10 испытаний (100000 элементов) Лучшее: 1184,767 ± 332,968 мкс / оп

```
Benchmark                              Score       Error  Units
test1_UsingWhileAndMapEntry         1184.767  ±  332.968  µs/op
test10_UsingEclipseMutableMap       1191.735  ±  304.273  µs/op
test2_UsingForEachAndMapEntry       1205.815  ±  366.043  µs/op
test6_UsingForAndIterator           1206.873  ±  367.272  µs/op
test8_UsingJava8StreamAPIParallel   1485.895  ±  233.143  µs/op
test5_UsingKeySetAndIterator        1540.281  ±  357.497  µs/op
test4_UsingKeySetAndForEach         1593.342  ±  294.417  µs/op
test3_UsingForEachAndJava8          1666.296  ±  126.443  µs/op
test7_UsingJava8StreamAPI           1706.676  ±  436.867  µs/op
test9_UsingApacheIterableMap        3289.866  ± 1445.564  µs/op
```

4. Сравнение вариаций производительности в зависимости от размера карты




```
      x: Size of Map
   f(x): Benchmark Score (µs/op)

                100      600     1100     1600     2100
       ------------------------------------------------
       10   |   0.333    1.631    2.752    5.937    8.024
        3   |   0.309    1.971    4.147    8.147   10.473
        6   |   0.372    2.190    4.470    8.322   10.531
        1   |   0.405    2.237    4.616    8.645   10.707
Tests   2   |   0.376    2.267    4.809    8.403   10.910
 f(x)   7   |   0.473    2.448    5.668    9.790   12.125
        9   |   0.565    2.830    5.952    13.22   16.965
        4   |   0.808    5.012    8.813   13.939   17.407
        5   |   0.81     5.104    8.533   14.064   17.422
        8   |   5.173   12.499   17.351   24.671   30.403
```

Использовать пользовательский объект в качестве ключа

Прежде чем использовать свой собственный объект в качестве ключа, вы должны переопределить метод hashCode () и equals () вашего объекта.

В простом случае у вас будет что-то вроде:

```
class MyKey {
    private String name;
    MyKey(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof MyKey) {
            return this.name.equals(((MyKey)obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

`hashCode` определит, какой хэш-ведро принадлежит ключу, и `equals` будет решать, какой объект внутри этого хэш-ведра.

Без этого метода ссылка на ваш объект будет использоваться для сравнения выше, которое не будет работать, если вы не используете одну и ту же ссылку на объект каждый раз.

Использование HashMap

HashMap - это реализация интерфейса Map, который предоставляет структуру данных для хранения данных в парах Key-Value.

1. Объявление HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType и ValueType должны быть допустимыми типами на Java, такими как - String, Integer, Float или любой пользовательский класс, например Employee, Student и т. Д.

Например: `Map<String,Integer> myMap = new HashMap<String,Integer>();`

2. Ввод значений в HashMap.

Чтобы поместить значение в HashMap, мы должны вызвать метод `put` в объекте HashMap, передав параметры Key и Value as.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

Если вы вызываете метод put с ключом, который уже существует в Map, метод переопределяет его значение и возвращает старое значение.

3. Получение значений из HashMap.

Для получения значения из HashMap вам нужно вызвать метод `get` , передав ключ в качестве параметра.

```
myMap.get("key1");    //return 1 (class Integer)
```

Если вы передадите ключ, который не существует в HashMap, этот метод вернет значение `null`

4. Проверьте, находится ли ключ на карте или нет.

```
myMap.containsKey(varKey);
```

5. Проверьте, находится ли значение на карте или нет.

```
myMap.containsValue(varValue);
```

Вышеуказанные методы возвращают `boolean` значение true или false, если ключ, значение существует на карте или нет.

Создание и инициализация карт

Вступление

`Maps` хранят пары ключ / значение, где каждый ключ имеет связанное значение. Учитывая определенный ключ, карта может быстро найти связанное значение.

`Maps` , также известные как ассоциированный массив, представляют собой объект, который хранит данные в виде ключей и значений. В Java карты представлены с использованием интерфейса Map, который не является расширением интерфейса коллекции.

- Путь 1: -

```
/*J2SE < 5.0*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Путь 2: -

```
/*J2SE 5.0+ style (use of generics):*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Путь 3: -

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Путь 4: -

```
Map<String, Object> map = new TreeMap<String, Object>();
    map.put("name", "A");
    map.put("address", "Malviya-Nagar");
    map.put("city", "Jaipur");
System.out.println(map);
```

- Путь 5: -

```
//Java 8
final Map<String, String> map =
    Arrays.stream(new String[][] {
        { "name", "A" },
        { "address", "Malviya-Nagar" },
        { "city", "jaipur" },
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));
System.out.println(map);
```

- Путь 6: -

```
//This way for initial a map in outside the function
final static Map<String, String> map;
static
{
    map = new HashMap<String, String>();
    map.put("a", "b");
    map.put("c", "d");
}
```

- Путь 7: - Создание неизменной карты с одним ключом.

```
  //Immutable single key-value map
  Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Обратите внимание, что **эту карту невозможно изменить** .

Любые attemts для изменения карты приведут к бросанию UnsupportedOperationException.

```
    //Immutable single key-value pair
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
    singletonMap.put("newKey", "newValue"); //will throw UnsupportedOperationException
    singletonMap.putAll(new HashMap<>()); //will throw UnsupportedOperationException
    singletonMap.remove("key"); //will throw UnsupportedOperationException
    singletonMap.replace("key", "value", "newValue"); //will throw
UnsupportedOperationException
    //and etc
```

Прочитайте Карты онлайн: https://riptutorial.com/ru/java/topic/105/карты

глава 87: Класс - отражение Java

Вступление

Класс java.lang.Class предоставляет множество методов, которые можно использовать для получения метаданных, изучения и изменения поведения времени выполнения класса.

Пакеты java.lang и java.lang.reflect предоставляют классы для отражения java.

Где он используется

API Reflection в основном используется в:

IDE (интегрированная среда разработки), например Eclipse, MyEclipse, NetBeans и т. Д. Инструменты тестирования отладчика и т. Д.

Examples

Метод getClass () класса Object

```
class Simple { }

class Test {
     void printName(Object obj){
         Class c = obj.getClass();
         System.out.println(c.getName());
     }
     public static void main(String args[]){
         Simple s = new Simple();

         Test t = new Test();
         t.printName(s);
    }
}
```

Прочитайте Класс - отражение Java онлайн: https://riptutorial.com/ru/java/topic/10151/класс---отражение-java

глава 88: Класс EnumSet

Вступление

Класс Java EnumSet - это специализированная реализация Set для использования с типами перечислений. Он наследует класс AbstractSet и реализует интерфейс Set.

Examples

Пример набора Enum

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

Прочитайте Класс EnumSet онлайн: https://riptutorial.com/ru/java/topic/10159/класс-enumset

глава 89: Класс java.util.Objects

Examples

Основное использование для проверки нулевого объекта

Для метода нулевой проверки

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

Недействительный метод проверки

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Использование ссылки метода Objects.nonNull () в потоке api

В старом способе для нулевой проверки коллекции

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

С `Objects.nonNull` метода `Objects.nonNull` и Java8 Stream API мы можем сделать следующее:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
           .filter(Objects::nonNull)
           .forEach(this::doSomething);
```

Прочитайте Класс java.util.Objects онлайн: https://riptutorial.com/ru/java/topic/5768/класс-java-util-objects

глава 90: Класс даты

Синтаксис

- `Date object = new Date();`
- `Date object = new Date(long date);`

параметры

параметр	объяснение
Нет параметров	Создает новый объект Date с использованием времени выделения (до ближайшей миллисекунды)
длительная дата	Создает новый объект Date с временем, установленным на миллисекунды с «эпохи» (1 января 1970 года, 00:00:00 GMT)

замечания

Представление

Внутренне объект Java Date представлен как длинный; это число миллисекунд с определенного времени (называемое *эпохой*). У исходного класса Java Date были методы для работы с часовыми поясами и т. Д., Но они были устаревшими в пользу нового класса Calendar.

Поэтому, если все, что вы хотите сделать в своем коде, представляет определенное время, вы можете создать класс Date и сохранить его и т. Д. Однако, если вы хотите распечатать человекочитаемую версию этой даты, вы создаете класс Calendar и использовать его форматирование для создания часов, минут, секунд, дней, часовых поясов и т. д. Имейте в виду, что конкретная миллисекунда отображается как разные часы в разных часовых поясах; обычно вы хотите отображать его в «локальном» часовом поясе, но методы форматирования должны учитывать, что вы можете отобразить его для какого-либо другого.

Также имейте в виду, что часы, используемые JVM, обычно не имеют миллисекундной точности; часы могут «гасить» каждые 10 миллисекунд, и поэтому, если вы делаете тактику времени, вы не можете полагаться на точные измерения на этом уровне.

Импортный отчет

```
import java.util.Date;
```

Класс `Date` может быть импортирован из пакета `java.util` .

предосторожность

Экземпляры `Date` изменяемы, поэтому их использование может затруднить запись потокобезопасного кода или может случайно предоставить доступ на запись во внутреннее состояние. Например, в классе ниже метод `getDate()` позволяет вызывающему пользователю изменить дату транзакции:

```
public final class Transaction {
  private final Date date;

  public Date getTransactionDate() {
    return date;
  }
}
```

Решение состоит в том, чтобы либо вернуть копию поля `date` либо использовать новые API в `java.time` введенные в Java 8.

Большинство методов конструктора в классе `Date` устарели и не должны использоваться. Практически во всех случаях рекомендуется использовать класс `Calendar` для операций с датой.

Java 8

Java 8 вводит новый API времени и даты в пакет `java.time` , включая LocalDate и LocalTime . Классы в пакете `java.time` предоставляют переработанный API, который проще в использовании. Если вы пишете на Java 8, настоятельно рекомендуется использовать этот новый API. См. « Даты и время» (java.time. *) .

Examples

Создание объектов Date

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Здесь этот объект `Date` содержит текущую дату и время создания этого объекта.

```
Calendar calendar = Calendar.getInstance();
calendar.set(90, Calendar.DECEMBER, 11);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990
```

Объекты `Date` лучше всего создавать с помощью экземпляра `Calendar` поскольку использование конструкторов данных устарело и не рекомендуется. Для этого нам нужно получить экземпляр класса `Calendar` из заводского метода. Затем мы можем установить

год, месяц и день месяца, используя числа или в случае месячных констант, предоставленных классу Calendar для повышения удобочитаемости и уменьшения ошибок.

```
calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);
Date myBirthDatenTime = calendar.getTime();
System.out.println(myBirthDatenTime); // Mon Dec 31 08:32:35 IST 1990
```

Наряду с датой мы также можем пропустить время в часах, минутах и секундах.

Сравнение объектов Date

Календарь, дата и локальная дата

Java SE 8

до, после, compareTo и equals методы

```
//Use of Calendar and Date objects
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//Before example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//After example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//Compare example
System.out.printf("Compare %1$tF to %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("Compare %1$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("Compare %2$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//Equal example
```

```
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", birthdate, samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
            "Because birthdate.getTime() -> %1$d is different from samebirthdate.getTime() ->
%2$d, there are millisecondes!%n",
            Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//Clear ms from calendars
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("Is %1$tF equal to %2$tF after clearing ms? %3$b%n", birthdate,
samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
```

Java SE 8

isBefore, isAfter, compareTo и equals методы

```
//Use of LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

//Hours, minutes, second and nanoOfsecond can also be configured with an other class
LocalDateTime
//LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

//isBefore example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo example
System.out.printf("Compare %1$tF to %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));
System.out.printf("Compare %1$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("Compare %2$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));
```

```
//equals example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));

//isEqual example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));
```

Сравнение даты перед Java 8

До Java 8 даты можно было сравнить с помощью классов java.util.Calendar и java.util.Date . Класс Date предлагает 4 метода сравнения дат:

- после (дата когда)
- before (Дата когда)
- compareTo (Date anotherDate)
- equals (Object obj)

`after` , `before` , `compareTo` и `equals` сравнивают значения, возвращаемые методом getTime () для каждой даты.

Метод `compareTo` возвращает положительное целое число.

- Значение больше 0: когда Date после аргумента Date
- Значение больше 0: когда дата предшествует аргументу Date
- Значение равно 0: когда Date равно аргументу Date

`equals` результаты могут быть неожиданными, как показано в примере, потому что значения, такие как миллисекунды, не инициализируются с тем же значением, если явно не указано.

Поскольку Java 8

С Java 8 доступен новый объект для работы с датой java.time.LocalDate . `LocalDate` реализует ChronoLocalDate , абстрактное представление даты, в которой хронология или система календаря подключается.

Чтобы иметь точность даты даты, необходимо использовать объект java.time.LocalDateTime . `LocalDate` и `LocalDateTime` используют `LocalDate` и то же имя метода для сравнения.

Сравнение дат с использованием `LocalDate` отличается от использования `ChronoLocalDate` потому что хронология или система календаря не учитываются в первом.

Поскольку большинство приложений должны использовать `LocalDate` , `ChronoLocalDate` не включен в примеры. Далее читайте здесь .

> Большинство приложений должны объявлять сигнатуры методов, поля и переменные как LocalDate, а не этот интерфейс [ChronoLocalDate].

`LocalDate` имеет 5 методов для сравнения дат:

- isAfter (ChronoLocalDate другой)
- isBefore (ChronoLocalDate другой)
- isEqual (ChronoLocalDate другой)
- compareTo (ChronoLocalDate другой)
- equals (Object obj)

В случае параметра `LocalDate` , `isAfter` , `isBefore` , `isEqual` , `equals` и `compareTo` теперь используют этот метод:

```
int compareTo0(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

`equals` method проверяет, соответствует ли значение параметра первой дате, тогда как `isEqual` напрямую вызывает `compareTo0` .

В случае другого экземпляра класса `ChronoLocalDate` даты сравниваются с использованием дня `Epoch Day` . Количество дней Epoch Day - это простое увеличение количества дней, в которых день 0 равен 1970-01-01 (ISO).

Преобразование даты в определенный формат String

`format()` из класса `SimpleDateFormat` помогает преобразовать объект `Date` в определенный объект `String` формата, используя прилагаемую *строку шаблона* .

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Шаблоны можно снова применить с помощью `applyPattern()`

```
dateFormat.applyPattern("dd-MM-yyyy");
System.out.println(dateFormat.format(today)); //25-02-2016

dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

Примечание: Здесь `mm` (малая буква m) обозначает минуты, а `MM` (капитал M) обозначает месяц. Обращайте пристальное внимание при форматировании лет: капитал «Y» (`Y`) обозначает «неделю в году», а нижний регистр «y» (`y`) обозначает год.

Преобразование строки в дату

`parse()` из класса `SimpleDateFormat` помогает преобразовать шаблон `String` в объект `Date` .

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // input String
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

Существует 4 разных стиля для текстового формата: `SHORT` , `MEDIUM` (это значение по умолчанию), `LONG` и `FULL` , все из которых зависят от языка. Если локаль не указана, используется языковая система по умолчанию.

Стиль	Locale.US	Locale.France
КОРОТКАЯ	6/30/09	30/06/09
СРЕДНЯЯ	30 июня 2009 г.	30 июня 2009 г.
ДОЛГО	30 июня 2009 г.	30 июня 2009 г.
ПОЛНЫЙ	Вторник, 30 июня 2009 г.	mardi 30 июня 2009

Основной вывод даты

Используя следующий код со строкой формата `yyyy/MM/dd hh:mm.ss` , мы получим следующий вывод:

2016/04/19 11: 45,36

```
// define the format to use
String formatString = "yyyy/MM/dd hh:mm.ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);
```

```
// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);


// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm.ss").format(Calendar.getInstance().getTime()));
```

Преобразование форматированного строкового представления объекта date to Date

Этот метод можно использовать для преобразования форматированного строкового представления даты в объект `Date` .

```
    /**
     * Parses the date using the given format.
     *
     * @param formattedDate the formatted date string
     * @param dateFormat the date format which was used to create the string.
     * @return the date
     */
    public static Date parseDate(String formattedDate, String dateFormat) {
        Date date = null;
        SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
        try {
          date = objDf.parse(formattedDate);
        } catch (ParseException e) {
          // Do what ever needs to be done with exception.
        }
        return date;
    }
```

Создание конкретной даты

Хотя класс Java Date имеет несколько конструкторов, вы заметите, что большинство из них устарело. Единственным приемлемым способом создания экземпляра Date напрямую является либо использование пустого конструктора, либо передача длинных (количество миллисекунд с момента стандартного базового времени). Ни один из них не является удобным, если вы не ищете текущую дату или не имеете другого экземпляра даты уже в руке.

Чтобы создать новую дату, вам понадобится экземпляр календаря. Оттуда вы можете установить экземпляр календаря на нужную вам дату.

```
Calendar c = Calendar.getInstance();
```

Это возвращает новый экземпляр календаря, установленный в текущее время. В календаре есть много способов для того, чтобы изменить дату и время или установить его прямо. В

этом случае мы установим его на определенную дату.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

Метод `getTime` возвращает экземпляр Date, который нам нужен. Имейте в виду, что методы набора календаря устанавливают только одно или несколько полей, они не устанавливают их всех. То есть, если вы установите год, остальные поля остаются неизменными.

западня

Во многих случаях этот фрагмент кода выполняет свою задачу, но имейте в виду, что две важные части даты / времени не определены.

- параметры `(1974, 6, 2, 8, 0, 0)` интерпретируются в часовом поясе по умолчанию, определяемом где-то еще,
- миллисекунды не устанавливаются на ноль, а заполняются из системных часов во время создания экземпляра календаря.

Объекты Java 8 LocalDate и LocalDateTime

Объекты Date и LocalDate **не могут** быть *точно* преобразованы между собой, поскольку объект Date представляет собой определенный день и время, в то время как объект LocalDate не содержит информацию о времени или часовом поясе. Тем не менее, может быть полезно преобразовать их между двумя, если вы заботитесь только о фактической информации о дате, а не о временной информации.

Создает LocalDate

```
// Create a default date
LocalDate lDate = LocalDate.now();

// Creates a date from values
lDate = LocalDate.of(2017, 12, 15);

// create a date from string
lDate = LocalDate.parse("2017-12-15");

// creates a date from zone
LocalDate.now(ZoneId.systemDefault());
```

Создает LocalDateTime

```
// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
```

```
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());
```

LocalDate to Date и наоборот

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDate
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();

// LocalDate to Date
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTime to Date и наоборот

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDateTime
LocalDateTime localDateTime =  date.toInstant().atZone(defaultZoneId).toLocalDateTime();

// LocalDateTime to Date
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

Часовой пояс и java.util.Date

Объект `java.util.Date` *не* имеет понятия часового пояса.

- Невозможно **установить** часовой пояс для даты
- Невозможно **изменить** часовой пояс объекта Date
- Объект Date, созданный с помощью `new Date()` умолчанию, будет инициализирован текущим временем в системном часовом поясе

Тем не менее, можно отобразить дату, представленную точкой во времени, описанную объектом Date в другом часовом поясе, например, `java.text.SimpleDateFormat` :

```
Date date = new Date();
//print default time zone
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //note: time zone not in
format!
//print date in the original time zone
System.out.println(sdf.format(date));
//current time in London
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));
```

Выход:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Преобразовать java.util.Date в java.sql.Date

`java.util.Date` для преобразования `java.sql.Date` обычно необходимо, когда объект Date необходимо записать в базу данных.

`java.sql.Date` - это оболочка с миллисекундным значением и используется `JDBC` для идентификации типа `SQL DATE`

В приведенном ниже примере мы используем конструктор `java.util.Date()` , который создает объект Date и инициализирует его для представления времени до ближайшей миллисекунды. Эта дата используется в методе `convert(java.util.Date utilDate)` для возврата объекта `java.sql.Date`

пример

```
public class UtilToSqlConversion {

    public static void main(String args[])
    {
        java.util.Date utilDate = new java.util.Date();
        System.out.println("java.util.Date is : " + utilDate);
        java.sql.Date sqlDate = convert(utilDate);
        System.out.println("java.sql.Date is : " + sqlDate);
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");
        System.out.println("dateFormated date is : " + df.format(utilDate));
    }

    private static java.sql.Date convert(java.util.Date uDate) {
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());
        return sDate;
    }

}
```

Выход

```
java.util.Date is : Fri Jul 22 14:40:35 IST 2016
java.sql.Date is : 2016-07-22
dateFormated date is : 22/07/2016 - 02:40:35
```

`java.util.Date` имеет информацию о дате и времени, тогда как `java.sql.Date` имеет только дату

Местное время

Чтобы использовать только временную часть Даты, используйте LocalTime. Вы можете создать экземпляр объекта LocalTime несколькими способами

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` также имеет встроенный метод toString, который отображает формат очень хорошо.

```
System.out.println(time);
```

вы также можете получать, добавлять и вычитать часы, минуты, секунды и наносекунды из объекта LocalTime, т. е.

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

Вы можете превратить его в объект Date со следующим кодом:

```
LocalTime lTime = LocalTime.now();
Instant instant = lTime.atDate(LocalDate.of(A_YEAR, A_MONTH, A_DAY)).
        atZone(ZoneId.systemDefault()).toInstant();
Date time = Date.from(instant);
```

этот класс отлично работает в классе таймера для имитации будильника.

Прочитайте Класс даты онлайн: https://riptutorial.com/ru/java/topic/164/класс-даты

глава 91: Класс свойств

Вступление

Объект properties содержит пару ключей и значений как строку. Класс java.util.Properties является подклассом Hashtable.

Его можно использовать для получения значения свойства на основе ключа свойства. Класс Properties предоставляет методы для получения данных из файла свойств и хранения данных в файле свойств. Более того, его можно использовать для получения свойств системы.

Преимущество файла свойств

Перекомпиляция не требуется, если информация изменяется из файла свойств: если какая-либо информация изменена с

Синтаксис

- В файле свойств:
- ключ = значение
- #комментарий

замечания

Объект Properties - это карта , ключи и значения которой являются строками по соглашению. Хотя методы карты могут быть использованы для доступа к данным, тем больше типизированные методы GetProperty , SetProperty и stringPropertyNames обычно используются вместо этого.

Свойства часто хранятся в файлах свойств Java, которые представляют собой простые текстовые файлы. Их формат подробно описан в методе Properties.load . В итоге:

- Каждая пара ключ / значение представляет собой строку текста с пробелами, равно (`=`) или двоеточие (`:`) между ключом и значением. Уравнение или двоеточие может иметь любое количество пробелов до и после него, которое игнорируется.
- Ведущие пробелы всегда игнорируются, всегда включается конечный пробел.
- Обратная косая черта может использоваться для удаления любого символа (кроме нижнего регистра `u`).
- Обратная косая черта в конце строки указывает, что следующая строка является продолжением текущей строки. Однако, как и во всех строках, ведущие пробелы в строке продолжения игнорируются.

- Как и в исходном коде Java, `\u` за которым следуют четыре шестнадцатеричных цифры, представляет символ UTF-16.

Большинство фреймворков, включая собственные средства Java SE, такие как java.util.ResourceBundle, загружают файлы свойств как InputStreams. При загрузке файла свойств из InputStream этот файл может содержать только символы ISO 8859-1 (то есть символы в диапазоне 0-255). Любые другие символы должны быть представлены как `\u` экранирование. Тем не менее, вы можете написать текстовый файл в любой кодировке и использовать инструмент native2ascii (который поставляется с каждым JDK), чтобы сделать это для вас.

Если вы загружаете файл свойств собственным кодом, он может быть в любой кодировке, если вы создаете Reader (например, InputStreamReader) на основе соответствующей кодировки . Затем вы можете загрузить файл с использованием load (Reader) вместо метода устаревшей загрузки (InputStream).

Вы также можете хранить свойства в простом XML-файле, который позволяет самому файлу определять кодировку. Такой файл может быть загружен с помощью метода loadFromXML . DTD, описывающий структуру таких файлов XML, находится по адресу http://java.sun.com/dtd/properties.dtd .

Examples

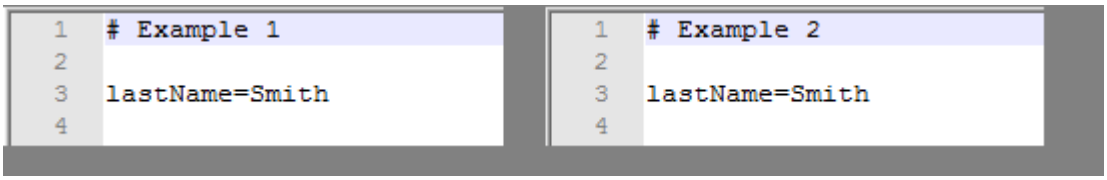
Загрузка свойств

Чтобы загрузить файл свойств в комплекте с вашим приложением:

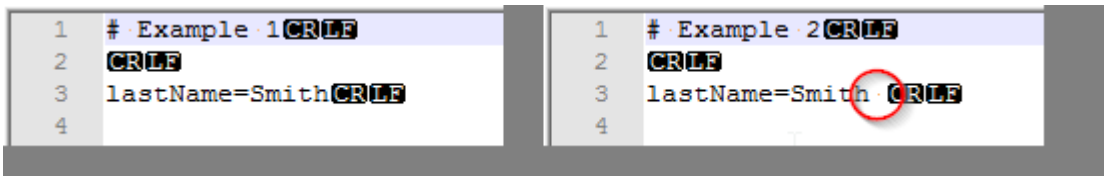
```
public class Defaults {

    public static Properties loadDefaults() {
        try (InputStream bundledResource =
            Defaults.class.getResourceAsStream("defaults.properties")) {

            Properties defaults = new Properties();
            defaults.load(bundledResource);
            return defaults;
        } catch (IOException e) {
            // Since the resource is bundled with the application,
            // we should never get here.
            throw new UncheckedIOException(
                "defaults.properties not properly packaged"
                + " with application", e);
        }
    }

}
```

Предоставление файла свойств: завершение пробела

Внимательно посмотрите на эти два файла свойств, которые кажутся полностью идентичными:



за исключением того, что они действительно не идентичны:



(скриншоты из Notepad ++)

Так как конечный пробел сохраняется, значение `lastName` будет `"Smith"` в первом случае, а `"Smith "` во втором случае.

Очень редко это то, что пользователи ожидают, и одно, и может только предположить, почему это поведение класса `Properties` по умолчанию. Однако легко создать расширенную версию `Properties` которая устраняет эту проблему. Следующий класс, **TrimmedProperties** , делает именно это. Это замена для стандартного класса Properties.

```
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * Properties class where values are trimmed for trailing whitespace if the
 * properties are loaded from a file.
 *
 * <p>
 * In the standard {@link java.util.Properties Properties} class trailing
 * whitespace is always preserved. When loading properties from a file such
 * trailing whitespace is almost always <i>unintentional</i>. This class fixes
 * this problem. The trimming of trailing whitespace only takes place if the
 * source of input is a file and only where the input is line oriented (meaning
 * that for example loading from XML file is <i>not</i> changed by this class).
 * For this reason this class is almost in all cases a safe drop-in replacement
 * for the standard <tt>Properties</tt>
 * class.
 *
 * <p>
 * Whitespace is defined here as any of space (U+0020) or tab (U+0009).
 * *
 */
public class TrimmedProperties extends Properties {
```

```
    /**
     * Reads a property list (key and element pairs) from the input byte stream.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.InputStream) }
     * with the exception that trailing whitespace is trimmed from property values
     * if <tt>inStream</tt> is an instance of <tt>FileInputStream</tt>.
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream the input stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(inStream);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(inStream);
        }
    }

    /**
     * Reads a property list (key and element pairs) from the input character stream in a
simple line-oriented format.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.Reader)}
     * with the exception that trailing whitespace is trimmed on property values
     * if <tt>reader</tt> is an instance of <tt>FileReader</tt>.
     *
     * @see java.util.Properties#load(java.io.Reader) }
     * @param reader the input character stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(Reader reader) throws IOException {
        if (reader instanceof FileReader) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(reader);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(reader);
        }
    }

    private void trimAndLoad(Properties p) {
        for (Entry<Object, Object> entry : p.entrySet()) {
            if (entry.getValue() instanceof String) {
                put(entry.getKey(), trimTrailing((String) entry.getValue()));
            } else {
                put(entry.getKey(), entry.getValue());
            }
        }
    }

    /**
     * Trims trailing space or tabs from a string.
```

```
     *
     * @param str
     * @return
     */
    public static String trimTrailing(String str) {
        if (str != null) {
            // read str from tail until char is no longer whitespace
            for (int i = str.length() - 1; i >= 0; i--) {
                if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                    return str.substring(0, i + 1);
                }
            }
        }
        return str;
    }
}
```

Сохранение свойств как XML

Сохранение свойств в файле XML

Способ хранения файлов свойств в виде файлов XML очень похож на способ хранения их как файлов `.properties` . Просто вместо использования `store()` вы будете использовать `storeToXML()` .

```
public void saveProperties(String location) throws IOException{
    // make new instance of properties
    Properties prop = new Properties();

    // set the property values
    prop.setProperty("name", "Steve");
    prop.setProperty("color", "green");
    prop.setProperty("age", "23");

    // check to see if the file already exists
    File file = new File(location);
    if (!file.exists()){
        file.createNewFile();
    }

    // save the properties
    prop.storeToXML(new FileOutputStream(file), "testing properties with xml");
}
```

Когда вы откроете файл, он будет выглядеть следующим образом.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>testing properties with xml</comment>
<entry key="age">23</entry>
<entry key="color">green</entry>
<entry key="name">Steve</entry>
</properties>
```

Загрузка свойств из файла XML

Теперь для загрузки этого файла в качестве `properties` вам нужно вызвать `loadFromXML()` вместо `load()` который вы будете использовать с обычными файлами `.propeties` .

```
public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // make new properties instance to load the file into
    Properties prop = new Properties();

    // check to make sure the file exists
    File file = new File(location);
    if (file.exists()){
        // load the file
        prop.loadFromXML(new FileInputStream(file));

        // print out all the properties
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("Error: No file found at: " + location);
    }
}
```

Когда вы запустите этот код, вы получите следующее в консоли:

```
age=23
color=green
name=Steve
```

Прочитайте Класс свойств онлайн: https://riptutorial.com/ru/java/topic/576/класс-свойств

глава 92: Классы и объекты

Вступление

Объекты имеют состояния и поведение. Пример: у собаки есть состояния - цвет, имя, порода, а также поведение - виляние хвоста, лай, еда. Объект является экземпляром класса.

Класс. Класс может быть определен как шаблон / план, который описывает поведение / состояние, которое поддерживает объект его типа.

Синтаксис

- class Example {} // ключевое слово, имя, тело

Examples

Самый простой возможный класс

```
class TrivialClass {}
```

Класс состоит как минимум из ключевого слова `class` , имени и тела, которое может быть пустым.

Вы создаете класс с помощью `new` оператора.

```
TrivialClass tc = new TrivialClass();
```

Участник объекта против статического члена

С этим классом:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter ++;
        memberCounter++;
    }
}
```

следующий фрагмент кода:

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();

o1.increment();

o2.increment();
o2.increment();

System.out.println("o1 static counter " + o1.staticCounter);
System.out.println("o1 member counter " + o1.memberCounter);
System.out.println();

System.out.println("o2 static counter " + o2.staticCounter);
System.out.println("o2 member counter " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// the following line does not compile. You need an object
// to access its members
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.memberCounter);
```

производит этот вывод:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Примечание. Вы не должны вызывать `static` члены на объектах, а на классах. Хотя это не имеет значения для JVM, читатели-люди оценят это.

`static` члены являются частью класса и существуют только один раз для каждого класса. ОТСУТСТВИЯ `static` члены существуют в случаях, существует независимая копия для каждого экземпляра. Это также означает, что вам нужен доступ к объекту этого класса для доступа к его членам.

Методы перегрузки

Иногда такие же функции должны быть написаны для разных типов входов. В то время можно использовать одно и то же имя метода с другим набором параметров. Каждый другой набор параметров известен как сигнатура метода. Как видно на примере, один метод может иметь несколько подписей.

```
public class Displayer {

    public void displayName(String firstName) {
        System.out.println("Name is: " + firstName);
```

```
    }

    public void displayName(String firstName, String lastName) {
        System.out.println("Name is: " + firstName + " " + lastName);
    }

    public static void main(String[] args) {
        Displayer displayer = new Displayer();
        displayer.displayName("Ram");          //prints "Name is: Ram"
        displayer.displayName("Jon", "Skeet"); //prints "Name is: Jon Skeet"
    }
}
```

Преимущество состоит в том, что одна и та же функциональность вызывается с двумя разными количествами входов. При вызове метода в соответствии с входом мы передаем (в этом случае либо одно строковое значение, либо два строковых значения) выполняется соответствующий метод.

Методы могут быть перегружены:

1. На основании **количества** переданных **параметров** .

   Пример: `method(String s)` и `method(String s1, String s2)` .

2. На основе **порядка параметров** .

   Пример: `method(int i, float f)` и `method(float f, int i))` .

Примечание. *Методы не могут быть перегружены путем изменения только типа возвращаемого значения (* `int method()` *считается таким же, как* `String method()` *и при попытке будет* `RuntimeException` *). Если вы измените тип возврата, вы также должны изменить параметры для перегрузки.*

Строительство и использование базовых объектов

Объекты входят в свой класс, поэтому простым примером может служить автомобиль (подробные пояснения ниже):

```
public class Car {

    //Variables describing the characteristics of an individual car, varies per  object
   private int milesPerGallon;
   private String name;
   private String color;
   public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

```
    //this is where an individual object is created
    public Car(int mpg, int, gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //methods to make the object more usable

    //Cars need to drive
    public void drive(int distanceInMiles){
        //get miles left in car
        int miles = numGallonsInTank * milesPerGallon;

        //check that car has enough gas to drive distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("Drove " + numGallonsInTank + " miles!");
        } else {
            System.out.println("Could not drive!");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }
        //set new Miles Per Gallon
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

       //set new number of Gallon In Tank
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }

    //Get the Car color
    public String getColor(){
        return color;
    }

    //Get the Car name
    public String getName(){
        return name;
    }

    //Get the number of Gallons
    public String getGallons(){
        return numGallonsInTank;
    }

}
```

Объектами являются **экземпляры** их класса. Таким образом, способ **создания объекта** был бы путем вызова класса Car **одним из двух способов** в вашем основном классе

(основной метод в Java или onCreate в Android).

Опция 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

Вариант 1 - это то, где вы по существу рассказываете программе все о Автомобиле при создании объекта. Для изменения любого свойства автомобиля потребуется вызвать один из методов, например метод `repaintCar` . Пример:

```
 newCar.repaintCar("Blue");
```

Примечание. Убедитесь, что вы передали правильный тип данных методу. В приведенном выше примере вы также можете передать переменную методу `repaintCar` **если тип данных правильный** .

Это был пример изменения свойств объекта, для получения свойств объекта потребовался бы метод из класса Car, у которого есть возвращаемое значение (что означает `void` метод). Пример:

```
String myCarName = newCar.getName();  //returns string "Ferrari"
```

Вариант 1 - **лучший** вариант, когда у вас есть **все данные объекта** во время создания.

Вариант 2

```
`Car newCar = new Car();
```

Вариант 2 получает тот же эффект, но требует больше работы для правильного создания объекта. Я хочу напомнить об этом Конструкторе в классе Car:

```
public void Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Обратите внимание, что вам не нужно передавать какие-либо параметры в объект для его создания. Это очень полезно, когда у вас нет всех аспектов объекта, но вам нужно использовать те части, которые у вас есть. Это устанавливает общие данные в каждую из переменных экземпляра объекта, так что, если вы вызываете фрагмент данных, который не существует, ошибки не генерируются.

Примечание. Не забывайте, что вам нужно установить части объекта позже, после чего вы не инициализировали его. Например,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

Это распространенная ошибка среди объектов, которые не инициализируются всеми их данными. Ошибок избегали, потому что есть Конструктор, который позволяет создать пустой объект Car с **переменными stand-in** (`public Car(){}`), но никакая часть myCar не была настроена. **Правильный пример создания объекта автомобиля:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

И, как напоминание, получить свойства объекта, вызвав метод в вашем основном классе. Пример:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Конструкторы

Конструкторы - это специальные методы, названные в честь класса и без возвращаемого типа, и используются для построения объектов. Конструкторы, подобно методам, могут принимать входные параметры. Конструкторы используются для инициализации объектов. Абстрактные классы также могут иметь конструкторы.

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
// instantiates the object during creating and prints out the content
// of wordToPrint
```

Важно понимать, что конструкторы отличаются от методов несколькими способами:

1. Конструкторы могут принимать только модификаторы `public` , `private` и `protected` и не могут быть объявлены `abstract` , `final` , `static` или `synchronized` .

2. Конструкторы не имеют типа возврата.

3. Конструкторы ДОЛЖНЫ называться так же, как имя класса. В `Hello` например, `Hello` имя конструктора объекта является таким же , как имя класса.

4. `this` ключевое слово имеет дополнительное использование внутри конструкторах. `this.method(...)`

вызывает метод в текущем экземпляре, тогда как `this(...)` ссылается на другой конструктор текущего класса с разными сигнатурами.

Конструкторы также можно вызывать через наследование, используя ключевое слово `super` .

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}


public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

См. Спецификацию Java Language # 8.8 и # 15.9

Инициализация статических конечных полей с использованием статического инициализатора

Чтобы инициализировать `static final` поля, которые требуют использования более одного выражения, для назначения значения может использоваться `static` инициализатор. В следующем примере инициализируется немодифицируемый набор `String` s:

```
public class MyClass {

    public static final Set<String> WORDS;

    static {
        Set<String> set = new HashSet<>();
        set.add("Hello");
        set.add("World");
        set.add("foo");
        set.add("bar");
        set.add("42");
        WORDS = Collections.unmodifiableSet(set);
    }
}
```

Объяснение, что такое перегрузка и переопределение метода.

Переопределение и перегрузка метода - это две формы полиморфизма, поддерживаемые Java.

Перегрузка метода

Перегрузка метода (также известный как статический полиморфизм) - это способ, которым вы можете иметь два (или более) метода (функции) с одним и тем же именем в одном классе. Да, это так просто.

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;

    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

     //To calculate area of a circle
     public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

Таким образом, пользователь может вызывать тот же метод для области в зависимости от типа формы, которую он имеет.

Но реальный вопрос заключается в следующем: как будет компилятор java отличить, какое тело метода должно быть выполнено?

Ну, Java ясно дал понять, что хотя **имена методов** (`area()` в нашем случае) **могут быть одинаковыми, но метод аргументов должен быть другим.**

> Перегруженные методы должны иметь список разных аргументов (количество и типы).

При этом мы не можем добавить еще один метод для вычисления площади квадрата: `public Double area(Long side)` потому что в этом случае он будет конфликтовать с методом области окружности и вызовет **двусмысленность** для java-компилятора.

Слава богу, есть некоторые релаксации при написании перегруженных методов, таких как

> Может иметь разные типы возврата.
>
> Могут быть разные модификаторы доступа.
>
> Может бросать разные исключения.

Почему это называется статическим полиморфизмом?

Ну, это потому, что перегруженные методы должны быть вызваны, определяется во время компиляции, основываясь на фактическом количестве аргументов и типах аргументов времени компиляции.

Одной из распространенных причин использования перегрузки метода является простота кода, который он предоставляет. Например, помните `String.valueOf()` который принимает почти любой тип аргумента? То, что написано за сценой, возможно, примерно так:

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

Переопределение метода

Ну, переопределение метода (да, вы догадались, что это правильно, он также известен как динамический полиморфизм) является несколько более интересной и сложной темой.

При переопределении метода мы перезаписываем тело метода, предоставляемое родительским классом. Понял? Нет? Давайте рассмотрим пример.

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }
}
```

Таким образом, у нас есть класс под названием Shape, и у него есть метод, называемый областью, которая, вероятно, вернет область формы.

Скажем, теперь у нас есть два класса, называемые Circle и Rectangle.

```
public class Circle extends Shape {
    private Double radius = 5.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}
```

Аналогично, класс прямоугольника:

```
 public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;
```

```
    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return length * breadth;
    }
}
```

Итак, теперь оба класса ваших детей обновили тело метода, предоставленное родительским (`Shape`) классом. Теперь вопрос заключается в том, как увидеть результат? Хорошо, давайте сделаем это старым способом `psvm` .

```
public class AreaFinder{

    public static void main(String[] args){

        //This will create an object of circle class
        Shape circle = new Circle();
        //This will create an object of Rectangle class
        Shape rectangle = new Rectangle();

        // Drumbeats ......
        //This should print 78.5
        System.out.println("Shape of circle : "+circle.area());

        //This should print 50.0
        System.out.println("Shape of rectangle: "+rectangle.area());

    }
}
```

Вот Это Да! разве это не здорово? Два объекта одного типа вызывают одни и те же методы и возвращают разные значения. Мой друг, это сила динамического полиморфизма.

Вот график, чтобы лучше сравнить различия между этими двумя:

Перегрузка метода	Переопределение метода
Перегрузка метода используется для повышения удобочитаемости программы.	Переопределение метода используется для обеспечения конкретной реализации метода, который уже предоставлен его суперклассом.
Перегрузка метода выполняется внутри класса.	Переопределение метода происходит в двух классах, имеющих отношение IS-A (наследование).
В случае перегрузки метода параметр должен быть другим.	В случае переопределения метода параметр должен быть таким же.

Перегрузка метода	Переопределение метода
Перегрузка метода является примером полиморфизма времени компиляции.	Переопределение метода - пример полиморфизма времени выполнения.
В java перегрузка метода не может быть выполнена путем изменения типа возвращаемого метода. Тип возврата может быть таким же или другим при перегрузке метода. Но вам придется изменить параметр.	Тип возврата должен быть таким же или ковариантным в переопределении метода.

Прочитайте Классы и объекты онлайн: https://riptutorial.com/ru/java/topic/114/классы-и-объекты

глава 93: Клонирование объектов

замечания

Клонирование может быть сложным, особенно когда в полях объекта хранятся другие объекты. Бывают ситуации, когда вы хотите выполнить глубокую копию вместо того, чтобы копировать только значения поля (т.е. ссылки на другие объекты).

Нижняя строка - это клон , и вам следует подумать дважды, прежде чем внедрять интерфейс `Cloneable` и переопределять метод `clone` . Метод `clone` объявлен в классе `Object` а не в интерфейсе `Cloneable` , поэтому `Cloneable` не может функционировать как интерфейс, потому что ему не хватает общедоступного метода `clone` . В результате контракт на использование `clone` тонко документирован и слабо соблюдается. Например, класс, который переопределяет `clone` иногда полагается на все его родительские классы, также перекрывающие `clone` . Они не применяются для этого, и если они не делают ваш код, вы можете исключать исключения.

Лучшим решением для обеспечения функциональности клонирования является создание *конструктора копирования* или *фабрики копирования* . См . Эффективный Java Joshua Bloch. Пункт 11: разумно переверните клон.

Examples

Клонирование с использованием конструктора копирования

Простым способом клонирования объекта является реализация конструктора копирования.

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // copy constructor
    // copies the fields of other into the new object
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// create a sheep
```

```
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = new Sheep(sheep); // dolly.name is "Dolly" and dolly.weight is 20
```

Клонирование путем реализации интерфейса Clonable

Клонирование объекта путем реализации интерфейса Cloneable .

```
public class Sheep implements Cloneable {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly =  (Sheep) sheep.clone(); // dolly.name is "Dolly" and dolly.weight is 20
```

Клонирование выполнения мелкой копии

Поведение по умолчанию при клонировании объекта заключается в выполнении мелкой копии полей объекта. В этом случае как исходный объект, так и клонированный объект содержат ссылки на одни и те же объекты.

Этот пример показывает это поведение.

```
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
```

```
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints true, both arrays contain the same objects
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Клонирование, выполняющее глубокую копию

Для копирования вложенных объектов необходимо выполнить глубокую копию , как показано в этом примере.

```
import java.util.ArrayList;
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
```

```
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // make a deep copy of the children
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
            clone.setChildren(cloneChildren);
        }
        return clone;
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints false, both arrays contain copies of the objects inside
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Клонирование с использованием фабрики копирования

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other);
        return new Sheep(other.name, other.weight)
    }
```

```
}
```

Прочитайте Клонирование объектов онлайн: https://riptutorial.com/ru/java/topic/2830/клонирование-объектов

глава 94: Кодировка символов

Examples

Чтение текста из файла, закодированного в UTF-8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Strange cyrillic symbol Ы");
        }
        /* First Way. For big files */
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.print(line);
            }
        }

        System.out.println(); //just separating output

        /* Second way. For small files */
        String s = new String(Files.readAllBytes(Paths.get("test.txt")),
StandardCharsets.UTF_8);
        System.out.print(s);
    }
}
```

Запись текста в файл в UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
```

```
StandardCharsets.UTF_8)) {
            wr.write("Cyrillic symbol Ы");
        }
    }
}
```

Получение байтового представления строки в UTF-8

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class GetUtf8BytesFromString {

    public static void main(String[] args) {
        String str = "Cyrillic symbol Ы";
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);

        System.out.println(Arrays.toString(textInUtf8));
    }
}
```

Прочитайте Кодировка символов онлайн: https://riptutorial.com/ru/java/topic/2735/кодировка-символов

глава 95: Коллекции

Вступление

Структура коллекций в `java.util` предоставляет ряд общих классов для наборов данных с функциональностью, которые не могут быть предоставлены регулярными массивами.

Структура коллекций содержит интерфейсы для `Collection<O>` , с основными суб-интерфейсами `List<O>` и `Set<O>` , а также картографическая коллекция `Map<K,V>` . Коллекции являются корневым интерфейсом и реализуются многими другими структурами коллекции.

замечания

Коллекции - это объекты, которые могут хранить коллекции других объектов внутри них. Вы можете указать тип данных, хранящихся в коллекции, с помощью Generics .

В коллекциях обычно используются пространства имен `java.util` или `java.util.concurrent` .

Java SE 1.4

Java 1.4.2 и ниже не поддерживают дженерики. Таким образом, вы не можете указать параметры типа, которые содержит коллекция. Помимо того, что у вас нет безопасности типов, вы также должны использовать отбрасывания, чтобы получить правильный тип из коллекции.

В дополнение к `Collection<E>` существует несколько основных типов коллекций, некоторые из которых имеют подтипы.

- `List<E>` - упорядоченный набор объектов. Он похож на массив, но не определяет ограничение по размеру. Реализации будут обычно расти в размерах внутри, чтобы разместить новые элементы.
- `Set<E>` - это набор объектов, которые не позволяют дублировать.
    - `SortedSet<E>` - это `Set<E>` который также задает порядок элементов.
- `Map<K,V>` представляет собой набор пар ключ / значение.
    - `SortedMap<K,V>` - это `Map<K,V>` которая также определяет упорядочение элементов.

Java SE 5

Java 5 добавляет в новый тип коллекции:

- `Queue<E>` представляет собой набор элементов, предназначенных для обработки в определенном порядке. Реализация определяет, является ли это FIFO или LIFO. Это устаревает класс `Stack` .

Java SE 6

Java 6 добавляет некоторые новые подтипы коллекций.

- `NavigableSet<E>` - это `Set<E>` со встроенными специальными навигационными методами.
- `NavigableMap<K,V>` - это `Map<K,V>` со встроенными специальными навигационными методами.
- `Deque<E>` - это `Queue<E>` которая может быть прочитана с любого конца.

Обратите внимание, что вышеупомянутые элементы - все интерфейсы. Чтобы использовать их, вы должны найти соответствующие классы реализации, такие как `ArrayList` , `HashSet` , `HashMap` или `PriorityQueue` .

Каждый тип коллекции имеет несколько реализаций, которые имеют разные показатели производительности и варианты использования.

Обратите внимание, что принцип замены Лискова применяется к подтипам коллекции. То есть, `SortedSet<E>` может быть передан функции, ожидающей `Set<E>` . Также полезно прочитать раздел « Ограниченные параметры в разделе« Общие »для получения дополнительной информации о том, как использовать коллекции с наследованием класса.

Если вы хотите создать свои собственные коллекции, может быть проще наследовать один из абстрактных классов (например, `AbstractList`) вместо реализации интерфейса.

Java SE 1.2

До 1.2 вам пришлось использовать следующие классы / интерфейсы:

- `Vector` вместо `ArrayList`
- `Dictionary` вместо `Map` . Обратите внимание, что словарь также является абстрактным классом, а не интерфейсом.
- `Hashtable` вместо `HashMap`

Эти классы устарели и не должны использоваться в современном коде.

Examples

Объявление ArrayList и добавление объектов

Мы можем создать `ArrayList` (после интерфейса `List`):

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Теперь используйте метод `add` для добавления `String` :

```
aListOfFruits.add("Melon");
aListOfFruits.add("Strawberry");
```

В приведенном выше примере `ArrayList` будет содержать `String` «Дыня» с индексом 0 и `String` «Клубника» по индексу 1.

Также мы можем добавить несколько элементов с помощью `addAll(Collection<? extends E> c)`

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Теперь «Лук» помещается в индекс 0 в `aListOfFruitsAndVeggies` , «Дыня» находится в индексе 1, а «Клубника» - с индексом 2.

Построение коллекций из существующих данных

Стандартные коллекции

Структура коллекций Java

Простым способом построения `List` из отдельных значений данных является использование метода `java.utils.Arrays` `Arrays.asList` :

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

Все стандартные реализации коллекции предоставляют конструкторы, которые берут другую коллекцию в качестве аргумента, добавляя все элементы в новую коллекцию во время построения:

```
List<String> list = new ArrayList<>(data); // will add data as is
Set<String> set1 = new HashSet<>(data); // will add data keeping only unique values
SortedSet<String> set2 = new TreeSet<>(data); // will add data keeping unique values and
sorting
Set<String> set3 = new LinkedHashSet<>(data); // will add data keeping only unique values and
preserving the original order
```

Система коллекций Google Guava

Еще одна отличная основа - это `Google Guava` который является удивительным классом

полезности (предоставляющим удобные статические методы) для построения различных типов стандартных коллекций. `Lists` и `Sets` :

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

Составление карт

Структура коллекций Java

Аналогично для карт, учитывая `Map<String, Object> map` новая карта может быть построена со всеми элементами следующим образом:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Общий раздел

Используя `Apache Commons` вы можете создать Map using array в `ArrayUtils.toMap` а также `MapUtils.toMap` :

```
import org.apache.commons.lang3.ArrayUtils;
...
// Taken from org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// Create a Map mapping colors.
Map colorMap = MapUtils.toMap(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}});
```

Каждый элемент массива должен быть либо Map.Entry, либо Array, содержащий как минимум два элемента, где первый элемент используется как ключ, а второй - как значение.

Система коллекций Google Guava

Класс полезности из системы `Google Guava` называется `Maps` :

```
import com.google.common.collect.Maps;
...
```

```
void howToCreateMapsMethod(Function<? super K,V> valueFunction,
          Iterable<K> keys1,
          Set<K> keys2,
          SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // Immutable copy
    Map<K, V> map2 = asMap(keys2, valueFunction); // Live Map view
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // Live Map view
}
```

Java SE 8

Используя `Stream` ,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

или же

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Регистрация списков

Следующие способы могут использоваться для объединения списков без изменения списка источников.

Первый подход. Имеет больше строк, но легко понять

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Второй подход. Имеет одну меньшую строку, но менее читаемую.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Третий подход. Требуется сторонняя библиотека коллекций коллекций Apache .

```
ListUtils.union(listOne,listTwo);
```

Java SE 8

Использование потоков также может быть достигнуто путем

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

Рекомендации. Список интерфейсов

Удаление элементов из списка в цикле

Излишне удалять элементы из списка в цикле, это связано с тем, что индекс и длина списка изменены.

Учитывая следующий список, вот несколько примеров, которые приведут к неожиданному результату, а некоторые - к правильному результату.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

НЕПРАВИЛЬНО

Удаление в итерации `for` утверждения *Пропускает «Банан»:*

Образец кода будет печатать только `Apple` и `Strawberry` . `Banana` пропускается, потому что он перемещается в индекс `0` когда `Apple` удаляется, но в то же время `i` увеличиваю до `1` .

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Удаление в расширении `for` оператора *Throws Exception:*

Из-за итерации над сборкой и ее модификацией в одно и то же время.

Броски: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

ПРАВИЛЬНЫЙ

Удаление во время цикла с использованием `Iterator`

```
Iterator<String> fruitIterator = fruits.iterator();
```

```
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruitIterator.remove();
    }
}
```

Интерфейс `Iterator` имеет метод `remove()` построенный только для этого случая. Однако этот метод помечен как «необязательный» в документации, и он может вызывать `UnsupportedOperationException` .

> Выбрасывает: UnsupportedOperationException - если операция удаления не поддерживается этим итератором

Поэтому рекомендуется проверить документацию, чтобы убедиться, что эта операция поддерживается (на практике, если коллекция не является неизменной, полученной через стороннюю библиотеку или использование одного из методов `Collections.unmodifiable...()` операция почти всегда поддерживается).

При использовании `Iterator` возникает `modCount ConcurrentModificationException` когда `modCount` из `List` изменяется с момента создания `Iterator` . Это могло произойти в том же потоке или в многопоточном приложении, использующем один и тот же список.

`modCount` - это переменная `int` которая подсчитывает количество структурных изменений этого списка. Структурное изменение по существу означает операцию `add()` или `remove()` , вызываемую в объекте `Collection` (изменения, сделанные `Iterator` , не учитываются). Когда `Iterator` создан, он сохраняет этот `modCount` и на каждой итерации `List` проверяет, `modCount` ли текущий `modCount` тем, как и когда был создан `Iterator` . Если `modCount` значение `modCount` оно выдает `modCount ConcurrentModificationException` .

Следовательно, для вышеопределенного списка операция, подобная приведенной ниже, не будет вызывать никаких исключений:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

Но добавление нового элемента в `List` после инициализации `Iterator` вызовет исключение `ConcurrentModificationException` :

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Итерация назад

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Это ничего не пропускает. Недостатком этого подхода является то, что выход обратный. Однако в большинстве случаев вы удаляете элементы, которые не имеют значения. Вы никогда не должны делать это с помощью `LinkedList` .

Итерация вперед, корректировка индекса цикла

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
         i--;
    }
}
```

Это ничего не пропускает. Когда `i` й элемент удаляется из `List` , элемент, первоначально расположенный в индексе `i+1` становится новым `i` м элементом. Поэтому цикл может уменьшать `i` чтобы следующая итерация обрабатывала следующий элемент без пропусков.

Использование списка «должно быть удалено»

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);
```

Это решение позволяет разработчику проверить, удалены ли правильные элементы более чистым способом.

Java SE 8

В Java 8 возможны следующие варианты. Они более чистые и более прямые, если удаление не должно происходить в цикле.

Фильтрация потока

`List` можно передавать и фильтровать. Для удаления всех нежелательных элементов можно использовать правильный фильтр.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Обратите внимание, что в отличие от всех других примеров здесь, этот пример создает новый экземпляр `List` и сохраняет исходный `List` без изменений.

Использование `removeIf`

Сохраняет накладные расходы на создание потока, если требуется только удалить набор элементов.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Немодифицированная коллекция

Иногда это не очень хорошая практика, выставляя внутреннюю коллекцию, поскольку это может привести к уязвимости вредоносного кода из-за изменчивой характеристики. Для обеспечения коллекций «только для чтения» java предоставляет свои неизменяемые версии.

Неизменяемая коллекция часто является копией модифицируемой коллекции, которая гарантирует, что сама коллекция не может быть изменена. Попытки изменить его приведет к исключению UnsupportedOperationException.

Важно заметить, что объекты, которые присутствуют внутри коллекции, могут быть изменены.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value){
        intList.add(value);
    }

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}
```

Следующая попытка изменить немодифицируемую коллекцию приведет к исключению:

```
import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}
```

выход:

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)
```

Итерирование над коллекциями

Итерирование по списку

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Если нам нужно использовать параллелизм

```
names.parallelStream().forEach(System.out::println);
```

Java SE 5

```
for (String name : names) {
    System.out.println(name);
}
```

Java SE 5

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

Java SE 1.2

```
//Creates ListIterator which supports both forward as well as backward traversel
ListIterator<String> listIterator = names.listIterator();

//Iterates list in forward direction
```

```
while(listIterator.hasNext()){
    System.out.println(listIterator.next());
}

//Iterates list in backward direction once reaches the last element from above iterator in
forward direction
while(listIterator.hasPrevious()){
    System.out.println(listIterator.previous());
}
```

Итерирование по множеству

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Java SE 5

```
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {
    System.out.println(iterator.next());
}

for (String name : names) {
    System.out.println(name);
}
```

Java SE 5

```
Iterator iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Итерация по карте

```
Map<Integer, String> names = new HashMap<>();
names.put(1, "Clementine");
names.put(2, "Duran");
names.put(3, "Mike");
```

Java SE 8

```
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));
```

Java SE 5

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {
    System.out.println(entry.getKey());
```

```
    System.out.println(entry.getValue());
}

// Iterating over only keys
for (Integer key : names.keySet()) {
    System.out.println(key);
}
// Iterating over only values
for (String value : names.values()) {
    System.out.println(value);
}
```

Java SE 5

```
Iterator entries = names.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry entry = (Map.Entry) entries.next();
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}
```

Неизменяемые пустые коллекции

Иногда целесообразно использовать неизменяемую пустую коллекцию. Класс `Collections` предоставляет методы для эффективного сбора таких коллекций:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

Эти методы являются общими и автоматически преобразуют возвращенную коллекцию в тип, которому она назначена. То есть, вызов, например, `emptyList()` может быть назначен любому типу `List` а также для `emptySet()` и `emptyMap()` .

Коллекции, возвращаемые этими методами, неизменны тем, что они будут вызывать `UnsupportedOperationException` если вы попытаетесь вызвать методы, которые изменили бы их содержимое (`add` , `put` и т. Д.). Эти коллекции в первую очередь полезны в качестве замены для результатов пустых методов или других значений по умолчанию, вместо того, чтобы использовать `null` или создавать объекты с `new` .

Коллекции и примитивные ценности

Коллекции в Java работают только для объектов. Т.е. в Java нет `Map<int, int>` . Вместо этого примитивные значения должны быть *помещены* в объекты, как в `Map<Integer, Integer>` . Java-бокс позволит прозрачному использованию этих коллекций:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

К сожалению, накладные расходы на это *существенны* . Для `HashMap<Integer, Integer>` потребуется около 72 байт на запись (например, на 64-битной JVM со сжатыми указателями и при условии, что целые числа больше 256 и предполагают 50% нагрузки на карту). Поскольку фактические данные составляют всего 8 байтов, это приводит к большим накладным расходам. Кроме того, для этого требуется два уровня косвенности (Map -> Entry -> Value), это излишне медленно.

Существует несколько библиотек с оптимизированными коллекциями для примитивных типов данных (для которых требуется только ~ 16 байт на запись при нагрузке 50%, т.е. на 4 раза меньше памяти и на один уровень косвенности меньше), что может принести значительные преимущества при использовании больших коллекций примитивных значения в Java.

Удаление совпадающих элементов из списков с помощью Iterator.

Выше я заметил пример удаления элементов из списка в цикле, и я подумал о другом примере, который может пригодиться на этот раз, используя интерфейс `Iterator` .
Это демонстрация трюка, который может пригодиться при работе с дублирующимися элементами в списках, от которых вы хотите избавиться.

Примечание. Это добавляется только к **удалению элементов из списка в** примере **цикла** :

Итак, давайте определим наши списки как обычно

```
    String[] names = {"James","Smith","Sonny","Huckle","Berry","Finn","Allan"};
    List<String> nameList = new ArrayList<>();

    //Create a List from an Array
    nameList.addAll(Arrays.asList(names));

    String[] removeNames = {"Sonny","Huckle","Berry"};
    List<String> removeNameList = new ArrayList<>();

    //Create a List from an Array
    removeNameList.addAll(Arrays.asList(removeNames));
```

Следующий метод принимает два объекта Collection и выполняет магию удаления элементов в нашем `removeNameList` которые соответствуют элементам в `nameList` .

```
private static void removeNames(Collection<String> collection1, Collection<String>
collection2) {
      //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

Вызов метода и передача в `nameList` и `removeNameList` следующим образом `removeNames(nameList,removeNameList);`
Произведет следующий вывод:

> Список массивов перед удалением имен: **James Smith Sonny Huckle Berry Finn Allan**
> Список массивов после удаления имен: **James Smith Finn Allan**

Простое использование для коллекций, которое может пригодиться для удаления повторяющихся элементов внутри списков.

Создание собственной структуры Iterable для использования с циклами Iterator или for-each.

Чтобы гарантировать, что наша коллекция может быть итерации с использованием итератора или для каждого цикла, мы должны позаботиться о следующих шагах:

1. Материал, который мы хотим повторить, должен быть `Iterable` и выставить `iterator()` .
2. `hasNext() java.util.Iterator` , переопределив `hasNext()` , `next()` и `remove()` .

Я добавил простой общий список связанных списков ниже, который использует выше, чтобы сделать связанный список итерабельным.

```
package org.algorithms.linkedlist;

import java.util.Iterator;
import java.util.NoSuchElementException;


public class LinkedList<T> implements Iterable<T> {

    Node<T> head, current;

    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
        }
    }

    public LinkedList(T data) {
        head = new Node<>(data);
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {

        Node<T> node = head;
```

```
        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Node<T> prevNode = node;
            node = node.next;
            return prevNode.data;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Removal logic not implemented.");
        }
    }

    public void add(T data) {
        Node current = head;
        while (current.next != null)
            current = current.next;
        current.next = new Node<>(data);
    }

}

class App {
    public static void main(String[] args) {

        LinkedList<Integer> list = new LinkedList<>(1);
        list.add(2);
        list.add(4);
        list.add(3);

        //Test #1
        System.out.println("using Iterator:");
        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer i = itr.next();
            System.out.print(i + " ");
        }

        //Test #2
        System.out.println("\n\nusing for-each:");
        for (Integer data : list) {
            System.out.print(data + " ");
        }
    }
}
```

Выход

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

Это будет работать в Java 7+. Вы можете запустить его на Java 5 и Java 6, заменив:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

с

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

или любую другую версию путем включения совместимых изменений.

Pitfall: исключения одновременной модификации

Это исключение возникает, когда коллекция модифицируется во время итерации по ней с использованием методов, отличных от тех, которые предоставляются объектом итератора. Например, у нас есть список шляп, и мы хотим удалить все те, которые имеют ушные заслонки:

```
List<IHat> hats = new ArrayList<>();
hats.add(new Ushanka()); // that one has ear flaps
hats.add(new Fedora());
hats.add(new Sombrero());
for (IHat hat : hats) {
    if (hat.hasEarFlaps()) {
        hats.remove(hat);
    }
}
```

Если мы запустим этот код, ***ConcurrentModificationException*** будет поднят, так как код изменяет коллекцию при ее итерации. Такое же исключение может возникнуть, если один из нескольких потоков, работающих с одним и тем же списком, пытается изменить коллекцию, а другие перебирают ее. Одновременная модификация коллекций в нескольких потоках является естественной вещью, но ее следует обрабатывать с помощью обычных инструментов из параллельного инструментария программирования, таких как блокировки синхронизации, специальные коллекции, принятые для одновременной модификации, изменение клонированной коллекции с начального и т. Д.

Подразделы

Список subList (int fromIndex, int toIndex)

Здесь fromIndex включен, а toIndex является эксклюзивным.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. Если список не существует в диапазоне выдачи, он генерирует

IndexOutofBoundException.
2. Все изменения, внесенные в список1, будут влиять на те же изменения в списке. Это называется резервными коллекциями.
3. Если fromnIndex больше, чем toIndex (fromIndex> toIndex), он выдает исключение IllegalArgumentException.

Пример:

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

Выход:
Перед Sublist [Hello1, Hello2]
После изменений подписок [Hello1, Hello3, Hello2]

Установите subSet (fromIndex, toIndex)

Здесь fromIndex включен, а toIndex является эксклюзивным.

```
Set set = new TreeSet();
Set set1 = set.subSet(fromIndex,toIndex);
```

Возвращенный набор будет вызывать исключение IllegalArgumentException при попытке вставить элемент за пределы его диапазона.

MapMapMap (fromKey, toKey)

fromKey является инклюзивным, а toKey является эксклюзивным

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

Если fromKey больше, чем toKey, или если сама эта карта имеет ограниченный диапазон, а fromKey или toKey лежит за пределами диапазона, тогда он выдает исключение IllegalArgumentException.

Все коллекции, поддерживаемые подкрепленными коллекциями, означают, что изменения, внесенные в подсечку, будут иметь одинаковые изменения в основной коллекции.

Прочитайте Коллекции онлайн: https://riptutorial.com/ru/java/topic/90/коллекции

глава 96: Команда Java - «java» и «javaw»

Синтаксис

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

замечания

Команда `java` используется для запуска приложения Java из командной строки. Он доступен как часть Java SE JRE или JDK.

В системах Windows есть два варианта команды `java` :

- Вариант `java` запускает приложение в новом окне консоли.
- Вариант `javaw` запускает приложение без создания нового окна консоли.

В других системах (например, Linux, Mac OSX, UNIX) предоставляется только команда `java` , и она не запускает новое консольное окно.

Символ `<opt>` в синтаксисе обозначает параметр в командной строке `java` . Темы «Параметры Java» и «Параметры выбора кучи и размера стека» охватывают наиболее часто используемые параметры. Другие описаны в теме JVM Flags .

Examples

Запуск исполняемого файла JAR

Исполняемые файлы JAR - это самый простой способ собрать Java-код в один файл, который можно выполнить. * (Редакционное примечание: создание JAR-файлов должно охватываться отдельной темой.) *

Предполагая, что у вас есть исполняемый JAR-файл с именем пути `<jar-path>` , вы должны иметь возможность запускать его следующим образом:

```
java -jar <jar-path>
```

Если команде требуются аргументы командной строки, добавьте их после `<jar-path>` . Например:

```
java -jar <jar-path> arg1 arg2 arg3
```

Если вам нужно предоставить дополнительные параметры JVM в командной строке `java` , они должны пройти *до* опции `-jar` . Обратите внимание , что `-cp` / `-classpath` опция будет игнорироваться , если вы используете `-jar` . Путь к классам приложения определяется манифестом JAR-файла.

Запуск приложений Java через «основной» класс

Когда приложение не было упаковано в качестве исполняемого JAR, вам необходимо указать имя класса точки входа в командной строке `java` .

Запуск класса HelloWorld

Пример «HelloWorld» описан в разделе «Создание новой программы Java» . Он состоит из одного класса `HelloWorld` который удовлетворяет требованиям для точки входа.

Предполагая, что (скомпилированный) файл HelloWorld.class находится в текущем каталоге, его можно запустить следующим образом:

```
java HelloWorld
```

Следует отметить следующие важные моменты:

- Мы должны указать имя класса: не путь к файлу «.class» или «.java».
- Если класс объявлен в пакете (как и большинство классов Java), то имя класса, которое мы поставляем команде `java` должно быть полным именем класса. Например, если `SomeClass` объявлен в пакете `com.example` , то полное имя класса будет `com.example.SomeClass` .

Указание пути к классам

Если мы не используем синтаксис команды `java -jar` , команда `java` ищет класс для загрузки путем поиска в пути к классам; см. « Класс» . Вышеприведенная команда опирается на путь по умолчанию, который является (или включает) текущий каталог. Мы можем быть более откровенными в этом вопросе, указав путь к классам, который будет использоваться с помощью опции `-cp` .

```
java -cp . HelloWorld
```

Это говорит о том, чтобы сделать текущий каталог (который является тем, «означает») единственную запись в пути к классам.

`-cp` - это опция, которая обрабатывается командой `java` . Все параметры, предназначенные для команды `java` должны быть перед именем класса. Все, что после класса будет рассматриваться как аргумент командной строки для приложения Java, и будет передано в

приложение в `String[]` , которое передается `main` методу.

(Если опция no `-cp` предоставлена, `java` будет использовать путь к классам, который задается переменной среды `CLASSPATH` . Если эта переменная не установлена или пуста, `java` использует «.» Как путь по умолчанию по умолчанию.)

Классы точек входа

Класс точки входа Java имеет `main` метод со следующими сигнатурами и модификаторами:

```
public static void main(String[] args)
```

> Sidenote: из-за того, как работают массивы, он также может быть `(String args[])`

Когда `java` команда запускает виртуальную машину, она загружает указанные классы начальной точки и пытается найти `main` . В случае успеха аргументы из командной строки преобразуются в объекты Java `String` и собираются в массив. Если `main` вызывается так, массив *не* будет `null` и не будет содержать `null` элементы.

Действительный метод класса входной точки должен выполнять следующие действия:

- Называть `main` (с учетом регистра)
- Быть `public` и `static`
- `void` тип возврата `void`
- Имейте один аргумент с массивом `String[]` . Аргумент должен присутствовать и допускается не более одного аргумента.
- Будьте универсальными: параметры типа не допускаются.
- Имейте не общий, верхний уровень (не вложенный или внутренний) класс

Традиционно объявлять класс `public` но это не является строго необходимым. Начиная с Java 5, тип аргумента `main` метода может быть переменным `String` вместо строкового массива. `main` может опционально генерировать исключения, а его параметр можно назвать чем угодно, но обычно это `args` .

Входные точки JavaFX

Начиная с Java 8, команда `java` также может непосредственно запускать приложение JavaFX. JavaFX документируется в теге JavaFX , но точка входа JavaFX должна выполнять следующие действия:

- Расширить `javafx.application.Application`
- Будьте `public` а не `abstract`
- Не быть родовым или вложенным
- Имейте явный или неявный `public` no-args

Устранение неполадок команды «java»

В этом примере рассматриваются общие ошибки с использованием команды «java».

"Команда не найдена"

Если вы получите сообщение об ошибке, например:

```
java: command not found
```

при попытке запустить `java` команду это означает, что на пути поиска команд вашей оболочки нет `java` команды. Причиной может быть:

- у вас нет Java JRE или JDK вообще,
- вы не обновили `PATH` среды `PATH` (правильно) в файле инициализации оболочки или
- вы не «получили» соответствующий файл инициализации в текущей оболочке.

Обратитесь к разделу «Установка Java» для шагов, которые необходимо предпринять.

«Не удалось найти или загрузить основной класс»

Это сообщение об ошибке выводится командой `java` если не удалось найти / загрузить класс точки входа, который вы указали. В общих чертах, есть три широкие причины, которые могут произойти:

- Вы указали класс точки входа, который не существует.
- Класс существует, но вы указали его неправильно.
- Класс существует, и вы указали его правильно, но Java не может найти его, потому что путь к классам неверен.

Вот процедура диагностики и решения проблемы:

1. Узнайте полное имя класса точки входа.

    - Если у вас есть исходный код для класса, то полное имя состоит из имени пакета и простого имени класса. Экземпляр класса «Основной» объявляется в пакете «com.example.myapp», тогда его полное имя «com.example.myapp.Main».
    - Если у вас есть скомпилированный файл класса, вы можете найти имя класса, запустив `javap` .
    - Если файл класса находится в каталоге, вы можете вывести полное имя класса из имен каталогов.
    - Если файл класса находится в JAR или ZIP-файле, вы можете сделать полное имя класса из пути к файлу в JAR или ZIP-файле.

2. Посмотрите сообщение об ошибке из команды `java` . Сообщение должно заканчиваться полным именем класса, которое пытается использовать `java` .

    - Убедитесь, что он точно соответствует полному классу для класса начальной точки.
    - Он не должен заканчиваться словами «.java» или «.class».
    - Он не должен содержать косой черты или любой другой символ, который не является законным в Java-идентификаторе [1] .
    - Корпус имени должен точно соответствовать полному названию класса.

3. Если вы используете правильное имя класса, убедитесь, что класс действительно находится в пути к классам:

    - Разработайте путь, к которому сопоставляется имя класса; см. Сопоставление имен классов с именами путей
    - Выясните, что такое classpath; см. этот пример: Различные способы указания пути к классам
    - Посмотрите на каждый из JAR и ZIP-файлов в classpath, чтобы узнать, содержат ли они класс с требуемым именем пути.
    - Посмотрите на каждую директорию, чтобы узнать, разрешает ли путь к файлу в каталоге.

Если проверка пути к классам вручную не найдена, вы можете добавить параметры `-Xdiag` и `-XshowSettings` . В первом перечислены все загружаемые классы, а в последнем - параметры, которые включают эффективный путь к классу JVM.

Наконец, есть некоторые *неясные* причины для этой проблемы:

- Исполняемый JAR-файл с атрибутом `Main-Class` который указывает класс, который не существует.
- Исполняемый JAR-файл с неправильным атрибутом `Class-Path` .
- Если вы испортили [2] опций перед именем класса, команда `java` может попытаться интерпретировать один из них как имя класса.
- Если кто-то проигнорировал правила стиля Java и использовал идентификаторы пакетов или классов, которые отличаются только буквенным случаем, и вы работаете на платформе, которая обрабатывает регистр букв в именах файлов как несущественные.
- Проблемы с гомоглифами в именах классов в коде или в командной строке.

«Основной метод не найден в классе <имя>"

Эта проблема возникает, когда команда `java` может находить и загружать назначенный вами класс, но затем не может найти метод точки входа.

Существует три возможных объяснения:

- Если вы пытаетесь запустить исполняемый JAR-файл, в манифесте JAR есть неправильный атрибут «Main-Class», который указывает класс, который не является допустимым классом точки входа.
- Вы сказали команде `java` класс, который не является классом точки входа.
- Класс точки входа неверен; см. классы точек входа для получения дополнительной информации.

Другие источники

- Что означает «Не удалось найти или загрузить основной класс»?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

---

1 - Начиная с Java 8 и более поздней версии, команда `java` поможет отобразить разделитель имен файлов («/» или «») на период («.»). Однако это поведение не описано на страницах руководства.

2 - Действительно непонятным случаем является то, что вы копируете и вставляете команду из отформатированного документа, где текстовый редактор использовал «длинный дефис» вместо обычного дефиса.

Запуск приложения Java с зависимостями библиотеки

Это продолжение примеров «основного класса» и «исполняемого JAR» .

Типичные Java-приложения состоят из кода, специфичного для приложения, и другого многократного кода библиотеки, который вы реализовали или который был реализован третьими лицами. Последние обычно называются библиотечными зависимостями и обычно упаковываются в виде файлов JAR.

Java - динамически связанный язык. Когда вы запускаете приложение Java с зависимостями библиотеки, JVM должен знать, где находятся зависимости, чтобы он мог загружать классы по мере необходимости. В широком смысле, есть два способа справиться с этим:

- Приложение и его зависимости можно переупаковать в один JAR-файл, содержащий все необходимые классы и ресурсы.
- JVM может быть рассказано, где найти зависимые файлы JAR через путь к среде выполнения.

Для исполняемого JAR-файла путь класса выполнения задается атрибутом манифеста класса-класса. *(Редакционное примечание: это должно быть описано в отдельной теме в команде `jar` .)* В противном случае путь к классам выполнения должен быть предоставлен с использованием параметра `-cp` или с помощью переменной среды `CLASSPATH` .

Например, предположим, что у нас есть приложение Java в файле «myApp.jar», чей класс

точки входа `com.example.MyApp` . Предположим также, что приложение зависит от библиотечных JAR-файлов «lib / library1.jar» и «lib / library2.jar». Мы могли бы запустить приложение, используя команду `java` как показано в командной строке:

```
$ # Alternative 1 (preferred)
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp

$ # Alternative 2
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar
$ java com.example.MyApp
```

(В Windows, вы должны использовать `;` вместо того , чтобы `:` как разделитель пути к классам, и вы должны установить (локальный) `CLASSPATH` переменную используя `set` , а не `export` .)

В то время как разработчику Java было бы удобно с этим, оно не «удобно». Поэтому обычной практикой является написать простой сценарий оболочки (или пакетный файл Windows), чтобы скрыть детали, о которых пользователю не нужно знать. Например, если вы поместили следующий сценарий оболочки в файл под названием «myApp», сделали его исполняемым и поместили его в каталог по пути поиска команд:

```
#!/bin/bash
# The 'myApp' wrapper script

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

то вы можете запустить его следующим образом:

```
$ myApp arg1 arg2 ...
```

Любые аргументы в командной строке будут переданы Java-приложению через расширение `"$@"` . (Вы можете сделать что-то подобное с пакетным файлом Windows, хотя синтаксис отличается.)

Пробелы и другие специальные символы в аргументах

Прежде всего, проблема обработки пространств в аргументах на самом деле НЕ является проблемой Java. Скорее, это проблема, которая должна выполняться командной оболочкой, которую вы используете при запуске Java-программы.

В качестве примера предположим, что мы имеем следующую простую программу, которая печатает размер файла:

```
import java.io.File;

public class PrintFileSizes {
```

```
    public static void main(String[] args) {
        for (String name: args) {
            File file = new File(name);
            System.out.println("Size of '" + file + "' is " + file.size());
        }
    }
}
```

Предположим теперь, что мы хотим напечатать размер файла, у которого в нем есть пробелы; например `/home/steve/Test File.txt` . Если мы запустим команду следующим образом:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

оболочка не будет знать, что `/home/steve/Test File.txt` на самом деле является одним из путей. Вместо этого он будет передавать два различных аргумента Java-приложению, которые будут пытаться найти их соответствующие размеры файлов и сбой, потому что файлы с этими путями (вероятно) не существуют.

Решения, использующие оболочку POSIX

Оболочки POSIX включают в себя `sh` а также производные, такие как `bash` и `ksh` . Если вы используете одну из этих оболочек, вы можете решить проблему, *указав* аргумент.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

Двойная кавычка вокруг имени пути сообщает оболочке, что она должна быть передана как один аргумент. Кавычки будут удалены, когда это произойдет. Есть несколько других способов сделать это:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Одиночные (прямые) кавычки рассматриваются как двойные кавычки, за исключением того, что они также подавляют различные расширения в аргументе.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

Обратная косая черта пропускает следующее пространство и не позволяет интерпретировать ее как разделитель аргументов.

Для более полной документации, включая описание того, как обращаться с другими специальными символами в аргументах, обратитесь к теме цитирования в документации Bash .

Решение для Windows

Основная проблема для Windows заключается в том, что на уровне ОС аргументы передаются дочернему процессу как одна строка (источник). Это означает, что конечная ответственность за синтаксический анализ (или повторный анализ) командной строки лежит на любой программе или ее библиотеках времени исполнения. Существует много несогласованности.

В случае Java, чтобы сократить длинную историю:

- Вы можете поместить двойные кавычки вокруг аргумента в команду `java` , и это позволит вам передавать аргументы с пробелами в них.
- По-видимому, сама команда `java` анализирует командную строку, и она становится более или менее правильной
- Однако, когда вы пытаетесь объединить это с использованием замены `SET` и переменной в пакетном файле, становится очень сложно определить, удаляются ли двойные кавычки.
- Оболочка `cmd.exe` видимому, имеет другие механизмы экранирования; например, удвоение двойных кавычек и использование `^` экранов.

Подробнее см. В документации к пакетному файлу .

Параметры Java

Команда `java` поддерживает широкий диапазон опций:

- Все параметры начинаются с одного символа дефиса или минус-знака (`-`): соглашение об использовании GNU / Linux для использования `--` для «длинных» вариантов не поддерживается.
- Параметры должны появляться перед признанием `<classname>` или `-jar <jarfile>` . Любые аргументы после них будут рассматриваться как аргументы, которые будут переданы в приложение Java, которое выполняется.
- Параметры, которые не начинаются с `-X` или `-XX` являются стандартными. Вы можете полагаться на все реализации Java [1] для поддержки любой стандартной опции.
- Параметры, начинающиеся с `-X` являются нестандартными параметрами и могут быть удалены из одной версии Java в другую.
- Опции, начинающиеся с `-XX` являются расширенными опциями и могут также быть сняты.

Настройка свойств системы с помощью `-D`

Параметр `-D<property>=<value>` используется для установки свойства в объекте `Properties` системы. Этот параметр можно повторить, чтобы установить разные свойства.

Варианты памяти, стека и мусора

Основные параметры управления размерами кучи и стека описаны в разделе «Размеры кучи, пермгена и стека» . *(Редакционное примечание: опции сборщика мусора должны быть описаны в той же теме.)*

Включение и отключение утверждений

Параметры `-ea` и `-da` соответственно включают и отключают проверку `assert` Java:

- По умолчанию проверка всех утверждений отключена.
- Параметр `-ea` позволяет проверять все утверждения
- `-ea:<packagename>...` позволяет проверять утверждения в пакете *и всех подпакетах* .
- `-ea:<classname>...` позволяет проверять утверждения в классе.
- Параметр `-da` отключает проверку всех утверждений
- `-da:<packagename>...` отключает проверку утверждений в пакете *и всех подпакетах* .
- `-da:<classname>...` отключает проверку утверждений в классе.
- Параметр `-esa` позволяет проверять все системные классы.
- Параметр `-dsa` отключает проверку всех системных классов.

Параметры можно комбинировать. Например.

```
$ # Enable all assertion checking in non-system classes
$ java -ea -dsa MyApp

$ # Enable assertions for all classes in a package except for one.
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

Обратите внимание, что включение проверки достоверности может повлиять на поведение Java-программирования.

- Это может сделать приложение более медленным в целом.
- Это может привести к тому, что определенные методы потребуют больше времени для запуска, что может изменить время потоков в многопоточном приложении.
- Это может привести к неожиданным *случаям - перед* отношениями, которые могут привести к исчезновению аномалий памяти.
- Неправильно выполненный `assert` может иметь нежелательные побочные эффекты.

Выбор типа виртуальной машины

Параметры `-client` и `-server` позволяют выбирать между двумя различными формами виртуальной машины HotSpot:

- Форма «клиент» настраивается для пользовательских приложений и предлагает более быстрый запуск.
- Форма «сервер» настроена для приложений с длительным сроком службы. В процессе JVM «разогревать» требуется больше времени для сбора статистики, что позволяет компилятору JIT лучше выполнять работу по оптимизации собственного кода.

По умолчанию JVM будет работать в 64-битном режиме, если это возможно, в зависимости от возможностей платформы. Параметры `-d32` и `-d64` позволяют вам выбрать режим явно.

---

1 - Проверьте официальное руководство для команды `java` . Иногда *стандартная* опция описывается как «подлежащая изменению».

Прочитайте Команда Java - «java» и «javaw» онлайн: https://riptutorial.com/ru/java/topic/5791/команда-java----java--и--javaw-

глава 97: Команды выполнения

Examples

Добавление стоп-логов

Иногда вам нужна часть кода для выполнения, когда программа останавливается, например, освобождение открытых вами системных ресурсов. Вы можете запустить поток, когда программа останавливается с `addShutdownHook` метода `addShutdownHook` :

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Прочитайте Команды выполнения онлайн: https://riptutorial.com/ru/java/topic/7304/команды-выполнения

глава 98: Компилятор Java - «javac»

замечания

Команда `javac` используется для компиляции исходных файлов Java в файлы байт-кода. Файлы байт-кода не зависят от платформы. Это означает, что вы можете скомпилировать свой код на одном оборудовании и операционной системе, а затем запустить код на любой другой платформе, поддерживающей Java.

Команда `javac` включена в дистрибутивы Java Development Kit (JDK).

Компилятор Java и остальная часть стандартной инструментальной привязки Java накладывают следующие ограничения на код:

- Исходный код хранится в файлах с суффиксом ".java"
- Байткоды хранятся в файлах с суффиксом «.class»
- Для файлов исходного и байт-кода в файловой системе имена файлов должны соответствовать именам пакетов и классов.

Примечание. Компилятор `javac` не следует путать с компилятором Just in Time (JIT), который компилирует байт-коды в собственный код.

Examples

Команда «javac» - начало работы

Простой пример

Предполагая, что «HelloWorld.java» содержит следующий источник Java:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

(Для объяснения приведенного выше кода обратитесь к началу работы с языком Java .)

Мы можем скомпилировать вышеуказанный файл, используя следующую команду:

```
$ javac HelloWorld.java
```

Это создает файл HelloWorld.class, который мы можем запустить следующим образом:

```
$ java HelloWorld
Hello world!
```

Ключевыми моментами этого примера являются:

1. Исходное имя файла «HelloWorld.java» должно соответствовать имени класса в исходном файле ... который является `HelloWorld` . Если они не совпадают, вы получите ошибку компиляции.
2. Имя файла байта «HelloWorld.class» соответствует имени класса. Если вы хотите переименовать «HelloWorld.class», вы получите сообщение об ошибке при попытке запустить его.
3. При запуске Java-приложения с использованием `java` вы указываете имя класса NOT байтовое имя файла.

Пример с пакетами

Наиболее практичный Java-код использует пакеты для организации пространства имен для классов и снижения риска случайного столкновения имени класса.

Если бы мы хотели объявить класс `HelloWorld` в вызове `com.example` пакета, «HelloWorld.java» будет содержать следующий источник Java:

```
package com.example;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Этот файл исходного кода должен храниться в дереве каталогов, структура которого соответствует названию пакета.

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
```

Мы можем скомпилировать вышеуказанный файл, используя следующую команду:

```
$ javac com/example/HelloWorld.java
```

Это создает файл с именем «com / example / HelloWorld.class»; т.е. после компиляции структура файла должна выглядеть так:

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

Затем мы можем запустить приложение следующим образом:

```
$ java com.example.HelloWorld
Hello world!
```

Дополнительные моменты, которые следует отметить из этого примера:

1. Структура каталогов должна соответствовать структуре имени пакета.
2. Когда вы запускаете класс, должно быть предоставлено полное имя класса; т.е. « com.example.HelloWorld» не «HelloWorld».
3. Вам не нужно компилировать и запускать код Java из текущего каталога. Мы просто делаем это здесь для иллюстрации.

Компиляция сразу нескольких файлов с помощью javac.

Если ваше приложение состоит из нескольких файлов исходного кода (и большинство из них!), Вы можете скомпилировать их по одному. Кроме того, вы можете скомпилировать несколько файлов одновременно, указав пути:

```
$ javac Foo.java Bar.java
```

или используя функциональные возможности подстановочных файлов в командной оболочке.

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

Это скомпилирует все исходные файлы Java в текущем каталоге, в каталоге «com / example» и рекурсивно в дочерних каталогах соответственно. Третьей альтернативой является предоставление списка исходных имен файлов (и параметров компилятора) в виде файла. Например:

```
$ javac @sourcefiles
```

где файл `sourcefiles` файлов содержит:

```
Foo.java
```

```
Bar.java
com/example/HelloWorld.java
```

Примечание: компиляция такого кода подходит для небольших проектов с одним человеком и для одноразовых программ. Кроме того, рекомендуется выбирать и использовать инструмент построения Java. В качестве альтернативы, большинство программистов используют Java IDE (например, NetBeans , eclipse , IntelliJ IDEA), которая предлагает встроенный компилятор и инкрементное построение «проектов».

Обычно используемые опции «javac»

Вот несколько вариантов команды `javac` , которые могут быть вам полезны

- Параметр `-d` устанавливает целевой каталог для записи файлов «.class».
- Параметр `-sourcepath` задает путь поиска исходного кода.
- `-cp` или `-classpath` опция устанавливает путь поиска для нахождения внешних и ранее скомпилированных классов. Для получения дополнительной информации о пути к классам и о том, как его указать, обратитесь к теме Classpath .
- Параметр `-version` выводит информацию о версии компилятора.

Более полный список параметров компилятора будет описан в отдельном примере.

Рекомендации

Определяющей ссылкой для команды `javac` является страница руководства Oracle для `javac` .

Компиляция для другой версии Java

Язык программирования Java (и его время выполнения) претерпел многочисленные изменения со времени его выпуска с момента его первоначального публичного выпуска. Эти изменения включают:

- Изменения в синтаксисе и семантике языка программирования Java
- Изменения в API, предоставляемые стандартными библиотеками классов Java.
- Изменения в наборе команд Java (байт-код) и в формате classfile.

За очень немногими исключениями (например, ключевое слово `enum` , изменения в некоторых «внутренних» классах и т. Д.) Эти изменения обратно совместимы.

- Java-программа, которая была скомпилирована с использованием старой версии инструментальной Java-технологии, будет работать на платформе Java новой версии без перекомпиляции.
- Программа Java, написанная в старой версии Java, будет успешно скомпилирована с

новым компилятором Java.

Компиляция старой Java с более новым компилятором

Если вам нужно (повторно) скомпилировать старый Java-код на более новой платформе Java для запуска на новой платформе, вам вообще не нужно давать какие-либо специальные флаги компиляции. В некоторых случаях (например, если вы использовали `enum` как идентификатор), вы можете использовать параметр `-source` для отключения нового синтаксиса. Например, учитывая следующий класс:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

для компиляции класса с использованием компилятора Java 5 (или более поздней) требуется следующее:

```
$ javac -source 1.4 OldSyntax.java
```

Компиляция для старой платформы выполнения

Если вам нужно скомпилировать Java для работы на старых платформах Java, самый простой подход - установить JDK для самой старой версии, которую вам нужно поддерживать, и использовать этот компилятор JDK в своих сборках.

Вы также можете скомпилировать новый Java-компилятор, но сложны. Прежде всего, есть некоторые важные предпосылки, которые должны быть выполнены:

- Код, который вы компилируете, не должен содержать конструкторы языка Java, которые не были доступны в версии Java, на которую вы нацеливаете.
- Код не должен зависеть от стандартных классов Java, полей, методов и т. Д., Которые не были доступны на старых платформах.
- Сторонние библиотеки, которые зависят от кода, также должны быть созданы для старой платформы и доступны во время компиляции и времени выполнения.

Учитывая, что выполнены предварительные условия, вы можете перекомпилировать код для более старой платформы, используя параметр `-target` . Например,

```
$ javac -target 1.4 SomeClass.java
```

скомпилирует вышеуказанный класс для создания байт-кодов, совместимых с Java 1.4 или более поздней версией JVM. (Фактически параметр `-source` подразумевает совместимый `-target` , поэтому `javac -source 1.4 ...` будет иметь тот же эффект. Связь между `-source` и `-target` описана в документации Oracle.)

Сказав это, если вы просто используете `-target` или `-source` , вы все равно будете компилировать против стандартных библиотек классов, предоставляемых JDK компилятора. Если вы не будете осторожны, вы можете получить классы с правильной версией байт-кода, но с зависимостями от API, которые недоступны. Решение состоит в использовании опции `-bootclasspath` . Например:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

будет скомпилирован против альтернативного набора библиотек времени исполнения. Если скомпилированный класс имеет (случайные) зависимости от более новых библиотек, это даст вам ошибки компиляции.

Прочитайте Компилятор Java - «javac» онлайн: https://riptutorial.com/ru/java/topic/4478/компилятор-java----javac-

глава 99: Компилятор Just in Time (JIT)

замечания

история

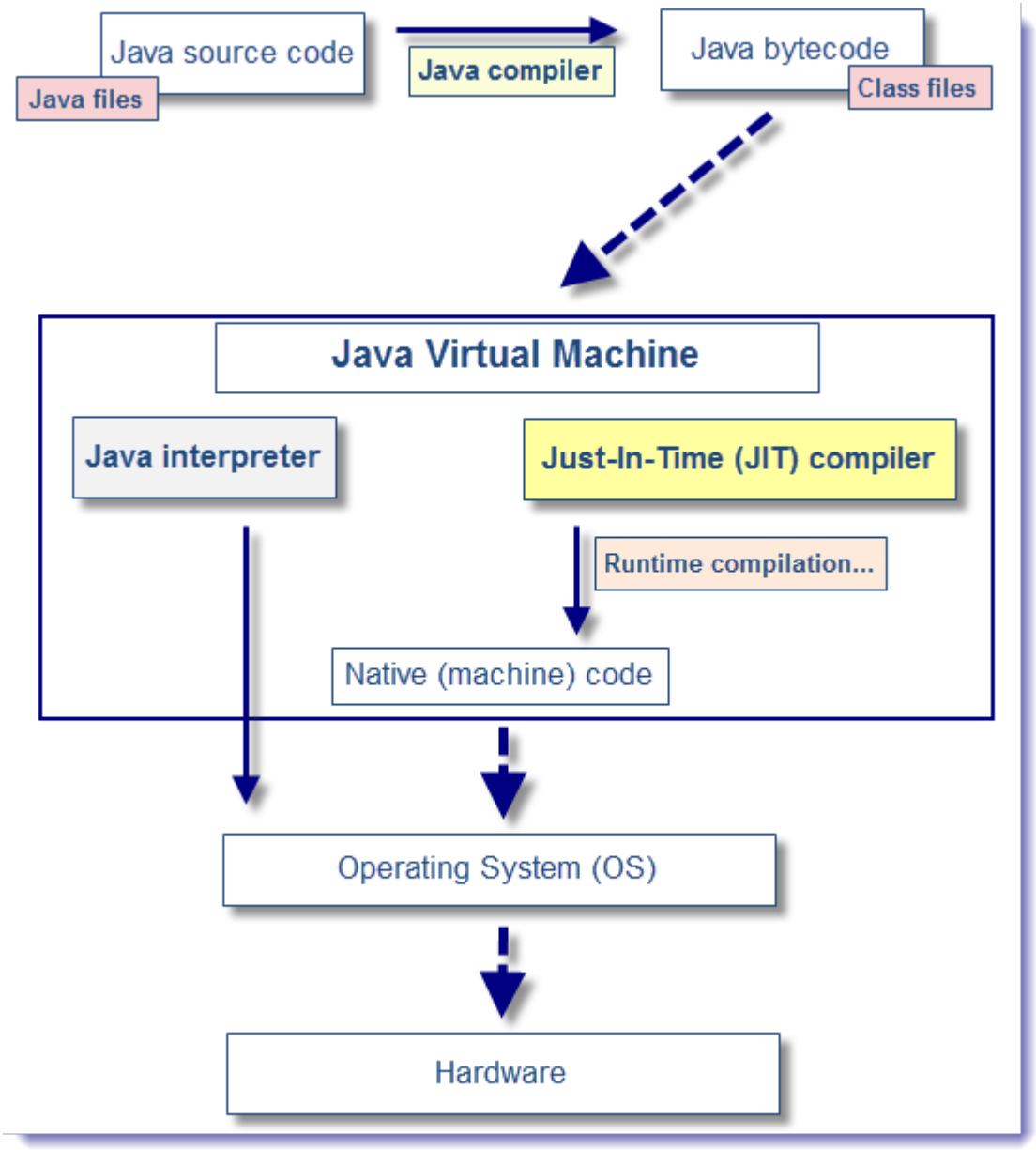
Компилятор Symantec JIT был доступен в Sun Java с версии 1.1.5, но у него были проблемы.

Компилятор Hotspot JIT был добавлен в Sun Java в 1.2.2 в качестве плагина. В Java 1.3 JIT был включен по умолчанию.

(Источник: Когда Java получил JIT-компилятор?

Examples

обзор



Компилятор Just-In-Time (JIT) является компонентом среды выполнения Java ™ Runtime Environment, которая повышает производительность приложений Java во время выполнения.

- Программы Java состоят из классов, которые содержат нейтрали, базирующиеся на платформе, которые могут быть интерпретированы JVM на разных компьютерных архитектурах.
- Во время выполнения JVM загружает файлы классов, определяет семантику каждого отдельного байт-кода и выполняет соответствующее вычисление.

  Дополнительный процессор и использование памяти во время интерпретации означает, что приложение Java работает медленнее, чем собственное приложение.

  Компилятор JIT помогает повысить производительность Java-программ, компилируя байт-коды в собственный машинный код во время выполнения.

Компилятор JIT включен по умолчанию и активируется при вызове метода Java. Компилятор JIT компилирует байт-коды этого метода в собственный машинный код, компилируя его `"just in time"` для запуска.

Когда метод был скомпилирован, JVM вызывает скомпилированный код этого метода непосредственно вместо его интерпретации. Теоретически, если компиляция не требовала процессорного времени и использования памяти, компиляция каждого метода могла бы позволить скорости Java-программы приближаться к скорости работы собственного приложения.

Для компиляции JIT требуется время процессора и использование памяти. Когда JVM запускается первым, вызывается тысячи методов. Компиляция всех этих методов может существенно повлиять на время запуска, даже если программа в конечном итоге достигает очень хорошей пиковой производительности.

- На практике методы не компилируются при первом вызове. Для каждого метода JVM поддерживает `call count` который увеличивается каждый раз при вызове метода.
- JVM интерпретирует метод до тех пор, пока его количество вызовов не превысит порог компиляции JIT.
- Поэтому часто используемые методы скомпилируются вскоре после запуска JVM, и менее используемые методы скомпилируются намного позже или вообще не выполняются.
- Порог компиляции JIT помогает быстрому запуску JVM и по-прежнему имеет улучшенную производительность.
- Порог был тщательно выбран для получения оптимального баланса между временем запуска и долгосрочной эксплуатацией.
- После компиляции метода его счетчик вызовов сбрасывается на ноль, а последующие вызовы метода продолжают увеличивать его количество.
- Когда количество вызовов метода достигает порога перекомпиляции JIT, компилятор JIT компилирует его второй раз, применяя больший выбор оптимизаций, чем в предыдущей компиляции.
- Этот процесс повторяется до тех пор, пока не будет достигнут максимальный уровень оптимизации.

  Самые загруженные методы Java-программы всегда оптимизируются наиболее агрессивно, что максимизирует выгоды от использования JIT-компилятора.

Компилятор JIT также может измерять `operational data at run time` и использовать эти данные для улучшения качества последующей перекомпиляции.

> Компилятор JIT может быть отключен, и в этом случае вся программа Java будет интерпретирована. Отключение JIT-компилятора не рекомендуется, кроме как для диагностики или решения проблем компиляции JIT.

Прочитайте Компилятор Just in Time (JIT) онлайн: https://riptutorial.com/ru/java/topic/5152/компилятор-just-in-time--jit-

глава 100: Консольный ввод-вывод

Examples

Чтение пользовательского ввода с консоли

Использование `BufferedReader` :

```
System.out.println("Please type your name and press Enter.");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("Hello, " + name + "!");
} catch(IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

Для этого кода необходимы следующие импорты:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Использование `Scanner` :

Java SE 5

```
System.out.println("Please type your name and press Enter");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");
```

Для этого примера нужен следующий импорт:

```
import java.util.Scanner;
```

Чтобы прочитать несколько строк, повторно вызовите `scanner.nextLine()` :

```
System.out.println("Please enter your first and your last name, on separate lines.");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();
```

```
System.out.println("Hello, " + firstName + " " + lastName + "!");
```

Существует два метода для получения `Strings` , `next()` и `nextLine()` . `next()` возвращает текст до первого пробела (также известного как «токен»), а `nextLine()` возвращает весь текст, введенный пользователем, до нажатия ввода.

`Scanner` также предоставляет методы утилиты для чтения типов данных, отличных от `String` . Они включают:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

Префикс любого из этих методов `has` (как в `hasNextLine()` , `hasNextInt()`) возвращает значение `true` если поток имеет больше типа запроса. Примечание. Эти методы приведут к сбою программы, если вход не соответствует запрошенному типу (например, набрав «a» для `nextInt()`). Вы можете использовать `try {} catch() {}` чтобы предотвратить это (см. Исключения)

```
Scanner scanner = new Scanner(System.in); //Create the scanner
scanner.useLocale(Locale.US); //Set number format excepted
System.out.println("Please input a float, decimal separator is .");
if (scanner.hasNextFloat()){ //Check if it is a float
    float fValue = scanner.nextFloat(); //retrive the value directly as float
    System.out.println(fValue + " is a float");
}else{
    String sValue = scanner.next(); //We can not retrive as float
    System.out.println(sValue + " is not a float");
}
```

Использование `System.console` :

Java SE 6

```
String name = System.console().readLine("Please type your name and press Enter%n");

System.out.printf("Hello, %s!", name);

//To read passwords (without echoing as in unix terminal)
char[] password = System.console().readPassword();
```

Преимущества :

- Методы чтения синхронизированы
- Можно использовать синтаксис строки формата

Примечание . Это будет работать, только если программа запускается из реальной командной строки без перенаправления стандартных потоков ввода и вывода. Он не работает, когда программа запускается из определенных IDE, например Eclipse. Для кода, который работает в среде IDE и с перенаправлением потока, см. Другие примеры.

Внедрение базового поведения командной строки

Для базовых прототипов или базового поведения в командной строке может понадобиться следующий цикл.

```
public class ExampleCli {

    private static final String CLI_LINE   = "example-cli>"; //console like string

    private static final String CMD_QUIT   = "quit";    //string for exiting the program
    private static final String CMD_HELLO  = "hello";    //string for printing "Hello World!"
on the screen
    private static final String CMD_ANSWER = "answer";    //string for printing 42 on the
screen

    public static void main(String[] args) {
        ExampleCli claimCli = new ExampleCli();    // creates an object of this class

        try {
            claimCli.start();    //calls the start function to do the work like console
        }
        catch (IOException e) {
            e.printStackTrace();    //prints the exception log if it is failed to do get the
user input or something like that
        }
    }

    private void start() throws IOException {
        String cmd = "";

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while (!cmd.equals(CMD_QUIT)) {    // terminates console if user input is "quit"
            System.out.print(CLI_LINE);    //prints the console-like string

            cmd = reader.readLine();    //takes input from user. user input should be started
with "hello",  "answer" or "quit"
            String[] cmdArr = cmd.split(" ");

            if (cmdArr[0].equals(CMD_HELLO)) {    //executes when user input starts with
"hello"
                hello(cmdArr);
            }
            else if (cmdArr[0].equals(CMD_ANSWER)) {    //executes when user input starts with
"answer"
                answer(cmdArr);
            }
        }
    }

    // prints "Hello World!" on the screen if user input starts with "hello"
    private void hello(String[] cmdArr) {
        System.out.println("Hello World!");
```

```
    }

    // prints "42" on the screen if user input starts with "answer"
    private void answer(String[] cmdArr) {
        System.out.println("42");
    }
}
```

Выравнивание строк в консоли

Метод `PrintWriter.format` (вызванный через `System.out.format`) может использоваться для печати выровненных строк в консоли. Метод получает `String` с информацией о формате и рядом объектов для форматирования:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3s";    // min 3 characters, left aligned
String column2Format = "%-5.8s";  // min 5 and max 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Выход:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

Использование форматированных строк с фиксированным размером позволяет печатать строки в виде таблиц с фиксированными размерами:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3.3s";  // fixed size 3 characters, left aligned
String column2Format = "%-8.8s";  // fixed size 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Выход:

```
1   1             1
123 1234       1234
123 1234567  123456
123 12345678 123456
```

Форматирование строк

- `%s` : просто строка без форматирования
- `%5s` : форматировать строку с **минимум** 5 символами; если строка короче, она будет **дополнена** 5 символами и выровнена **вправо**
- `%-5s` : форматировать строку с **минимум** 5 символами; если строка короче, она будет **дополнена** 5 символами и выровнена **влево**
- `%5.10s` : форматирование строки с **минимумом** 5 символов и не **более** 10 символов; если строка короче 5, она будет **дополнена** 5 символами и выровнена **вправо** ; если строка длиннее 10, она будет **усечена** до 10 символов и выровнена **вправо**
- `%-5.5s` : форматировать строку с **фиксированным** размером 5 символов (минимальный и максимальный равны); если строка короче 5, она будет **дополнена** 5 символами и выровнена **влево** ; если строка длиннее 5, она будет **усечена** до 5 символов и выровнена **влево**

Прочитайте Консольный ввод-вывод онлайн: https://riptutorial.com/ru/java/topic/126/консольный-ввод-вывод

глава 101: Конструкторы

Вступление

Хотя это и не требуется, конструкторы в Java - это методы, распознаваемые компилятором для создания конкретных значений для класса, которые могут быть существенными для роли объекта. В этом разделе показано правильное использование конструкторов классов Java.

замечания

Спецификация языка Java подробно обсуждает точный характер семантики конструктора. Их можно найти в JLS §8.8

Examples

Конструктор по умолчанию

«Default» для конструкторов состоит в том, что у них нет аргументов. Если вы не укажете **какой-либо** конструктор, компилятор создаст для вас конструктор по умолчанию.
Это означает, что следующие два фрагмента семантически эквивалентны:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

Видимость конструктора по умолчанию такая же, как видимость класса. Таким образом, для класса, определенного для пакета, в частном порядке есть конструктор по умолчанию для пакета-частного

Однако, если у вас есть конструктор, отличный от стандартного, компилятор не создаст для вас конструктор по умолчанию. Таким образом, они не эквивалентны:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
    }
    public TestClass(String arg) {
    }
}
```

Помните, что сгенерированный конструктор не выполняет нестандартную инициализацию. Это означает, что все поля вашего класса будут иметь значение по умолчанию, если у них нет инициализатора.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

Конструкторы называются так:

```
TestClass testClass = new TestClass();
```

Конструктор с аргументами

Конструкторы могут быть созданы с любыми аргументами.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

Вызывается следующим образом:

```
TestClass testClass = new TestClass("Test Data");
```

Класс может иметь несколько конструкторов с разными сигнатурами. Чтобы вызвать вызовы конструктора (вызовите другой конструктор того же класса при создании экземпляра), используйте `this()` .

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
```

```
        this.testData = testData;
    }

    public TestClass() {
        this("Test"); // testData defaults to "Test"
    }
}
```

Вызывается следующим образом:

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

Вызов родительского конструктора

Скажем, у вас есть класс родителя и класс ребенка. Для создания экземпляра Child всегда требуется, чтобы какой-либо родительский конструктор запускался с самого начала конструктора Child. Мы можем выбрать конструктор родителя, который мы хотим, явным образом вызываем `super(...)` с соответствующими аргументами как наш первый оператор конструктора Child. Это экономит время, повторно используя конструктор родительских классов вместо того, чтобы переписать один и тот же код в конструкторе дочерних классов.

Без `super(...)` метода:

(неявно, версия no-args `super()` называется невидимо)

```
class Parent {
    private String name;
    private int age;

    public Parent() {} // necessary because we call super() without arguments

    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

// This does not even compile, because name and age are private,
// making them invisible even to the child class.
class Child extends Parent {
    public Child() {
        // compiler implicitly calls super() here
        name = "John";
        age = 42;
    }
}
```

С методом `super()` :

```
class Parent {
```

```
    private String name;
    private int age;
    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

class Child extends Parent {
    public Child() {
        super("John", 42);   // explicit super-call
    }
}
```

Примечание. Вызовы к другому конструктору (цепочке) или суперконструктору **ДОЛЖНЫ** быть первым утверждением внутри конструктора.

Если вы явно вызываете конструктор `super(...)` , должен существовать соответствующий родительский конструктор (это просто, не так ли?).

Если вы не вызываете конструктор `super(...)` явно, ваш родительский класс должен иметь конструктор no-args - и это может быть либо явно написано, либо создано по умолчанию компилятором, если родительский класс не предоставляет любой конструктор.

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

Класс Parent не имеет конструктора по умолчанию, поэтому компилятор не может добавить `super` в конструктор Child. Этот код не будет компилироваться. Вы должны изменить конструкторы, чтобы они соответствовали обеим сторонам, или написать собственный `super` , например:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Прочитайте Конструкторы онлайн: https://riptutorial.com/ru/java/topic/682/конструкторы

глава 102: литералы

Вступление

Литерал Java является синтаксическим элементом (т.е. тем, что вы находите в *исходном коде* программы Java), который представляет значение. Примеры: `1` , `0.333F` , `false` , `'X'` и `"Hello world\n"` .

Examples

Шестнадцатеричные, восьмеричные и двоичные литералы

`hexadecimal` число - это значение в базе 16. Есть 16 цифр, `0-9` и буквы `AF` (случай не имеет значения). `AF` представляет `10-16` .

`octal` число - это значение в базе-8 и использует цифры `0-7` .

`binary` число - это значение в базе-2 и использует цифры `0` и `1` .

Все эти числа приводят к тому же значению, `110` :

```
int dec = 110;            // no prefix   --> decimal literal
int bin = 0b1101110;      // '0b' prefix --> binary literal
int oct = 0156;           // '0' prefix  --> octal literal
int hex = 0x6E;           // '0x' prefix --> hexadecimal literal
```

Обратите внимание, что бинарный литерал синтаксис был введен в Java 7.

Октальный литерал может быть легко ловушкой для семантических ошибок. Если вы определяете ведущее `'0'` для ваших десятичных литералов, вы получите неправильное значение:

```
int a = 0100;        // Instead of 100, a == 64
```

Использование подчеркивания для улучшения удобочитаемости

Начиная с Java 7, было возможно использовать один или несколько символов подчеркивания (_) для разделения групп цифр в литерале примитивного числа, чтобы улучшить их читаемость.

Например, эти два объявления эквивалентны:

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

Это можно применить ко всем литералам примитивных чисел, как показано ниже:

Java SE 7

```
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurtyNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals =  3.14_15F;
double piTenDecimals =  3.14_15_92_65_35;
```

Это также работает с использованием префиксов для двоичных, восьмеричных и шестнадцатеричных баз:

Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

Существует несколько правил о подчеркиваниях, которые **запрещают** их размещение в следующих местах:

- В начале или в конце ряда (например , `_123` или `123_` *не* действительны)
- Рядом с десятичной точкой в плавающей точке буквальной (например , `1._23` или `1_.23` *не* действительны)
- Перед суффиксом F или L (например , `1.23_F` или `9999999_L` *не* действительны)
- В местах , где ожидается строка цифр (например , `0_xFFFF` *не* действует)

Эквивалентные последовательности в литералах

Строковые и символьные литералы обеспечивают механизм эвакуации, который позволяет выразить коды символов, которые в противном случае не были бы разрешены в литерале. Эквивалентная последовательность состоит из символа обратной косой черты (`\`), за которым следует одна или несколько других символов. В обоих символах одинаковые последовательности являются строковыми литералами.

Полный набор управляющих последовательностей выглядит следующим образом:

Эквивалентная последовательность	Имея в виду
`\\`	Обозначает символ обратной косой черты (`\`)
`\'`	Обозначает символ одной кавычки (`'`)

Эквивалентная последовательность	Имея в виду
`\"`	Обозначает символ с двойной кавычкой (`"`)
`\n`	Обозначает символ линии (`LF`)
`\r`	Обозначает символ возврата каретки (`CR`)
`\t`	Обозначает символ горизонтальной вкладки (`HT`)
`\f`	Обозначает символ подачи формы (`FF`)
`\b`	Обозначает символ backspace (`BS`)
`\<octal>`	Обозначает код символа в диапазоне от 0 до 255.

`<octal>` в приведенном выше состоит из одной, двух или трех восьмеричных цифр (от 0 до 7), которые представляют число от 0 до 255 (десятичное).

Обратите внимание, что обратная косая черта, сопровождаемая любым другим символом, является недопустимой управляющей последовательностью. Неверные escape-последовательности рассматриваются как ошибки компиляции JLS.

Ссылка:

- JLS 3.10.6. Последовательности выхода для символов и строк

Юникодные экраны

В дополнение к последовательностям escape-последовательности строк и символов, описанным выше, Java имеет более общий механизм экранирования Unicode, как определено в JLS 3.3. Unicode Escapes . Выделение Unicode имеет следующий синтаксис:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

где `<hex-digit>` является одним из `'0'` , `'1'` , `'2'` , `'3'` , `'4'` , `'5'` , `'6'` , `'7'` , `'8'` , `'9'` , `'a'` , `'b'` , `'c'` , `'d'` , `'e'` , `'f'` , `'A'` , `'B'` , `'C'` , `'D'` , `'E'` , `'F'` .

Выделение Unicode сопоставляется компилятором Java с символом (строго говоря, 16-разрядным *блоком кода* Unicode) и может использоваться в любом месте исходного кода, в котором действительный отображаемый символ действителен. Он обычно используется в символьных и строковых литералах, когда вам нужно представить символ не-ASCII в литерале.

Исключение в регулярных выражениях

TBD

Десятичные целые литеры

Целочисленные литералы предоставляют значения, которые могут использоваться там, где вам нужен `byte` , `short` , `int` , `long` или `char` . (В этом примере основное внимание уделяется простым десятичным формам. В других примерах объясняется, как литералы в восьмеричных, шестнадцатеричных и двоичных выражениях, а также использование подчеркиваний для повышения удобочитаемости.)

Обычные целые литералы

Простейшей и наиболее распространенной формой целочисленного литерала является десятичный целочисленный литерал. Например:

```
0       // The decimal number zero       (type 'int')
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

Вы должны быть осторожны с ведущими нулями. Ведущий нуль приводит к тому, что целочисленный литерал интерпретируется как *восьмеричный,* а не десятичный.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Целочисленные литералы неподписанны. Если вы видите что - то вроде `-10` или `+10` , это на самом деле *выражение* , использующее одинарных `-` и унарных `+` операторов.

Диапазон целочисленных литералов этой формы имеет внутренний тип `int` и должен находиться в диапазоне от 0 до 2 ^{31} или 2 147 483 648.

Обратите внимание, что 2 ^{31} равно `1` больше `Integer.MAX_VALUE` . Литералы от 0 до до `2147483647` можно использовать в любом месте, но это ошибка компиляции использовать `2147483648` без предшествующего одноместной `-` оператора. (Другими словами, он зарезервирован для выражения значения `Integer.MIN_VALUE` .)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Длинные целые литералы

Литералы типа `long` выражаются добавлением суффикса `L` Например:

```
0L          // The decimal number zero       (type 'long')
1L          // The decimal number one        (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648;    // ERROR
long big2 = 2147483648L;  // OK
```

Обратите внимание, что различие между `int` и `long` литералами значимо в других местах. Например

```
int i = 2147483647;
long l = i + 1;           // Produces a negative value because the operation is
                          // performed using 32 bit arithmetic, and the
                          // addition overflows
long l2 = i + 1L;         // Produces the (intuitively) correct value.
```

Ссылка: JLS 3.10.1 - Целочисленные литералы

Булевы литералы

Булевы литералы являются простейшим из литералов на языке программирования Java. Два возможных `boolean` значения представлены литералами `true` и `false` . Они чувствительны к регистру. Например:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Строковые литералы

Строковые литералы предоставляют наиболее удобный способ представления строковых значений в исходном коде Java. Строковый литерал состоит из:

- Открывающий символ двойной кавычки (`"`).
- Нулевой или более других символов, которые не являются ни двойным кавычкой, ни символом прерывания строки. (Символ обратной косой черты (`\`) изменяет значение последующих символов, см. Последовательности Escape в литералах .)
- Закрывающий символ двойной кавычки.

Например:

```
"Hello world"   // A literal denoting an 11 character String
""              // A literal denoting an empty (zero length) String
"\""            // A literal denoting a String consisting of one
                //     double quote character
"1\t2\t3\n"     // Another literal with escape sequences
```

Обратите внимание, что один строковый литерал может не охватывать несколько строк исходного кода. Это ошибка компиляции для разрыва строки (или конца исходного файла)

перед закрывающей двойной кавычкой литерала. Например:

```
"Jello world    // Compilation error (at the end of the line!)
```

Длинные строки

Если вам нужна строка, которая слишком длинная, чтобы поместиться на линии, обычный способ выразить ее состоит в том, чтобы разбить ее на несколько литералов и использовать оператор конкатенации (`+`) для объединения фигур. Например

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

Выражение, подобное выше, состоящее из строковых литералов и `+` удовлетворяет требованиям как константное выражение . Это означает, что выражение будет оцениваться компилятором и представлено во время выполнения одним объектом `String` .

Интернирование строковых литералов

Когда файл класса, содержащий строковые литералы, загружается JVM, соответствующие объекты `String` *интернируются* системой выполнения. Это означает, что строковый литерал, используемый в нескольких классах, занимает больше места, чем если бы он использовался в одном классе.

Дополнительные сведения о интернировании и пуле строк см. В примере String pool и кучи памяти в разделе «Строки».

Нулевой литерал

Нулевой литерал (записанный как `null`) представляет одно и единственное значение нулевого типа. вот несколько примеров

```
    MyClass object = null;
    MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

    myMethod(null);

    if (objects != null) {
         // Do something
    }
```

Нулевой тип довольно необычен. У него нет имени, поэтому вы не можете выразить его в исходном коде Java. (И у него нет представления во время выполнения).

Единственной целью нулевого типа является тип `null` . Это присвоение, совместимое со

всеми ссылочными типами, и может быть введено для любого ссылочного типа. (В последнем случае приведение не требует проверки типа времени выполнения).

Наконец, `null` имеет свойство, что `null instanceof <SomeReferenceType>` будет оцениваться как `false` , независимо от типа.

Литералы с плавающей запятой

Литералы с плавающей запятой предоставляют значения, которые можно использовать там, где вам нужен `float` или `double` экземпляр. Существует три типа литералов с плавающей запятой.

- Простые десятичные формы
- Масштабированные десятичные формы
- Шестнадцатеричные формы

(Синтаксические правила JLS объединяют две десятичные формы в одну форму. Мы относимся к ним отдельно для удобства объяснения.)

Существуют разные литературные типы для `float` и `double` литералов, выраженные с использованием суффиксов. Различные формы используют буквы для выражения разных вещей. Эти буквы нечувствительны к регистру.

Простые десятичные формы

Простейшая форма литерала с плавающей запятой состоит из одной или нескольких десятичных цифр и десятичной точки (`.`) И необязательного суффикса (`f` , `F` , `d` или `D`). Необязательный суффикс позволяет указать, что литерал представляет собой значение `float` (`f` или `F`) или `double` (`d` или `D`). Значение по умолчанию (если суффикс не указан) является `double` .

Например

```
0.0     // this denotes zero
.0      // this also denotes zero
0.      // this also denotes zero
3.14159 // this denotes Pi, accurate to (approximately!) 5 decimal places.
1.0F    // a `float` literal
1.0D    // a `double` literal.  (`double` is the default if no suffix is given)
```

Фактически, десятичные цифры, за которыми следует суффикс, также являются литералами с плавающей запятой.

```
1F      // means the same thing as 1.0F
```

Значение десятичного литерала - это число с плавающей запятой IEEE, которое *ближе*

всего к математическому вещественному числу бесконечной точности, обозначенному десятичной формой с плавающей запятой. Это концептуальное значение преобразуется в двоичное представление с плавающей точкой IEEE с *округлением до ближайшего* . (Точная семантика десятичного преобразования указана в javadocs для `Double.valueOf(String)` и `Float.valueOf(String)` , имея в виду, что существуют различия в синтаксисе числа.)

Масштабированные десятичные формы

Масштабированные десятичные формы состоят из простой десятичной дроби с частью экспоненты, введенной `E` или `e` , и за ней следует целое число со знаком. Часть экспоненты представляет собой короткую руку для умножения десятичной формы на десять, как показано в приведенных ниже примерах. Существует также дополнительный суффикс, чтобы отличать `float` и `double` литералы. Вот некоторые примеры:

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)
1E-1D    // this means 1.0 x 10^(-1) ... or 0.1 (double)
1.0e10f  // this means 1.0 x 10^(10) ... or 10000000000.0 (float)
```

Размер литерала ограничен представлением (`float` или `double`). Это ошибка компиляции, если масштабный коэффициент приводит к слишком большому или слишком маленькому значению.

Шестнадцатеричные формы

Начиная с Java 6, можно выразить литералы с плавающей запятой в шестнадцатеричном формате. Шестнадцатеричная форма имеет аналогичный синтаксис для простых и масштабированных десятичных форм со следующими отличиями:

1. Каждый шестнадцатеричный литерал с плавающей запятой начинается с нуля (`0`), а затем `x` или `X`
2. Цифры номера (но *не* части экспоненты!) Также включают шестнадцатеричные цифры от `a` до `f` и их прописные эквиваленты.
3. Показатель является *обязательным* и вводится буквой `p` (или `P`) вместо `e` или `E` Показатель представляет собой коэффициент масштабирования, который представляет собой мощность 2 вместо мощности 10.

Вот некоторые примеры:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Совет. Поскольку шестнадцатеричные формы с плавающей запятой не знакомы большинству программистов на Java, рекомендуется использовать их экономно.

подчеркивания

Начиная с Java 7, символы подчеркивания допускаются в цифровых строках во всех трех формах литералов с плавающей запятой. Это относится и к «экспоненциальным» частям. См. Раздел Использование подчеркивания для повышения удобочитаемости .

Особые случаи

Это ошибка компиляции, если литерал с плавающей запятой обозначает число, которое слишком велико или слишком мало для представления в выбранном представлении; т.е. если число будет переполняться до + INF или -INF или underflow до 0.0. Тем не менее, для литерала законно представлять ненулевое денормализованное число.

Синтаксис букв с плавающей запятой не предоставляет буквенных представлений для специальных значений IEEE 754, таких как значения INF и NaN. Если вам нужно выразить их в исходном коде, рекомендуется использовать константы, определенные `java.lang.Float` и `java.lang.Double` ; например `Float.NaN` , `Float.NEGATIVE_INFINITY` и `Float.POSITIVE_INFINITY` .

Литералы символов

Литералы символов предоставляют наиболее удобный способ выражения значений `char` в исходном коде Java. Литеральный символ состоит из:

- Открывающий символ одной кавычки (`'`).
- Представление символа. Это представление не может быть символом одиночной кавычки или строки, но это может быть escape-последовательность, введенная символом обратной косой черты (`\`); см. последовательности Escape в литералах .
- Закрывающий символ одной кавычки (`'`).

Например:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

Разрыв строки в символьном литерале является ошибкой компиляции:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Прочитайте литералы онлайн: https://riptutorial.com/ru/java/topic/8250/литералы

глава 103: Локализация и интернационализация

замечания

Java поставляется с мощным и гибким механизмом для локализации ваших приложений, но также легко злоупотреблять и завершать работу с программой, которая игнорирует или изменяет локаль пользователя, и, следовательно, как они ожидают, что ваша программа будет вести себя.

Ваши пользователи ожидают увидеть данные, локализованные в форматах, к которым они привыкли, и попытка поддерживать это вручную - это безумное поручение. Вот лишь небольшой пример того, как пользователи ожидают увидеть контент, который вы можете предположить, «всегда» отображается определенным образом:

	Даты	чисел	Местная валюта	Иностранная валюта	Расстояния
Бразилия					
Китай					
Египет					
Мексика	20/3/16	1.234,56	$ 1,000.50	1,000.50 USD	
Соединенное Королевство	20/3/16	1,234.56	£ 1,000.50		100 км
Соединенные Штаты Америки	3/20/16	1,234.56	$ 1,000.50	1,000.50 MXN	60 миль

Общие ресурсы

- Википедия: интернационализация и локализация

Ресурсы Java

- Учебник по Java: интернационализация
- Oracle: интернационализация: понимание языка в платформе Java
- JavaDoc: `Locale`

Examples

Автоматически отформатированные даты с использованием «locale»

`SimpleDateFormatter` отлично подходит, но, как и название, он не очень хорошо масштабируется.

Если вы внедрили `"MM/dd/yyyy"` всем своем приложении, ваши международные пользователи не будут счастливы.

Пусть Java сделает для вас работу

Используйте `static` методы в `DateFormat` чтобы получить правильное форматирование для вашего пользователя. Для настольного приложения (где вы будете полагаться на локаль по умолчанию) просто вызовите:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Где `style` - одна из констант форматирования (`FULL` , `LONG` , `MEDIUM` , `SHORT` и т. Д.), Указанных в `DateFormat` .

Для приложения на стороне сервера, где пользователь указывает свой локаль как часть запроса, вы должны передать его явно в `getDateInstance()` :

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Сравнение строк

Сравните два случая игнорирования строк:

```
"School".equalsIgnoreCase("school"); // true
```

Не использовать

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Языки имеют разные правила для преобразования верхнего и нижнего регистров. «I» будет преобразован в «i» на английском языке. Но на турецком языке «я» становится «ı». Если вам нужно использовать `toLowerCase()` используйте перегрузку, `String.toLowerCase(Locale) Locale` : `String.toLowerCase(Locale)` .

Сравнение двух строк, игнорирующих незначительные отличия:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Сортировка строк в соответствии с порядком естественного языка, игнорируя регистр (используйте ключ сопоставления для:

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // ignore case
Arrays.sort(texts, collator::compare); // will return {"Apfel", "äther", "Birne"}
```

место действия

Класс `java.util.Locale` используется для представления «географического, политического или культурного» региона для локализации данного текста, числа, даты или операции. Таким образом, объект Locale может содержать страну, регион, язык, а также вариант языка, например диалект, произнесенный в определенном регионе страны, или разговариваемый в другой стране, чем страна, из которой происходит этот язык.

Экземпляр Locale передается компонентам, которые должны локализовать свои действия, независимо от того, преобразует ли он вход, выход или просто нуждается в его внутренних операциях. Класс Locale не может выполнять какую-либо интернационализацию или локализацию самостоятельно

язык

Язык должен быть ISO 639 2 или 3 символьным языковым кодом или зарегистрированным языковым subtag длиной до 8 символов. В случае, если на языке есть код с кодом 2 и 3 символа, используйте 2 символьный код. Полный список кодов языков можно найти в реестре субтега языка IANA.

Коды языков нечувствительны к регистру, но класс Locale всегда использует строчные версии кодов языков

Создание локали

Создание экземпляра `java.util.Locale` можно выполнить четырьмя различными способами:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

Вы создаете экземпляр ResourceBundle следующим образом:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

У меня есть файл `i18n.properties` :

```
message=This is locale
```

Выход:

```
This is locale
```

Настройка локали

Если вы хотите воспроизвести состояние с использованием других языков, вы можете использовать **`setDefault()`** . Его использование:

```
 setDefault(Locale.JAPANESE); //Set Japanese
```

Прочитайте Локализация и интернационализация онлайн:
https://riptutorial.com/ru/java/topic/4086/локализация-и-интернационализация

глава 104: Лямбда-выражения

Вступление

Лямбда-выражения обеспечивают четкий и лаконичный способ реализации интерфейса одного метода с использованием выражения. Они позволяют вам уменьшить количество кода, который вы должны создавать и поддерживать. Хотя они похожи на анонимные классы, они сами не имеют информации о типе. Необходимо ввести вывод типа.

Ссылки на методы реализуют функциональные интерфейсы, используя существующие методы, а не выражения. Они также принадлежат к лямбда-семье.

Синтаксис

- () -> {return expression; } // Zero-arity с телом функции, чтобы вернуть значение.
- () -> выражение // Сокращение для указанного объявления; для выражений нет точки с запятой.
- () -> {function-body} // Побочное действие в выражении лямбда для выполнения операций.
- parameterName -> expression // Одномерное лямбда-выражение. В лямбда-выражениях с одним аргументом скобки могут быть удалены.
- (Тип parameterName, Тип secondParameterName, ...) -> выражение // lambda, оценивающее выражение с параметрами, перечисленными слева
- (parameterName, secondParameterName, ...) -> expression // Сокращение, которое удаляет типы параметров для имен параметров. Может использоваться только в контекстах, которые могут быть выведены компилятором, где размер списка заданных параметров соответствует одному (и только одному) размера ожидаемых функциональных интерфейсов.

Examples

Использование выражений лямбда для сортировки коллекции

Сортировка списков

До Java 8 необходимо было реализовать интерфейс `java.util.Comparator` с анонимным (или названным) классом при сортировке списка [1] :

Java SE 1.2

```
List<Person> people = ...
```

```
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

Начиная с Java 8, анонимный класс можно заменить выражением лямбда. Обратите внимание, что типы параметров `p1` и `p2` могут быть опущены, так как компилятор автоматически выведет их:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

Пример можно упростить, используя ссылки `Comparator.comparing` и методы, выраженные с помощью символа `::` (двойной двоеточие).

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

Статический импорт позволяет выразить это более сжато, но это спорно это улучшает ли общую читаемость:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

Компараторы, построенные таким образом, также могут быть соединены вместе. Например, после сравнения людей по их имени, если есть люди с одинаковым именем, метод `thenComparing` также сравнивается по имени:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

---

1 - Обратите внимание, что Collections.sort (...) работает только с коллекциями, которые являются подтипами `List` . API `Set` и `Collection` не подразумевает упорядочения элементов.

Сортировка карт

Вы можете сортировать записи `HashMap` по значению аналогичным образом. (Обратите внимание , что `LinkedHashMap` должен быть использован в качестве мишени. Ключи в

обычной `HashMap` являются неупорядоченными.)

```
Map<String, Integer> map = new HashMap();  // ... or any other Map class
// populate the map
map = map.entrySet()
    .stream()
    .sorted(Map.Entry.<String, Integer>comparingByValue())
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),
                              (k, v) -> k, LinkedHashMap::new));
```

Введение в Java lambdas

Функциональные интерфейсы

Lambdas может работать только на функциональном интерфейсе, который является интерфейсом только с одним абстрактным методом. Функциональные интерфейсы могут иметь любое количество `default` или `static` методов. (По этой причине они иногда называются интерфейсами единого абстрактного метода или интерфейсами SAM).

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

При объявлении функционального интерфейса может быть добавлена аннотация `@FunctionalInterface` . Это не имеет особого эффекта, но ошибка компилятора будет сгенерирована, если эта аннотация применяется к интерфейсу, который не является функциональным, тем самым действуя как напоминание о том, что интерфейс не должен изменяться.

```
@FunctionalInterface
interface Foo5 {
    void bar();
}

@FunctionalInterface
interface BlankFoo1 extends Foo3 { // inherits abstract method from Foo3
```

```
}

@FunctionalInterface
interface Foo6 {
    void bar();
    boolean equals(Object obj); // overrides one of Object's method so not counted
}
```

И наоборот, это **не** функциональный интерфейс, так как он имеет более **одного абстрактного** метода:

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

Это **также не** функциональный интерфейс, так как он не имеет никаких методов:

```
interface BlankFoo2 { }
```

Обратите внимание на следующее. Предположим, что у вас есть

```
interface Parent { public int parentMethod(); }
```

а также

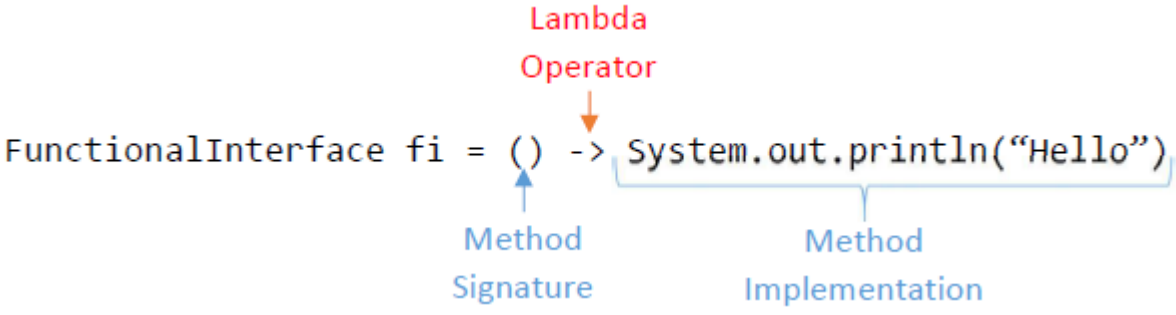
```
interface Child extends Parent { public int ChildMethod(); }
```

Тогда `Child` **не может** быть функциональным интерфейсом, поскольку он имеет два указанных метода.

Java 8 также предоставляет ряд общих шаблонных функциональных интерфейсов в пакете `java.util.function` . Например, встроенный интерфейс `Predicate<T>` обертывает один метод, который вводит значение типа `T` и возвращает `boolean` .

Лямбда-выражения

Основная структура выражения Лямбды:



`fi` затем проведет одиночный экземпляр класса, аналогичный анонимному классу, который реализует `FunctionalInterface` и где определение одного метода `{ System.out.println("Hello"); }` . Другими словами, вышесказанное в основном эквивалентно:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

Лямбда только « в основном эквивалент» анонимного класса , потому что в лямбда, смысл выражений , как `this` , `super` или `toString()` ссылается на класс , внутри которого назначение происходит, а не вновь созданный объект.

Вы не можете указать имя метода при использовании лямбда-но вам не нужно, потому что функциональный интерфейс должен иметь только один абстрактный метод, поэтому Java переопределяет его.

В случаях, когда тип лямбда не определен (например, перегруженные методы), вы можете добавить бросок в лямбда, чтобы сообщить компилятору, каким должен быть его тип, например:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

Если единственный метод функционального интерфейса принимает параметры, локальные формальные имена должны появляться между скобками лямбда. Нет необходимости объявлять тип параметра или возвращать, поскольку они взяты из интерфейса (хотя это не ошибка, чтобы объявлять типы параметров, если вы хотите). Таким образом, эти два примера эквивалентны:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

Скобки вокруг аргумента могут быть опущены, если функция имеет только один аргумент:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Неявные возвращения

Если код, помещенный внутри лямбда, является *выражением* Java, а не *оператором* , он рассматривается как метод, который возвращает значение выражения. Таким образом, следующие два эквивалентны:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Доступ к локальным переменным (закрытие значений)

Поскольку лямбды являются синтаксическими сокращениями для анонимных классов, они следуют тем же правилам для доступа к локальным переменным в охватывающей области; переменные должны рассматриваться как `final` и не модифицироваться внутри лямбда.

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
                                // because amount is not modified
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

Если необходимо обернуть переменную изменения таким образом, следует использовать обычный объект, который хранит копию переменной. Читайте больше в Java Closures с лямбда-выражениями.

Принятие Lambdas

Поскольку лямбда - это реализация интерфейса, ничего особенного не нужно делать, чтобы метод принимал лямбда: любая функция, которая принимает функциональный интерфейс, также может принимать лямбда.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

Тип выражения лямбда

Выражение лямбда само по себе не имеет определенного типа. Хотя верно, что типы и

количество параметров вместе с типом возвращаемого значения могут передавать некоторую информацию о типе, такая информация будет ограничивать только те типы, которым она может быть назначена. Лямбда получает тип, когда ему назначается тип функционального интерфейса одним из следующих способов:

- Прямое присвоение функциональному типу, например `myPredicate = s -> s.isEmpty()`
- Передача его как параметра, который имеет функциональный тип, например `stream.filter(s -> s.isEmpty())`
- Возврат его из функции, возвращающей функциональный тип, например `return s -> s.isEmpty()`
- Передача его функциональному типу, например `(Predicate<String>) s -> s.isEmpty()`

Пока не будет выполнено такое присвоение функциональному типу, лямбда не имеет определенного типа. Для иллюстрации рассмотрим лямбда-выражение `o -> o.isEmpty()`. Одно и то же выражение лямбда может быть присвоено многим различным функциональным типам:

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // return value is ignored!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

Теперь, когда они назначены, показанные примеры имеют совершенно разные типы, хотя выражения лямбда выглядят одинаково, и они не могут быть назначены друг другу.

Ссылки на методы

Ссылки на методы позволяют предопределять статические или экземплярные методы, которые соответствуют совместимому функциональному интерфейсу, которые передаются как аргументы, а не анонимное выражение лямбда.

Предположим, что у нас есть модель:

```
class Person {
    private final String name;
    private final String surname;

    public Person(String name, String surname){
        this.name = name;
        this.surname = surname;
    }

    public String getName(){ return name; }
    public String getSurname(){ return surname; }
}

List<Person> people = getSomePeople();
```

Ссылка метода экземпляра (на произвольный экземпляр)

```
people.stream().map(Person::getName)
```

Эквивалентная лямбда:

```
people.stream().map(person -> person.getName())
```

В этом примере `getName()` ссылка метода на метод `getName()` экземпляра типа `Person` . Поскольку известно, что это тип коллекции, будет вызван метод экземпляра (известный позже).

Справочник метода экземпляра (к конкретному экземпляру)

```
people.forEach(System.out::println);
```

Поскольку `System.out` является экземпляром `PrintStream` , ссылка метода на этот конкретный экземпляр передается в качестве аргумента.

Эквивалентная лямбда:

```
people.forEach(person -> System.out.println(person));
```

Ссылка на статический метод

Также для преобразования потоков мы можем применить ссылки на статические методы:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

В этом примере приведена ссылка на статический метод `valueOf()` для типа `String` . Следовательно, объект экземпляра в коллекции передается как аргумент `valueOf()` .

Эквивалентная лямбда:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Ссылка на конструктор

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Прочитайте « Собрать элементы потока в коллекцию», чтобы увидеть, как собирать элементы в коллекцию.

Здесь используется единственный конструктор аргументов String типа `Integer` , чтобы построить целое число, заданное строкой, предоставленной в качестве аргумента. В этом случае, пока строка представляет число, поток будет отображаться в «Целые». Эквивалентная лямбда:

```
strings.stream().map(s -> new Integer(s));
```

Чит-лист

Формат ссылки метода	Код	Эквивалентная лямбда
Статический метод	`TypeName::method`	`(args) -> TypeName.method(args)`
Нестатический метод (например, *)	`instance::method`	`(args) -> instance.method(args)`
Нестатический метод (нет экземпляра)	`TypeName::method`	`(instance, args) -> instance.method(args)`
Конструктор **	`TypeName::new`	`(args) -> new TypeName(args)`
Конструктор массива	`TypeName[]::new`	`(int size) -> new TypeName[size]`

---

* `instance` может быть любым выражением, которое оценивает ссылку на экземпляр, например `getInstance()::method` , `this::method`

** Если `TypeName` является `TypeName` внутренним классом, ссылка на конструктор действительна только в пределах экземпляра внешнего класса

Реализация нескольких интерфейсов

Иногда вам может понадобиться выражение лямбда, реализующее более одного интерфейса. Это в основном полезно с интерфейсами маркеров (например, java.io.Serializable), поскольку они не добавляют абстрактные методы.

Например, вы хотите создать `TreeSet` с пользовательским `Comparator` а затем сериализовать его и отправить по сети. Тривиальный подход:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

не работает, поскольку лямбда для компаратора не реализует `Serializable` . Вы можете исправить это, используя типы пересечений и явно указав, что этот лямбда должен быть сериализуемым:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

Если вы часто используете типы пересечений (например, если вы используете фреймворк, такой как Apache Spark, где почти все должно быть сериализуемым), вы можете создавать пустые интерфейсы и вместо этого использовать их в своем коде:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

Таким образом, вы гарантируете, что переданный компаратор будет сериализуемым.

Ламбдас и шаблон выживания

Существует несколько хороших примеров использования lambdas как FunctionalInterface в простых сценариях. Довольно распространенный вариант использования, который может быть улучшен lambdas, - это так называемый шаблон Execute-Around. В этом шаблоне у вас есть набор стандартного кода установки / разрыва, который необходим для нескольких сценариев, связанных с конкретным кодом конкретного случая использования. Несколько распространенных примеров этого - файлы io, базы данных io, try / catch.

```
interface DataProcessor {
    void process( Connection connection ) throws SQLException;;
}

public void doProcessing( DataProcessor processor ) throws SQLException{
    try (Connection connection = DBUtil.getDatabaseConnection();) {
        processor.process(connection);
        connection.commit();
    }
}
```

Затем, чтобы вызвать этот метод с лямбдой, он может выглядеть так:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

Это не ограничивается операциями ввода-вывода. Он может применяться к любому сценарию, где аналогичные задачи установки / срыва применяются с незначительными

вариациями. Основное преимущество этого шаблона - повторное использование кода и принудительное использование DRY (Do not Repeat Yourself).

Использование выражения лямбда с вашим собственным функциональным интерфейсом

Lambdas предназначены для предоставления встроенного кода реализации для интерфейсов с одним интерфейсом и способности передавать их, как мы делали с обычными переменными. Мы называем их функциональным интерфейсом.

Например, запись Runnable в анонимный класс и начало Thread выглядит так:

```
//Old way
new Thread(
        new Runnable(){
            public void run(){
                System.out.println("run logic...");
            }
        }
).start();

//lambdas, from Java 8
new Thread(
        ()-> System.out.println("run logic...")
).start();
```

Теперь, в соответствии с выше, скажем, у вас есть пользовательский интерфейс:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

Как вы используете лямбда для реализации этого интерфейса в своем коде? То же, что и пример Runnable, показанный выше. Смотрите программу драйвера ниже:

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("Divisor can not be 0");
            return a/b;
        };

        System.out.println("Plus operation of 3 and 5 is: " + plusOperation.operate(3, 5));
        System.out.println("Divide operation 50 by 25 is: " + divideOperation.operate(50,
25));

    }
}
```

`return` возвращается только из лямбда, а не из внешнего метода

Метод `return` возвращается только из лямбда, а не из внешнего метода.

Остерегайтесь, что это *отличается* от Scala и Kotlin!

```
void threeTimes(IntConsumer r) {
  for (int i = 0; i < 3; i++) {
    r.accept(i);
  }
}

void demo() {
  threeTimes(i -> {
    System.out.println(i);
    return; // Return from lambda to threeTimes only!
  });
}
```

Это может привести к неожиданному поведению при попытке записи собственных языковых конструкций, как в конструкциях встроенных , такие как `for` петель `return` ведет себя по- разному:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

В Scala и Kotlin `demo` и `demo2` будут только печатать `0` . Но это *не* является более последовательным. Подход Java совместим с рефакторингом и использованием классов - `return` в код вверху, а приведенный ниже код ведет себя одинаково:

```
void demo3() {
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
    System.out.println(i);
    return;
  }
}
```

Таким образом, `return` Java более совместим с методами класса и рефакторингом, но меньше с функциями `for` и `while` builtins, они остаются особенными.

Из-за этого следующие два эквивалентны в Java:

```
IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
```

```
    .forEach(System.out::println);
```

Кроме того, использование try-in-resources безопасно в Java:

```
class Resource implements AutoCloseable {
  public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
  try (Resource r = new Resource()) {
    System.out.print("before ");
    f.accept(r);
    System.out.print("after ");
  }
}

void demo4() {
  executeAround(r -> {
    System.out.print("accept() ");
    return; // Does not return from demo4, but frees the resource.
  });
}
```

будет печатать `before accept() after close()` . В семантике Scala и Kotlin средства try-with-resources не будут закрыты, но они будут печататься `before accept()` .

Закрытие Java с лямбда-выражениями.

Закрытие лямбда создается, когда выражение лямбда ссылается на переменные охватывающей области (глобальной или локальной). Правила для этого те же, что и для встроенных методов и анонимных классов.

Локальные переменные из охватывающей области, которые используются в лямбда, должны быть `final` . С Java 8 (самая ранняя версия, поддерживающая lambdas), они не обязательно должны быть *объявлены* `final` во внешнем контексте, но с этим нужно обращаться. Например:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

Это законно, если значение переменной `n` не изменяется. Если вы попытаетесь изменить переменную, внутри или вне лямбда, вы получите следующую ошибку компиляции:

> «Локальные переменные, на которые ссылается выражение лямбда, должны быть *окончательными* или *фактически окончательными* ».

Например:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

Если необходимо использовать переменную изменения в лямбда, нормальный подход заключается в объявлении `final` копии переменной и использовании копии. Например

```
int n = 0;
final int k = n; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = k;
    // do something
};
n++;      // Now will not generate an error
r.run();  // Will run with i = 0 because k was 0 when the lambda was created
```

Естественно, тело лямбда не видит изменений исходной переменной.

Обратите внимание: Java не поддерживает истинные закрытия. Java-лямбда не может быть создана таким образом, чтобы она могла видеть изменения в среде, в которой она была создана. Если вы хотите реализовать закрытие, которое отслеживает или вносит изменения в его среду, вы должны имитировать его с использованием обычного класса. Например:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

Вышеприведенный пример не будет компилироваться по причинам, обсуждавшимся ранее. Мы можем обойти ошибку компиляции следующим образом:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

Проблема заключается в том, что это нарушает контракт на `IntUnaryOperator` интерфейса `IntUnaryOperator` котором говорится, что экземпляры должны быть функциональными и неактивными. Если такое закрытие передается во встроенные функции, которые принимают функциональные объекты, это может привести к сбоям или ошибочному

поведению. Закрытия, которые инкапсулируют изменяемое состояние, должны быть реализованы как обычные классы. Например.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Лямбда - пример слушателя

Анонимный слушатель

До Java 8 очень распространено, что анонимный класс используется для обработки события click JButton, как показано в следующем коде. В этом примере показано, как реализовать анонимный прослушиватель в области `btn.addActionListener` .

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Лямбда-слушатель

Поскольку `ActionListener` интерфейс определяет только один метод `actionPerformed()` , он представляет собой функциональный интерфейс , который означает , что есть место , чтобы использовать лямбда - выражений для замены стандартного кода. Вышеприведенный пример можно переписать с использованием выражений Lambda следующим образом:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Традиционный стиль в стиле Лямбда

Традиционный способ

```
interface MathOperation{
    boolean unaryOperation(int num);
}
```

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

Лямбда-стиль

1. Удалить имя класса и тело функционального интерфейса.

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

2. Объявление необязательного типа

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Необязательная скобка вокруг параметра, если она является единственным параметром

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Дополнительные фигурные скобки, если в теле функции есть только одна строка
5. Необязательное ключевое слово return, если в теле функции есть только одна строка

```
MathOperation isEven = num -> num%2 == 0;
```

Использование Lambdas и памяти

Поскольку Java lambdas являются закрытием, они могут «захватывать» значения переменных в охватывающей лексической области. В то время как не все лямбды захватывают что угодно - простые лямбды, такие как `s -> s.length()` ничего не захватывают

и называются *апатридами* - захват lambdas требует временного объекта для хранения захваченных переменных. В этом фрагменте кода лямбда `() -> j` является захватывающей лямбдой и может вызывать выделение объекта при его оценке:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Хотя это может быть не сразу очевидным, так как `new` ключевое слово не появляется нигде в фрагменте, этот код может создать 1,000,000,000 отдельных объектов, чтобы представлять экземпляры выражения `() -> j` lambda. Однако следует также отметить, что будущие версии Java [1] могут оптимизировать это, чтобы *во время выполнения* экземпляры лямбда были повторно использованы или были представлены каким-то другим способом.

---

1 - Например, Java 9 вводит факультативную фазу «ссылка» в последовательность сборки Java, которая предоставит возможность для глобальных оптимизаций, подобных этому.

Использование лямбда-выражений и предикатов для получения определенного значения (-ов) из списка

Начиная с Java 8, вы можете использовать лямбда-выражения и предикаты.

Пример. Используйте лямбда-выражения и предикат, чтобы получить определенное значение из списка. В этом примере каждый человек будет распечатан с фактом, если ему 18 лет и старше или нет.

Класс личности:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() { return age; }
    public String getName() { return name; }
}
```

Встроенный интерфейс Predicate из пакета java.util.function.Predicate представляет собой функциональный интерфейс с методом `boolean test(T t)` .

Пример использования:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " matches your expression.");
            } else {
                System.out.println(person  + " doesn't match your expression.");
            }
        }
    }
}
```

`print(personList, p -> p.getAge() >= 18);` метод принимает лямбда-выражение (потому что в Predicate используется параметр), где вы можете определить требуемое выражение. Метод проверки checker проверяет правильность этого выражения или нет: `checker.test(person)` .

Вы можете легко изменить это на что-то другое, например, для `print(personList, p -> p.getName().startsWith("J"));` , Это проверяет, начинается ли имя человека с буквы «J».

Прочитайте Лямбда-выражения онлайн: https://riptutorial.com/ru/java/topic/91/лямбда-выражения

глава 105: Массивы

Вступление

Массивы позволяют хранить и извлекать произвольное количество значений. Они аналогичны векторам в математике. Массивы массивов аналогичны матрицам и действуют как многомерные массивы. Массивы могут хранить любые данные любого типа: примитивы, такие как `int` или ссылочные типы, такие как `Object` .

Синтаксис

- `ArrayType[] myArray;` // Объявление массивов
- `ArrayType myArray[];` // Другой допустимый синтаксис (менее часто используемый и обескураженный)
- `ArrayType[][][] myArray;` // Объявление многомерных зубчатых массивов (repeat [] s)
- `ArrayType myVar = myArray[index];` // Доступ к элементу (считыванию) по индексу
- `myArray[index] = value;` // Присвоить значение `index` позиции массива
- `ArrayType[] myArray = new ArrayType[arrayLength];` // Синтаксис инициализации массива
- `int[] ints = {1, 2, 3};` // Синтаксис инициализации массива с предоставленными значениями, длина выводится из числа предоставленных значений: {[value1 [, value2] *]}
- `new int[]{4, -5, 6} // Can be used as argument, without a local variable`
- `int[] ints = new int[3]; // same as {0, 0, 0}`
- `int[][] ints = {{1, 2}, {3}, null};` // Инициализация многомерных массивов. int [] extends Object (а также anyType []), поэтому значение null является допустимым.

параметры

параметр	подробности
`ArrayType`	Тип массива. Это может быть примитивным (`int` , `long` , `byte`) или Object (`String` , `MyObject` и т. Д.).
индекс	Индекс относится к позиции определенного объекта в массиве.
длина	Каждому массиву при создании требуется заданная длина. Это делается либо при создании пустого массива (`new int[3]`), либо подразумевается при указании значений (`{1, 2, 3}`).

Examples

Основные случаи

```
int[]   numbers1 = new int[3];                // Array for 3 int values, default value is 0
int[]   numbers2 = { 1, 2, 3 };               // Array literal of 3 int values
int[]   numbers3 = new int[] { 1, 2, 3 };     // Array of 3 int values initialized
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // Jagged array literal
int[][] numbers5 = new int[5][];              // Jagged array, one dimension 5 long
int[][] numbers6 = new int[5][4];             // Multidimensional array: 5x4
```

Массивы могут быть созданы с использованием любого примитивного или ссылочного типа.

```
float[]  boats = new float[5];          // Array of five 32-bit floating point numbers.
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
                                        // Array of five 64-bit floating point numbers.
String[] theory = new String[] { "a", "b", "c" };
                                        // Array of three strings (reference type).
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
                                        // Array of three Objects (reference type).
```

В последнем примере обратите внимание, что подтипы объявленного типа массива разрешены в массиве.

Массивы для пользовательских типов также могут быть построены подобно примитивным типам

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

Массивы, коллекции и потоки

Java SE 1.2

```
// Parameters require objects, not primitives

// Auto-boxing happening for int 127 here
Integer[]       initial        = { 127, Integer.valueOf( 42 ) };
List<Integer>   toList         = Arrays.asList( initial );  // Fixed size!

// Note: Works with all collections
Integer[]       fromCollection = toList.toArray( new Integer[toList.size()] );

//Java doesn't allow you to create an array of a parameterized type
List<String>[]  list = new ArrayList<String>[2];  // Compilation error!
```

Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream       = Arrays.stream( initial );
```

```
Integer[]       fromStream     = toStream.toArray( Integer[]::new );
```

вступление

Массив - это структура данных, которая содержит фиксированное число примитивных значений **или** ссылок на экземпляры объектов.

Каждый элемент в массиве называется элементом, и каждый элемент получает доступ к его числовому индексу. Длина массива устанавливается при создании массива:

```
int size = 42;
int[] array = new int[size];
```

Размер массива фиксируется во время выполнения при инициализации. Он не может быть изменен после инициализации. Если размер должен быть изменен во время выполнения, вместо него следует использовать класс `Collection` такой как `ArrayList` . `ArrayList` хранит элементы в массиве и поддерживает изменение размера путем выделения нового массива и копирования элементов из старого массива.

Если массив имеет примитивный тип, т. Е.

```
int[] array1 = { 1,2,3 };
int[] array2 = new int[10];
```

значения сохраняются в самом массиве. В отсутствие инициализатора (как в `array2` выше) значение по умолчанию, присвоенное каждому элементу, равно `0` (ноль).

Если тип массива является ссылкой на объект, как в

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

то массив содержит *ссылки* на объекты типа `SomeClassOrInterface` . Эти ссылки могут ссылаться на экземпляр `SomeClassOrInterface` *или любого подкласса (для классов) или на реализацию класса (для интерфейсов)* `SomeClassOrInterface` . Если объявление массива не имеет инициализатора, то каждому элементу присваивается значение по умолчанию `null` .

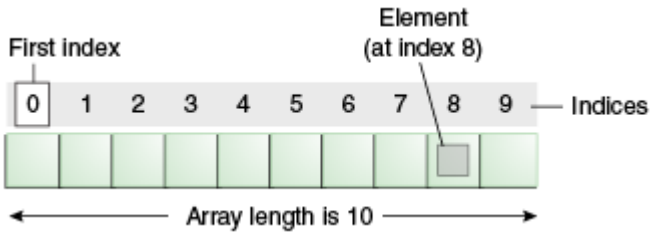
Поскольку все массивы являются `int` -индексированными, размер массива должен быть задан с помощью `int` . Размер массива не может быть задан как `long` :

```
long size = 23L;
int[] array = new int[size]; // Compile-time error:
                             // incompatible types: possible lossy conversion from
                             // long to int
```

В массивах используется **индексная** система с **нулевым значением** , что означает, что

индексирование начинается с `0` и заканчивается на `length - 1` .

Например, следующее изображение представляет массив с размером `10` . Здесь первый элемент находится в индексе `0` а последний элемент находится в индексе `9` , а первый элемент находится в индексе `1` и последний элемент в индексе `10` (см. Рисунок ниже).



Доступ к элементам массивов осуществляется в **постоянное время** . Это означает, что доступ к первому элементу массива имеет одинаковую стоимость (по времени) доступа к второму элементу, третьему элементу и так далее.

Java предлагает несколько способов определения и инициализации массивов, включая **буквенные** и **конструкторские** обозначения. При объявлении массивов с использованием `new Type[length]` каждый элемент будет инициализирован со следующими значениями по умолчанию:

- `0` для примитивных числовых типов : `byte` , `short` , `int` , `long` , `float` и `double` .
- `'\u0000'` (нулевой символ) для типа `char` .
- `false` для `boolean` типа.
- `null` для ссылочных типов .

Создание и инициализация массивов примитивных типов

```
int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and
                                      // array initializer.
int[] array2 = { 1, 2, 3 };           // Shortcut syntax with array initializer.
int[] array3 = new int[3];            // Equivalent to { 0, 0, 0 }
int[] array4 = null;                  // The array itself is an object, so it
                                      // can be set as null.
```

При объявлении массива `[]` будет отображаться как часть типа в начале объявления (после имени типа) или как часть декларатора для определенной переменной (после имени переменной) или обоих:

```
int array5[];       /* equivalent to */  int[] array5;
int a, b[], c[][];  /* equivalent to */  int a; int[] b; int[][] c;
int[] a, b[];       /* equivalent to */  int[] a; int[][] b;
int a, []b, c[][];  /* Compilation Error, because [] is not part of the type at beginning
                       of the declaration, rather it is before 'b'. */
```

```
// The same rules apply when declaring a method that returns an array:
int foo()[] { ... } /* equivalent to */  int[] foo() { ... }
```

В следующем примере обе декларации правильны и могут компилироваться и запускаться без каких-либо проблем. Однако как Конвенция по кодированию Java, так и Руководство по стилю Google Java препятствуют форме с помощью скобок после имени переменной - скобки идентифицируют тип массива и должны появляться с обозначением типа . То же самое следует использовать для сигнатур возврата метода.

```
float array[]; /* and */ int foo()[] { ... } /* are discouraged */
float[] array; /* and */ int[] foo() { ... } /* are encouraged */
```

Воспринимаемый тип предназначен для размещения переходных пользователей C , которые знакомы с синтаксисом C, который имеет скобки после имени переменной.

В Java возможно иметь массивы размером `0` :

```
int[] array = new int[0]; // Compiles and runs fine.
int[] array2 = {};        // Equivalent syntax.
```

Однако, поскольку это пустой массив, никакие элементы не могут быть прочитаны или назначены ему:

```
array[0] = 1;     // Throws java.lang.ArrayIndexOutOfBoundsException.
int i = array2[0]; // Also throws ArrayIndexOutOfBoundsException.
```

Такие пустые массивы обычно полезны в качестве возвращаемых значений, поэтому вызывающий код должен беспокоиться только о работе с массивом, а не о потенциальном `null` значении, которое может привести к исключению `NullPointerException` .

Длина массива должна быть неотрицательным целым числом:

```
int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException
```

Размер массива можно определить, используя публичное конечное поле с названием `length` :

```
System.out.println(array.length); // Prints 0 in this case.
```

Примечание . `array.length` возвращает фактический размер массива, а не количество элементов массива, которым было присвоено значение, в отличие от `ArrayList.size()` которое возвращает количество элементов массива, которым было присвоено значение.

Создание и инициализация многомерных

массивов

Самый простой способ создания многомерного массива состоит в следующем:

```
int[][] a = new int[2][3];
```

Это создаст две три длины `int` arrays- a `a[0]` и `a[1]` . Это очень похоже на классическую инициализацию прямоугольных многомерных массивов в стиле C.

Вы можете создавать и инициализировать одновременно:

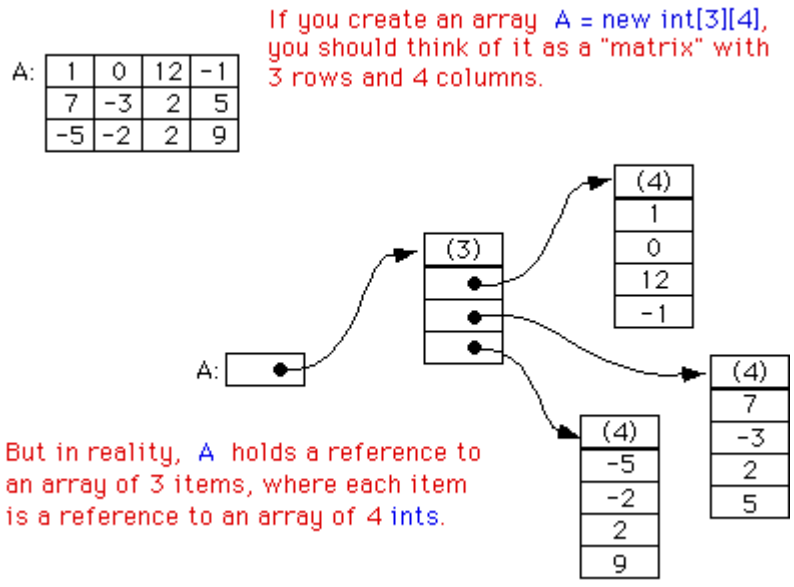
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

В отличие от C , где поддерживаются только прямоугольные многомерные массивы, внутренние массивы не должны быть одинаковой длины или даже определены:

```
int[][] a = { {1}, {2, 3}, null };
```

Здесь, `a[0]` является одной длиной `int` массива, в то время как `a[1]` представляет собой две длины `int` массива и `a[2]` является `null` . Такие массивы называются зубчатыми массивами или оборванными массивами , то есть массивами массивов. Многомерные массивы в Java реализованы как массивы массивов, т. `array[i][j][k]` эквивалентен `((array[i])[j])[k]` . В отличие от C # , синтаксический `array[i,j]` не поддерживается в Java.

Представление многомерного массива в Java



Источник - Live on Ideone

Создание и инициализация массивов ссылочных типов

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new
                                                      // operator and array initializer.
String[] array7 = { "Laurel", "Hardy" };              // Shortcut syntax with array
                                                      // initializer.
String[] array8 = new String[3];                      // { null, null, null }
String[] array9 = null;                               // null
```

Живой на Ideone

В дополнение к `String` литералам и примитивам, показанным выше, синтаксис ярлыков для инициализации массива также работает с каноническими типами `Object` :

```
Object[] array10 = { new Object(), new Object() };
```

Поскольку массивы являются ковариантными, массив ссылочного типа может быть инициализирован как массив подкласса, хотя `ArrayStoreException` будет `ArrayStoreException` , если вы попытаетесь установить элемент в нечто иное, чем `String` :

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

Синтаксис ярлыка не может использоваться для этого, потому что синтаксис ярлыка будет иметь неявный тип `Object[]` .

Массив может быть инициализирован нулевыми элементами, используя `String[] emptyArray = new String[0]` . Например, массив с нулевой длиной, подобный этому, используется для создания `Array` из `Collection` когда методу нужен тип среды выполнения объекта.

В обоих примитивных и ссылочных типах пустая инициализация массива (например, `String[] array8 = new String[3]`) инициализирует массив значением по умолчанию для каждого типа данных .

Создание и инициализация массивов типичного типа

В родовых классах массивы генерических типов **не могут** быть инициализированы,

например, из-за стирания типа :

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
    }
}
```

Вместо этого они могут быть созданы с использованием одного из следующих способов: (обратите внимание, что они будут генерировать непроверенные предупреждения)

1. Создав массив `Object` и переведя его в общий тип:

   ```
   a = (T[]) new Object[5];
   ```

   Это самый простой метод, но поскольку базовый массив по-прежнему имеет тип `Object[]` , этот метод не обеспечивает безопасность типов. Поэтому этот метод создания массива лучше всего использовать только в универсальном классе - не публично публиковаться.

2. Используя `Array.newInstance` с параметром класса:

   ```
   public MyGenericClass(Class<T> clazz) {
       a = (T[]) Array.newInstance(clazz, 5);
   }
   ```

   Здесь класс `T` должен быть явно передан конструктору. Возвращаемый тип `Array.newInstance` всегда является `Object` . Однако этот метод более безопасен, потому что вновь созданный массив всегда имеет тип `T[]` и поэтому может быть безопасно экстернализирован.

Заполнение массива после инициализации

Java SE 1.2

`Arrays.fill()` может использоваться для заполнения массива с **тем же значением** после инициализации:

```
Arrays.fill(array8, "abc");         // { "abc", "abc", "abc" }
```

Живой на Ideone

`fill()` также может присваивать значение каждому элементу указанного диапазона массива:

```
Arrays.fill(array8, 1, 2, "aaa");  // Placing "aaa" from index 1 to 2.
```

Живой на Ideone

Java SE 8

Поскольку Java-версия 8, метод `setAll` и ее `Concurrent` эквивалент `parallelSetAll` , могут использоваться для установки каждого элемента массива на сгенерированные значения. Этим методам передается функция генератора, которая принимает индекс и возвращает желаемое значение для этой позиции.

Следующий пример создает целочисленный массив и устанавливает все его элементы в соответствующее значение индекса:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

Живой на Ideone

Отдельная декларация и инициализация массивов

Значение индекса для элемента массива должно быть целым числом (0, 1, 2, 3, 4, ...) и меньше длины массива (индексы основаны на нуле). В противном случае будет выбрано исключение ArrayIndexOutOfBoundsException :

```
int[] array9;              // Array declaration - uninitialized
array9 = new int[3];       // Initialize array  - { 0, 0, 0 }
array9[0] = 10;            // Set index 0 value - { 10, 0, 0 }
array9[1] = 20;            // Set index 1 value - { 10, 20, 0 }
array9[2] = 30;            // Set index 2 value - { 10, 20, 30 }
```

Массивы не могут быть повторно инициализированы синтаксисом ярлыка инициализатора массива

Невозможно повторно инициализировать массив с помощью синтаксиса ярлыков с инициализатором массива, поскольку инициализатор массива может быть указан только в объявлении поля или объявлении локальной переменной или как часть выражения создания массива.

Тем не менее, можно создать новый массив и назначить его переменной, используемой для ссылки на старый массив. Хотя это приводит к тому, что массив, на который ссылается эта переменная, повторно инициализируется, содержимое переменной представляет собой совершенно новый массив. Для этого `new` оператор может использоваться с инициализатором массива и назначается переменной массива:

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };

// Prints "1 2 3 ".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6 ".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut
                        // syntax with array initializer.
```

Живой на Ideone

Создание массива из коллекции

Два метода в `java.util.Collection` создают массив из коллекции:

- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

`Object[] toArray()` может использоваться следующим образом:

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

`<T> T[] toArray(T[] a)` можно использовать следующим образом:

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");
```

```
// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

Разница между ними больше, чем просто отсутствие нетипизированных или типизированных результатов. Их производительность может также отличаться (подробности см. В этом разделе анализа эффективности):

- `Object[] toArray()` использует векторизованную `arraycopy` , которая намного быстрее, чем проверенная `arraycopy` используемая в `T[] toArray(T[] a)` .
- `T[] toArray(new T[non-zero-size])` должен обнулить массив во время выполнения, а `T[] toArray(new T[0])` . Такое избегание заставляет последнего звонить быстрее первого. Подробный анализ здесь: Массивы Мудрости Древних .

Java SE 8

Начиная с Java SE 8+, где была введена концепция `Stream` , можно использовать `Stream` созданный сборкой, для создания нового массива с использованием метода `Stream.toArray` .

```
String[] strings = list.stream().toArray(String[]::new);
```

Примеры, взятые из двух ответов (1 , 2) на преобразование 'ArrayList в' String [] 'в Java на переполнение стека.

Массивы для строки

Java SE 5

Начиная с Java 1.5 вы можете получить представление `String` содержимого указанного массива без итерации по каждому элементу. Просто используйте `Arrays.toString(Object[])` или `Arrays.deepToString(Object[])` для многомерных массивов:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr));      // [1, 2, 3, 4, 5]

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr));  // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`Arrays.toString()` использует метод `Object.toString()` для получения значений `String` каждого

элемента в массиве, помимо массива примитивного типа, его можно использовать для всех типов массивов. Например:

```
public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
      return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr));        // [CAT!, CAT!]
```

Если для класса не существует переопределенного `toString()` , тогда будет использоваться унаследованная `toString()` из `Object` . Обычно выход не очень полезен, например:

```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));        // [Dog@17ed40e0]
```

Создание списка из массива

Метод `Arrays.asList()` может использоваться для возврата `List` фиксированного размера, содержащего элементы данного массива. Полученный `List` будет иметь тот же тип параметра, что и базовый тип массива.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Примечание . Этот список поддерживается (*представлением*) исходного массива, что означает, что любые изменения в списке изменят массив и наоборот. Однако изменения в списке, который изменит его размер (и, следовательно, длину массива), вызовут исключение.

Чтобы создать копию списка, используйте конструктор `java.util.ArrayList` взяв `Collection` в качестве аргумента:

Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Java SE 7

В Java SE 7 и более поздних версиях может использоваться пара угловых скобок `<>` (пустой набор аргументов типа), который называется Diamond . Компилятор может определить аргументы типа из контекста. Это означает, что информация о типе может

быть опущена при вызове конструктора `ArrayList` и будет автоматически выведена во время компиляции. Это называется Type Inference, который является частью Java Generics .

```
// Using Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

Точка стоит отметить о Даймонд , что она не может быть использована с классами Anonymous .

Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Важные замечания, связанные с использованием метода Arrays.asList ()

- Этот метод возвращает `List` , который является экземпляром `Arrays$ArrayList` (статический внутренний класс `Arrays`), а не `java.util.ArrayList` . Полученный `List` имеет фиксированный размер. Это означает, что добавление или удаление элементов не поддерживается и будет `UnsupportedOperationException` исключение `UnsupportedOperationException` :

  ```
  stringList.add("something"); // throws java.lang.UnsupportedOperationException
  ```

- Новый `List` можно создать, передав `List` с поддержкой массива в конструктор нового `List` . Это создает новую копию данных, которая имеет изменяемый размер и не поддерживается исходным массивом:

  ```
  List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
  ```

- Вызов `<T> List<T> asList(T... a)` в примитивном массиве, такой как `int[]` , приведет к созданию `List<int[]>` , единственным элементом которого является исходный примитивный массив вместо фактических элементов исходного массива.

  Причиной такого поведения является то, что примитивные типы не могут использоваться вместо параметров типового типа, поэтому весь примитивный массив заменяет параметр типового типа в этом случае. Чтобы преобразовать примитивный массив в `List` , прежде всего, преобразовать примитивный массив в массив соответствующего типа-оболочки (т. `Arrays.asList` в `Integer[]` вместо `int[]`).

  Поэтому это будет напечатать `false` :

  ```
  int[] arr = {1, 2, 3};      // primitive array of int
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  Посмотреть демо

  С другой стороны, это будет `true` :

  ```
  Integer[] arr = {1, 2, 3};  // object array of Integer (wrapper for int)
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  Посмотреть демо

  Это также напечатает `true` , потому что массив будет интерпретироваться как `Integer[]`):

  ```
  System.out.println(Arrays.asList(1,2,3).contains(1));
  ```

  Посмотреть демо

Многомерные и зубчатые массивы

Можно определить массив с более чем одним измерением. Вместо того, чтобы получать доступ, предоставляя единый индекс, доступ к многомерному массиву можно получить, указав индекс для каждого измерения.

Объявление многомерного массива можно сделать, добавив `[]` для каждого измерения к регулярному объявлению массива. Например, чтобы создать двумерный массив `int` , добавьте еще один набор скобок в объявление, например `int[][]` . Это продолжается для 3-мерных массивов (`int[][][]`) и т. Д.

Чтобы определить двумерный массив с тремя строками и тремя столбцами:

```
int rows = 3;
int columns = 3;
```

```
int[][] table = new int[rows][columns];
```

Массив можно индексировать и присваивать ему значения с помощью этой конструкции. Обратите внимание, что неназначенные значения являются значениями по умолчанию для типа массива, в этом случае `0` для `int` .

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

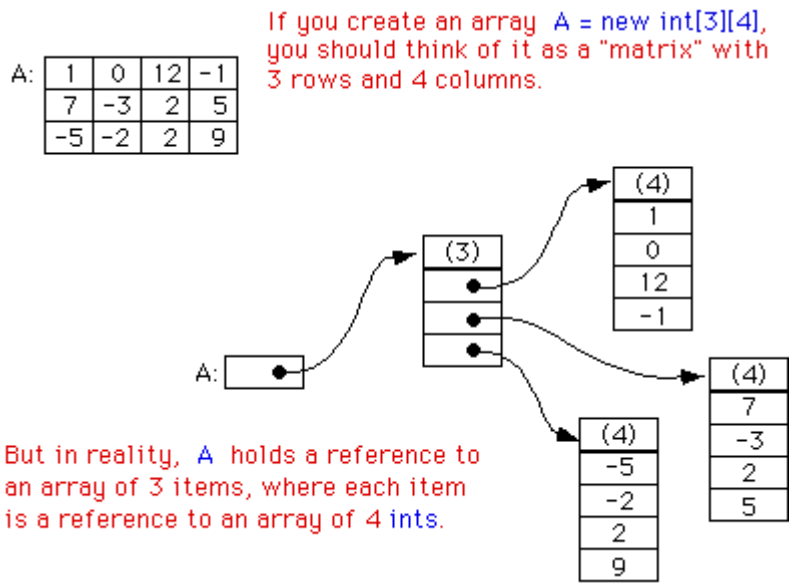
Также возможно одновременное создание измерений и даже создание непрямоугольных массивов. Они чаще всего называются **зубчатыми массивами** .

```
int[][] nonRect = new int[4][];
```

Важно отметить, что, хотя можно определить любой размер массива с зубчатым контуром, ***должен*** быть определен предшествующий уровень.

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

Как многомерные массивы представлены в Java



Источник изображения: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

Литературная инициализация массива Jagged

Многомерные массивы и зубчатые массивы также могут быть инициализированы литеральным выражением. Следующее объявляет и заполняет массив 2x3 `int` :

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Примечание : Subarrays с зубцами также может быть `null` . Например, следующий код объявляет и заполняет двумерный массив `int` чей первый подмассив имеет значение `null` , второй субарейр имеет нулевую длину, третий подмассив имеет одну длину, а последний подмассива представляет собой массив из двух длин:

```
int[][] table = {
    null,
    {},
    {1},
    {1,2}
};
```

Для многомерного массива можно выделить массивы измерения нижнего уровня по их индексам:

```
int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

ArrayIndexOutOfBoundsException

`ArrayIndexOutOfBoundsException` генерируется при доступе к несуществующему индексу массива.

Массивы индексируются с нулевым индексом, поэтому индекс первого элемента равен `0` а индекс последнего элемента - `array.length - 1` массива минус `1` (т. `array.length - 1`).

Поэтому любой запрос элемента массива индексом `i` должен удовлетворять условию `0 <= i < array.length` , иначе будет `0 <= i < array.length` `ArrayIndexOutOfBoundsException` .

Следующий код - простой пример, в котором генерируется `ArrayIndexOutOfBoundsException` .

```
String[] people = new String[] { "Carol", "Andy" };

// An array will be created:
// people[0]: "Carol"
// people[1]: "Andy"
```

```
// Notice: no item on index 2. Trying to access it triggers the exception:
System.out.println(people[2]);  // throws an ArrayIndexOutOfBoundsException.
```

Выход:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at your.package.path.method(YourClass.java:15)
```

Обратите внимание, что незаконный индекс, к которому обращаются, также включен в исключение (`2` в примере); эта информация может быть полезна, чтобы найти причину исключения.

Чтобы этого избежать, просто проверьте, что индекс находится в пределах массива:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Получение длины массива

Массивы - это объекты, которые обеспечивают пространство для хранения до размера элементов указанного типа. Размер массива не может быть изменен после создания массива.

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

Поле `length` в массиве сохраняет размер массива. Это `final` поле и не может быть изменено.

Этот код показывает разницу между `length` массива и количеством объектов, хранящихся в массиве.

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrElt = arr[i];
        if (arrElt != null) {
```

```
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
                           + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

Результат:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Сравнение массивов для равенства

Типы массивов наследуют реализации `equals()` (и `hashCode()`) из java.lang.Object , поэтому `equals()` возвращает true только при сравнении с тем же самым объектом массива. Чтобы сравнить массивы для равенства на основе их значений, используйте `java.util.Arrays.equals` , который перегружен для всех типов массивов.

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); //prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); //prints "true" because the elements of a and b have
the same values
```

Когда тип элемента является ссылочным типом, `Arrays.equals()` вызывает `equals()` для элементов массива для определения равенства. В частности, если тип элемента сам по себе является типом массива, будет использоваться сравнение идентичности. Чтобы сравнить многомерные массивы для равенства, используйте `Arrays.deepEquals()` как `Arrays.deepEquals()` ниже:

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };

Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject));// true
```

Поскольку наборы и карты используют `equals()` и `hashCode()` , массивы обычно не полезны в качестве заданных элементов или клавиш карты. Либо оберните их в класс-помощник, который реализует `equals()` и `hashCode()` в терминах элементов массива или преобразует их в экземпляры `List` и сохраняет списки.

Массивы для потока

Java SE 8

Преобразование массива объектов в `Stream` :

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

Преобразование массива примитивов в `Stream` с использованием `Arrays.stream()` преобразует массив в примитивную специализацию Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

Вы также можете ограничить `Stream` для диапазона элементов в массиве. Начальный индекс является включительным, а конечный индекс - исключительным:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

В классе `Stream` появляется метод, аналогичный `Arrays.stream()` : `Stream.of()` . Разница в том, что `Stream.of()` использует параметр varargs, поэтому вы можете написать что-то вроде:

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

Итерация по массивам

Вы можете перебирать массивы либо с помощью расширенного цикла for (aka foreach), либо с использованием индексов массива:

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

Java SE 5

```
// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

Здесь стоит отметить, что нет прямого способа использовать Iterator в массиве, но через библиотеку Arrays его можно легко преобразовать в список, чтобы получить объект `Iterable` .

Для массивов в коробке используйте Arrays.asList :

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

Для примитивных массивов (с использованием java 8) используйте потоки (в частности, в этом примере - Arrays.stream -> IntStream):

```
int[] primitives = {1, 2, 3};
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

Если вы не можете использовать потоки (без java 8), вы можете использовать goava-библиотеку guava :

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

В двумерных массивах или более, обе методики могут быть использованы несколько более сложным образом.

Пример:

```
int[][] array = new int[10][10];

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

Невозможно установить Array на любое неравномерное значение без использования цикла, основанного на индексе.

Конечно, вы также можете использовать `while` или `do-while` циклы при итерации с использованием индексов.

Одно примечание: при использовании индексов массива убедитесь, что индекс находится между `0` и `array.length - 1` (оба `array.length - 1`). Не делайте жестко закодированные предположения относительно длины массива, иначе вы можете сломать свой код, если длина массива изменится, но ваши жестко закодированные значения не будут.

Пример:

```
int[] numbers = {1, 2, 3, 4};

public void incrementNumbers() {
    // DO THIS :
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] += 1; //or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }

    // DON'T DO THIS :
    for (int i = 0; i < 4; i++) {
        numbers[i] += 1;
    }
}
```

Это также лучше, если вы не используете причудливые вычисления для получения индекса, но используете индекс для итерации, и если вам нужны разные значения, подсчитайте их.

Пример:

```
public void fillArrayWithDoubleIndex(int[] array) {
    // DO THIS :
    for (int i = 0; i < array.length; i++) {
        array[i] = i * 2;
    }

    // DON'T DO THIS :
    int doubleLength = array.length * 2;
    for (int i = 0; i < doubleLength; i += 2) {
        array[i / 2] = i;
    }
}
```

Доступ к массивам в обратном порядке

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
}
```

Использование временных массивов для уменьшения повторения кода

Итерация по временному массиву вместо повторения кода может сделать ваш код более чистым. Он может использоваться, когда одна и та же операция выполняется для нескольких переменных.

```
// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
```

```
int arms = 5;


// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);


// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));
```

Имейте в виду, что этот код не должен использоваться в критически важных средах, поскольку массив создается каждый раз, когда вводится цикл, и что примитивные переменные будут скопированы в массив и, следовательно, не могут быть изменены.

Копирование массивов

Java предоставляет несколько способов копирования массива.

для цикла

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

Обратите внимание, что использование этой опции с массивом Object вместо примитивного массива будет заполнять копию ссылкой на исходное содержимое вместо его копии.

Object.clone ()

Поскольку массивы `Object` в Java, вы можете использовать `Object.clone()` .

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

Обратите внимание, что метод `Object.clone` для массива выполняет **мелкую копию** , то есть возвращает ссылку на новый массив, который ссылается на те **же** элементы, что и исходный массив.

Arrays.copyOf ()

`java.util.Arrays` предоставляет простой способ выполнить копию массива в другой. Вот основное использование:

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Обратите внимание, что `Arrays.copyOf` также обеспечивает перегрузку, которая позволяет вам изменять тип массива:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

System.arraycopy ()

> `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` Копирует массив из указанного исходного массива, начиная с указанной позиции, в указанную позицию целевого массива.

Ниже приведен пример использования

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

Arrays.copyOfRange ()

В основном используется для копирования части массива, вы также можете использовать его для копирования целого массива в другой, как показано ниже:

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

Литейные массивы

Массивы - это объекты, но их тип определяется типом содержащихся объектов. Поэтому нельзя просто отбрасывать `A[]` в `T[]` , но каждый член `A[]` конкретного `A[]` должен быть переведен в объект `T` Общий пример:

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
```

```
        target[i] = (T) array[i];
    }
    return target;
}
```

Таким образом, с учетом массива `A[]` :

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE предоставляет для этой цели метод `Arrays.copyOf(original, newLength, newType)` :

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

Удалить элемент из массива

Java не предоставляет прямой метод в `java.util.Arrays` для удаления элемента из массива. Для его выполнения вы можете скопировать исходный массив в новый, не удаляя элемент или преобразовывая его в другую структуру, позволяющую удалить.

Использование ArrayList

Вы можете преобразовать массив в `java.util.List` , удалить элемент и преобразовать список обратно в массив следующим образом:

```
String[] array = new String[]{"foo", "bar", "baz"};

List<String> list = new ArrayList<>(Arrays.asList(array));
list.remove("foo");

// Creates a new array with the same size as the list and copies the list
// elements to it.
array = list.toArray(new String[list.size()]);

System.out.println(Arrays.toString(array)); //[bar, baz]
```

Использование System.arraycopy

`System.arraycopy()` можно использовать для создания копии исходного массива и удаления `System.arraycopy()` элемента. Ниже пример:

```
int[] array = new int[] { 1, 2, 3, 4 }; // Original array.
int[] result = new int[array.length - 1]; // Array which will contain the result.
int index = 1; // Remove the value "2".

// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
```

```
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); //[1, 3, 4]
```

Использование Apache Commons Lang

Чтобы легко удалить элемент, вы можете использовать библиотеку Apache Commons Lang и особенно статический метод `removeElement()` класса `ArrayUtils` . Ниже пример:

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

Ковариантность массива

Объектные массивы являются ковариантными, что означает, что так же, как `Integer` является подклассом `Number` , `Integer[]` является подклассом `Number[]` . Это может показаться интуитивным, но может привести к неожиданному поведению:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray;  // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L;                  // throws ArrayStoreException at runtime
```

Хотя `Integer[]` является подклассом `Number[]` , он может содержать только `Integer` s, и попытка назначить элемент `Long` выдает исключение во время выполнения.

Обратите внимание, что это поведение уникально для массивов, и его можно избежать, используя вместо этого общий `List` :

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList;  // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L);                  // compile error
```

Для всех элементов массива не обязательно использовать один и тот же тип, если они являются подклассом типа массива:

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
                                                     // operator and array initializer.
```

```
I[] array11 = { new A(), new B(), new C() };          // Shortcut syntax with array
                                                      // initializer.

I[] array12 = new I[3];                               // { null, null, null }

I[] array13 = new A[] { new A(), new A() };           // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
                                                      // new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" };      // Shortcut syntax
                                                      // with array initializer.
```

Как изменить размер массива?

Простой ответ заключается в том, что вы не можете этого сделать. После создания массива его размер не может быть изменен. Вместо этого массив может быть только «изменен», создав новый массив с соответствующим размером и скопировав элементы из существующего массива в новый.

```
String[] listOfCities = new String[3];   // array created with size 3.
listOfCities[0] = "New York";
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Предположим (например), что новый элемент необходимо добавить в массив `listOfCities` как указано выше. Для этого вам необходимо:

1. создайте новый массив размером 4,
2. скопируйте существующие 3 элемента старого массива в новый массив при смещениях 0, 1 и 2 и
3. добавьте новый элемент в новый массив со смещением 3.

Существуют различные способы сделать это. До появления Java 6 наиболее кратким способом было:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

Начиная с Java 6 методы `Arrays.copyOf` и `Arrays.copyOfRange` могут сделать это проще:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

Для других способов копирования массива см. Следующий пример. Имейте в виду, что при изменении размера вам понадобится копия массива с другой длиной до оригинала.

- Копирование массивов

Лучшие альтернативы изменению размера массива

Существуют два основных недостатка с изменением размера массива, как описано выше:

- Это неэффективно. Создание массива больше (или меньше) включает в себя копирование многих или всех существующих элементов массива и выделение нового объекта массива. Чем больше массив, тем он дороже.
- Вы должны иметь возможность обновлять любые «живые» переменные, содержащие ссылки на старый массив.

Один из вариантов - создать массив с достаточно большим размером для начала. Это возможно только в том случае, если вы можете точно определить этот размер *до выделения массива* . Если вы не можете этого сделать, возникает проблема изменения размера массива.

Другой альтернативой является использование класса структуры данных, предоставляемого библиотекой классов Java SE или сторонней библиотекой. Например, структура «коллекций» Java SE предоставляет ряд реализаций API-интерфейсов `List` , `Set` и `Map` с различными свойствами среды выполнения. Класс `ArrayList` ближе всего к характеристикам производительности простого массива (например, O (N) lookup, O (1) get и set, O (N) случайная вставка и удаление), обеспечивая при этом более эффективное изменение размера без проблемы с эталонным обновлением.

(Эффективность изменения размера для `ArrayList` исходит из стратегии удвоения размера массива поддержки при каждом изменении размера. Для типичного варианта использования это означает, что вы иногда изменяете размер. Когда вы амортизируете за весь срок службы списка, стоимость изменения размера для каждой вставки `O(1)` . При изменении размера простого массива может быть использована одна и та же стратегия.)

Поиск элемента в массиве

Существует множество способов найти местоположение значения в массиве. В приведенных ниже примерах показано, что массив является одним из следующих:

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

Кроме того, каждый устанавливает `index` или `index2` либо индексу требуемого элемента, либо `-1` если элемент отсутствует.

Использование `Arrays.binarySearch` (только для отсортированных массивов)

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Использование массива `Arrays.asList` (только для не примитивных массивов)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1);  // compilation error
```

Использование `Stream`

Java SE 8

```
int index = IntStream.range(0, strings.length)
                     .filter(i -> "A".equals(strings[i]))
                     .findFirst()
                     .orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

Линейный поиск с использованием цикла

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

Линейный поиск с использованием сторонних библиотек, таких как org.apache.commons

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Примечание. Использование прямого линейного поиска более эффективно, чем перенос в список.

Тестирование, если массив содержит элемент

Вышеприведенные примеры могут быть адаптированы для проверки того, содержит ли массив элемент, просто проверяя, вычисляется ли вычисляемый индекс больше или равен нулю.

Кроме того, есть также несколько более сжатых вариантов:

```
boolean isPresent = Arrays.asList(strings).contains("A");
```

Java SE 8

```
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
```

```
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}

boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

Сортировка массивов

Сортировка массивов можно легко сделать с Массивы апи.

```
import java.util.Arrays;

// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));
```

Сортировка массивов строк:

`String` - это не числовые данные, она определяет ее собственный порядок, который называется лексикографическим порядком, также известным как алфавитный порядок. Когда вы сортируете массив String с помощью метода `sort()` , он сортирует массив в естественный порядок, определенный интерфейсом Comparable, как показано ниже:

Увеличение порядка

```
String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("String array before sorting : " + Arrays.toString(names));
Arrays.sort(names);
System.out.println("String array after sorting in ascending order : " +
Arrays.toString(names));
```

Выход:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

Уменьшение порядка

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " +
Arrays.toString(names));
```

Выход:

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

Сортировка массива объектов

Чтобы отсортировать массив объектов, все элементы должны реализовать интерфейс `Comparable` или `Comparator` для определения порядка сортировки.

Мы можем использовать любой метод `sort(Object[])` для сортировки массива объектов в его естественном порядке, но вы должны убедиться, что все элементы в массиве должны реализовать `Comparable` .

Кроме того, они должны быть взаимно сопоставимыми, например, `e1.compareTo(e2)` не должен бросать *`ClassCastException`* для любых элементов e1 и e2 в массиве. В качестве альтернативы вы можете отсортировать массив объектов в пользовательском порядке, используя метод `sort(T[], Comparator)` как показано в следующем примере.

```
// How to Sort Object Array in Java using Comparator and Comparable
Course[] courses = new Course[4];
courses[0] = new Course(101, "Java", 200);
courses[1] = new Course(201, "Ruby", 300);
courses[2] = new Course(301, "Python", 400);
courses[3] = new Course(401, "Scala", 500);

System.out.println("Object array before sorting : " + Arrays.toString(courses));

Arrays.sort(courses);
System.out.println("Object array after sorting in natural order : " +
Arrays.toString(courses));

Arrays.sort(courses, new Course.PriceComparator());
System.out.println("Object array after sorting by price : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.NameComparator());
System.out.println("Object array after sorting by name : " + Arrays.toString(courses));
```

Выход:

```
Object array before sorting : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting in natural order : [#101 Java@200 , #201 Ruby@300 , #301 Python@400
, #401 Scala@500 ]
Object array after sorting by price : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
```

```
Object array after sorting by name : [#101 Java@200 , #301 Python@400 , #201 Ruby@300 , #401
Scala@500 ]
```

Преобразование массивов между примитивами и коробочными типами

Иногда требуется преобразование примитивных типов в бокс- типы.

Для преобразования массива можно использовать потоки (в Java 8 и выше):

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

С более низкими версиями это может быть путем итерации примитивного массива и явного копирования его в массив в штучной упаковке:

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

Точно так же массив в коробке может быть преобразован в массив его примитивной копии:

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray =
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray = new int[boxedArray.length];
for (int i = 0; i < boxedArray.length; ++i) {
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here
}
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/java/topic/99/массивы

глава 106: Менеджер по безопасности

Examples

Включение SecurityManager

Виртуальные машины Java (JVM) могут быть запущены с установленным SecurityManager. SecurityManager управляет тем, что разрешен коду, запущенному в JVM, на основе таких факторов, как загрузка кода и какие сертификаты были использованы для подписи кода.

SecurityManager можно установить, установив свойство системы java.security.manager в командной строке при запуске JVM:

```
java -Djava.security.manager <main class name>
```

или программно из кода Java:

```
System.setSecurityManager(new SecurityManager())
```

Стандартный Java SecurityManager предоставляет разрешения на основе политики, которая определена в файле политики. Если файл политики не указан, будет использоваться файл политики по умолчанию в разделе `$JAVA_HOME/lib/security/java.policy` .

Класс Sandboxing, загружаемый ClassLoader

ClassLoader должен предоставить `ProtectionDomain` идентифицирующий источник кода:

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;


    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }


    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

Переопределяя `findClass` а не `loadClass` `findClass` модель сохраняется, и PluginClassLoader

сначала запрашивает систему и родительский загрузчик классов для определений классов.

Создание политики:

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // amend this as appropriate
        appPermissions.add(new AllPermission());
        // add any permissions plugins should have to pluginPermissions
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return new Permissions();
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return isPlugin(domain)?pluginPermissions:appPermissions;
    }

    private boolean isPlugin(ProtectionDomain pd){
        return pd.getClassLoader() instanceof PluginClassLoader;
    }

}
```

Наконец, установите политику и SecurityManager (реализация по умолчанию выполнена нормально):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

Внедрение правил отклонения политики

Иногда желательно *отклонить* определенное `Permission` на какой-либо `ProtectionDomain` , *независимо* от каких-либо других разрешений, которые начисляются доменом. Этот пример демонстрирует только один из возможных подходов к удовлетворению такого рода

требований. Он вводит «отрицательный» класс разрешений вместе с оболочкой, которая позволяет использовать `Policy` по умолчанию в качестве репозитория таких разрешений.

Заметки:

- Синтаксис стандартного файла политики и механизм назначения разрешений вообще не затрагиваются. Это означает, что правила *запрета* в файлах политики все еще выражаются в виде *грантов* .
- Полицейская оболочка предназначена для инкапсуляции `Policy` по умолчанию, поддерживаемой файлами (предполагается, что это `com.sun.security.provider.PolicyFile`).
- Отклоненные разрешения обрабатываются только на уровне политики. Если они статически назначены для домена, они по умолчанию будут обрабатываться этим доменом как обычные «положительные» разрешения.

Класс `DeniedPermission`

```
package com.example;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Modifier;
import java.security.BasicPermission;
import java.security.Permission;
import java.security.UnresolvedPermission;
import java.text.MessageFormat;

/**
 * A representation of a "negative" privilege.
 * <p>
 * A <code>DeniedPermission</code>, when "granted" (to some <code>ProtectionDomain</code>
and/or
 * <code>Principal</code>), represents a privilege which <em>cannot</em> be exercised,
regardless of
 * any positive permissions (<code>AllPermission</code> included) possessed. In other words,
if a
 * set of granted permissions, <em>P</em>, contains a permission of this class, <em>D</em>,
then the
 * set of effectively granted permissions is<br/>
 * <br/>
 *     <em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * Each instance of this class encapsulates a <em>target permission</em>, representing the
 * "positive" permission being denied.
 * </p>
 * Denied permissions employ the following naming scheme:<br/>
 * <br/>
 *
    <em>&lt;target_class_name&gt;:&lt;target_name&gt;(:&lt;target_actions&gt;)</em>

 * <br/>
 * where:
 * <ul>
 * <li><em>target_class_name</em> is the name of the target permission's class,</li>
```

```
 * <li><em>target_name</em> is the name of the target permission, and</li>
 * <li><em>target_actions</em> is, optionally, the actions string of the target
permission.</li>
 * </ul>
 * A denied permission, having a target permission <em>t</em>, is said to <em>imply</em>
another
 * permission <em>p</em>, if:
 * <ul>
 * <li>p <em>is not</em> itself a denied permission, and <code>(t.implies(p) == true)</code>,
 * or</li>
 * <li>p <em>is</em> a denied permission, with a target <em>t1</em>, and
 * <code>(t.implies(t1) == true)</code>.
 * </ul>
 * <p>
 * It is the responsibility of the policy decision point (e.g., the <code>Policy</code>
provider) to
 * take denied permission semantics into account when issuing authorization statements.
 * </p>
 */
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 47362516386980067 9L;

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of
the
     * indicated class, specified name and, optionally, actions.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>targetClassName</code> is <code>null</code>, the empty string,
does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>targetName</code> is <code>null</code>.</li>
     *             <li><code>targetClassName</code> cannot be instantiated, and it's the
caller's fault;
     *             e.g., because <code>targetName</code> and/or <code>targetActions</code> do
not adhere
     *             to the naming constraints of the target class; or due to the target class
not
     *             exposing a <code>(String name)</code>, or <code>(String name, String
actions)</code>
     *             constructor, depending on whether <code>targetActions</code> is
<code>null</code> or
     *             not.</li>
     *             </ul>
     */
    public static DeniedPermission newDeniedPermission(String targetClassName, String
targetName,
            String targetActions) {
        if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null)
{
            throw new IllegalArgumentException(
                    "Null or empty [targetClassName], or null [targetName] argument was
supplied.");
        }
        StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
```

```
        if (targetName != null) {
            sb.append(":").append(targetName);
        }
        return new DeniedPermission(sb.toString());
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of
the class,
     * name and, optionally, actions, collectively provided as the <code>name</code> argument.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>name</code>'s target permission class name component is empty,
does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>name</code>'s target name component is <code>empty</code></li>
     *             <li>the target permission class cannot be instantiated, and it's the
caller's fault;
     *             e.g., because <code>name</code>'s target name and/or target actions
component(s) do
     *             not adhere to the naming constraints of the target class; or due to the
target class
     *             not exposing a <code>(String name)</code>, or
     *             <code>(String name, String actions)</code> constructor, depending on
whether the
     *             target actions component is empty or not.</li>
     *             </ul>
     */
    public DeniedPermission(String name) {
        super(name);
        String[] comps = name.split(":");
        if (comps.length < 2) {
            throw new IllegalArgumentException(MessageFormat.format("Malformed name [{0}]
argument.", name));
        }
        this.target = initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates the given target
permission.
     *
     * @throws IllegalArgumentException
     *             if <code>target</code> is <code>null</code>, a
<code>DeniedPermission</code>, or an
     *             <code>UnresolvedPermission</code>.
     */
    public static DeniedPermission newDeniedPermission(Permission target) {
        if (target == null) {
            throw new IllegalArgumentException("Null [target] argument.");
        }
        if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
            throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
        }
        StringBuilder sb = new
StringBuilder(target.getClass().getName()).append(":").append(target.getName());
```

```
        String targetActions = target.getActions();
        if (targetActions != null) {
            sb.append(":").append(targetActions);
        }
        return new DeniedPermission(sb.toString(), target);
    }

    private DeniedPermission(String name, Permission target) {
        super(name);
        this.target = target;
    }

    private Permission initTarget(String targetClassName, String targetName, String
targetActions) {
        Class<?> targetClass;
        try {
            targetClass = Class.forName(targetClassName);
        }
        catch (ClassNotFoundException cnfe) {
            if (targetClassName.trim().isEmpty()) {
                targetClassName = "<empty>";
            }
            throw new IllegalArgumentException(
                    MessageFormat.format("Target Permission class [{0}] not found.",
targetClassName));
        }
        if (!Permission.class.isAssignableFrom(targetClass) ||
Modifier.isAbstract(targetClass.getModifiers())) {
            throw new IllegalArgumentException(MessageFormat
                    .format("Target Permission class [{0}] is not a (concrete) Permission.",
targetClassName));
        }
        if (targetClass == DeniedPermission.class || targetClass ==
UnresolvedPermission.class) {
            throw new IllegalArgumentException("Target Permission class cannot be a
DeniedPermission itself.");
        }
        Constructor<?> targetCtor;
        try {
            if (targetActions == null) {
                targetCtor = targetClass.getConstructor(String.class);
            }
            else {
                targetCtor = targetClass.getConstructor(String.class, String.class);
            }
        }
        catch (NoSuchMethodException nsme) {
            throw new IllegalArgumentException(MessageFormat.format(
                    "Target Permission class [{0}] does not provide or expose a (String name)
or (String name, String actions) constructor.",
                    targetClassName));
        }
        try {
            return (Permission) targetCtor
                    .newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] {
targetName }
                            : new Object[] { targetName, targetActions }));
        }
        catch (ReflectiveOperationException roe) {
            if (roe instanceof InvocationTargetException) {
                if (targetName == null) {
```

```
                    targetName = "<null>";
                }
                else if (targetName.trim().isEmpty()) {
                    targetName = "<empty>";
                }
                if (targetActions == null) {
                    targetActions = "<null>";
                }
                else if (targetActions.trim().isEmpty()) {
                    targetActions = "<empty>";
                }
                throw new IllegalArgumentException(MessageFormat.format(
                        "Could not instantiate target Permission class [{0}]; provided target
name [{1}] and/or target actions [{2}] potentially erroneous.",
                        targetClassName, targetName, targetActions), roe);
            }
            throw new RuntimeException(
                    "Could not instantiate target Permission class [{0}]; an unforeseen error
occurred - see attached cause for details",
                    roe);
        }
    }

    /**
     * Checks whether the given permission is implied by this one, as per the {@link
DeniedPermission
     * overview}.
     */
    @Override
    public boolean implies(Permission p) {
        if (p instanceof DeniedPermission) {
            return target.implies(((DeniedPermission) p).target);
        }
        return target.implies(p);
    }

    /**
     * Returns this denied permission's target permission (the actual positive permission
which is not
     * to be granted).
     */
    public Permission getTargetPermission() {
        return target;
    }

}
```

Класс `DenyingPolicy`

```
package com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;
```

```
/**
 * Wrapper that adds rudimentary {@link DeniedPermission} processing capabilities to the
standard
 * file-backed <code>Policy</code>.
 */
public final class DenyingPolicy extends Policy {

    {
        try {
            defaultPolicy = Policy.getInstance("javaPolicy", null);
        }
        catch (NoSuchAlgorithmException nsae) {
            throw new RuntimeException("Could not acquire default Policy.", nsae);
        }
    }

    private final Policy defaultPolicy;

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return defaultPolicy.getPermissions(codesource);
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return defaultPolicy.getPermissions(domain);
    }

    /**
     * @return
     *         <ul>
     *         <li><code>true</code> if:</li>
     *         <ul>
     *         <li><code>permission</code> <em>is not</em> an instance of
     *         <code>DeniedPermission</code>,</li>
     *         <li>an <code>implies(domain, permission)</code> invocation on the system-
default
     *         <code>Policy</code> yields <code>true</code>, and</li>
     *         <li><code>permission</code> <em>is not</em> implied by any
<code>DeniedPermission</code>s
     *         having potentially been assigned to <code>domain</code>.</li>
     *         </ul>
     *         <li><code>false</code>, otherwise.
     *         </ul>
     */
    @Override
    public boolean implies(ProtectionDomain domain, Permission permission) {
        if (permission instanceof DeniedPermission) {
            /*
             * At the policy decision level, DeniedPermissions can only themselves imply, not
be implied (as
             * they take away, rather than grant, privileges). Furthermore, clients aren't
supposed to use this
             * method for checking whether some domain _does not_ have a permission (which is
what
             * DeniedPermissions express after all).
             */
            return false;
        }
```

```
        if (!defaultPolicy.implies(domain, permission)) {
            // permission not granted, so no need to check whether denied
            return false;
        }

        /*
         * Permission granted--now check whether there's an overriding DeniedPermission. The
following
         * assumes that previousPolicy is a sun.security.provider.PolicyFile (different
implementations
         * might not support #getPermissions(ProtectionDomain) and/or handle
UnresolvedPermissions
         * differently).
         */

        Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
        while (perms.hasMoreElements()) {
            Permission p = perms.nextElement();
            /*
             * DeniedPermissions will generally remain unresolved, as no code is expected to
check whether other
             * code has been "granted" such a permission.
             */
            if (p instanceof UnresolvedPermission) {
                UnresolvedPermission up = (UnresolvedPermission) p;
                if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
                    // force resolution
                    defaultPolicy.implies(domain, up);
                    // evaluate right away, to avoid reiterating over the collection
                    p = new DeniedPermission(up.getUnresolvedName());
                }
            }
            if (p instanceof DeniedPermission && p.implies(permission)) {
                // permission denied
                return false;
            }
        }
        // permission granted
        return true;
    }

    @Override
    public void refresh() {
        defaultPolicy.refresh();
    }

}
```

демонстрация

```
package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
        System.setSecurityManager(new SecurityManager());
```

```
        // should fail
        System.getProperty("foo.bar");
    }

}
```

Назначьте некоторые разрешения:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

И наконец, запустите `Main` и наблюдайте, как он терпит неудачу, из-за правила «deny» (`DeniedPermission`), переопределяющего `grant` (его `PropertyPermission`). Обратите внимание, что `setProperty("foo.baz", "xyz")` вместо этого преуспел, поскольку разрешенное разрешение распространяется только на действие «читать» и исключительно для свойства «foo.bar».

Прочитайте Менеджер по безопасности онлайн: https://riptutorial.com/ru/java/topic/5712/менеджер-по-безопасности

глава 107: Местное время

Синтаксис

- LocalTime time = LocalTime.now (); // Инициализирует текущие системные часы
- LocalTime time = LocalTime.MIDNIGHT; // 00:00
- LocalTime time = LocalTime.NOON; // 12:00
- LocalTime time = LocalTime.of (12, 12, 45); // 12:12:45

параметры

метод	Выход
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	12:00
`LocalTime.now()`	Текущее время от системных часов
`LocalTime.MAX`	Максимальное поддерживаемое местное время 23: 59: 59.999999999
`LocalTime.MIN`	Минимальное поддерживаемое местное время 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59, Получает Время от значения второго дня
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02, Получает Время от значения нано-дня

замечания

Как и имя класса, `LocalTime` представляет собой время без временной зоны. Он не представляет дату. Это простая метка для данного времени.

Класс основан на значении, и при выполнении сравнений должен использоваться метод `equals` .

Этот класс из пакета java.time.

Examples

Модификация времени

Вы можете добавить часы, минуты, секунды и наносекунды:

```
LocalTime time = LocalTime.now();
LocalTime addHours = time.plusHours(5); // Add 5 hours
LocaLTime addMinutes = time.plusMinutes(15) // Add 15 minutes
LocalTime addSeconds = time.plusSeconds(30) // Add 30 seconds
LocalTime addNanoseconds = time.plusNanos(150_000_000) // Add 150.000.000ns (150ms)
```

Временные зоны и их разность во времени

```
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
            ZoneId zone1 = ZoneId.of("Europe/Berlin");
            ZoneId zone2 = ZoneId.of("Brazil/East");

            LocalTime now = LocalTime.now();
            LocalTime now1 = LocalTime.now(zone1);
            LocalTime now2 = LocalTime.now(zone2);

            System.out.println("Current Time : " + now);
            System.out.println("Berlin Time : " + now1);
            System.out.println("Brazil Time : " + now2);

            long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
            System.out.println("Minutes Between Berlin and Brazil : " + minutesBetween
+"mins");
    }
}
```

Количество времени между двумя LocalTime

Существует два эквивалентных способа расчета суммы единицы времени между двумя `LocalTime` : (1) `until(Temporal, TemporalUnit)` и через (2) `TemporalUnit.between(Temporal, Temporal)` .

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // hour, minute, second
        LocalTime end = LocalTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
```

```
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millisecs1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millisecs2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 4220000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 4220000000000

        // Using others ChronoUnit will be thrown UnsupportedTemporalTypeException.
        // The following methods are examples thereof.
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

вступление

LocalTime - неизменный класс и поточно-безопасный, используемый для представления времени, часто рассматриваемого как hour-min-sec. Время представлено до наносекундной точности. Например, значение «13: 45.30.123456789» может быть сохранено в LocalTime.

Этот класс не хранит или не представляет дату или часовую зону. Вместо этого это описание местного времени, которое видно на настенных часах. Он не может представлять мгновение на временной линии без дополнительной информации, такой как смещение или временная зона. Это класс, основанный на значении, метод equals должен использоваться для сравнения.

поля

MAX - Максимальное поддерживаемое LocalTime, '23: 59: 59.999999999 '. MIDNIGHT, MIN, NOON

Важные статические методы

now (), now (Clock clock), теперь (ZoneId zone), parse (текст CharSequence)

Важные методы

isAfter (LocalTime other), isBefore (LocalTime other), минус (TemporalAmount amountToSubtract), минус (longToSubtract, TemporalUnit unit), плюс (TemporalAmount amountToAdd), плюс (long amountToAdd, TemporalUnit unit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

Разницу во времени можно рассчитать любым из следующих способов:

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

Вы также можете добавлять / вычитать часы, минуты или секунды из любого объекта LocalTime.

minusHours (long hoursToSubtract), минус минуты (long hoursToMinutes), минусNanos (long nanosToSubtract), минус секунды (длинные секундыToSubtract), plusHours (long hoursToSubtract), плюсMinutes (long hoursToMinutes), плюсNanos (long nanosToSubtract), плюсSeconds (long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Прочитайте Местное время онлайн: https://riptutorial.com/ru/java/topic/3065/местное-время

глава 108: Местный внутренний класс

Вступление

Класс, т. Е. Созданный внутри метода, называется локальным внутренним классом в java. Если вы хотите вызвать методы локального внутреннего класса, вы должны создать экземпляр этого класса внутри метода.

Examples

Местный внутренний класс

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Прочитайте Местный внутренний класс онлайн: https://riptutorial.com/ru/java/topic/10160/местный-внутренний-класс

глава 109: Методы и конструкторы классов объектов

Вступление

На этой странице документации приведены сведения о конструкторах классов java и методах класса объектов, которые автоматически унаследованы от `Object` суперкласса любого вновь созданного класса.

Синтаксис

- public final native Class <?> getClass ()
- public final native void notify ()
- public final native void notifyAll ()
- public final native void wait (long timeout) throws InterruptedException
- public final void wait () throws InterruptedException
- public final void wait (long timeout, int nanos) бросает InterruptedException
- public native int hashCode ()
- public boolean equals (Object obj)
- public String toString ()
- защищенный native Object clone () бросает CloneNotSupportedException
- protected void finalize () throws Throwable

Examples

Метод toString ()

`toString()` метод используется для создания `String` представления объекта с помощью object's контента. Этот метод следует переопределить при написании вашего класса. `toString()` называется неявно, когда объект конкатенируется с строкой, как в `"hello " + anObject` .

Рассмотрим следующее:

```
public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
```

```
    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        User user = new User("John", "Doe");
        System.out.println(user.toString()); // Prints "John Doe"
    }
}
```

Здесь `toString()` из класса `Object` переопределяется в классе `User` для предоставления значимых данных об объекте при его печати.

При использовании функции `println()` `toString()` метод `toString()` объекта неявно называется. Поэтому эти утверждения делают то же самое:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

Если `toString()` не переопределяется в вышеупомянутом классе `User` , `System.out.println(user)` может возвращать `User@659e0bfd` или аналогичную строку без почти полезной информации, кроме имени класса. Это будет связано с тем, что вызов будет использовать реализацию `toString()` базового класса Java `Object` который ничего не знает о структуре или бизнес-правилах класса `User` . Если вы хотите изменить эту функциональность в своем классе, просто переопределите метод.

метод equals ()

TL; DR

`==` тесты для ссылочного равенства (являются ли они ***одним и*** тем ***же объектом***)

`.equals()` для равенства значений (независимо от того, являются ли они ***логически «равными»***)

`equals()` - метод, используемый для сравнения двух объектов для равенства. По умолчанию реализация метода `equals()` в классе `Object` возвращает `true` тогда и только тогда, когда обе ссылки указывают на один и тот же экземпляр. Поэтому он ведет себя так же, как и сравнение `==` .

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }
```

```
    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints false
    }
}
```

Хотя `foo1` и `foo2` создаются с одинаковыми полями, они указывают на два разных объекта в памяти. Таким образом, реализация `equals()` умолчанию вычисляет значение `false` .

Чтобы сравнить содержимое объекта для равенства, `equals()` должно быть переопределено.

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3));
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + this.field1;
        hash = 31 * hash + this.field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints true
    }
}
```

Здесь переопределенный метод `equals()` решает, что объекты равны, если их поля

одинаковы.

Обратите внимание, что метод `hashCode()` также был перезаписан. В договоре для этого метода указано, что, когда два объекта равны, их хэш-значения также должны быть одинаковыми. Вот почему нужно почти всегда переопределять `hashCode()` и `equals()` вместе.

Обратите особое внимание на тип аргумента метода `equals` . Это `Object obj` , а не `Foo obj` . Если вы поместите последнее в свой метод, это не является переопределением метода `equals` .

При написании собственного класса вам придется писать аналогичную логику при переопределении `equals()` и `hashCode()` . Большинство IDE могут автоматически генерировать это для вас.

Пример реализации `equals()` можно найти в классе `String` , который является частью основного Java API. Вместо сравнения указателей класс `String` сравнивает содержимое `String` .

Java SE 7

В Java 1.7 представлен класс `java.util.Objects` который обеспечивает метод удобства, `equals` , который сравнивает две потенциально `null` ссылки, поэтому его можно использовать для упрощения реализации метода `equals` .

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}
```

Сравнение классов

Поскольку метод equals может работать против любого объекта, одна из первых вещей, которую метод часто делает (после проверки на `null`), - это проверить, соответствует ли класс сравниваемого объекта текущему классу.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
```

```
}
```

Обычно это делается, как указано выше, путем сравнения объектов класса. Тем не менее, это может потерпеть неудачу в нескольких особых случаях, которые могут быть не очевидны. Например, некоторые фреймворки генерируют динамические прокси классов, и эти динамические прокси-серверы фактически являются другим классом. Вот пример использования JPA.

```
Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //Can never get here if equality is tested with getClass()
    //as mergedInstance is a proxy (subclass) of Foo
}
```

Один из механизмов, позволяющих обойти это ограничение, заключается в сравнении классов с использованием `instanceof`

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

Тем не менее, есть несколько подводных камней, которых следует избегать при использовании `instanceof` . Поскольку Foo потенциально может иметь другие подклассы, и эти подклассы могут переопределять `equals()` вы можете попасть в случай, когда `Foo` равен `FooSubclass` но `FooSubclass` не равен `Foo` .

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

Это нарушает свойства симметрии и транзитивности и, таким образом, является недопустимой реализацией метода `equals()` . В результате при использовании `instanceof` хорошая практика заключается в том, чтобы сделать метод `equals() final` (как в приведенном выше примере). Это гарантирует, что никакие переопределения подклассов не `equals()` и нарушают ключевые допущения.

метод hashCode ()

Когда класс Java переопределяет метод `equals` , он также должен переопределять метод `hashCode` . Как определено в контракте метода :

- Всякий раз, когда он вызывается на одном и том же объекте более одного

раза во время выполнения приложения Java, метод `hashCode` должен последовательно возвращать одно и то же целое число, если информация, используемая при равных сравнениях на объекте, не изменяется. Это целое число не должно оставаться согласованным с одним исполнением приложения на другое выполнение одного и того же приложения.
- Если два объекта равны в соответствии с методом `equals(Object)` , то вызов метода `hashCode` для каждого из двух объектов должен давать одинаковый целочисленный результат.
- Не требуется, чтобы, если два объекта неравны в соответствии с методом `equals(Object)` , то вызов метода `hashCode` для каждого из двух объектов должен производить различные целочисленные результаты. Тем не менее, программист должен знать, что получение отдельных целых результатов для неравных объектов может улучшить производительность хеш-таблиц.

Хэш коды используются в хэш-реализации, такие как `HashMap` , `HashTable` и `HashSet` . Результат функции `hashCode` определяет ведро, в которое будет помещен объект. Эти хэш-реализации более эффективны, если реализована реализация `hashCode` . Важным свойством хорошей `hashCode` является то, что распределение значений `hashCode` является однородным. Другими словами, существует небольшая вероятность того, что многочисленные экземпляры будут сохранены в одном ковше.

Алгоритм вычисления значения хэш-кода может быть аналогичен следующему:

```
public class Foo {
    private int field1, field2;
    private String field3;

    public Foo(int field1, int field2, String field3) {
        this.field1 = field1;
        this.field2 = field2;
        this.field3 = field3;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3);
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + field1;
```

```
        hash = 31 * hash + field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }
}
```

Использование Arrays.hashCode () в виде короткой вырезки

Java SE 1.2

В Java 1.2 и выше вместо разработки алгоритма для вычисления хеш-кода можно сгенерировать с помощью `java.util.Arrays#hashCode` , предоставив массив Object или primitives, содержащий значения полей:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Java SE 7

В Java 1.7 представлен класс `java.util.Objects` который предоставляет метод удобства `hash(Object... objects)` , который вычисляет хэш-код на основе значений объектов, предоставленных ему. Этот метод работает так же, как `java.util.Arrays#hashCode` .

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Примечание: этот подход неэффективен и создает объекты мусора каждый раз, когда вы вызываете свой собственный метод `hashCode()` :

- Создается временный `Object[]` . (В версии `Objects.hash()` массив создается механизмом «varargs».)
- Если какое-либо из полей является примитивными типами, они должны быть помещены в бокс и могут создавать более временные объекты.
- Массив должен быть заполнен.
- Массив должен быть повторен с помощью `Arrays.hashCode` или `Objects.hash` .
- `Object.hashCode()` который `Arrays.hashCode` или `Objects.hash` должен сделать (возможно) не может быть встроен.

Внутреннее кэширование хеш-кодов

Поскольку вычисление хеш-кода объекта может быть дорогостоящим, может быть

привлекательным кэшировать значение хеш-кода внутри объекта при первом его вычислении. Например

```
public final class ImmutableArray {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableArray(int[] initial) {
        array = initial.clone();
    }

    // Other methods

    @Override
    public boolean equals(Object obj) {
         // ...
    }

    @Override
    public int hashCode() {
        int h = hash;
        if (h == 0) {
            h = Arrays.hashCode(array);
            hash = h;
        }
        return h;
    }
}
```

Этот подход отменяет стоимость (многократно) вычисления хеш-кода накладных расходов дополнительного поля для кэширования хеш-кода. Независимо от того, будет ли это окупаться, так как оптимизация производительности будет зависеть от того, насколько часто данный объект хэшируется (искал) и другие факторы.

Вы также заметите, что если истинный хэш-код объекта `ImmutableArray` равен нулю (один шанс в 2^{32}), кеш неэффективен.

Наконец, этот подход намного сложнее реализовать правильно, если объект, который мы хешируем, изменен. Однако, если хэш-коды меняются; см. контракт выше.

wait () и notify ()

`wait()` и `notify()` работают в тандеме - когда один поток вызовов `wait()` на объекте, что поток будет заблокирован до тех пор , другой поток вызовов `notify()` или `notifyAll()` на тот же объект.

(См. Также: wait () / notify ())

```
package com.example.examples.object;

import java.util.concurrent.atomic.AtomicBoolean;

public class WaitAndNotify {
```

```
    public static void main(String[] args) throws InterruptedException {
        final Object obj = new Object();
        AtomicBoolean aHasFinishedWaiting = new AtomicBoolean(false);

        Thread threadA = new Thread("Thread A") {
            public void run() {
                System.out.println("A1: Could print before or after B1");
                System.out.println("A2: Thread A is about to start waiting...");
                try {
                    synchronized (obj) { // wait() must be in a synchronized block
                        // execution of thread A stops until obj.notify() is called
                        obj.wait();
                    }
                    System.out.println("A3: Thread A has finished waiting. "
                            + "Guaranteed to happen after B3");
                } catch (InterruptedException e) {
                    System.out.println("Thread A was interrupted while waiting");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("Thread B") {
            public void run() {
                System.out.println("B1: Could print before or after A1");

                System.out.println("B2: Thread B is about to wait for 10 seconds");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // sleep for 1 second
                    } catch (InterruptedException e) {
                        System.err.println("Thread B was interrupted from waiting");
                    }
                }

                System.out.println("B3: Will ALWAYS print before A3 since "
                        + "A3 can only happen after obj.notify() is called.");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // notify ONE thread which has called obj.wait()
                        obj.notify();
                    }
                }
            }
        };

        threadA.start();
        threadB.start();

        threadA.join();
        threadB.join();

        System.out.println("Finished!");
    }
}
```

Пример вывода:

```
A1: Could print before or after B1
B1: Could print before or after A1
A2: Thread A is about to start waiting...
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
A1: Could print before or after B1
A2: Thread A is about to start waiting...
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

A1: Could print before or after B1
A2: Thread A is about to start waiting...
B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!
```

Метод getClass ()

Метод `getClass()` может использоваться для поиска типа класса выполнения для объекта. См. Пример ниже:

```
public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }
}

public class SpecificUser extends User {
    private String specificUserID;

    public SpecificUser(String specificUserID, long userID, String name) {
        super(userID, name);
        this.specificUserID = specificUserID;
    }
}

public static void main(String[] args){
    User user = new User(879745, "John");
    SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
    User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

    System.out.println(user.getClass()); //Prints "class User"
    System.out.println(specificUser.getClass()); //Prints "class SpecificUser"
    System.out.println(anotherSpecificUser.getClass()); //Prints "class SpecificUser"
}
```

Метод `getClass()` возвращает наиболее специфический тип класса, поэтому, когда `getClass()` вызывается в `anotherSpecificUser` , возвращаемое значение является `class SpecificUser` потому что это ниже дерева наследования, чем `User` .

Стоит отметить, что, хотя метод `getClass` объявлен как:

```
public final native Class<?> getClass();
```

Действительным статическим типом, возвращаемым вызовом `getClass` является `Class<? extends T>` где `T` - статический тип объекта, на который вызывается `getClass` .

т.е. следующее:

```
Class<? extends String> cls = "".getClass();
```

метод clone ()

Метод `clone()` используется для создания и возврата копии объекта. Этот метод следует избегать, так как он проблематичен, и конструктор копирования или какой-либо другой подход к копированию следует использовать в пользу `clone()` .

Для использования метода все классы, вызывающие метод, должны реализовывать интерфейс `Cloneable` .

Сам интерфейс `Cloneable` - это просто интерфейс тега, используемый для изменения поведения метода `native clone()` который проверяет, реализует ли класс вызывающих объектов `Cloneable` . Если вызывающий абонент не реализует этот интерфейс, будет `CloneNotSupportedException` .

Сам класс `Object` не реализует этот интерфейс, поэтому `CloneNotSupportedException` будет вызываться, если вызывающий объект имеет класс `Object` .

Чтобы клон был прав, он должен быть независим от объекта, из которого он клонируется, поэтому может потребоваться изменить объект до его возвращения. Это означает, что по существу создать «глубокую копию», также копируя любой из *изменяемых* объектов, которые составляют внутреннюю структуру клонируемого объекта. Если это не выполняется правильно, клонированный объект не будет независимым и будет иметь те же ссылки на изменяемые объекты, что и объект, из которого он был клонирован. Это приведет к непоследовательному поведению, так как любые изменения в них повлияют на другие.

```
class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
```

```
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {
              result.y = this.y.clone();
            }
            if (this.z != null) {
              result.z = this.z.clone();
            }

            // Done, return the new object
            return result;

        } catch (CloneNotSupportedException e) {
            // in case any of the cloned mutable fields do not implement Cloneable
            throw new AssertionError(e);
        }
    }
}
```

метод finalize ()

Это *защищенный* и *нестатический* метод класса `Object` . Этот метод используется для выполнения некоторых окончательных операций или очистки операций над объектом до его удаления из памяти.

> Согласно документу, этот метод вызывается сборщиком мусора на объекте, когда сбор мусора определяет, что больше нет ссылок на объект.

Но нет никаких гарантий того, что метод `finalize()` будет вызван, если объект по-прежнему доступен или нет, когда сборщик мусора запускается, когда объект становится подходящим. Вот почему лучше **не полагаться** на этот метод.

В основных библиотеках Java могут быть найдены некоторые примеры использования, например, в `FileInputStream.java` :

```
protected void finalize() throws IOException {
    if ((fd != null) &&  (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

В этом случае это последний шанс закрыть ресурс, если этот ресурс не был закрыт раньше.

Как правило, считается неправильной практикой использовать метод `finalize()` в приложениях любого типа и его следует избегать.

Финализаторы *не* предназначены для освобождения ресурсов (например, закрытие файлов). Сборщик мусора вызывается, когда (если!) Система работает на куче. Вы не можете полагаться на это, чтобы вызываться, когда система работает на ручках файлов или по какой-либо другой причине.

Предполагаемый прецедент для финализаторов предназначен для объекта, который должен быть возвращен, чтобы уведомить какой-либо другой объект о его предстоящей гибели. Для этого теперь существует лучший механизм - класс `java.lang.ref.WeakReference<T>` . Если вы считаете, что вам нужно написать метод `finalize()` , вы должны изучить, можете ли вы решить ту же проблему, используя `WeakReference` . Если это не решит вашу проблему, вам может потребоваться переосмыслить свой дизайн на более глубоком уровне.

Для дальнейшего чтения здесь приведен пункт о методе `finalize()` из книги «Эффективная Java» Джошуа Блоха.

Конструктор объектов

Все конструкторы в Java должны сделать вызов конструктору `Object` . Это делается с помощью вызова `super()` . Это должна быть первая строка в конструкторе. Причиной этого является то, что объект может быть фактически создан в куче до выполнения любой дополнительной инициализации.

Если вы не укажете вызов `super()` в конструкторе, компилятор поместит его для вас.

Таким образом, все три из этих примеров функционально идентичны

с явным вызовом конструктора `super()`

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

с неявным вызовом конструктора `super()`

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

с неявным конструктором

```
public class MyClass {

}
```

Как насчет Constructor-Chaining?

В качестве первой инструкции конструктора можно вызвать другие конструкторы. Поскольку и явный вызов супер-конструктора, и вызов другому конструктору должны быть как первыми инструкциями, так и взаимоисключающими.

```
public class MyClass {

    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

Вызов нового `MyClass(Arrays.asList("a", "b", "c"))` вызовет второй конструктор с аргументом List, который, в свою очередь, делегирует первому конструктору (который будет делегировать неявно `super()`), а затем вызовите `addInitialValues(int size)` со вторым размером списка. Это используется для уменьшения дублирования кода, когда несколько конструкторов должны выполнять одну и ту же работу.

Как вызвать конкретный конструктор?

В приведенном выше примере можно либо вызвать `new MyClass("argument")` либо `new MyClass("argument", 0)` . Другими словами, подобно перегрузке метода , вы просто вызываете конструктор с параметрами, которые необходимы для вашего выбранного конструктора.

Что произойдет в конструкторе класса Object?

В подклассе, который имеет пустой конструктор по умолчанию (минус вызов `super()`), ничего не может произойти.

Пустой конструктор по умолчанию может быть явно определен, но если он не будет компилятором, он будет установлен для вас до тех пор, пока не будут определены другие конструкторы.

Как объект создается из конструктора в Object?

Фактическое создание объектов зависит от JVM. Каждый конструктор Java появляется как специальный метод с именем `<init>` который отвечает за инициализацию экземпляра. Этот метод `<init>` предоставляется компилятором, и поскольку `<init>` не является допустимым идентификатором в Java, он не может использоваться непосредственно на языке.

> Как JVM вызывает этот метод `<init>` ?

JVM будет вызывать метод `<init>` используя `invokespecial` инструкцию `invokespecial` и может быть вызван только в неинициализированных экземплярах класса.

Для получения дополнительной информации см. Спецификацию JVM и спецификацию Java Language:

- Специальные методы (JVM) - JVMS - 2.9
- Конструкторы - JLS - 8.8

Прочитайте Методы и конструкторы классов объектов онлайн:
https://riptutorial.com/ru/java/topic/145/методы-и-конструкторы-классов-объектов

глава 110: Методы по умолчанию

Вступление

Метод по умолчанию, введенный в Java 8, позволяет разработчикам добавлять новые методы в интерфейс без нарушения существующих реализаций этого интерфейса. Он обеспечивает гибкость, позволяющую интерфейсу определять реализацию, которая будет использоваться по умолчанию, когда класс, который реализует этот интерфейс, не может обеспечить реализацию этого метода.

Синтаксис

- public default void methodName () {/ * метод body * /}

замечания

Методы по умолчанию

- Может использоваться в интерфейсе, чтобы ввести поведение, не заставляя существующие подклассы реализовывать его.
- Может быть переопределено подклассами или под-интерфейсом.
- Не разрешено переопределять методы в классе java.lang.Object.
- Если класс, реализующий более одного интерфейса, наследует методы по умолчанию с идентичными сигнатурами методов из каждой из intefaces, тогда он должен переопределять и предоставлять свой собственный интерфейс, как если бы они не были методами по умолчанию (как часть разрешения множественного наследования).
- Хотя они предназначены для введения поведения без нарушения существующих реализаций, существующие подклассы со статическим методом с той же сигнатурой метода, что и новый метод по умолчанию, будут по-прежнему нарушаться. Однако это верно даже в случае введения метода экземпляра в суперкласс.

Статические методы

- Может использоваться в интерфейсе, в первую очередь предназначенном для использования в качестве метода утилиты для методов по умолчанию.
- Не может быть переопределен подклассами или под-интерфейсом (скрыт от них). Однако, как и в случае со статическими методами даже сейчас, каждый класс или интерфейс могут иметь свои собственные.

- Не разрешено переопределять методы экземпляра в классе java.lang.Object (как и в случае с подклассами).

Ниже приведена таблица, суммирующая взаимодействие между подклассом и суперклассом.

-	SUPER_CLASS-INSTANCE-METHOD	SUPER_CLASS-STATIC-МЕТОД
SUB_CLASS-INSTANCE-METHOD	*Переопределение*	*генерирует-compiletime-ошибку*
SUB_CLASS-STATIC-МЕТОД	*генерирует-compiletime-ошибку*	*шкуры*

Ниже приведена таблица, суммирующая взаимодействие между интерфейсом и классом реализации.

-	ИНТЕРФЕЙС-DEFAULT-МЕТОД	ИНТЕРФЕЙС-STATIC-МЕТОД
IMPL_CLASS-INSTANCE-METHOD	*Переопределение*	*шкуры*
IMPL_CLASS-STATIC-МЕТОД	*генерирует-compiletime-ошибку*	*шкуры*

Рекомендации :

- http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Examples

Основное использование методов по умолчанию

```
/**
 * Interface with default method
 */
```

```
public interface Printable {
    default void printString() {
        System.out.println( "default implementation" );
    }
}

/**
 * Class which falls back to default implementation of {@link #printString()}
 */
public class WithDefault
    implements Printable
{
}

/**
 * Custom implementation of {@link #printString()}
 */
public class OverrideDefault
    implements Printable {
    @Override
    public void printString() {
        System.out.println( "overridden implementation" );
    }
}
```

Следующие утверждения

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Будет производить этот вывод:

```
default implementation
overridden implementation
```

Доступ к другим методам интерфейса в рамках метода по умолчанию

Вы также можете получить доступ к другим методам интерфейса из вашего метода по умолчанию.

```
public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }
```

```
    @Override
    public int getB() {
        return 2;
    }
}
```

Следующий оператор напечатает *3* :

```
System.out.println(new Sum().calculateSum());
```

Методы по умолчанию могут использоваться вместе со статическими методами интерфейса:

```
public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}
```

Следующее утверждение также напечатает 3:

```
System.out.println(new Sum().calculateSum());
```

Доступ к переопределенным методам по умолчанию из класса реализации

В классах `super.foo()` будет выглядеть только в суперклассах. Если вы хотите вызвать реализацию по умолчанию из суперинтерфейса, вам нужно квалифицировать `super` с именем интерфейса: `Fooable.super.foo()` .

```
public interface Fooable {
    default int foo() {return 3;}
}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //error: no method foo() in java.lang.Object
        return Fooable.super.foo() + 1; //okay, returns 4
    }
}
```

Зачем использовать методы по умолчанию?

Простой ответ заключается в том, что он позволяет вам развивать существующий интерфейс без нарушения существующих реализаций.

Например, у вас есть интерфейс `Swim` который вы опубликовали 20 лет назад.

```
public interface Swim {
    void backStroke();
}
```

Мы отлично поработали, наш интерфейс очень популярен, в мире есть много реализаций, и вы не контролируете их исходный код.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

Через 20 лет вы решили добавить новые функции в интерфейс, но похоже, что наш интерфейс заморожен, потому что он нарушит существующие реализации.

К счастью, Java 8 представляет новую функцию, называемую методом по умолчанию.

Теперь мы можем добавить новый метод в интерфейс `Swim` .

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Теперь все существующие реализации нашего интерфейса могут по-прежнему работать. Но самое главное, они могут реализовать недавно добавленный метод в свое время.

Одна из основных причин этого изменения и одно из его самых больших применений - в структуре Java Collections. Oracle не смог добавить метод `foreach` к существующему интерфейсу Iterable, не нарушая при этом весь существующий код, который реализовал Iterable. Добавляя методы по умолчанию, существующая реализация Iterable наследует реализацию по умолчанию.

Класс, абстрактный класс и метод интерфейса

Реализации в классах, включая абстрактные объявления, имеют приоритет над всеми значениями по умолчанию интерфейса.

- Метод абстрактного класса имеет приоритет над методом интерфейса по умолчанию .

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
    }
}

public class FooSwimmer extends AbstractSwimmer {
}
```

Следующее утверждение

```
new FooSwimmer().backStroke();
```

Будет производить

```
AbstractSwimmer.backStroke
```

- Метод класса имеет приоритет над методом интерфейса по умолчанию

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
}

public class FooSwimmer extends AbstractSwimmer {
    public void backStroke() {
        System.out.println("FooSwimmer.backStroke");
    }
}
```

Следующее утверждение

```
new FooSwimmer().backStroke();
```

Будет производить

```
FooSwimmer.backStroke
```

Метод множественного наследования по умолчанию метода по умолчанию

Рассмотрим следующий пример:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}
```

Вот два интерфейса, объявляющих метод `foo` `default` с той же сигнатурой.

Если вы попытаетесь `extend` эти интерфейсы в новом интерфейсе, вам нужно сделать выбор из двух, потому что Java заставляет вас явно разрешить это столкновение.

Во-первых , вы можете объявить метод `foo` с той же сигнатурой, что и `abstract` , что переопределит поведение `A` и `B`

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

И когда вы будете `implement` `ABExtendsAbstract` в `class` вам придется обеспечить реализацию `foo` :

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

Или, во- **вторых** , вы можете обеспечить полностью новую реализацию по `default` . Вы также можете повторно использовать код методов `A` и `B` `foo` путем доступа к переопределенным методам по умолчанию из класса реализации .

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

И когда вы будете `implement` `ABExtends` в `class` вам `not` придется выполнять `foo` реализацию:

```
public class ABExtendsImpl implements ABExtends {}
```

Прочитайте Методы по умолчанию онлайн: https://riptutorial.com/ru/java/topic/113/методы-по-

умолчанию

глава 111: Методы сбора коллекции

Вступление

Приход Java 9 привносит много новых функций в API коллекций Java, одним из которых является сборка заводских методов. Эти методы позволяют легко инициализировать **неизменяемые** коллекции, независимо от того, являются ли они пустыми или непустыми.

Обратите внимание, что эти заводские методы доступны только для следующих интерфейсов: `List<E>` , `Set<E>` и `Map<K, V>`

Синтаксис

- `static <E> List<E> of()`
- `static <E> List<E> of(E e1)`
- `static <E> List<E> of(E e1, E e2)`
- `static <E> List<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> List<E> of(E... elements)`
- `static <E> Set<E> of()`
- `static <E> Set<E> of(E e1)`
- `static <E> Set<E> of(E e1, E e2)`
- `static <E> Set<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> Set<E> of(E... elements)`
- `static <K,V> Map<K,V> of()`
- `static <K,V> Map<K,V> of(K k1, V v1)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, ..., K k9, V v9, K k10, V v10)`
- `static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)`

параметры

Метод w / Параметр	Описание
`List.of(E e)`	Общий тип, который может быть классом или интерфейсом.
`Set.of(E e)`	Общий тип, который может быть классом или интерфейсом.
`Map.of(K k, V v)`	Пара ключ-значение общих типов, каждый из которых может быть классом или интерфейсом.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	Экземпляр `Map.Entry` где его ключ может быть `K` или одним из его дочерних элементов, а его значение может быть `V` или любым из его дочерних элементов.

Examples

Список Примеры заводских методов

- `List<Integer> immutableEmptyList = List.of();`
  - Инициализирует пустой, неизменный `List<Integer>` .
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Инициализирует неизменяемый `List<Integer>` с пятью начальными элементами.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
  - Инициализирует измененный `List<Integer>` из неизменяемого `List<Integer>` .

Задавать Примеры заводских методов

- `Set<Integer> immutableEmptySet = Set.of();`
  - Инициализирует пустой, неизменяемый `Set<Integer>` .
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
  - Инициализирует неизменяемый `Set<Integer>` с пятью начальными элементами.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
  - Инициализирует измененный `Set<Integer>` из неизменяемого `Set<Integer>` .

карта Примеры заводских методов

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
  - Инициализирует пустой, неизменяемый `Map<Integer, Integer>` .
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
  - Инициализирует неизменяемую `Map<Integer, Integer>` с двумя начальными ключевыми значениями.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
  - Инициализирует неизменяемую `Map<Integer, Integer>` с двумя начальными ключевыми значениями.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
  - Инициализирует измененную `Map<Integer, Integer>` из неизменяемого `Map<Integer, Integer>` .

Прочитайте Методы сбора коллекции онлайн: https://riptutorial.com/ru/java/topic/9783/методы-сбора-коллекции

глава 112: Модель памяти Java

замечания

Модель памяти Java - это раздел JLS, который определяет условия, при которых одному потоку гарантированно видят эффекты записи в памяти, сделанные другим потоком. Соответствующий раздел в последних выпусках - «Модель памяти JLS 17.4» (в Java 8 , Java 7 , Java 6)

Был капитальный ремонт Java Memory Model в Java 5, который (среди прочего) изменил способ работы `volatile` . С тех пор модель памяти практически не изменилась.

Examples

Мотивация для модели памяти

Рассмотрим следующий пример:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

Если этот класс используется, это однопоточное приложение, то наблюдаемое поведение будет таким, каким вы ожидали. Например:

```
public class SingleThreaded {
    public static void main(String[] args) {
        Example eg = new Example();
        System.out.println(eg.a + ", " + eg.c);
        eg.doIt();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

выведет:

```
0, 0
1, 1
```

Что касается «основного» потока , то инструкции в методе `main()` метод `doIt()` будут выполняться в том порядке, в котором они записаны в исходном коде. Это явное требование спецификации языка Java (JLS).

Теперь рассмотрим тот же класс, который используется в многопоточном приложении.

```
public class MultiThreaded {
    public static void main(String[] args) {
        final Example eg = new Example();
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    eg.doIt();
                }
            }
        }).start();
        while (true) {
            System.out.println(eg.a + ", " + eg.c);
        }
    }
}
```

Что будет печатать?

На самом деле, согласно JLS, невозможно предсказать, что это будет печатать:

- Вероятно, вы увидите несколько строк `0, 0` для начала.
- Тогда вы, вероятно, увидите строки, такие как `N, N` или `N, N + 1` .
- Вы можете видеть строки типа `N + 1, N`
- В теории вы даже можете увидеть, что линии `0, 0` продолжаются навсегда [1] .

1 - На практике наличие операторов `println` может привести к некорректной синхронизации и кэш памяти. Вероятно, это скроет некоторые из эффектов, которые приведут к вышеуказанному поведению.

Итак, как мы можем объяснить это?

Переопределение заданий

Одним из возможных объяснений неожиданных результатов является то, что компилятор JIT изменил порядок присвоений в методе `doIt()` . JLS требует , чтобы заявления , как *представляется для* исполнения в порядке *с точки зрения текущего* потока. В этом случае ничто в коде метода `doIt()` может наблюдать эффект (гипотетического) переупорядочения этих двух утверждений. Это означает, что JIT-компилятор будет разрешен для этого.

Зачем это делать?

На типичном современном оборудовании машинные инструкции выполняются с использованием конвейера команд, который позволяет последовательности инструкций находиться на разных этапах. Некоторые этапы выполнения команд занимают больше времени, чем другие, и операции с памятью имеют тенденцию занимать больше времени. Интеллектуальный компилятор может оптимизировать пропускную способность инструкции по конвейеру, заказывая инструкции, чтобы максимизировать количество перекрытий. Это может привести к тому, что выполняемые части инструкций будут

неработоспособны. JLS позволяет это обеспечить, что не влияет на результат вычисления *с точки зрения текущей нити* .

Эффекты кэшей памяти

Вторым возможным объяснением является эффект кэширования памяти. В классической компьютерной архитектуре каждый процессор имеет небольшой набор регистров и больший объем памяти. Доступ к регистрам намного быстрее, чем доступ к основной памяти. В современных архитектурах есть кэши памяти, которые медленнее регистра, но быстрее, чем основная память.

Компилятор будет использовать это, пытаясь сохранить копии переменных в регистрах или в кэшах памяти. Если переменную не *нужно* очищать в основной памяти, или ее не *нужно* читать из памяти, это может привести к значительным преимуществам в производительности, если вы этого не сделаете. В случаях, когда JLS не требует, чтобы операции с памятью были видимыми для другого потока, компилятор Java JIT скорее всего не добавит инструкции «барьер чтения» и «барьер записи», которые заставят читать и записывать основную память. Еще раз, преимущества производительности при этом важны.

Правильная синхронизация

До сих пор мы видели, что JLS позволяет компилятору JIT генерировать код, который делает однопоточный код быстрее, переупорядочивая или избегая операций с памятью. Но что происходит, когда другие потоки могут наблюдать состояние (общих) переменных в основной памяти?

Ответ заключается в том, что другие потоки могут наблюдать переменные состояния, которые выглядят невозможными ... на основе кодового порядка операторов Java. Решением этого является использование соответствующей синхронизации. Три основных подхода:

- Использование примитивных мьютексов и `synchronized` конструкций.
- Использование `volatile` переменных.
- Использование поддержки параллелизма на более высоком уровне; например, классы в пакетах `java.util.concurrent` .

Но даже при этом важно понять, где необходима синхронизация, и на какие последствия вы можете положиться. Здесь находится модель памяти Java.

Модель памяти

Модель памяти Java - это раздел JLS, который определяет условия, при которых одному

потоку гарантированно видят эффекты записи в памяти, сделанные другим потоком. Модель памяти задается с достаточной степенью *формальной строгости* , и (в результате) требуется подробное и тщательное чтение для понимания. Но основным принципом является то, что некоторые конструкции создают связь «между событиями» перед записью переменной одним потоком и последующее чтение одной и той же переменной другим потоком. Если существует отношение «произойдет до», компилятор JIT *обязан* сгенерировать код, который гарантирует, что операция чтения увидит значение, записанное записью.

Вооружившись этим, можно рассуждать о когерентности памяти в программе Java и решить, будет ли это предсказуемым и последовательным для *всех* платформ исполнения.

Происходит до отношений

(Ниже приведена упрощенная версия того, что говорит спецификация языка Java. Для более глубокого понимания вам необходимо прочитать спецификацию.)

Случаи-до отношений являются частью модели памяти, которые позволяют нам понять и понять видимость памяти. Как говорит JLS (JLS 17.4.5):

> «Два *действия* могут быть упорядочены с помощью отношения« *произойдет раньше »* . Если *произойдет* одно действие - *перед* другим, то первое будет видимым и упорядоченным до второго».

Что это значит?

действия

Действия, указанные выше в цитируемой цитате, указаны в JLS 17.4.2 . Существует 5 видов действий, указанных в спецификации:

- Чтение: чтение неизменяемой переменной.
- Запись: запись нелетучей переменной.
- Действия синхронизации:
  - Volatile read: Чтение изменчивой переменной.
  - Volatile write: запись изменчивой переменной.
  - Замок. Блокировка монитора
  - Разблокировка. Разблокировка монитора.
  - (Синтетическое) первое и последнее действие потока.

    - Действия, которые запускают поток или обнаруживают, что поток завершен.

- Внешние действия. Действие, которое имеет результат, зависящий от среды, в которой программа.
- Действия дивергенции потока. Они моделируют поведение определенных видов бесконечного цикла.

Заказ программы и синхронизация

Эти два порядка (JLS 17.4.3 и JLS 17.4.4) управляют выполнением операторов в Java

Заказ программы описывает порядок выполнения инструкции в одном потоке.

Порядок синхронизации описывает порядок выполнения оператора для двух операторов, связанных синхронизацией:

- Действие разблокировки на мониторе *синхронизируется со* всеми последующими действиями блокировки на этом мониторе.
- *Синхронизация* записи в изменчивую переменную - *со* всеми последующими чтениями одной и той же переменной любым потоком.
- Действие, которое запускает поток (т. `Thread.start()` Вызов `Thread.start()`), *синхронизируется с* первым действием в `Thread.start()` потоке (т. `Thread.start()` метода `run()` потока).
- По умолчанию инициализация полей *синхронизируется с* первым действием в каждом потоке. (См. JLS для объяснения этого.)
- Заключительное действие в потоке *синхронизируется с* любым действием в другом потоке, который обнаруживает завершение; например, возврат вызова `join()` или `isTerminated()` который возвращает `true` .
- Если один поток прерывает другой поток, вызов прерывания в первом потоке *синхронизируется - с* точкой, в которой другой поток обнаруживает, что поток был прерван.

Бывает-до заказа

Это упорядочение (JLS 17.4.5) определяет, гарантируется ли запись в памяти для последующего чтения в памяти.

Более конкретно, чтение переменной `v` гарантируется наблюдением записи в `v` тогда и только тогда, когда `write(v)` *происходит - перед* `read(v)` И нет промежуточной записи в `v` . Если есть промежуточные записи, то `read(v)` может видеть результаты из них, а не

предыдущие.

Правила, определяющие порядок *до-* заказа, следующие:

- **Happens-Before Rule # 1** - Если x и y - действия одного и того же потока, а x - до y в *программном порядке* , то x *происходит до* y.
- **Happens-Before Rule # 2** - Происходит до конца от конца конструктора объекта до начала финализатора для этого объекта.
- **Happens-Before Rule # 3** - Если действие x *синхронизируется с* последующим действием y, то x *происходит до* y.
- **Happens-Before Rule # 4** - Если x *происходит - до того, как* y и y *произойдет - до* z, то x *произойдет - до* z.

Кроме того, различные классы в стандартных библиотеках Java указываются как определяющие как *-либо* отношения. Вы можете интерпретировать это как означающее, что это происходит *так или иначе* , без необходимости точно знать, как гарантия будет выполнена.

Бывает - прежде чем рассуждения применимы к некоторым примерам

Мы представим несколько примеров, чтобы показать, как применять *, прежде чем* рассуждать, чтобы проверить, что записи видны для последующих чтений.

Однопоточный код

Как и следовало ожидать, записи всегда видны для последующих чтений в однопоточной программе.

```
public class SingleThreadExample {
    public int a, b;

    public int add() {
       a = 1;          // write(a)
       b = 2;          // write(b)
       return a + b;  // read(a) followed by read(b)
    }
}
```

By Happens-Before Правило № 1:

1. Действие `write(a)` *происходит - перед* действием `write(b)` .
2. Действие `write(b)` *происходит - перед* действием `read(a)` .
3. `read(a)` *происходит - перед* действием `read(a)` .

By Happens-Before Правило № 4:

4. `write(a)` *происходит - перед* `write(b)` И `write(b)` *происходит - перед* `read(a)` ПРОБЛЕМЫ `write(a)` *происходит - перед* `read(a)` .
5. `write(b)` *происходит - перед* `read(a)` И `read(a)` *происходит - перед* `read(b)` ПРОБЛЕМЫ `write(b)` *происходит - перед* `read(b)` .

Подведение итогов:

6. Отношение `write(a)` *-before* `read(a)` означает, что `a + b` гарантированно видит правильное значение `a` .
7. Отношение `write(b)` *-before* `read(b)` означает, что `a + b` гарантированно видит правильное значение `b` .

Поведение «volatile» в примере с 2 потоками

Мы будем использовать следующий примерный код, чтобы изучить некоторые последствия модели памяти для `volatile.

```
public class VolatileExample {
    private volatile int a;
    private int b;         // NOT volatile

    public void update(int first, int second) {
       b = first;         // write(b)
       a = second;         // write-volatile(a)
    }

    public int observe() {
       return a + b;       // read-volatile(a) followed by read(b)
    }
}
```

Во-первых, рассмотрим следующую последовательность операторов с участием 2 потоков:

1. Создается один экземпляр `VolatileExample` ; назовите это `ve` ,
2. `ve.update(1, 2)` вызывается в одном потоке и
3. `ve.observe()` вызывается в другом потоке.

By Happens-Before Правило № 1:

1. Действие `write(a)` *происходит - перед* действием `volatile-write(a)` .
2. `volatile-read(a)` - *перед* действием `read(b)` .

«Бывает» - до правила № 2:

3. Действие `volatile-write(a)` в первом потоке *происходит до* выполнения `volatile-read(a)` действия во втором потоке.

By Happens-Before Правило № 4:

4. Действие `write(b)` в первом потоке *происходит - перед* действием `read(b)` во втором потоке.

Другими словами, для этой конкретной последовательности мы гарантируем, что второй поток увидит обновление для нелетучей переменной `b` сделанной первым потоком. Тем не менее, также должно быть ясно, что если назначения в методе `update` были наоборот, или метод `observe()` прочитал переменную `b` перед `a` , то цепочка, которая произошла *раньше* , будет нарушена. Цепь также будет разбита, если во втором потоке `volatile-read(a)` будет следовать `volatile-write(a)` в первом потоке.

Когда цепь сломана, нет *гарантии,* что `observe()` увидит правильное значение `b` .

Летучие с тремя нитями

Предположим, что мы добавим третий поток в предыдущий пример:

1. Создается один экземпляр `VolatileExample` ; назовите это `ve` ,
2. Два потока требуют `update` :
    - `ve.update(1, 2)` вызывается в одном потоке,
    - `ve.update(3, 4)` вызывается во втором потоке,
3. `ve.observe()` впоследствии вызывается в третьем потоке.

Чтобы полностью проанализировать это, нам нужно рассмотреть все возможные перемежения операторов в первом и втором потоках. Вместо этого мы рассмотрим только два из них.

Сценарий №1 - предположим, что `update(1, 2)` предшествует `update(3,4)` мы получаем следующую последовательность:

```
write(b, 1), write-volatile(a, 2)     // first thread
write(b, 3), write-volatile(a, 4)     // second thread
read-volatile(a), read(b)             // third thread
```

В этом случае, легко видеть , что существует непрерывная *происходит до-цепь* от `write(b, 3)` , чтобы `read(b)` . Кроме того, нет промежуточной записи в `b` . Таким образом, для этого сценария третий поток гарантированно видит, что `b` имеет значение `3` .

Сценарий № 2 - предположим, что `update(1, 2)` и `update(3,4)` перекрываются, а элементы чередуются следующим образом:

```
write(b, 3)                           // second thread
write(b, 1)                           // first thread
write-volatile(a, 2)                  // first thread
write-volatile(a, 4)                  // second thread
read-volatile(a), read(b)             // third thread
```

Теперь, в то время как есть *происходит прежде* , *чем-цепь* от `write(b, 3)` , чтобы `read(b)` ,

существует промежуточные `write(b, 1)` действие , выполняемое другой нить. Это означает, что мы не можем быть уверены, какое значение `read(b)` .

(Помимо этого: это демонстрирует, что мы не можем полагаться на `volatile` для обеспечения видимости энергонезависимых переменных, за исключением очень ограниченных ситуаций.)

Как избежать необходимости понимать модель памяти

Модель памяти трудно понять и ее трудно применить. Это полезно, если вам нужно рассуждать о правильности многопоточного кода, но вы не хотите, чтобы это объяснение для каждого многопоточного приложения, которое вы пишете.

Если вы принимаете следующие принципы при написании параллельного кода на Java, *во многом* вы можете избежать необходимости прибегать к *случаям - до* рассуждений.

- По возможности используйте неизменяемые структуры данных. Правильно внедренный неизменяемый класс будет потокобезопасным и не будет вводить проблемы безопасности потоков при использовании его с другими классами.
- Понимать и избегать «небезопасной публикации».
- Используйте примитивные мьютексы или объекты `Lock` для синхронизации доступа к состоянию в изменяемых объектах, которые должны быть потокобезопасными [1] .
- Используйте `Executor` / `ExecutorService` или платформу fork join, а не пытайтесь напрямую создавать потоки управления.
- Используйте классы `java.util.concurrent, которые предоставляют расширенные блокировки, семафоры, защелки и барьеры, вместо прямого использования wait / notify / notifyAll.
- Используйте версии `java.util.concurrent` карт, наборов, списков, очередей и требований, а не внешнюю синхронизацию неконкурентных коллекций.

Общий принцип - попытаться использовать встроенные библиотеки параллельного использования Java, а не «сворачивать свой собственный» параллелизм. Вы можете полагаться на их работу, если вы используете их правильно.

---

1 - Не все объекты должны быть потокобезопасными. Например, если объект или объекты *ограничены потоком* (т. *Е.* Доступен только для одного потока), то его безопасность потока не имеет отношения к делу.

Прочитайте Модель памяти Java онлайн: https://riptutorial.com/ru/java/topic/6829/модель-памяти-java

глава 113: Модификаторы без доступа

Вступление

Модификаторы без доступа ***не изменяют доступность переменных*** и методов, но они предоставляют им ***особые свойства*** .

Examples

окончательный

`final` в Java может относиться к переменным, методам и классам. Существует три простых правила:

- конечная переменная не может быть переназначена
- окончательный метод нельзя переоценить
- окончательный класс не может быть расширен

Обычаи

Хорошая практика программирования

Некоторые разработчики считают хорошей практикой отмечать переменную final, когда можете. Если у вас есть переменная, которая не должна быть изменена, вы должны отметить ее окончательную.

Важное использование `final` ключевого слова if для параметров метода. Если вы хотите подчеркнуть, что метод не меняет свои входные параметры, пометьте свойства как final.

```
public int sumup(final List<Integer> ints);
```

Это подчеркивает, что метод `sumup` не собирается изменять `ints` .

Доступ к внутреннему классу

Если ваш анонимный внутренний класс хочет получить доступ к переменной, переменная должна быть отмечена `final`

```
public IPrintName printName(){
  String name;
  return new IPrintName(){
      @Override
      public void printName(){
          System.out.println(name);
      }
  };
```

```
}
```

Этот класс не компилируется, поскольку `name` переменной не является окончательным.

Java SE 8

Эффективно конечные переменные являются исключением. Это локальные переменные, которые записываются только один раз и поэтому могут быть сделаны окончательными. Эффективно конечные переменные можно получить и из классов анонимуса.

`final static` переменная

Несмотря на то, что приведенный ниже код полностью легален, когда `final` переменная `foo` не является `static` , в случае `static` она не будет компилироваться:

```
class TestFinal {
    private final static List foo;

    public Test() {
        foo = new ArrayList();
    }
}
```

Причина в том, что, повторим еще раз, *конечная переменная не может быть переназначена* . Поскольку `foo` является статическим, он разделяется между всеми экземплярами класса `TestFinal` . Когда создается новый экземпляр класса `TestFinal` , его конструктор вызывается и, следовательно, foo получает переназначение, компилятор которого не разрешает. Правильный способ инициализации переменной `foo` в этом случае:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

или с помощью статического инициализатора:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

`final` методы полезны, когда базовый класс реализует некоторые важные функции, которые производный класс не должен изменять. Они также быстрее, чем не конечные методы, потому что нет никакой концепции виртуальной таблицы.

Все классы-оболочки в Java являются окончательными, например `Integer` , `Long` и т. Д. Создатели этих классов не хотели, чтобы кто-либо мог, например, расширить Integer в свой

класс и изменить основное поведение класса Integer. Одним из требований к созданию неизменяемого класса является то, что подклассы не могут переопределять методы. Самый простой способ сделать это - объявить класс `final` .

летучий

Модификатор `volatile` используется в многопоточном программировании. Если вы объявляете поле `volatile` это сигнал потокам, который должен прочитать последнее значение, а не локально кэшированное. Кроме того, `volatile` чтения и записи гарантированно являются атомарными (доступ к `volatile long` или `double` не является атомарным), что позволяет избежать некоторых ошибок чтения / записи между несколькими потоками.

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run is called in one thread
        active = true;
        while (active){
            // some code here
        }
    }

    public void stop(){ // stop() is called from another thread
        active = false;
    }
}
```

статический

`static` ключевое слово используется для класса, метода или поля, чтобы заставить их работать независимо от любого экземпляра класса.

- Статические поля являются общими для всех экземпляров класса. Им не нужен экземпляр для доступа к ним.
- Статические методы могут выполняться без экземпляра класса, в котором они находятся. Однако они могут получить доступ только к статическим полям этого класса.
- Статические классы могут быть объявлены внутри других классов. Им не нужен экземпляр класса, в котором они должны быть созданы.

```
public class TestStatic
{
    static int staticVariable;

    static {
        // This block of code is run when the class first loads
        staticVariable = 11;
    }
```

```
    int nonStaticVariable = 5;

    static void doSomething() {
        // We can access static variables from static methods
        staticVariable = 10;
    }

    void add() {
        // We can access both static and non-static variables from non-static methods
        nonStaticVariable += staticVariable;
    }

    static class StaticInnerClass {
        int number;
        public StaticInnerClass(int _number) {
            number = _number;
        }

        void doSomething() {
            // We can access number and staticVariable, but not nonStaticVariable
            number += staticVariable;
        }

        int getNumber() {
            return number;
        }
    }
}


// Static fields and methods
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10

// Static inner classes
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200
```

Аннотация

Абстракция - это процесс скрытия деталей реализации и отображения пользователю только функциональных возможностей. Абстрактный класс никогда не может быть создан. Если класс объявлен как абстрактный, единственной целью является расширение класса.

```
abstract class Car
{
    abstract void tagLine();
}

class Honda extends Car
{
    void tagLine()
    {
        System.out.println("Start Something Special");
    }
}

class Toyota extends Car
{
    void tagLine()
    {
        System.out.println("Drive Your Dreams");
    }
}
```

синхронизированный

Синхронизированный модификатор используется для управления доступом определенного метода или блока несколькими потоками. Только один поток может входить в метод или блок, который объявляется как синхронизированный. синхронизированное ключевое слово работает на внутренней блокировке объекта, в случае синхронного метода блокировка текущих объектов и статический метод использует объект класса. Любой поток, пытающийся выполнить синхронизированный блок, должен сначала получить блокировку объекта.

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }

    void SharedMethod2()
    {
        synchronized (this)
```

```
        {
            System.out.println("Thais access to currect object is synchronize "+this);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        Thread t2 = new Thread("Thread - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        t1.start();

        t2.start();
    }
}
```

преходящий

Переменная, объявленная как переходная, не будет сериализована во время сериализации объекта.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

strictfp

Java SE 1.2

Для вычисления с плавающей запятой используется модификатор strictfp. Этот модификатор делает переменную с плавающей запятой более согласованной на нескольких платформах и гарантирует, что все вычисления с плавающей запятой выполняются в соответствии со стандартами IEEE 754, чтобы избежать ошибок вычисления (ошибки округления), переполнения и переполнения как на 32-битной, так и на 64-битной архитектуре. Это нельзя применить к абстрактным методам, переменным или

конструкторам.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Прочитайте Модификаторы без доступа онлайн: https://riptutorial.com/ru/java/topic/4401/модификаторы-без-доступа

глава 114: Модули

Синтаксис

- требует java.xml;
- требует публичного java.xml; # предоставляет модуль иждивенцам для использования
- экспортирует com.example.foo; # иждивенцы могут использовать общедоступные типы в этом пакете
- экспортирует com.example.foo.impl в com.example.bar; # ограничение использования модуля

замечания

Использование модулей поощряется, но не требуется, это позволяет существующему коду продолжать работать на Java 9. Он также обеспечивает постепенный переход к модульному коду.

Любой немодульный код помещается в *неназванный модуль* при компиляции. Это специальный модуль, который может использовать типы из всех других модулей, но **только из пакетов с декларацией** `exports` .

Все пакеты в *неназванном модуле* экспортируются автоматически.

Ключевые слова, например `module` т. Д., Ограничены в использовании в объявлении модуля, но могут быть использованы в качестве идентификаторов в другом месте.

Examples

Определение базового модуля

Модули определяются в файле с именем `module-info.java` , называемом дескриптором модуля. Он должен быть помещен в корень исходного кода:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Вот простой описатель модуля:

```
module com.example {
```

```
    requires java.httpclient;
    exports com.example.foo;
}
```

Имя модуля должно быть уникальным, и рекомендуется использовать одну и ту же нотацию имен DNS-реверса, используемую пакетами, чтобы обеспечить это.

Модуль `java.base` , который содержит базовые классы Java, неявно виден любому модулю и не нуждается в его включении.

`requires` декларация позволяет использовать другие модули, в примере модуль `java.httpclient` импортируется.

Модуль также может указывать, какие пакеты он `exports` и, следовательно, делает его видимым для других модулей.

Пакет `com.example.foo` объявлен в `exports` пункта будет виден другим модулям. Любые `com.example.foo` не будут экспортироваться, им нужны собственные декларации `export` .

И наоборот, `com.example.bar` который не указан в предложениях `exports` , не будет видимым для других модулей.

Прочитайте Модули онлайн: https://riptutorial.com/ru/java/topic/5286/модули

глава 115: наборы

Examples

Объявление HashSet со значениями

Вы можете создать новый класс, который наследуется от HashSet:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

Однострочное решение:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Использование guava:

```
Sets.newHashSet("a", "b", "c")
```

Использование потоков:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Типы и использование наборов

Как правило, наборы - это тип коллекции, в котором хранятся уникальные значения. Уникальность определяется методами `equals()` и `hashCode()` .

Сортировка определяется типом набора.

`HashSet` - случайная сортировка

Java SE 7

```
Set<String> set = new HashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

`LinkedHashSet` - порядок вставки

Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Banana", "Apple", "Strawberry"]
```

`TreeSet` - с помощью `compareTo()` или `Comparator`

Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Apple", "Banana", "Strawberry"]
```

Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

инициализация

Набор - это сборник, который не может содержать повторяющиеся элементы. Он моделирует математическую абстрактную абстракцию.

`Set` есть его реализация в различных классах , как `HashSet` , `TreeSet` , `LinkedHashSet` .

Например:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Здесь `T` может быть `String` , `Integer` или любым другим **объектом** . **HashSet** позволяет быстро найти O (1), но не сортирует данные, добавленные к нему, и теряет порядок

вставки элементов.

TreeSet:

Он хранит данные отсортированным образом, жертвуя некоторой скоростью для основных операций, которые принимают O (lg (n)). Он не поддерживает порядок вставки элементов.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

Это реализация связанного списка `HashSet` После этого можно перебирать элементы в том порядке, в котором они были добавлены. Сортировка не предоставляется для ее содержимого. O (1) предоставляются основные операции, однако при сохранении связанного списка поддержки существует более высокая стоимость, чем `HashSet` .

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Основы набора

Что такое набор?

Набор представляет собой структуру данных, которая содержит набор элементов с важным свойством, что ни один из двух элементов в множестве не равен.

Типы комплектов:

1. **HashSet:** набор, поддерживаемый хэш-таблицей (на самом деле экземпляр HashMap)
2. **Связанный HashSet:** набор, поддерживаемый таблицей Hash и связанным списком, с предсказуемым порядком итерации
3. **TreeSet:** реализация **NavigableSet** на основе TreeMap.

Создание набора

```
Set<Integer> set = new HashSet<Integer>(); // Creates an empty Set of Integers

Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(); //Creates a empty Set of Integers,
with predictable iteration order
```

Добавление элементов в набор

Элементы могут быть добавлены в набор, используя метод `add()`

```
 set.add(12); //  - Adds element 12 to the set
 set.add(13); //  - Adds element 13 to the set
```

Наш набор после выполнения этого метода:

```
set = [12,13]
```

Удалить все элементы набора

```
set.clear();  //Removes all objects from the collection.
```

После этого набора будет:

```
set = []
```

Проверьте, является ли элемент частью набора

Существование элемента в наборе можно проверить с помощью метода `contains()`

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Выход: `False`

Проверьте, не установлен ли набор

Метод `isEmpty()` может использоваться для проверки того, пуст ли Set.

```
set.isEmpty();  //Returns true if the set has no elements
```

Выход: True

Удалите элемент из набора

```
 set.remove(0); // Removes first occurrence of a specified object from the collection
```

Проверьте размер набора

```
set.size(); //Returns the number of elements in the collection
```

Выход: 0

Создайте список из существующего набора

Использование нового списка

```
List<String> list = new ArrayList<String>(listOfElements);
```

Использование метода List.addAll ()

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
```

```
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Использование Java 8 Steam API

```
List<String> list = set.stream().collect(Collectors.toList());
```

Устранение дубликатов с использованием Set

Предположим, что у вас есть `elements` коллекции, и вы хотите создать другую коллекцию, содержащую те же элементы, но со всеми **дублирующимися исключениями** :

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Пример :

```
List<String> names = new ArrayList<>(
        Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));
Set<String> noDuplicates = new HashSet<>(names);
System.out.println("noDuplicates = " + noDuplicates);
```

Выход :

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Прочитайте наборы онлайн: https://riptutorial.com/ru/java/topic/3102/наборы

глава 116: наследование

Вступление

Наследование - это базовая объектно-ориентированная функция, в которой один класс приобретает и расширяет свойства другого класса, используя ключевое слово `extends` . Для интерфейсов и `implements` ключевых слов см. Интерфейсы .

Синтаксис

- класс ClassB расширяет ClassA {...}
- класс ClassB реализует InterfaceA {...}
- интерфейс InterfaceB расширяет интерфейсA {...}
- класс ClassB расширяет ClassA реализует InterfaceC, InterfaceD {...}
- абстрактный класс AbstractClassB расширяет ClassA {...}
- абстрактный класс AbstractClassB расширяет AbstractClassA {...}
- абстрактный класс AbstractClassB расширяет ClassA реализует InterfaceC, InterfaceD {...}

замечания

Наследование часто сочетается с дженериками, поэтому базовый класс имеет один или несколько параметров типа. См. Создание общего класса .

Examples

Абстрактные классы

Абстрактным классом является класс, помеченный ключевым словом `abstract` . Он, вопреки не-абстрактному классу, может содержать абстрактные методы без реализации. Однако справедливо создание абстрактного класса без абстрактных методов.

Абстрактный класс не может быть создан. Он может быть подклассифицирован (расширен), пока подкласс является либо абстрактным, либо реализует все методы, помеченные как абстрактные суперклассы.

Пример абстрактного класса:

```
public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
```

```
        this.y = y;
    }

    public abstract void render();
}
```

Класс должен быть отмечен абстрактным, если он имеет хотя бы один абстрактный метод. Абстрактным методом является метод, который не имеет реализации. Другие методы могут быть объявлены в абстрактном классе, который имеет реализацию, чтобы обеспечить общий код для любых подклассов.

Попытка создать экземпляр этого класса приведет к ошибке компиляции:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

Однако класс, который расширяет `Component` , и обеспечивает реализацию для всех его абстрактных методов и может быть создан.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Экземпляры наследующих классов также могут быть представлены как родительский класс (нормальное наследование), и они обеспечивают полиморфный эффект при вызове абстрактного метода.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Абстрактные классы vs Интерфейсы

Абстрактные классы и интерфейсы обеспечивают способ определения сигнатур методов, в то же время требуя, чтобы класс расширения / реализации обеспечивал реализацию.

Существует два основных различия между абстрактными классами и интерфейсами:

- Класс может распространять только один класс, но может реализовывать множество интерфейсов.
- Абстрактный класс может содержать экземпляры (`static`) поля, но интерфейсы могут содержать только `static` поля.

Java SE 8

Методы, объявленные в интерфейсах, не могут содержать реализации, поэтому абстрактные классы использовались, когда было полезно предоставить дополнительные методы, реализация которых называется абстрактными методами.

Java SE 8

Java 8 позволяет интерфейсам содержать методы по умолчанию , обычно реализуемые с использованием других методов интерфейса , что делает интерфейсы и абстрактные классы одинаково мощными в этом отношении.

Анонимные подклассы абстрактных классов

В качестве удобства java позволяет создавать экземпляры анонимных подклассов абстрактных классов, которые обеспечивают реализацию абстрактных методов при создании нового объекта. Используя приведенный выше пример, это может выглядеть так:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Статическое наследование

Статический метод может быть унаследован аналогично обычным методам, однако в отличие от обычных методов невозможно создать « абстрактные » методы, чтобы заставить статический метод переопределить. Написание метода с той же сигнатурой, что и статический метод в суперклассе, представляется формой переопределения, но на самом деле это просто создает новую функцию, которая скрывает другую.

```
public class BaseClass {

    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);
```

```
        SubClass.sayHello();
        //This will be different than the above statement's output, since it runs
        //A different method
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public  class SubClass extends BaseClass {

    //Inherits the sayHello function, but does not override it
    public static void sayHello(boolean test) {
        System.out.println("Hey");
    }
}

public static class StaticOverride extends BaseClass {

    //Hides the num field from BaseClass
    //You can even change the type, since this doesn't affect the signature
    public static String num = "test";

    //Cannot use @Override annotation, since this is static
    //This overrides the sayHello method from BaseClass
    public static void sayHello() {
        System.out.println("Static says Hi");
    }

}
```

Запуск любого из этих классов дает результат:

```
Hello
BaseClass's num: 5
Hello
Hey
Static says Hi
StaticOverride's num: test
```

Обратите внимание, что в отличие от обычного наследования методы статического наследования не скрыты. Вы всегда можете вызвать базовый метод `sayHello` , используя `BaseClass.sayHello()` . Но классы наследуют статические методы, *если* в подклассе не найдены методы с одной и той же сигнатурой. Если разные сигнатуры метода различаются, оба метода можно запускать из подкласса, даже если имя одного и того же.

Статические поля скрыть друг друга аналогичным образом.

Использование «final» для ограничения наследования и переопределения

Финальные классы

При использовании в объявлении `class final` модификатор запрещает объявлять другие классы, `extend` класс. `final` класс - это «лист» в иерархии классов наследования.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

Варианты использования для финальных классов

Заключительные классы можно комбинировать с `private` конструктором для управления или предотвращения создания экземпляра класса. Это можно использовать для создания так называемого «класса утилиты», который определяет только статические элементы; т.е. константы и статические методы.

```
public final class UtilityClass {

    // Private constructor to replace the default visible constructor
    private UtilityClass() {}

    // Static members can still be used as usual
    public static int doSomethingCool() {
        return 123;
    }

}
```

Неизменяемые классы также должны быть объявлены `final` . (Неизменяемый класс - это тот, чьи экземпляры не могут быть изменены после их создания, см. Тему I mmutable Objects .) При этом вы не можете создать изменяемый подкласс неизменяемого класса. Это нарушит Принцип замещения Лискова, который требует, чтобы подтип должен подчиняться «поведенческому контракту» его супертипов.

С практической точки зрения, объявляя неизменный класс `final` легче рассуждать о поведении программы. Он также рассматривает проблемы безопасности в сценарии, где ненадежный код выполняется в изолированной программной среде. (Например, поскольку `String` объявляется `final` , доверенный класс не должен беспокоиться о том, что его можно обмануть в принятии изменяемого подкласса, который ненадежный вызывающий абонент мог бы тайно изменить.)

Одним из недостатков `final` классов является то, что они не работают с некоторыми насмешливыми фреймворками, такими как Mockito. Обновление: версия Mockito 2 теперь поддерживает насмешку над финальными классами.

Конечные методы

`final` модификатор также может применяться к методам предотвращения их переопределения в подклассах:

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

Конечные методы обычно используются, когда вы хотите ограничить то, что подкласс может изменить в классе, без полного запрещения подклассов.

`final` модификатор также может применяться к переменным, но значение `final` для переменных не связано с наследованием.

Принцип замещения Лискова

Подменяемость - это принцип в объектно-ориентированном программировании, введенный Барбарой Лисковым в майне конференции 1987 года, в которой говорится, что если класс `B` является подклассом класса `A` , то везде, где ожидается `A` , вместо `B` можно использовать `B` :

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

Это также применяется, когда тип является интерфейсом, где нет необходимости в иерархической взаимосвязи между объектами:

```
interface Foo {
    void bar();
}

class A implements Foo {
    void bar() {...}
}
```

```
class B implements Foo {
    void bar() {...}
}

List<Foo> foos = new ArrayList<>();
foos.add(new A()); // OK
foos.add(new B()); // OK
```

Теперь список содержит объекты, которые не принадлежат к одной и той же иерархии классов.

наследование

С использованием ключевого слова `extends` среди классов все свойства суперкласса (также называемые *родительским классом* или *базовым классом*) присутствуют в подклассе (также известном как *дочерний класс* или *производный класс*)

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Экземпляры `SubClass` *унаследовали* метод `baseMethod()` :

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Дополнительный контент может быть добавлен в подкласс. Это позволяет использовать дополнительные функции в подклассе без каких-либо изменений в базовом классе или любых других подклассах из того же базового класса:

```
public class Subclass2 extends BaseClass {

    public void anotherMethod() {
        System.out.println("Doing subclass2 stuff");
    }
}

Subclass2 s2 = new Subclass2();
s2.baseMethod(); //Still valid , prints "Doing base class stuff"
s2.anotherMethod(); //Also valid, prints "Doing subclass2 stuff"
```

Поля также унаследованы:

```
public class BaseClassWithField {
```

```
    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

`private` поля и методы все еще существуют в подклассе, но недоступны:

```
public class BaseClassWithPrivateField {

    private int x = 5;

    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.
```

В Java каждый класс может распространяться не более чем на один класс.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

Это известно как множественное наследование, и, хотя оно является законным на некоторых языках, Java не разрешает его с классами.

В результате этого у каждого класса есть неразветвленная цепочка предков классов, ведущих к `Object` , из которого все классы спускаются.

Наследование и статические методы

В Java родительский и дочерний классы могут иметь статические методы с тем же именем. Но в таких случаях реализация статического метода в дочерней среде скрывает реализацию родительского класса, это не метод переопределения. Например:

```
class StaticMethodTest {
```

```
  // static method and inheritance
  public static void main(String[] args) {
    Parent p = new Child();
    p.staticMethod(); // prints Inside Parent
    ((Child) p).staticMethod(); // prints Inside Child
  }

  static class Parent {
    public static void staticMethod() {
      System.out.println("Inside Parent");
    }
  }

  static class Child extends Parent {
    public static void staticMethod() {
      System.out.println("Inside Child");
    }
  }
}
```

Статические методы привязываются к классу не к экземпляру, и привязка этого метода происходит во время компиляции. Так как в первом обращении к `staticMethod()` , родительская ссылка на класс `p` был использован, `Parent` версия «x `staticMethod()` вызывается. Во втором случае, мы клали `p` в `Child` класс, `Child` «s `staticMethod()` выполняются.

Переменная затенение

Переменные SHADOWED и методы OVERRIDDEN. Какая переменная будет использоваться, зависит от класса, объявленного переменной. Какой метод будет использоваться, зависит от фактического класса объекта, на который ссылается переменная.

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {

    }


    public static void main(String[] args) {
```

```
        Car car = new SportsCar();
        System.out.println(car.gearRatio + "  " + car.accelerate());
        // will print out 8  Accelerate : SportsCar
    }
}
```

Сужение и расширение ссылок на объекты

Передача экземпляра базового класса в подкласс, как в: `b = (B) a;` называется *сужением* (поскольку вы пытаетесь сузить объект базового класса до более конкретного объекта класса) и нуждается в явном типе.

Передача экземпляра подкласса базовому классу, как в: `A a = b;` называется *расширением* и не нуждается в типе.

Для иллюстрации рассмотрим следующие объявления классов и тестовый код:

```
class Vehicle {
}

class Car extends Vehicle {
}

class Truck extends Vehicle {
}

class MotorCycle extends Vehicle {
}

class Test {

    public static void main(String[] args) {

        Vehicle vehicle = new Car();
        Car car = new Car();

        vehicle = car; // is valid, no cast needed

        Car c = vehicle // not valid
        Car c = (Car) vehicle; //valid
    }
}
```

Заявление `Vehicle vehicle = new Car();` является допустимым оператором Java. Каждый экземпляр `Car` также является `Vehicle` . Следовательно, назначение является законным без необходимости явного приведения типов.

С другой стороны, `Car c = vehicle;` не является действительным. Статическим типом `vehicle` является `Vehicle` средство, которое означает, что оно может ссылаться на экземпляр `Car` , грузовика , мотоцикла , `or any other current or future subclass of` транспортного средства . `(Or indeed, an instance of` автомобиля `itself, since we did not declare it as an` абстрактным `class.) The assignment cannot be allowed, since that might lead to` автомобиль `referring to a`

экземпляр Truck.

Чтобы предотвратить эту ситуацию, нам нужно добавить явный тип-cast:

```
Car c = (Car) vehicle;
```

Тип-cast говорит компилятору, что мы *ожидаем,* что стоимость `vehicle` будет `Car` или подклассом `Car` . При необходимости компилятор вставляет код для выполнения проверки типа времени выполнения. Если проверка `ClassCastException` неудачей, тогда при выполнении `ClassCastException` будет `ClassCastException` .

Обратите внимание, что не все титровальные выражения действительны. Например:

```
String s = (String) vehicle;  // not valid
```

Компилятор Java знает, что экземпляр типа, совместимый с `Vehicle` *никогда не может быть* совместимым со `String` . Тип-литье никогда не будет успешным, и JLS обязывает, что это дает ошибку компиляции.

Программирование на интерфейс

Идея программирования для интерфейса состоит в том, чтобы основывать код в основном на интерфейсах и использовать только конкретные классы во время создания экземпляра. В этом контексте хороший код, касающийся, например, наборов Java, будет выглядеть примерно так (не то, что сам метод вообще никому не нужен, просто иллюстрация):

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

в то время как плохой код может выглядеть так:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

Его результаты могут быть более совместимы с кодом, предоставляемым другими разработчиками, которые в целом придерживаются концепции программирования для интерфейса. Однако наиболее важными причинами использования первых являются:

- в большинстве случаев контекст, в котором используется результат, не требует и не нуждается в том, чтобы многие детали были представлены в конкретной реализации;
- присоединение к интерфейсу создает более чистый код и меньше хаков, таких как еще один общедоступный метод, добавляется в класс, обслуживающий определенный сценарий;
- код более подвержен тестированию, так как интерфейсы легко макетируются;

- наконец, концепция помогает, даже если ожидается только одна реализация (по крайней мере, для проверки).

Итак, как можно легко применить концепцию программирования к интерфейсу при написании нового кода, имея в виду одну конкретную реализацию? Один из вариантов, который мы обычно используем, представляет собой комбинацию следующих шаблонов:

- программирование на интерфейс
- завод
- строитель

Следующий пример, основанный на этих принципах, представляет собой упрощенную и усеченную версию реализации RPC, написанную для нескольких разных протоколов:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

Вышеупомянутый интерфейс не должен создаваться непосредственно через фабрику, вместо этого мы выводим еще более конкретные интерфейсы: один для HTTP-вызова и один для AMQP, каждый из которых имеет фабрику и строитель для создания экземпляров, которые, в свою очередь, также являются экземплярами приведенный выше интерфейс:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Экземпляры `RemoteInvoker` для использования с AMQP теперь могут быть сконструированы так легко, как (или более задействованы в зависимости от компоновщика):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

И вызов запроса так же просто, как:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Из-за того, что Java 8 разрешает размещение статических методов непосредственно в интерфейсах, промежуточный завод становится неявным в вышеупомянутом коде, замененном на `AmqpInvoker.with()` . В Java до версии 8 такой же эффект может быть достигнут с помощью внутреннего класса `Factory` :

```
public interface AmqpInvoker extends RemoteInvoker {
```

```
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
}
```

Соответствующая инстанция затем превратится в:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Строитель, используемый выше, может выглядеть так (хотя это упрощение, так как фактическое позволяет определить до 15 параметров, отклоняющихся от значений по умолчанию). Обратите внимание, что конструкция не является общедоступной, поэтому ее можно использовать только с вышеупомянутого интерфейса `AmqpInvoker` :

```
public class AmqpInvokerBuilder {
  ...
  AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {
    this.instanceId = instanceId;
    this.factory = factory;
  }

  public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {
    this.requestRouter = requestRouter;
    return this;
  }

  public AmqpInvoker build() throws TimeoutException, IOException {
    return new AmqpInvokerImpl(instanceId, factory, requestRouter);
  }
}
```

Как правило, построитель также может быть сгенерирован с использованием инструмента, такого как FreeBuilder.

Наконец, стандартная (и единственная ожидаемая) реализация этого интерфейса определяется как локальный класс пакета для обеспечения использования интерфейса, фабрики и строителя:

```
class AmqpInvokerImpl implements AmqpInvoker {
  AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
    ...
  }

  @Override
  public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
    ...
  }
}
```

Между тем, эта модель оказалась очень эффективной при разработке всего нашего нового кода, независимо от того, насколько проста или сложна функциональность.

Абстрактный класс и использование интерфейса: отношение «Is-a» vs «Has-a»

Когда использовать абстрактные классы: реализовать одно и то же или другое поведение среди нескольких связанных объектов

Когда использовать интерфейсы: реализовать контракт несколькими несвязанными объектами

Абстрактные классы создают «это» отношения, в то время как интерфейсы предоставляют «возможность».

Это можно увидеть в приведенном ниже коде:

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){

        Dog dog = new Dog("Jack",16);
        Cat cat = new Cat("Joe",20);

        System.out.println("Dog:"+dog);
        System.out.println("Cat:"+cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra",40);
        System.out.println(man);

        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name,int lifeExpentency ){
```

```
        this.name = name;
        this.lifeExpentency=lifeExpentency;
    }
    public abstract void remember();
    public abstract void protectOwner();

    public String toString(){
        return this.getClass().getSimpleName()+":"+name+":"+lifeExpentency;
    }
}
class Dog extends Animal implements Learn{

    public Dog(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName()+" can remember for 5 minutes");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " will protect owner");
    }
    public void learn(){
        System.out.println(this.getClass().getSimpleName()+ " can learn:");
    }
}
class Cat extends Animal implements Climb {
    public Cat(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName() + " can remember for 16 hours");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " won't protect owner");
    }
    public void climb(){
        System.out.println(this.getClass().getSimpleName()+ " can climb");
    }
}
interface Climb{
    void climb();
}
interface Think {
    void think();
}

interface Learn {
    void learn();
}
interface Apply{
    void apply();
}

class Man implements Think,Learn,Apply,Climb{
    String name;
    int age;

    public Man(String name,int age){
        this.name = name;
        this.age = age;
    }
```

```
    public void think(){
        System.out.println("I can think:"+this.getClass().getSimpleName());
    }
    public void learn(){
        System.out.println("I can learn:"+this.getClass().getSimpleName());
    }
    public void apply(){
        System.out.println("I can apply:"+this.getClass().getSimpleName());
    }
    public void climb(){
        System.out.println("I can climb:"+this.getClass().getSimpleName());
    }
    public String toString(){
        return "Man :"+name+":Age:"+age;
    }
}
```

выход:

```
Dog:Dog:Jack:16
Cat:Cat:Joe:20
Dog can remember for 5 minutes
Dog will protect owner
Dog can learn:
Cat can remember for 16 hours
Cat won't protect owner
Cat can climb
Man :Ravindra:Age:40
I can climb:Man
I can think:Man
I can learn:Man
I can apply:Man
```

Ключевые заметки:

1. `Animal` - это абстрактный класс с общими атрибутами: `name` и жизнь. `lifeExpectancy` и абстрактные методы: `remember()` и `protectOwner()` . `Dog` и `Cat` - это `Animals` , которые внедрили методы `remember()` и `protectOwner()` .

2. `Cat` может `climb()` но `Dog` не может. `Dog` может `think()` но `Cat` не может. Эти специфические возможности добавляются в `Cat` и `Dog` путем реализации.

3. `Man` не `Animal` но он может `Think` , `Learn` , `Apply` и `Climb` .

4. `Cat` не `Man` но он может `Climb` .

5. `Dog` не `Man` но она может `Learn`

6. `Man` является ни `Cat` ни `Dog` но может иметь некоторые из возможностей последних двух, не расширяя `Animal` , `Cat` или `Dog` . Это делается с интерфейсами.

7. Несмотря на то, что `Animal` является абстрактным классом, он имеет конструктор, в отличие от интерфейса.

TL; DR:

Несвязанные классы могут иметь возможности через интерфейсы, но связанные классы меняют поведение посредством расширения базовых классов.

Обратитесь к странице документации Java, чтобы понять, какой из них следует использовать в конкретном случае использования.

Рассмотрим использование абстрактных классов, если ...

1. Вы хотите поделиться кодом между несколькими тесно связанными классами.
2. Вы ожидаете, что классы, которые расширяют ваш абстрактный класс, имеют много общих методов или полей или требуют модификаторов доступа, кроме публичных (например, защищенных и приватных).
3. Вы хотите объявить нестатические или нефинальные поля.

Рассмотрите возможность использования интерфейсов, если ...

1. Вы ожидаете, что не связанные классы будут реализовывать ваш интерфейс. Например, многие несвязанные объекты могут реализовать интерфейс `Serializable` .
2. Вы хотите указать поведение конкретного типа данных, но не обеспокоены тем, кто реализует его поведение.
3. Вы хотите использовать множественное наследование типа.

Перекрытие в наследовании

Overriding in Inheritance используется, когда вы используете уже определенный метод из суперкласса в подклассе, но по-другому, чем то, как метод был первоначально разработан в суперклассе. Переопределение позволяет пользователю повторно использовать код, используя существующий материал и модифицируя его в соответствии с потребностями пользователя.

В следующем примере показано, как `ClassB` переопределяет функциональность `ClassA` , изменяя то, что отправляется через метод печати:

Пример:

```
public static void main(String[] args) {
    ClassA a = new ClassA();
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
```

```
}

class ClassB extends ClassA {
    public void printing() {
         System.out.println("B");
    }
}
```

Выход:

B

Прочитайте наследование онлайн: https://riptutorial.com/ru/java/topic/87/наследование

глава 117: Настройка производительности Java

Examples

Общий подход

В Интернете собраны советы по повышению производительности программ Java. Возможно, в первую очередь это осознание. Это означает:

- Определите возможные проблемы производительности и узкие места.
- Используйте инструменты для анализа и тестирования.
- Знать хорошие практики и плохие практики.

Первый момент должен быть сделан на этапе проектирования, если говорить о новой системе или модуле. Если говорить о устаревших кодах, то на экране появляются инструменты анализа и тестирования. Самый простой инструмент для анализа производительности JVM - это JVisualVM, который включен в JDK.

Третий момент - это, в основном, опыт и обширные исследования, и, конечно же, сырые подсказки, которые будут отображаться на этой странице и другие, как это .

Уменьшение количества строк

В Java слишком «легко» создать множество экземпляров String, которые не нужны. Это и другие причины могут привести к тому, что ваша программа будет иметь множество строк, которые GC занят очисткой.

Некоторые способы создания экземпляров String:

```
myString += "foo";
```

Или, что еще хуже, в цикле или рекурсии:

```
for (int i = 0; i < N; i++) {
    myString += "foo" + i;
}
```

Проблема в том, что каждый `+` создает новую строку (обычно, поскольку новые компиляторы оптимизируют некоторые случаи). Возможную оптимизацию можно выполнить с помощью `StringBuilder` или `StringBuffer` :

```
StringBuffer sb = new StringBuffer(myString);
```

```
for (int i = 0; i < N; i++) {
    sb.append("foo").append(i);
}
myString = sb.toString();
```

Если вы часто строите длинные строки (например, SQL), используйте API-интерфейс String.

Другие вещи, которые необходимо учитывать:

- Уменьшите использование `replace` , `substring` и т. Д.
- Избегайте `String.toArray()` , особенно в часто `String.toArray()` коде.
- Журнальные отпечатки, которые предназначены для фильтрации (из-за уровня журнала, например), не должны генерироваться (уровень журнала должен быть проверен заранее).
- Используйте библиотеки , как это в случае необходимости.
- StringBuilder лучше, если переменная используется не общим способом (по потокам).

Основанный на доказательствах подход к настройке производительности Java

Дональд Кнут часто цитирует это:

> «Программисты тратят огромное количество времени, думая о скорости некритических частей своих программ или беспокоясь о них, и эти попытки эффективности действительно оказывают сильное негативное влияние при отладке и обслуживании. *Мы должны забыть о небольшой эффективности, например о 97% времени* : преждевременная оптимизация - это корень всего зла, но мы не должны упускать наши возможности в этих критических 3% ».

ИСТОЧНИК

Принимая во внимание этот совет мудреца, вот рекомендуемая процедура оптимизации программ:

1. Прежде всего, создавайте и кодируйте свою программу или библиотеку с упором на простоту и правильность. Для начала не тратьте много усилий на производительность.

2. Получите его в рабочее состояние и (в идеале) разработайте модульные тесты для ключевых частей кодовой базы.

3. Разработайте ориентир производительности на уровне приложений. Тест должен охватывать критически важные аспекты вашего приложения и должен выполнять ряд задач, которые типичны для того, как приложение будет использоваться в производстве.

4. Измерьте производительность.
5. Сравните измеренную производительность с вашими критериями того, насколько быстро приложение должно быть. (Избегайте нереалистичных, недостижимых или не поддающихся оценке критериев, таких как «как можно быстрее»).
6. Если вы выполнили критерии, STOP. Вы работаете. (Любые дополнительные усилия, вероятно, пустая трата времени.)
7. Профилируйте приложение во время выполнения вашего теста производительности.
8. Изучите результаты профилирования и выберите самые большие (неоптимизированные) «горячие точки производительности»; т.е. разделы кода, в котором приложение, как представляется, проводит больше всего времени.
9. Проанализируйте секцию кода точки доступа, чтобы попытаться понять, почему это узкое место, и подумайте о том, как сделать это быстрее.
10. Реализуйте это как предлагаемое изменение кода, проверку и отладку.
11. Повторите тест, чтобы узнать, улучшилось ли изменение кода:
    - Если «Да», вернитесь к шагу 4.
    - Если нет, отмените изменение и вернитесь к шагу 9. Если вы не достигли прогресса, выберите другую точку доступа для вашего внимания.

В конце концов вы доберетесь до точки, где приложение либо достаточно быстро, либо вы рассмотрели все важные горячие точки. На этом этапе вам нужно остановить этот подход. Если часть кода потребляет (скажем) 1% от общего времени, то даже 50% -ное улучшение только сделает приложение на 0,5% быстрее в целом.

Ясно, что есть точка, за которой оптимизация точек доступа является пустой тратой усилий. Если вы доберетесь до этого момента, вам нужно принять более радикальный подход. Например:

- Посмотрите на алгоритмическую сложность ваших основных алгоритмов.
- Если приложение тратит много времени на сборку мусора, ищите способы уменьшить скорость создания объекта.
- Если ключевые части приложения имеют интенсивность процессора и однопоточность, ищите возможности для параллелизма.
- Если приложение уже многопоточное, обратите внимание на узкие места для параллелизма.

Но везде, где это возможно, полагайтесь на инструменты и измерения, а не на интуицию, чтобы направлять ваши усилия по оптимизации.

Прочитайте Настройка производительности Java онлайн:

https://riptutorial.com/ru/java/topic/4160/настройка-производительности-java

глава 118: Неизменяемые объекты

замечания

Неизменяемые объекты имеют фиксированное состояние (без сеттеров), поэтому все состояние должно быть известно во время создания объекта.

Хотя это технически не требуется, лучше всего сделать все поля `final` . Это сделает безопасный класс потокобезопасным (см. Java Concurrency in Practice, 3.4.1).

В примерах показаны несколько шаблонов, которые могут помочь в достижении этого.

Examples

Создание неизменяемой версии типа с использованием защитного копирования.

Некоторые базовые типы и классы в Java существенно изменяемы. Например, все типы массивов являются изменяемыми, а также такими классами, как `java.util.Data` . Это может быть неудобно в ситуациях, когда требуется неизменный тип.

Один из способов борьбы с этим - создать неизменяемую оболочку для изменяемого типа. Вот простая оболочка для массива целых чисел

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

Этот класс работает с помощью *защитного копирования,* чтобы изолировать изменяемое состояние (`int[]`) от любого кода, который может его изменить:

- Конструктор использует `clone()` для создания отдельной копии массива параметров. Если вызывающий объект конструктора впоследствии изменил массив параметров, это не повлияет на состояние объекта `ImmutableIntArray` .
- Метод `getValue()` также использует `clone()` для создания возвращаемого массива. Если вызывающий абонент должен был изменить массив результатов, это не повлияет на состояние объекта `ImmutableIntArray` .

Мы могли бы также добавить методы для `ImmutableIntArray` для выполнения операций только для чтения на обернутом массиве; например, получить его длину, получить значение по определенному индексу и т. д.

Обратите внимание, что неизменяемый тип-оболочка, реализованный таким образом, не совместим с исходным типом. Вы не можете просто заменить первое для последнего.

Рецепт создания неизменяемого класса

Неизменяемым объектом является объект, состояние которого не может быть изменено. Неизменяемый класс - это класс, экземпляры которого неизменны по дизайну и реализации. Класс Java, который чаще всего представлен в качестве примера неизменяемости, - это java.lang.String .

Ниже приведен стереотипный пример:

```
public final class Person {
    private final String name;
    private final String ssn;     // (SSN == social security number)

    public Person(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }

    public String getName() {
        return name;
    }

    public String getSSN() {
        return ssn;
    }
}
```

Вариант этого заключается в том, чтобы объявить конструктор как `private` и вместо этого предоставить `public static` заводский метод.

Стандартный рецепт неизменяемого класса выглядит следующим образом:

- Все свойства должны быть заданы в конструкторе (конструкторах) или заводских методах.
- Не должно быть никаких сеттеров.
- Если необходимо включить сеттеры для соображений совместимости интерфейса, они не должны ничего делать или бросать исключение.
- Все свойства должны быть объявлены как `private` и `final` .
- Для всех свойств, которые являются ссылками на изменяемые типы:
    - свойство должно быть инициализировано с глубокой копией значения, переданного через конструктор, и

    - геттер свойства должен вернуть глубокую копию значения свойства.
- Класс должен быть объявлен `final` чтобы кто-то не создал изменяемый подкласс неизменяемого класса.

Несколько других замечаний:

- Неизменяемость не препятствует обнулению объекта; например, `null` может быть присвоен переменной `String` .
- Если свойства неизменяемых классов объявляются `final` , экземпляры по сути являются потокобезопасными. Это делает неизменные классы хорошим строительным блоком для реализации многопоточных приложений.

Типичные дефекты дизайна, которые препятствуют тому, чтобы класс был непреложным

Используя некоторые сеттеры, не устанавливая все необходимые свойства в конструкторе (конструкторах)

```
public final class Person { // example of a bad immutability
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
    public String getSurname() { return surname;}
    public void setSurname(String surname) { this.surname = surname); }
}
```

Легко показать, что класс `Person` не является неизменным:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

Чтобы исправить это, просто удалите `setSurname()` и реорганизуйте конструктор следующим образом:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
```

Не указывать переменные экземпляра как частные, так и окончательные

Взгляните на следующий класс:

```
public final class Person {
    public String name;
```

```
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

Следующий фрагмент показывает, что указанный класс не является неизменным:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

Чтобы исправить это, просто пометьте свойство name как `private` и `final` .

Выявление изменчивого объекта класса в геттере

Взгляните на следующий класс:

```
import java.util.List;
import java.util.ArrayList;
public final class Names {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
    public List<String> getNames() {
        return names;
    }
    public int size() {
        return names.size();
    }
}
```

Класс `Names` кажется неизменным с первого взгляда, но это не так, как показано в следующем коде:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
Names names = new Names(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList = names.getNames();
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

Это произошло потому, что изменение ссылочного списка, возвращаемого `getNames()` может изменить фактический список `Names` .

Чтобы исправить это, просто избегайте возврата ссылок на изменяемые объекты ссылочного класса *либо* путем создания защитных копий, как показано ниже:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

или путем создания геттеров таким образом, чтобы возвращались только другие *неизменные объекты* и *примитивы* :

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Конструктор инжекции с объектами (объектами), которые могут быть изменены вне неизменяемого класса

Это вариация предыдущего недостатка. Взгляните на следующий класс:

```
import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
    public int size() {
        return names.size();
    }
}
```

В качестве класса `Names` ранее класс `NewNames` кажется неизменным с первого взгляда, но это не так, на самом деле следующий фрагмент доказывает обратное:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

Чтобы исправить это, как и в предыдущем изъяне, просто сделайте защитные копии объекта, не присваивая его непосредственно неизменному классу, т. Е. Конструктор можно изменить следующим образом:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Предотвращение переопределения методов класса

Взгляните на следующий класс:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
}
```

Класс `Person` кажется неизменным с первого взгляда, но предположим, что новый подкласс `Person` определен:

```
public class MutablePerson extends Person {
    private String newName;
    public MutablePerson(String name) {
        super(name);
    }
    @Override
    public String getName() {
        return newName;
    }
    public void setName(String name) {
        newName = name;
    }
}
```

теперь изменчивость `Person` (im) может быть использована посредством полиморфизма с использованием нового подкласса:

```
Person person = new MutablePerson("Average Joe");
System.out.println(person.getName()); prints Average Joe
person.setName("Magic Mike"); // NOT OK, person has now a new name!
System.out.println(person.getName()); // prints Magic Mike
```

Чтобы исправить это, *либо* пометьте класс как `final` чтобы он не мог быть расширен *или* объявить все его конструкторы (-и) как `private` .

Прочитайте Неизменяемые объекты онлайн: https://riptutorial.com/ru/java/topic/2807/неизменяемые-объекты

глава 119: Неизменяемый класс

Вступление

Неизменяемыми объектами являются экземпляры, состояние которых не изменяется после его инициализации. Например, String является неизменяемым классом, и после его создания значение никогда не изменяется.

замечания

Некоторые непреложные классы в Java:

1. java.lang.String
2. Классы-оболочки для примитивных типов: java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Boolean, java.lang.Long, java.lang.Double, java.lang.Float
3. Большинство классов перечисления неизменяемы, но это фактически зависит от конкретного случая.
4. java.math.BigInteger и java.math.BigDecimal (по крайней мере, объекты самих этих классов)
5. java.io.File. Обратите внимание, что это представляет объект, внешний по отношению к виртуальной машине (файл в локальной системе), который может или не может существовать, и имеет некоторые методы, изменяющие и запрашивающие состояние этого внешнего объекта. Но сам объект File остается неизменным.

Examples

Правила определения неизменяемых классов

Следующие правила определяют простую стратегию создания неизменяемых объектов.

1. Не предоставляйте методы «setter» - методы, которые изменяют поля или объекты, на которые ссылаются поля.
2. Сделайте все поля окончательными и частными.
3. Не допускайте переопределение подклассов. Самый простой способ сделать это - объявить класс окончательным. Более сложный подход заключается в том, чтобы сделать конструктор частным и построить экземпляры в заводских методах.
4. Если поля экземпляра включают ссылки на изменяемые объекты, не разрешайте изменять эти объекты:
5. Не предоставляйте методы, изменяющие изменяемые объекты.
6. Не используйте ссылки на изменяемые объекты. Никогда не храните ссылки на внешние, изменяемые объекты, переданные конструктору; при необходимости,

создавать копии и хранить ссылки на копии. Аналогичным образом создайте копии своих внутренних изменяемых объектов, когда это необходимо, чтобы избежать возврата оригиналов в ваши методы.

Пример без изменяемых ссылок

```
public final class Color {
    final private int red;
    final private int green;
    final private int blue;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public Color(int red, int green, int blue) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public Color invert() {
        return new Color(255 - red, 255 - green, 255 - blue);
    }
}
```

Пример с mutable refs

В этом случае класс Point изменен, и некоторый пользователь может изменить состояние объекта этого класса.

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
```

```
    }
}

//...

public final class ImmutableCircle {
    private final Point center;
    private final double radius;

    public ImmutableCircle(Point center, double radius) {
        // we create new object here because it shouldn't be changed
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
```

В чем преимущество неизменности?

Преимущество непреложности заключается в параллелизме. Трудно поддерживать правильность в изменяемых объектах, поскольку несколько потоков могут пытаться изменить состояние одного и того же объекта, приводя к тому, что некоторые потоки видят другое состояние одного и того же объекта, в зависимости от времени чтения и записи в упомянутый объект.

Имея неизменный объект, можно гарантировать, что все потоки, которые смотрят на объект, будут видеть одно и то же состояние, так как состояние неизменяемого объекта не изменится.

Прочитайте Неизменяемый класс онлайн: https://riptutorial.com/ru/java/topic/10561/неизменяемый-класс

глава 120: Необязательный

Вступление

`Optional` - это контейнерный объект, который может содержать или не содержать ненулевое значение. Если значение присутствует, `isPresent()` вернет `true` и `get()` вернет значение.

`orElse()` дополнительные методы, которые зависят от наличия содержащегося значения, например `orElse()` , который возвращает значение по умолчанию, если значение отсутствует, и `ifPresent()` который выполняет блок кода, если это значение присутствует.

Синтаксис

- Необязательный.путь () // Создает пустой Необязательный экземпляр.
- Необязательно. (Значение) // Возвращает необязательный параметр с указанным ненулевым значением. Исключение NullPointerException будет выбрано, если переданное значение равно null.
- Необязательный.ofNullable (значение) // Возвращает необязательный параметр с указанным значением, которое может быть нулевым.

Examples

Возврат значения по умолчанию, если Необязательный пуст

Не используйте параметр `Optional.get()` поскольку это может `NoSuchElementException` . Способы `Optional.orElse(T)` и `Optional.orElseGet(Supplier<? extends T>)` обеспечивают способ предоставления значения по умолчанию в случае, если Опция пуста.

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// returns "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// returns "something" (never calls the getDefaultValue() method)
```

```
String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// returns "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// calls getDefaultValue() and returns its results
```

`orElse` различие между `orElse` и `orElseGet` заключается в том, что последнее оценивается только тогда, когда параметр Необязательный пуст, в то время как аргумент, предоставленный первому, оценивается, даже если опция не является пустой. Поэтому `orElse` следует использовать только для констант и никогда не предоставлять значение, основанное на каких-либо вычислениях.

карта

Используйте метод `map()` для параметра « `Optional` для работы со значениями, которые могут быть `null` без явных проверок `null` :

(Обратите внимание, что операции `map()` и `filter()` вычисляются сразу же, в отличие от их экземпляров Stream, которые оцениваются только при *операции терминала* .)

Синтаксис:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Примеры кода:

```
String value = null;

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "NONE"
```

```
String value = "something";

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "SOMETHING"
```

Поскольку Option.map () возвращает пустую опцию, когда ее функция отображения возвращает значение null, вы можете связать несколько операций map () как форму нуль-безопасного разыменования. Это также известно как **надежная цепочка** .

Рассмотрим следующий пример:

```
String value = foo.getBar().getBaz().toString();
```

Любой из `getBar` , `getBaz` и `toString` потенциально может генерировать `getBaz` `NullPointerException` .

Вот альтернативный способ получить значение `toString()` используя `Optional` :

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

Это вернет пустую строку, если какая-либо из функций отображения вернет null.

Ниже приведен еще один пример, но немного другой. Он напечатает значение только в том случае, если ни одна из функций сопоставления не вернула значение null.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Выбросьте исключение, если нет значения

Используйте метод `orElseThrow()` `Optional` чтобы получить содержащееся значение или выбросить исключение, если оно не было установлено. Это похоже на вызов `get()` , за исключением того, что он допускает произвольные типы исключений. Метод берет поставщика, который должен вернуть исключение, которое нужно выбросить.

В первом примере метод просто возвращает содержащееся значение:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

Во втором примере метод генерирует исключение, потому что значение не задано:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

Вы также можете использовать синтаксис лямбда, если требуется выброс исключения с сообщением:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Фильтр

`filter()` используется, чтобы указать, что вы хотите получить значение *только в том случае, если* оно соответствует вашему предикату.

Подумайте об этом, как `if (!somePredicate(x)) { x = null; }` .

Примеры кода:

```
String value = null;
Optional.ofNullable(value) // nothing
```

```
        .filter(x -> x.equals("cool string"))// this is never run since value is null
        .isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and passes
        .isPresent(); // true
```

```
String value = "hot string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and fails
        .isPresent(); // false
```

Использование дополнительных контейнеров для примитивных типов номеров

`OptionalDouble` , `OptionalInt` и `OptionalLong` работают как `Optional` , но специально предназначены для переноса примитивных типов:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Поскольку числовые типы имеют значение, для null нет специальной обработки. Пустые контейнеры можно проверить с помощью:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

Точно так же существуют сокращения для управления стоимостью:

```
// Prints the value since it is provided on creation.
presentInt.ifPresent(System.out::println);

// Gives the other value as the original Optional is empty.
int finalValue = absentInt.orElseGet(this::otherValue);

// Will throw a NoSuchElementException.
int nonexistentValue = absentInt.getAsInt();
```

Выполнить код, только если имеется значение

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println);//Prints "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println);//Does nothing.
```

Ленько предоставлять значение по умолчанию с помощью поставщика

Нормальный `orElse` метод принимает `Object` , так что вы могли бы задаться вопросом, почему есть возможность предоставить `Supplier` здесь (`orElseGet` метод).

Рассматривать:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

Он все равно вызовет `getValueThatIsHardToCalculate()` хотя результат не используется, поскольку необязательный не пуст.

Чтобы избежать этого наказания, вы предоставляете поставщика:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

Таким образом `getValueThatIsHardToCalculate()` будет вызываться только в том случае, если параметр `Optional` пуст.

FlatMap

`flatMap` похож на `map` . Разница описывается javadoc следующим образом:

> Этот метод похож на `map(Function)` , но предоставленный сопоставитель - это тот, результат которого уже является `Optional` , и если вызывается, `flatMap` не обертывает его дополнительным `Optional` .

Другими словами, когда вы связываете вызов метода, который возвращает `Optional` , использование `Optional.flatMap` позволяет избежать создания вложенных `Optionals` .

Например, учитывая следующие классы:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Если вы используете `Optional.map` , вы получите вложенный `Optional` ; т.е. `Optional<Optional<Bar>>` .

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

Однако, если вы используете `Optional.flatMap` , вы получите простой `Optional` ; т.е. `Optional<Bar>` .

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Прочитайте Необязательный онлайн: https://riptutorial.com/ru/java/topic/152/необязательный

глава 121: Новый ввод-вывод файлов

Синтаксис

- Paths.get (String first, String ... more) // Создает экземпляр Path своими элементами String
- Paths.get (URI uri) // Создает экземпляр пути с помощью URI

Examples

Создание путей

Класс `Path` используется для программного представления пути в файловой системе (и поэтому может указывать на файлы, а также на каталоги, даже на несуществующие)

Путь может быть получен с использованием класса-помощника `Paths` :

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

Получение информации о пути

Информацию о пути можно получить с помощью методов объекта `Path` :

- `toString()` возвращает строковое представление пути

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- `getFileName()` возвращает имя файла (или, более конкретно, последний элемент пути

  ```
  Path p1 = Paths.get("/var/www"); // p1.getFileName() returns "www"
  Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); // p3.getFileName()
  returns "HHGTDG.odt"
  ```

- `getNameCount()` возвращает количество элементов, которые образуют путь

  ```
  Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
  ```

- `getName(int index)` возвращает элемент по данному индексу

  ```
  Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
  "www"
  ```

- `getParent()` возвращает путь родительского каталога

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` возвращает корень пути

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() returns "/"
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); //
p3.getRoot().toString() returns "C:\\"
```

Пути манипулирования

Присоединение к двум путям

Пути можно объединить с помощью метода `resolve()` . Прошедший путь должен быть частичным путем, который представляет собой путь, который не включает корневой элемент.

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");
```

```
joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

Нормализация пути

Пути могут содержать элементы `.` (который указывает на каталог, в котором вы сейчас находитесь) и `..` (что указывает на родительский каталог).

При использовании в пути, `.` может быть удален в любое время , без изменения назначения по пути, и `..` может быть удален вместе с предыдущим элементом.

С API-интерфейсом Paths это выполняется с использованием `.normalize()` :

```
Path p7 = Paths.get("/home/./arthur/../ford/files");
Path p8 = Paths.get("C:\\Users\\.\\..\\Program Files");
```

```
p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

Получение информации с использованием файловой системы

Чтобы взаимодействовать с файловой системой, вы используете методы класса `Files` .

Проверка существования

Чтобы проверить наличие файла или каталога, на который указывает путь, вы используете следующие методы:

```
Files.exists(Path path)
```

а также

```
Files.notExists(Path path)
```

`!Files.exists(path)` необязательно должен быть равен `Files.notExists(path)` , поскольку существует три возможных сценария:

- Существование файла или каталога проверено (`exists` возврат `true` а `notExists` возвращает `false` в этом случае)
- Неизвестность файла или каталога verfied (`exists` возвращает `false` а `notExists` возвращает `true`)
- Ни существование, ни отсутствие файла или каталога не могут быть проверены (например, из-за ограничений доступа): Both `exists` и `nonExists` возвращают false.

Проверка того, указывает ли путь к файлу или каталогу

Это делается с использованием `Files.isDirectory(Path path)` и `Files.isRegularFile(Path path)`

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get("/home/testuser/File.txt");
```

```
Files.isDirectory(p1) == true
Files.isRegularFile(p1) == false

Files.isDirectory(p2) == false
Files.isRegularFile(p2) == true
```

Получение свойств

Это можно сделать, используя следующие методы:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

Получение типа MIME

```
Files.probeContentType(Path path)
```

Это пытается получить MIME-тип файла. Он возвращает строку типа MIME, например:

- `text/plain` для текстовых файлов
- `text/html` для HTML-страниц
- `application/pdf` для файлов PDF
- `image/png` для файлов PNG

Чтение файлов

Файлы можно читать по байтам и по линиям, используя класс « `Files` ».

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
byte[] content = Files.readAllBytes(p2);
List<String> linesOfContent = Files.readAllLines(p2);
```

`Files.readAllLines()` необязательно принимает параметр charset в качестве параметра (по умолчанию используется `StandardCharsets.UTF_8`):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Написание файлов

Файлы можно записывать укусом и по очереди с использованием класса « `Files` »

```
Path p2 = Paths.get("/home/testuser/File.txt");
List<String> lines = Arrays.asList(
    new String[]{"First line", "Second line", "Third line"});

Files.write(p2, lines);
```

```
Files.write(Path path, byte[] bytes)
```

Существующие файлы будут переопределены, будут созданы несуществующие файлы.

Прочитайте Новый ввод-вывод файлов онлайн: https://riptutorial.com/ru/java/topic/5519/

новый-ввод-вывод-файлов

глава 122: Обработка аргументов командной строки

Синтаксис

- public static void main (String [] args)

параметры

параметр	подробности
арг	Аргументы командной строки. Предполагая, что `main` метод вызывается Java-пусковой установкой, `args` будет не нулевым и не будет иметь `null` элементов.

замечания

Когда обычное Java-приложение запускается с использованием команды `java` (или эквивалента), вызывается `main` метод, передающий аргументы из командной строки в массиве `args` .

К сожалению, библиотеки классов Java SE не обеспечивают прямой поддержки обработки аргументов команды. Это дает вам две альтернативы:

- Реализовать обработку аргументов вручную в Java.
- Используйте стороннюю библиотеку.

Эта тема попытается охватить некоторые из более популярных сторонних библиотек. Подробный список альтернатив см. В этом ответе на вопрос StackOverflow «Как анализировать аргументы командной строки в Java?» ,

Examples

Обработка аргументов с использованием GWT ToolBase

Если вы хотите анализировать более сложные аргументы командной строки, например, с необязательными параметрами, лучше всего использовать подход GWT Google. Все классы доступны по адресу:

https://gwt.googlesource.com/gwt/+/2.8.0-

beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

Пример для обработки командной строки `myprogram -dir "~/Documents" -port 8888` является:

```
public class MyProgramHandler extends ToolBase {
   protected File dir;
   protected int port;
   // getters for dir and port
   ...

   public MyProgramHandler() {
       this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }
       });
       this.registerHandler(new ArgHandlerInt() {
            @Override
            public String[] getTagArgs() {
               return new String[]{"port"};
            }
            @Override
            public void setInt(int value) {
               this.port = value;
            }
       });
   }
   public static void main(String[] args) {
      MyProgramHandler myShell = new MyProgramHandler();
      if (myShell.processArgs(args)) {
         // main program operation
         System.out.println(String.format("port: %d; dir: %s",
            myShell.getPort(), myShell.getDir()));
      }
      System.exit(1);
   }
}
```

`ArgHandler` также имеет метод `isRequired()` , который может быть переписан , чтобы сказать , что требуется аргумент командной строки (возврат по умолчанию является `false` , так что аргумент является необязательным.

Обработка аргументов вручную

Когда синтаксис командной строки для приложения прост, разумно полностью выполнять обработку аргумента команды в пользовательском коде.

В этом примере мы представим серию простых тематических исследований. В каждом случае код будет выдавать сообщения об ошибках, если аргументы неприемлемы, а затем вызвать `System.exit(1)` чтобы сообщить оболочке, что команда не выполнена. (Мы будем предполагать в каждом случае, что Java-код вызывается с помощью оболочки, имя которой «myapp».)

Команда без аргументов

В данном случае команда не требует аргументов. Код показывает, что `args.length` дает нам количество аргументов командной строки.

```
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.err.println("usage: myapp");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked");
    }
}
```

Команда с двумя аргументами

В этом случае-исследовании команда требует только двух аргументов.

```
public class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: myapp <arg1> <arg2>");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked: " + args[0] + ", " + args[1]);
    }
}
```

Обратите внимание, что если мы пренебрегли проверкой `args.length` , команда выйдет из строя, если пользователь запустит ее с слишком небольшим количеством аргументов командной строки.

Команда с параметрами «флаг» и, по крайней мере, один аргумент

В данном случае команда имеет пару (необязательных) опций флага и требует по крайней мере одного аргумента после опций.

```
package tommy;
public class Main {
    public static void main(String[] args) {
        boolean feelMe = false;
        boolean seeMe = false;
        int index;
        loop: for (index = 0; index < args.length; index++) {
            String opt = args[index];
```

```
            switch (opt) {
            case "-c":
               seeMe = true;
               break;
            case "-f":
               feelMe = true;
               break;
            default:
               if (!opts.isEmpty() && opts.charAt(0) == '-') {
                   error("Unknown option: '" + opt + "'");
               }
               break loop;
            }
        }
        if (index >= args.length) {
            error("Missing argument(s)");
        }

        // Run the application
        // ...
    }

    private static void error(String message) {
        if (message != null) {
            System.err.println(message);
        }
        System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
        System.exit(1);
   }
}
```

Как вы можете видеть, обработка аргументов и опций становится довольно громоздкой, если синтаксис команды является сложным. Целесообразно использовать библиотеку «синтаксический анализ командной строки»; см. другие примеры.

Прочитайте Обработка аргументов командной строки онлайн:
https://riptutorial.com/ru/java/topic/4775/обработка-аргументов-командной-строки

глава 123: Общие ошибки Java

Вступление

В этом разделе описываются некоторые распространенные ошибки, допущенные новичками на Java.

Это включает в себя любые распространенные ошибки в использовании языка Java или понимание среды выполнения.

Ошибки, связанные с конкретными API-интерфейсами, могут быть описаны в разделах, относящихся к этим API. Строки - это особый случай; они описаны в Спецификации языка Java. Подробности, отличные от распространенных ошибок, можно описать в этом разделе на строках .

Examples

Pitfall: использование == для сравнения объектов примитивных оберток, таких как Integer

(Эта ошибка относится одинаково ко всем примитивным типам обертки, но мы проиллюстрируем ее для `Integer` и `int` .)

При работе с объектами `Integer` возникает соблазн использовать `==` для сравнения значений, потому что это то, что вы бы сделали с значениями `int` . И в некоторых случаях это будет работать:

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));          // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));   // true
```

Здесь мы создали два объекта `Integer` со значением `1` и сравним их (в этом случае мы создали один из `String` и один из `int` literal. Существуют и другие альтернативы). Кроме того, мы видим, что оба метода сравнения (`==` и `equals`) дают `true` .

Такое поведение меняется, когда мы выбираем разные значения:

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));   // true
```

В этом случае, только `equals` сравнение дает правильный результат.

Причиной этого различия в поведении является то, что JVM поддерживает кеш объектов `Integer` для диапазона от -128 до 127. (Верхнее значение может быть переопределено системным свойством «java.lang.Integer.IntegerCache.high» или JVM-аргумент "-XX: AutoBoxCacheMax = размер"). Для значений в этом диапазоне `Integer.valueOf()` вернет кешированное значение, а не создает новый.

Таким образом, в первом примере `Integer.valueOf(1)` и `Integer.valueOf("1")` возвращают тот же кешированный экземпляр `Integer` . Напротив, во втором примере `Integer.valueOf(1000)` и `Integer.valueOf("1000")` создали и вернули новые объекты `Integer` .

Оператор `==` для эталонных типов тестов для ссылочного равенства (т. Е. Одного и того же объекта). Поэтому в первом примере `int1_1 == int1_2 true` потому что ссылки одинаковы. Во втором примере `int2_1 == int2_2` является ложным, потому что ссылки разные.

Pitfall: забыв о свободных ресурсах

Каждый раз, когда программа открывает ресурс, такой как файл или сетевое соединение, важно освободить ресурс, как только вы закончите его использование. Аналогичная осторожность должна быть предпринята, если во время операций на таких ресурсах следует выбросить какие-либо исключения. Можно утверждать, что `FileInputStream` имеет финализатор, который вызывает метод `close()` в событии сбора мусора; однако, поскольку мы не можем быть уверены, когда начнется цикл сбора мусора, поток ввода может потреблять компьютерные ресурсы в течение неопределенного периода времени. Ресурс должен быть закрыт в `finally` разделе блока try-catch:

Java SE 7

```
private static void printFileJava6() throws IOException {
    FileInputStream input;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Так как Java 7 действительно полезный и аккуратный оператор, введенный в Java 7, особенно для этого случая, называемый try-with-resources:

Java SE 7

```
private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

Оператор *try-with-resources* может использоваться с любым объектом, который реализует интерфейс `Closeable` или `AutoCloseable` . Он гарантирует, что каждый ресурс будет закрыт до конца инструкции. Разница между двумя интерфейсами заключается в том, что метод `close() Closeable` вызывает `Closeable IOException` которое должно быть обработано каким-то образом.

В тех случаях, когда ресурс уже открыт, но после его использования он должен быть безопасно закрыт, его можно назначить локальной переменной внутри try-with-resources

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

Локальная переменная ресурса, созданная в конструкторе try-with-resources, фактически является окончательной.

Pitfall: утечки памяти

Java автоматически управляет памятью. Вы не обязаны освобождать память вручную. Память объекта в куче может быть освобождена сборщиком мусора, когда объект больше не *доступен доступной* нитью.

Тем не менее, вы можете предотвратить освобождение памяти, позволяя объектам быть доступными, которые больше не нужны. Если вы называете это утечкой памяти или упаковкой памяти, результат будет таким же - ненужное увеличение выделенной памяти.

Утечки памяти в Java могут происходить по-разному, но наиболее распространенной причиной являются вечные ссылки на объекты, поскольку сборщик мусора не может удалить объекты из кучи, пока есть ссылки на них.

Статические поля

Можно создать такую ссылку путем определения класса со `static` полем, содержащим некоторую коллекцию объектов, и забыть установить это `static` поле в `null` после того, как сбор больше не нужен. `static` поля считаются корнями GC и никогда не собираются.

Другой проблемой является утечка в памяти без кучи при использовании JNI .

Утечка класса загрузчика

Тем не менее, самым коварным типом утечки памяти является утечка загрузчика класса. Класс loader содержит ссылку на каждый класс, который он загрузил, и каждый класс содержит ссылку на свой загрузчик классов. У каждого объекта есть ссылка на его класс. Поэтому, если даже *один* объект класса, загружаемый загрузчиком классов, не является мусором, может быть собрано не один класс, загруженный загрузчиком этого класса. Поскольку каждый класс также ссылается на его статические поля, они также не могут быть собраны.

Утечка утечки. Пример утечки утечки может выглядеть следующим образом:

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();
final BigDecimal divisor = new BigDecimal(51);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    BigDecimal number = numbers.peekLast();
    if (number != null && number.remainder(divisor).byteValue() == 0) {
        System.out.println("Number: " + number);
        System.out.println("Deque size: " + numbers.size());
    }
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

В этом примере создаются две запланированные задачи. Первая задача берет последнее число из дека, называемого `numbers` , и, если число делится на 51, оно печатает число и размер дека. Вторая задача помещает числа в deque. Обе задачи запланированы с фиксированной скоростью, и они запускаются каждые 10 мс.

Если код выполнен, вы увидите, что размер deque постоянно увеличивается. Это в конечном итоге приведет к тому, что deque будет заполнено объектами, которые потребляют всю доступную память кучи.

Чтобы предотвратить это при сохранении семантики этой программы, мы можем использовать другой метод для `pollLast` чисел из deque: `pollLast` . В отличие от метода `peekLast` , `pollLast` возвращает элемент и удаляет его из deque, в то время как `peekLast` возвращает только последний элемент.

Pitfall: использование == для сравнения строк

Общей ошибкой для начинающих Java является использование оператора `==` чтобы проверить, равны ли две строки. Например:

```
public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

Вышеупомянутая программа должна проверять первый аргумент командной строки и печатать разные сообщения, когда она не является словом «привет». Но проблема в том, что это не сработает. Эта программа выведет «Вы чувствуете себя сердитой сегодня?» независимо от того, что первый аргумент командной строки.

В этом конкретном случае `String` «hello» помещается в пул строк, в то время как `String` args [0] находится в куче. Это означает, что есть два объекта, представляющих один и тот же литерал, каждый со своей ссылкой. Поскольку `==` тесты для ссылок, а не фактическое равенство, сравнение даст ложь большую часть времени. Это не означает, что это всегда будет так.

Когда вы используете `==` для тестирования строк, то, что вы на самом деле тестируете, - это два объекта `String` - один и тот же объект Java. К сожалению, это не то, что означает равенство строк в Java. Фактически, правильным способом тестирования строк является использование метода `equals(Object)` . Для пары строк мы обычно хотим проверить, состоят ли они из одних и тех же символов в том же порядке.

```
public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

Но на самом деле это становится хуже. Проблема заключается в том, что `==` *даст* ожидаемый ответ в некоторых обстоятельствах. Например

```
public class Test1 {
```

```
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}
```

Интересно, что это напечатает «тот же», хотя мы тестируем строки неверным образом. Это почему? Поскольку спецификация языка Java (раздел 3.10.5: литералы строк) предусматривает, что любые две строки >> литералы <<, состоящие из одних и тех же символов, будут фактически представлены одним и тем же объектом Java. Следовательно, тест `==` даст истину для равных литералов. (Строковые литералы «интернированы» и добавляются в общий «пул строк», когда ваш код загружен, но это фактически деталь реализации.)

Чтобы добавить к путанице, спецификация языка Java также предусматривает, что когда у вас есть выражение постоянной времени компиляции, которое объединяет два строковых литерала, это эквивалентно одному литералу. Таким образом:

```
    public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hel" + "lo";
        String s3 = " mum";
        if (s1 == s2) {
            System.out.println("1. same");
        } else {
            System.out.println("1. different");
        }
        if (s1 + s3 == "hello mum") {
            System.out.println("2. same");
        } else {
            System.out.println("2. different");
        }
    }
}
```

Это будет выводить «1. same» и «2. different». В первом случае выражение `+` оценивается во время компиляции, и мы сравниваем один объект `String` с самим собой. Во втором случае он оценивается во время выполнения, и мы сравниваем два разных объекта `String`

Таким образом, использование `==` для тестирования строк в Java почти всегда неверно, но не гарантированно дает неправильный ответ.

Pitfall: тестирование файла перед попыткой его открыть.

Некоторые люди рекомендуют вам применять различные тесты к файлу, прежде чем

пытаться открыть его, чтобы обеспечить лучшую диагностику или избежать устранения исключений. Например, этот метод пытается проверить, соответствует ли `path` читаемому файлу:

```
public static File getValidatedFile(String path) throws IOException {
    File f = new File(path);
    if (!f.exists()) throw new IOException("Error: not found: " + path);
    if (!f.isFile()) throw new IOException("Error: Is a directory: " + path);
    if (!f.canRead()) throw new IOException("Error: cannot read file: " + path);
    return f;
}
```

Вы можете использовать вышеупомянутый метод следующим образом:

```
File f = null;
try {
    f = getValidatedFile("somefile");
} catch (IOException ex) {
    System.err.println(ex.getMessage());
    return;
}
try (InputStream is = new FileInputStream(file)) {
    // Read data etc.
}
```

Первая проблема заключается в сигнатуре для `FileInputStream(File)` потому что компилятор по-прежнему настаивает на том, что мы поймаем `IOException` здесь или дальше по стеку.

Вторая проблема заключается в том, что проверки, выполняемые методом `getValidatedFile` , не гарантируют успеха `FileInputStream` .

- Условия гонки: другой поток или отдельный процесс может переименовать файл, удалить файл или удалить доступ для чтения после возвращения `getValidatedFile` . Это приведет к « `IOException` » `IOException` без специального сообщения.
- Есть те случаи, которые не охватываются этими тестами. Например, в системе с SELinux в режиме принудительного исполнения попытка чтения файла может завершиться неудачей, несмотря на то, что `canRead()` возвращает `true` .

Третья проблема заключается в том, что тесты неэффективны. Например, `exists` вызовы `isFile` и `canRead` , каждый из которых выполняет команду syscall для выполнения требуемой проверки. Затем открывается другой syscall, чтобы открыть файл, который повторяет те же проверки за кулисами.

Короче говоря, такие методы, как `getValidatedFile` , ошибочны. Лучше просто попытаться открыть файл и обработать исключение:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
```

```
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

Если вы хотите различать ошибки ввода-вывода при открытии и чтении, вы можете использовать вложенный try / catch. Если вы хотите улучшить диагностику открытых сбоев, вы можете выполнить проверки `exists` , `isFile` и `canRead` в обработчике.

Pitfall: мышление переменных как объектов

Никакая переменная Java не представляет объект.

```
String foo;   // NOT AN OBJECT
```

Ни один Java-массив не содержит объектов.

```
String bar[] = new String[100];  // No member is an object.
```

Если вы ошибочно считаете переменные как объекты, то реальное поведение языка Java удивит вас.

- Для переменных Java, которые имеют примитивный тип (например, `int` или `float`), переменная содержит копию значения. Все копии примитивного значения неразличимы; т.е. для номера один существует только одно значение `int` . Примитивные значения не являются объектами, и они не ведут себя как объекты.
- Для переменных Java, которые имеют ссылочный тип (либо класс, либо тип массива), переменная содержит ссылку. Все копии ссылки неразличимы. Ссылки могут указывать на объекты, или они могут быть `null` что означает, что они указывают на отсутствие объекта. Однако они не являются объектами, и они не ведут себя как объекты.

Переменные не являются объектами в любом случае, и они не содержат объектов в любом случае. Они могут содержать *ссылки на объекты* , но это говорит что-то другое.

Пример класса

В следующих примерах используется этот класс, который представляет точку в 2D пространстве.

```
public final class MutableLocation {
   public int x;
   public int y;

   public MutableLocation(int x, int y) {
       this.x = x;
       this.y = y;
```

```
    }

    public boolean equals(Object other) {
        if (!(other instanceof MutableLocation) {
            return false;
        }
        MutableLocation that = (MutableLocation) other;
        return this.x == that.x && this.y == that.y;
    }
}
```

Экземпляр этого класса представляет собой объект, который имеет два поля `x` и `y` которые имеют тип `int` .

У нас может быть много экземпляров класса `MutableLocation` . Некоторые из них будут представлять одинаковые местоположения в 2D-пространстве; т.е. соответствующие значения `x` и `y` будут совпадать. Другие будут представлять разные местоположения.

Несколько переменных могут указывать на один и тот же объект

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);
```

В приведенном выше `MutableLocation` мы объявили `here` три переменные, `there` и в `elsewhere` , `elsewhere` могут храниться ссылки на объекты `MutableLocation` .

Если вы (неправильно) считаете эти переменные объектами, то вы, скорее всего, неправильно понимаете утверждения:

1. Скопируйте местоположение «[1, 2]» `here`
2. Скопируйте местоположение «[1, 2]» `there`
3. Скопируйте местоположение «[1, 2]» в `elsewhere`

Из этого вы можете сделать вывод, что у нас есть три независимых объекта в трех переменных. Фактически, *только два объекта созданы* выше. Переменные `here` и `there` действительно относятся к одному и тому же объекту.

Мы можем это продемонстрировать. Предполагая объявления переменных, как указано выше:

```
System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
```

Это выведет следующее:

```
BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1
```

Мы присвоили новое значение `here.x` и изменили значение, которое мы видим через `there.x` . Они относятся к одному и тому же объекту. Но значение, которое мы видим через `elsewhere.x` , не изменилось, поэтому в `elsewhere` должно быть ссылка на другой объект.

Если переменная была объектом, тогда присваивание `here.x = 42` не изменилось бы `there.x` . `there.x`

Оператор равенства НЕ проверяет, что два объекта равны

Применение оператора равенства (`==`) для сравнения значений значений, если значения относятся к одному и тому же объекту. Он *не* проверяет, являются ли два (разных) объекта «равными» в интуитивном смысле.

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

Это напечатает «здесь есть», но он не будет печатать «здесь где-то еще». (Ссылки `here` и в `elsewhere` предназначены для двух разных объектов.)

Напротив, если мы назовем метод `equals(Object)` который мы реализовали выше, мы будем тестировать, если два экземпляра `MutableLocation` имеют одинаковое расположение.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

Это напечатает оба сообщения. В частности, здесь `here.equals(elsewhere)` возвращает `true` потому что семантические критерии, которые мы выбрали для равенства двух объектов `MutableLocation` , были выполнены.

Вызов метода НЕ пропускает объекты вообще

Вызов метода Java использует *pass по значению* [1] для передачи аргументов и возврата результата.

Когда вы передаете ссылочное значение методу, вы фактически передаете ссылку на объект *по значению* , а это значит, что он создает копию ссылки на объект.

Пока обе ссылки на объекты все еще указывают на один и тот же объект, вы можете изменить этот объект из любой ссылки, и это то, что вызывает путаницу для некоторых.

Однако вы *не* передаете объект по ссылке [2] . Различие заключается в том, что если копия ссылки на объект модифицирована, чтобы указать на другой объект, исходная ссылка на объект все равно укажет на исходный объект.

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4);   // Point local foo at a different object.
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 1".
}
```

Также вы не передаете копию объекта.

```
void f(MutableLocation foo) {
    foo.x = 42;
}

void g() {
    MutableLocation foo = new MutableLocation(0, 0);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"
}
```

---

1 - В таких языках, как Python и Ruby, термин «pass by sharing» является предпочтительным для «pass by value» объекта / ссылки.

2 - Термин «передавать по ссылке» или «вызов по ссылке» имеет очень специфическое значение в терминологии языка программирования. Фактически это означает, что вы передаете адрес *переменной или элемента массива* , так что, когда вызываемый метод присваивает новое значение формальному аргументу, он изменяет значение в исходной переменной. Java не поддерживает это. Для более подробного описания различных механизмов передачи параметров см. Https://en.wikipedia.org/wiki/Evaluation_strategy .

Pitfall: объединение назначений и побочных эффектов

Иногда мы видим вопросы StackOverflow Java (и вопросы C или C ++), которые задают что-

то вроде этого:

```
i += a[i++] + b[i--];
```

оценивает ... для некоторых известных начальных состояний `i` , `a` и `b` .

Вообще говоря:

- для Java ответ всегда задается [1] , но неочевидно, и часто трудно понять
- для C и C ++ ответ часто не указан.

Такие примеры часто используются на экзаменах или собеседованиях в качестве попытки выяснить, действительно ли учащийся или собеседник понимает, как выражение оценки действительно работает на языке программирования Java. Это, возможно, законно, как «тест знаний», но это не значит, что вы должны когда-либо делать это в реальной программе.

Чтобы проиллюстрировать, следующий простой, казалось бы, простой пример появился несколько раз в вопросах StackOverflow (например, этот). В некоторых случаях это кажется подлинной ошибкой в чьем-то коде.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

Большинство программистов (включая экспертов Java), *быстро* читающих эти утверждения, скажут, что он выводит `2` . Фактически, он выводит `1` . Подробное объяснение причин, пожалуйста, прочитайте этот ответ .

Однако реальный вынос из этого и подобных примеров является то , что *любое* заявление Java , что *и* присваивает *и* побочные эффекты и та же переменная будет *в лучшем* случае трудно понять, а *в худшем случае* совершенно ввести в заблуждение. Вы должны избегать написания кода, подобного этому.

---

1 - по модулю потенциальных проблем с моделью памяти Java, если переменные или объекты видны другим потокам.

Pitfall: Не понимая, что String является неизменным классом

Новые программисты Java часто забывают или не могут полностью понять, что класс Java `String` неизменен. Это приводит к таким проблемам, как в следующем примере:

```
public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
```

```
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

Вышеприведенный код должен печатать аргументы командной строки в верхнем регистре. К сожалению, это не сработает, случай аргументов не изменяется. Проблема заключается в следующем:

```
s.toUpperCase();
```

Вы можете подумать, что вызов `toUpperCase()` изменит `s` на верхнюю строку. Это не так. Это не может! `String` объекты неизменяемы. Они не могут быть изменены.

На самом деле метод `toUpperCase()` *возвращает* объект `String` который является строчной версией `String` которую вы вызываете ее. Вероятно, это будет новый объект `String` , но если `s` уже был в верхнем регистре, результатом может быть существующая строка.

Поэтому, чтобы эффективно использовать этот метод, вам нужно использовать объект, возвращенный вызовом метода; например:

```
s = s.toUpperCase();
```

Фактически правило «строки никогда не изменяется» применяется ко всем методам `String` . Если вы помните это, тогда вы можете избежать ошибок целой категории начинающих.

Прочитайте Общие ошибки Java онлайн: https://riptutorial.com/ru/java/topic/4388/общие-ошибки-java

глава 124: Одиночки

Вступление

Синглтон - это класс, в котором только один экземпляр. Для получения дополнительной информации о *шаблоне проектирования* Singleton см. Тему Singleton в теге Design Patterns .

Examples

Enum Singleton

Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Enums имеют частные конструкторы, являются окончательными и обеспечивают надлежащие механизмы сериализации. Они также очень сжаты и лениво инициализированы поточно-безопасным способом.

JVM обеспечивает гарантию того, что значения перечисления не будут создаваться более одного раза каждый, что дает синтаксическому шаблону enum очень сильную защиту от атак отражения.

То, что шаблон перечисления *не* защищает от других разработчиков, физически добавляет больше элементов в исходный код. Следовательно, если вы выберете этот стиль реализации для своих синглетов, вам очень важно четко указать, что новые значения не должны добавляться к этим перечислениям.

Это рекомендуемый способ реализации одноэлементного шаблона, как объяснил Джошуа Блох в «Эффективной Java».

Резьбоволокно Синглтон с двойной проверкой блокировки

Этот тип Singleton является потокобезопасным и предотвращает ненужную блокировку после создания экземпляра Singleton.

Java SE 5

```
public class MySingleton {
```

```
    // instance of class
    private static volatile MySingleton instance = null;

    // Private constructor
    private MySingleton() {
        // Some code for constructing object
    }

    public static MySingleton getInstance() {
        MySingleton result = instance;

        //If the instance already exists, no locking is necessary
        if(result == null) {
            //The singleton instance doesn't exist, lock and check again
            synchronized(MySingleton.class) {
                result = instance;
                if(result == null) {
                    instance = result = new MySingleton();
                }
            }
        }
        return result;
    }
}
```

Следует подчеркнуть - в версиях до Java SE 5 реализация выше неверна и ее следует избегать. Невозможно реализовать двойную проверку блокировки правильно в Java до Java 5.

Singleton без использования Enum (ожидающая инициализация)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Можно утверждать, что этот пример является *фактически* ленивой инициализацией. Раздел 12.4.1 Спецификации языка Java гласит:

> Класс или тип интерфейса T будут инициализированы непосредственно перед первым вхождением любого из следующих:
>
> - T - класс, и создается экземпляр T
> - T - класс, и статический метод, объявленный T, вызывается
> - Статическое поле, объявленное T, назначается
> - Статическое поле, объявленное T, используется, и поле не является постоянной переменной

- T - класс верхнего уровня, и выполняется инструкция assert, лексически вложенная в T.

Поэтому, если в классе нет других статических полей или статических методов, экземпляр `Singleton` не будет инициализирован до тех пор, пока метод `getInstance()` будет вызван в первый раз.

Инициализация потоков с лёгкой лентой с использованием класса держателей | Реализация Билла Пью Синглтона

```
public class Singleton {
    private static class InstanceHolder {
        static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

    private Singleton() {}
}
```

Это инициализирует переменную `INSTANCE` при первом вызове `Singleton.getInstance()` , используя преимущества безопасности потоков на языке для статической инициализации без необходимости дополнительной синхронизации.

Эта реализация также известна как синглтонная модель Билла Пью. [Вики]

Расширение одноэлементного (singleton наследования)

В этом примере базовый класс `Singleton` предоставляет метод `getMessage()` который возвращает `"Hello world!"` сообщение.

Это подклассы `UppercaseSingleton` и `LowercaseSingleton` переопределяют метод getMessage (), чтобы обеспечить соответствующее представление сообщения.

```
//Yeah, we'll need reflection to pull this off.
import java.lang.reflect.*;

/*
Enumeration that represents possible classes of singleton instance.
If unknown, we'll go with base class - Singleton.
*/
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//Base class
class Singleton{
```

```
/*
Extended classes has to be private inner classes, to prevent extending them in
uncontrolled manner.
 */
private class UppercaseSingleton extends Singleton {

    private UppercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toUpperCase();
    }
}

//Another extended class.
private class LowercaseSingleton extends Singleton
{
    private LowercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toLowerCase();
    }
}

//Applying Singleton pattern
private static SingletonKind kind = SingletonKind.UNKNOWN;

private static Singleton instance;

/*
By using this method prior to getInstance() method, you effectively change the
type of singleton instance to be created.
 */
public static void setKind(SingletonKind kind) {
    Singleton.kind = kind;
}

/*
If needed, getInstance() creates instance appropriate class, based on value of
singletonKind field.
 */
public static Singleton getInstance()
    throws  NoSuchMethodException,
            IllegalAccessException,
            InvocationTargetException,
            InstantiationException {

    if(instance==null){
        synchronized (Singleton.class){
            if(instance==null){
                Singleton singleton = new Singleton();
                switch (kind){
                    case UNKNOWN:

                        instance = singleton;
                        break;
```

```
                        case LOWERCASE:

                            /*
                             I can't use simple

                             instance = new LowercaseSingleton();

                             because java compiler won't allow me to use
                             constructor of inner class in static context,
                             so I use reflection API instead.

                             To be able to access inner class by reflection API,
                             I have to create instance of outer class first.
                             Therefore, in this implementation, Singleton cannot be
                             abstract class.
                             */

                            //Get the constructor of inner class.
                            Constructor<LowercaseSingleton> lcConstructor =

LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

                            //The constructor is private, so I have to make it accessible.
                            lcConstructor.setAccessible(true);

                            // Use the constructor to create instance.
                            instance = lcConstructor.newInstance(singleton);

                            break;

                        case UPPERCASE:

                            //Same goes here, just with different type
                            Constructor<UppercaseSingleton> ucConstructor =

UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
                            ucConstructor.setAccessible(true);
                            instance = ucConstructor.newInstance(singleton);
                    }
                }
            }
        }
        return instance;
    }

    //Singletons state that is to be used by subclasses
    protected String message;

    //Private constructor prevents external instantiation.
    private Singleton()
    {
        message = "Hello world!";
    }

    //Singleton's API. Implementation can be overwritten by subclasses.
    public String getMessage() {
        return message;
    }
}
```

```
//Just a small test program
public class ExtendingSingletonExample {

    public static void main(String args[]){

        //just uncomment one of following lines to change singleton class

        //Singleton.setKind(SingletonKind.UPPERCASE);
        //Singleton.setKind(SingletonKind.LOWERCASE);

        Singleton singleton = null;
        try {
            singleton = Singleton.getInstance();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
        System.out.println(singleton.getMessage());
    }
}
```

Прочитайте Одиночки онлайн: https://riptutorial.com/ru/java/topic/130/одиночки

глава 125: операторы

Вступление

Операторы на языке программирования Java - это специальные символы, которые выполняют определенные операции над одним, двумя или тремя операндами, а затем возвращают результат.

замечания

Оператор - это символ (или символы), который сообщает программе Java выполнить *операцию* над одним, двумя или тремя *операндами* . Оператор и его операнды образуют *выражение* (см. Раздел «Выражения»). Операнды оператора сами являются выражениями.

В этом разделе описываются 40 или около того различных операторов, определенных Java. В разделе «Отдельные выражения» объясняется:

- как операторы, операнды и другие вещи объединяются в выражения,
- как оцениваются выражения, и
- как работают выражения, преобразования и выражения.

Examples

Оператор конкатенации строк (+)

Символ `+` может означать три разных оператора в Java:

- Если нет операнда до `+` , то это унарный оператор Plus.
- Если есть два операнда, они оба являются числовыми. то он является двоичным оператором сложения.
- Если есть два операнда, и по крайней мере один из них является `String` , то он является двоичным оператором Concatenation.

В простом случае оператор Concatenation соединяет две строки, чтобы дать третью строку. Например:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

Если один из двух операндов не является строкой, он преобразуется в `String` следующим образом:

- Операнд, тип которого является примитивным типом, преобразуется, *как если бы он*

вызывал `toString()` по размеру в штучной упаковке.

- Операнд, тип которого является ссылочным типом, преобразуется путем вызова метода `toString()` операнда. Если операнд имеет значение `null` , или если метод `toString()` возвращает значение `null` , вместо него используется строковый литерал `"null"` .

Например:

```
int one = 1;
String s3 = "One is "  + one;         // s3 contains "One is 1"
String s4 = null + " is null";        // s4 contains "null is null"
String s5 = "{1} is " + new int[]{1}; // s5 contains something like
                                      // "{} is [I@xxxxxxxx"
```

Объяснение примера `s5` заключается в том, что метод `toString()` для типов массивов наследуется из `java.lang.Object` , и поведение заключается в создании строки, состоящей из имени типа и идентификатора hashcode объекта.

Оператор Concatenation задается для создания нового объекта `String` , за исключением случая, когда выражение является константным выражением. В последнем случае выражение оценивается по типу компиляции, а его значение времени выполнения эквивалентно строковому литералу. Это означает, что для разделения длинного строкового литерала нет накладных расходов во время выполнения:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Оптимизация и эффективность

Как отмечено выше, за исключением постоянных выражений, каждое выражение конкатенации строк создает новый объект `String` . Рассмотрим этот код:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

В вышеприведенном методе каждая итерация цикла создаст новую `String` которая будет на один символ длиннее предыдущей итерации. Каждая конкатенация копирует все символы в строках операндов, чтобы сформировать новую `String` . Таким образом, `stars(N)` будут:

- создать `N` новых объектов `String` и выбросить все, кроме последнего,
- копировать `N * (N + 1) / 2` символа и

- сгенерировать `O(N^2)` байты мусора.

Это очень дорого для больших `N` Действительно, любой код, который объединяет строки в цикле, может иметь эту проблему. Лучший способ написать это будет следующим:

```
public String stars(int count) {
    // Create a string builder with capacity 'count'
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

В идеале вы должны установить емкость `StringBuilder` , но если это нецелесообразно, класс автоматически *вырастет* массив поддержки, который строитель использует для хранения символов. (Примечание: реализация расширяет базовый массив экспоненциально. Эта стратегия сохраняет количество копий символов в `O(N)` а не `O(N^2)` .)

Некоторые люди применяют этот шаблон для всех конкатенаций строк. Однако это не нужно, поскольку JLS *позволяет* компилятору Java оптимизировать конкатенации строк в одном выражении. Например:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

как правило , оптимизируется компилятором байт-кода на что-то вроде этого;

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
tmp.append(s2 == null ? "null" + s2);
tmp.append("\n");
String test = tmp.toString();
```

(Компилятор JIT может оптимизировать это, если он сможет определить, что `s1` или `s2` не может быть `null` .) Но обратите внимание, что эта оптимизация разрешена только в одном выражении.

Короче говоря, если вас беспокоит эффективность конкатенаций строк:

- Ручная оптимизация, если вы повторяете конкатенацию в цикле (или аналогичном).
- Не ручная оптимизация одного выражения конкатенации.

Арифметические операторы (+, -, *, /,%)

Язык Java предоставляет 7 операторов, выполняющих арифметические операции с целыми

и плавающими значениями.

- Есть два `+` операторов:
  - Оператор двоичного сложения добавляет одно число к другому. (Существует также двоичный `+` оператор, который выполняет конкатенацию строк. Это описано в отдельном примере.)
  - Оператор унарного плюса ничего не делает, кроме запуска числового продвижения (см. Ниже)
- Есть два `-` операторы:
  - Оператор двоичного вычитания вычитает одно число из другого.
  - Унарный минус-оператор эквивалентен вычитанию его операнда с нуля.
- Оператор двоичного умножения (*) умножает одно число на другое.
- Оператор двоичного деления (/) делит одно число на другое.
- Оператор двоичного остатка [1] (%) вычисляет остаток, когда одно число делится на другое.

1. Это часто неправильно называют оператором «модуль». «Остаток» - это термин, который используется JLS. «Модуль» и «остаток» - это не одно и то же.

Операнд и типы результатов, а также числовое продвижение

Операторы требуют числовых операндов и получают числовые результаты. Типы операндов могут быть любыми примитивными числовыми типами (то есть `byte` , `short` , `char` , `int` , `long` , `float` или `double`) или любым числовым типом оболочки, определяемым в `java.lang` ; т.е. (`Byte` , `Character` , `Short` , `Integer` , `Long` , `Float` или `Double` .

Тип результата определяется базой по типам операнда или операндов следующим образом:

- Если любой из операндов является `double` или `Double` , тогда тип результата будет `double` .
- В противном случае, если любой из операндов является `float` или `Float` , тогда тип результата будет `float` .
- В противном случае, если любой из операндов `long` или `Long` , тогда тип результата `long` .
- В противном случае тип результата - `int` . Это охватывает `byte` , `short` и `char` операнды, а также `int.

Тип результата операции определяет, как выполняется арифметическая операция, и как обрабатываются операнды

- Если тип результата `double` , операнды повышаются до `double` , а операция выполняется с использованием 64-битной (двойной точности двоичной) IEE 754 с

плавающей запятой.
- Если тип результата является `float` , операнды продвигаются на `float` , и операция выполняется с использованием 32-битной (одиночной точности) IEE 754 с плавающей точкой арифметики.
- Если тип результата `long` , операнды продвигаются до `long` , и операция выполняется с использованием битовой целочисленной арифметики двоичного кода с двоичным кодом, состоящей из 64 бит.
- Если типом результата является `int` , операнды продвигаются до `int` , а операция выполняется с использованием 32-разрядной двоичной двоичной двоичной арифметики двоичного кода.

Продвижение осуществляется в два этапа:

- Если тип операнда является типом-оболочкой, значение операнда будет *опущено* на значение соответствующего примитивного типа.
- При необходимости примитивный тип продвигается до требуемого типа:
  - Продвижение целых чисел в `int` или `long` потерь.
  - Продвижение `float` в `double` потерь.
  - Продвижение целого числа в значение с плавающей запятой может привести к потере точности. Преобразование выполняется с использованием семантики «круг-к-ближайшему» IEE 768.

Смысл разделения

Оператор / делит левый операнд `n` (*дивиденд*) и правый операнд `d` (*делитель*) и выдает результат `q` (*фактор*).

Явное целочисленное деление округляется до нуля. В разделе JLS 15.17.2 указано поведение целочисленного деления Java следующим образом:

> Фактор, созданный для операндов `n` и `d` представляет собой целое значение `q` , величина которого как можно больше, при условии, что `|d · q| ≤ |n|` , Более того, `q` положительно, когда `|n| ≥ |d|` и `n` и `d` имеют один и тот же знак, но `q` отрицателен при `|n| ≥ |d|` и `n` и `d` имеют противоположные знаки.

Есть несколько особых случаев:

- Если `n` - `MIN_VALUE` , а делитель равен -1, то происходит переполнение целого числа, а результат - `MIN_VALUE` . Никакое исключение не возникает в этом случае.
- Если `d` равно 0, то генерируется «ArithmeticException».

Разделение с плавающей запятой Java требует рассмотрения более кратных случаев. Однако основная идея состоит в том, что результат `q` является значением, наиболее близким к удовлетворяющему `d . q = n` .

Разделение с плавающей точкой никогда не приведет к исключению. Вместо этого операции, делящиеся на ноль, приводят к значениям INF и NaN; увидеть ниже.

Значение остатка

В отличие от C и C ++, оператор остатка в Java работает как с целыми, так и с плавающей запятой.

Для целых случаев результат `a % b` определяется как число `r` такое, что `(a / b) * b + r` равно `a` , где `/` , `*` и `+` - соответствующие Java-целые операторы. Это применяется во всех случаях, кроме случаев, когда `b` равно нулю. В этом случае остаток приводит к `ArithmeticException` .

Из приведенного выше определения следует, что `a % b` может быть отрицательным, только если `a` отрицательно, и оно положительно, только если `a` положительно. Более того, величина `a % b` всегда меньше величины `b` .

Операция останова с плавающей запятой является обобщением целочисленного случая. Результат `a % b` - остаток `r` определяется математическим соотношением `r = a - (b * q)` где:

- `q` - целое число,
- это отрицательно, только если `a / b` отрицательно положительно, только если `a / b` положительно и
- его величина как можно больше, не превышающая величину истинного математического отношения `a` и `b` .

Остаток с плавающей точкой может производить значения `INF` и `NaN` в крайних случаях, например, когда `b` равно нулю; увидеть ниже. Это исключение не будет.

Важная заметка:

> Результат операции останова с плавающей запятой, вычисляемой по `%` **, не совпадает** с результатом операции остатка, определенной IEEE 754. Остаток IEEE 754 может быть вычислен с использованием библиотечного метода `Math.IEEEremainder` .

Целочисленное переполнение

Значения целых чисел на Java 32 и 64 бит подписаны и используют двоичное представление двоичного кода. Например, диапазон чисел, представляемых как (32 бит) `int` -2^{31} - $+2^{31} - 1$.

Когда вы добавляете, вычитаете или несколько двух N битовых целых чисел (N == 32 или 64), результат операции может быть слишком большим для представления в виде N

битового целого. В этом случае операция приводит к *целочисленному переполнению* , и результат может быть вычислен следующим образом:

- Математическая операция выполняется, чтобы дать промежуточное представление с двумя дополнениями всего числа. Это представление будет больше, чем N бит.
- В результате используются нижние 32 или 64 бита промежуточного представления.

Следует отметить, что целочисленное переполнение не приводит к исключениям ни при каких обстоятельствах.

Значения INF и NAN с плавающей запятой

Java использует представления с плавающей точкой IEE 754 для `float` и `double` . Эти представления имеют некоторые специальные значения для представления значений, которые выходят за пределы области действительных чисел:

- Значения «бесконечный» или «INF» означают слишком большие числа. Значение `+INF` обозначает слишком большие и положительные числа. Значение `-INF` обозначает слишком большие и отрицательные числа.
- «Неопределенный» / «не число» или NaN обозначают значения, возникающие в результате бессмысленных операций.

Значения INF производятся плавающими операциями, которые вызывают переполнение или делением на ноль.

Значения NaN производятся путем деления нуля на ноль или вычисления нуля нуля.

Удивительно, но можно выполнить арифметику с использованием операндов INF и NaN без возникновения исключений. Например:

- Добавление + INF и конечное значение дает + INF.
- Добавление + INF и + INF дает + INF.
- Добавление + INF и -INF дает NaN.
- Разделение по INF дает либо +0.0, либо -0.0.
- Все операции с одним или несколькими операндами NaN дают NaN.

Для получения полной информации см. Соответствующие подразделы JLS 15 . Обратите внимание, что это в значительной степени «академическое». Для типичных вычислений `INF` или `NaN` означает, что что-то пошло не так; например, у вас есть неполные или неправильные входные данные, или расчет был запрограммирован неправильно.

Операторы равенства (==,! =)

Операторы `==` и `!=` - это двоичные операторы, которые оценивают `true` или `false` зависимости от того, являются ли операнды равными. Оператор `==` задает значение `true`

если операнды равны и `false` противном случае. Оператор `!=` Дает `false` если операнды равны и `true` противном случае.

Этими операторами могут быть использованы операнды с примитивными и ссылочными типами, но поведение существенно отличается. Согласно JLS, на самом деле существует три различных набора этих операторов:

- Операторы Boolean `==` и `!=` .
- Операторы Numeric `==` и `!=` .
- Операторы Reference `==` и `!=` .

Однако во всех случаях тип результата операторов `==` и `!=` Является `boolean` .

Операторы Numeric `==` и `!=`

Когда один (или оба) операндов оператора `==` или `!=` Является примитивным числовым типом (`byte` , `short` , `char` , `int, long` , `float` или `double`), оператор представляет собой числовое сравнение. Второй операнд должен быть либо примитивным числовым типом, либо коробочным числовым типом.

Поведение других числовых операторов выглядит следующим образом:

1. Если один из операндов является коробочным, он распаковывается.
2. Если любой из операндов теперь является `byte` , `short` или `char` , он продвигается до `int` .
3. Если типы операндов не совпадают, то операнд с «меньшим» типом продвигается к «более крупному» типу.
4. Затем сравнение проводится следующим образом:
    - Если продвинутые операнды являются `int` или `long` тогда значения тестируются, чтобы убедиться, что они идентичны.
    - Если продвинутые операнды являются `float` или `double` тогда:
        - две версии нуля (`+0.0` и `-0.0`) считаются равными
        - значение `NaN` рассматривается как не равное чему-либо, и
        - другие значения равны, если их представления IEEE 754 идентичны.

Примечание: вам нужно быть осторожным при использовании `==` и `!=` Для сравнения значений с плавающей запятой.

Операторы Boolean `==` и `!=`

Если оба операнда являются `boolean` , или один является `boolean` а другой `Boolean` , эти операторы являются операторами Boolean `==` и `!=` . Поведение выглядит следующим образом:

1. Если один из операндов является `Boolean` , он распаковывается.
2. Проверяются незанятые операнды и вычисляется логический результат в соответствии со следующей таблицей истинности

	B	A == B	A! = B
ложный	ложный	правда	ложный
ложный	правда	ложный	правда
правда	ложный	ложный	правда
правда	правда	правда	ложный

Есть две «подводные камни», которые позволяют использовать `==` и `!=` Экономно с значениями истинности:

- Если вы используете `==` или `!=` Для сравнения двух `Boolean` объектов, то используются операторы Reference. Это может дать неожиданный результат; см. Pitfall: использование == для сравнения объектов примитивных оберток, таких как Integer
- Оператор `==` может быть легко ошибочен как `=` . Для большинства типов операндов эта ошибка приводит к ошибке компиляции. Однако для `boolean` и `Boolean` операндов ошибка приводит к неправильному поведению во время выполнения; см. Pitfall - использование '==' для проверки логического

Операторы Reference `==` и `!=`

Если оба операнда являются объектными ссылками, операторы `==` и `!=` Проверяют, **относятся** ли эти два операнда к **одному и тому же объекту** . Это часто не то, что вы хотите. Чтобы проверить, являются ли два объекта равными *по значению* , вместо этого следует использовать метод `.equals()` .

```
String s1 = "We are equal";
String s2 = new String("We are equal");

s1.equals(s2); // true

// WARNING - don't use == or != with String values
s1 == s2;      // false
```

Предупреждение: использование `==` и `!=` Для сравнения значений `String` в большинстве случаев **неверно** ; см. http://www.riptutorial.com/java/example/16290/pitfall--using----to-compare-strings . Аналогичная проблема применима к примитивным типам обертки; см. http://www.riptutorial.com/java/example/8996/pitfall--using----to-compare-primitive-wrappers-objects-such-as-integer .

О краях NaN

В JLS 15.21.1 указано следующее:

> Если либо операнд `NaN` , то результат `==` является `false` но результат `!= true` . Действительно, тест `x != x true` тогда и только тогда, когда значение `x` равно `NaN` .

Такое поведение (для большинства программистов) неожиданно. Если вы проверите, соответствует ли значение `NaN` самому себе, ответ будет «Нет, это не так!». Другими словами, `==` не *рефлексивно* для значений `NaN` .

Однако это не «странность» Java, это поведение указано в стандартах с плавающей точкой IEEE 754, и вы обнаружите, что оно реализовано большинством современных языков программирования. (Для получения дополнительной информации см. Http://stackoverflow.com/a/1573715/139985 ... отметив, что это написано кем-то, кто был «в комнате, когда были приняты решения»!)

Операторы Increment / Decrement (++ / -)

Переменные могут быть увеличены или уменьшены на 1 с помощью операторов `++` и `--` соответственно.

Когда операторы `++` и `--` следуют за переменными, они называются **post-increment** и **post-декремент** соответственно.

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

Когда операторы `++` и `--` предшествуют переменным, операции называются **пред-приращениями** и **пред-декрементами** соответственно.

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

Если оператор предшествует переменной, значение выражения будет значением переменной после того, как будет увеличиваться или уменьшаться. Если оператор следует за переменной, значение выражения будет значением переменной до того, как будет увеличиваться или уменьшаться.

```
int x=10;

System.out.println("x=" + x + " x=" + x++ + " x=" + x); // outputs x=10 x=10 x=11
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // outputs x=11 x=12 x=12
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // outputs x=12 x=12 x=11
```

```
System.out.println("x=" + x + " x=" + --x + " x=" + x); // outputs x=11 x=10 x=10
```

Будьте осторожны, чтобы не перезаписывать пост-приращения или декременты. Это происходит, если вы используете оператор post-in / decment в конце выражения, которое переназначается самой переменной in / decmented. Значение in / decment не будет иметь эффекта. Несмотря на то, что переменная с левой стороны увеличивается правильно, ее значение будет немедленно перезаписано ранее оцененным результатом с правой стороны выражения:

```
int x = 0;
x = x++ + 1 + x++;       // x = 0 + 1 + 1
                         // do not do this - the last increment has no effect (bug!)
System.out.println(x);   // prints 2 (not 3!)
```

Правильный:

```
int x = 0;
x = x++ + 1 + x;         // evaluates to x = 0 + 1 + 1
x++;                     // adds 1
System.out.println(x);   // prints 3
```

Условный оператор (? :)

Синтаксис

{условие для оценки} **?** *{statement-execute-on-true}* **:** *{statement-execute-on-false}*

Как показано в синтаксисе, Условный оператор (также известный как Тернарный оператор [1]) использует `?` (знак вопроса) и `:` (двоеточие), чтобы разрешить условное выражение двух возможных результатов. Он может использоваться для замены более длинных блоков `if-else` для возврата одного из двух значений на основе условия.

```
result = testCondition ? value1 : value2
```

Эквивалентно

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

Его можно прочитать как ***«Если testCondition истинно, установите результат в value1; в противном случае установите результат в значение2 ".***

Например:

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Эквивалентно

```
// get absolute value using if/else loop
a = -10;
int absValue;
if (a < 0) {
    absValue = -a;
} else {
    absValue = a;
}
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Общее использование

Вы можете использовать условный оператор для условных присвоений (например, для проверки нуля).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

Этот пример эквивалентен:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Поскольку Условный оператор имеет второе наименьшее приоритетное значение, выше Операторов присваивания , редко требуется использовать скобки вокруг *условия* , но скобки требуются вокруг всей конструкции условного оператора в сочетании с другими операторами:

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

Условные операторы вложения также могут выполняться в третьей части, где она больше похожа на цепочку или как оператор switch.

```
a ? "a is true" :
b ? "a is false, b is true" :
```

```
c ? "a and b are false, c is true" :
    "a, b, and c are false"

//Operator precedence can be illustrated with parenthesis:

a ? x : (b ? y : (c ? z : w))
```

Сноска:

1 - Как Java Language Specification, так и Java Tutorial вызывают оператор (`? :`) *Условный оператор* . В учебнике говорится, что он также известен как Тернарный оператор, поскольку он (в настоящее время) является единственным тройным оператором, определенным Java. Терминология «Условный оператор» согласуется с C и C ++ и другими языками с эквивалентным оператором.

Побитовые и логические операторы (~, &, |, ^)

Язык Java предоставляет 4 оператора, которые выполняют побитовые или логические операции с целыми или булевыми операндами.

- Оператор дополнения (`~`) является унарным оператором, который выполняет поразрядное или логическое обращение битов одного операнда; см. JLS 15.15.5. ,
- Оператор AND (`&`) является двоичным оператором, который выполняет побитовое или логическое «и» из двух операндов; см. JLS 15.22.2. ,
- Оператор OR (`|`) является двоичным оператором, который выполняет побитовое или логическое «включение» или «из двух операндов»; см. JLS 15.22.2. ,
- Оператор XOR (`^`) является двоичным оператором, который выполняет побитовое или логическое «исключение» или «из двух операндов»; см. JLS 15.22.2. ,

Логические операции, выполняемые этими операторами, когда операнды являются логическими, можно суммировать следующим образом:

	B	~ A	A & B	A | B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Обратите внимание, что для целых операндов приведенная выше таблица описывает, что происходит для отдельных бит. Операторы фактически работают со всеми 32 или 64 битами операнда или операндов параллельно.

Типы операндов и типы результатов.

Обычные арифметические преобразования применяются, когда операнды являются целыми числами. Обычные прецеденты для побитовых операторов

Оператор `~` используется для изменения логического значения или изменения всех битов в целочисленном операнде.

Оператор `&` используется для «маскировки» некоторых битов целочисленного операнда. Например:

```
int word = 0b00101010;
int mask = 0b00000011;   // Mask for masking out all but the bottom
                         // two bits of a word
int lowBits = word & mask;            // -> 0b00000010
int highBits = word & ~mask;          // -> 0b00101000
```

`|` оператор используется для объединения значений истинности двух операндов. Например:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask);   // -> 0b01011110
```

Оператор `^` используется для переключения или «переворачивания» битов:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

Дополнительные примеры использования побитовых операторов см. В разделе « Манипуляция бит»

Оператор экземпляра

Этот оператор проверяет, имеет ли объект определенный тип класса / интерфейса. Оператор **instanceof** записывается как:

```
( Object reference variable ) instanceof  (class/interface type)
```

Пример:

```
public class Test {

   public static void main(String args[]){
      String name = "Buyya";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

Это приведет к следующему результату:

```
true
```

Этот оператор по-прежнему будет возвращать true, если сравниваемый объект - это присвоение, совместимое с типом справа.

Пример:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

Это приведет к следующему результату:

```
true
```

Операторы присваивания (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = и ^ =)

Левый операнд для этих операторов должен быть либо не конечной переменной, либо элементом массива. Правый операнд должен быть *совместимым* с левым операндом. Это означает, что либо типы должны быть одинаковыми, либо правильный тип операнда должен быть конвертирован в тип левых операндов комбинацией бокса, распаковки или расширения. (Подробнее см. JLS 5.2 .)

Точное значение операторов «операция и назначение» указано в JLS 15.26.2 следующим образом:

> Составляющее выражение присваивания формы `E1 op= E2` эквивалентно `E1 = (T) ((E1) op (E2))` , где `T` - тип `E1` , за исключением того, что `E1` оценивается только один раз.

Обратите внимание, что перед окончательным присваиванием подразумевается неявный тип.

1. `=`

Простой оператор присваивания: присваивает значение правого операнда левому операнду.

> Пример: `c = a + b` добавит значение `a + b` к значению `c` и назначит его `c`

2. `+=`

Оператор «добавить и присваивать»: добавляет значение правого операнда к значению левого операнда и присваивает результат левому операнду. Если левый операнд имеет тип `String` , то это оператор «конкатенировать и присваивать».

Пример: `c += a` примерно такой же, как `c = c + a`

3. `-=`

Оператор «вычесть и присваивать»: вычитает значение правого операнда из значения левого операнда и присваивает результат левому операнду.

Пример: `c -= a` примерно совпадает с `c = c - a`

4. `*=`

Оператор «умножить и присваивать»: умножает значение правого операнда на значение левого операнда и присваивает результат левому операнду. ,

Пример: `c *= a` примерно совпадает с `c = c * a`

5. `/=`

Оператор «делить и присваивать»: делит значение правого операнда на значение левого операнда и присваивает результат левому операнду.

Пример: `c /*= a` примерно совпадает с `c = c / a`

6. `%=`

Оператор «модуль и назначение» вычисляет модуль значения правого операнда по значению левого операнда и присваивает результат левому операнду.

Пример: `c %*= a` примерно совпадает с `c = c % a`

7. `<<=`

Оператор «сдвиг влево и назначение».

Пример: `c <<= 2` примерно совпадает с `c = c << 2`

8. `>>=`

Оператор «арифметический сдвиг вправо и назначение».

Пример: `c >>= 2` примерно совпадает с `c = c >> 2`

9. `>>>=`

Оператор «Логический сдвиг вправо и назначение».

Пример: `c >>>= 2` примерно совпадает с `c = c >>> 2`

10. `&=`

Оператор «побитовой и назначающий».

Пример: `c &= 2` примерно совпадает с `c = c & 2`

11. `|=`

Оператор «побитовое или назначить».

Пример: `c |= 2` примерно совпадает с `c = c | 2`

12. `^=`

Оператор «побитовое исключение и присваивание».

Пример: `c ^= 2` примерно такой же, как `c = c ^ 2`

Условные и условные операторы (&& и ||)

Java предоставляет условный и условный или оператор, которые берут один или два операнда типа `boolean` и создают `boolean` результат. Это:

- `&&` - оператор условного-И,
- `||` - операторы условного ИЛИ. Оценка `<left-expr> && <right-expr>` эквивалентна следующему псевдокоду:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
}
```

Оценка `<left-expr> || <right-expr>` эквивалентен следующему псевдокоду:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
```

```
    }
```

Как видно из приведенного выше псевдокода, поведение операторов короткого замыкания эквивалентно использованию операторов `if` / `else` .

Пример - использование && в качестве защиты в выражении

В следующем примере показан наиболее распространенный шаблон использования для оператора `&&` . Сравните эти две версии метода, чтобы проверить, является ли поставленное `Integer` равным нулю.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

Первая версия работает в большинстве случаев, но если аргумент `value` имеет `value` `null` , тогда будет `NullPointerException` .

Во второй версии мы добавили тест «guard». Выражение `value != null && value == 0` оценивается, сначала выполняя `value != null` test. Если `null` тест завершается успешно (т.е. он оценивается как `true`), тогда вычисляется выражение `value == 0` . Если `null` тест терпит неудачу, то оценка `value == 0` пропущена (закорочена), и мы *не* получаем `NullPointerException` .

Пример. Используя &&, чтобы избежать дорогостоящего расчета

В следующем примере показано, как `&&` можно использовать, чтобы избежать относительно дорогостоящего расчета:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

В первой версии оба операнда `|` всегда будет оцениваться, поэтому (дорогостоящий) метод `isPrime` будет вызван без необходимости. Вторая версия позволяет избежать ненужного вызова, используя `||` вместо `|` ,

Операторы сдвига (<<, >> и >>>)

Язык Java предоставляет три оператора для выполнения поразрядного смещения по 32 и 64-битным целым значениям. Это все двоичные операторы, первый операнд которых является смещаемым значением, а второй операнд говорит, как далеко сдвинуться.

- Оператор `<<` или *левого сдвига* сдвигает значение, заданное первым операндом *влево* , на число битных позиций, заданных вторым операндом. Пустые позиции на правом конце заполняются нулями.
- Оператор «>>» или *арифметического сдвига* сдвигает значение, заданное первым операндом, *вправо* , на количество битных позиций, заданных вторым операндом. Пустые позиции на левом конце заполняются копированием самого левого разряда. Этот процесс известен как *расширение знака* .
- Оператор «>>>» или *логического сдвига* сдвигает значение, заданное первым операндом, *вправо* на число битных позиций, заданных вторым операндом. Пустые позиции в левом конце заполняются нулями.

Заметки:

1. Этим операторам требуется значение `int` или `long` в качестве первого операнда и выдает значение с тем же типом, что и первый операнд. (Вам нужно будет использовать явный тип приведения при назначении результата сдвига `byte` , `short` или переменной `char` .)
2. Если вы используете оператор сдвига с первым операндом, который является `byte` , `char` или `short` , он продвигается до `int` а операция производит `int` .)
3. Второй операнд уменьшается *по модулю количества бит операции,* чтобы дать величину сдвига. Подробнее о **математической концепции мод** см. Примеры модулей .
4. Биты, сдвинутые с левого или правого конца с помощью операции, отбрасываются. (Java не предоставляет примитивный оператор «rotate».)
5. Оператор арифметического сдвига эквивалентен делению числа (двух) дополнений на мощность 2.
6. Оператор сдвига влево эквивалентен, умножая число (2) дополнител на 2.

Следующая таблица поможет вам увидеть эффекты трех операторов сдвига. (Числа были выражены в двоичной нотации, чтобы помочь визуализации.)

Operand1	operand2	<<	>>	>>>
0b0000000000001011	0	0b0000000000001011	0b0000000000001011	0b0000000000001011

Operand1	operand2	<<	>>	>>>
0b0000000000001011	1	0b0000000000010110	0b0000000000000101	0b0000000000000101
0b0000000000001011	2	0b0000000000101100	0b0000000000000010	0b0000000000000010
0b0000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	32	0b0000000000001011	0b0000000000001011	0b0000000000001011
...	...	...	...	...
0b1000000000001011	0	0b1000000000001011	0b1000000000001011	0b1000000000001011
0b1000000000001011	1	0b0000000000010110	0b1100000000000101	0b0100000000000101
0b1000000000001011	2	0b0000000000101100	0b1110000000000010	0b0010000000000100
0b1000000000001011	31	0b1000000000000000	0b1111111111111111	0b0000000000000001

Там приведены примеры пользователя операторов сдвига в манипуляции бит

Оператор Лямбды (->)

Начиная с Java 8, оператор лямбды (`->`) является оператором, используемым для введения выражения Лямбды. Существуют два распространенных синтаксиса, которые иллюстрируются следующими примерами:

Java SE 8

```
a -> a + 1               // a lambda that adds one to its argument
a -> { return a + 1; }   // an equivalent lambda using a block.
```

Выражение лямбда определяет анонимную функцию или, вернее, экземпляр анонимного класса, который реализует *функциональный интерфейс* .

(Этот пример включен здесь для полноты. См. Раздел « Лямбда-выражения» для полного лечения.)

Реляционные операторы (<, <=,>,> =)

Операторы `<` , `<=` , `>` и `>=` являются двоичными операторами для сравнения числовых типов. Смысл операторов, как и следовало ожидать. Например, если `a` и `b` объявлены как `byte` , `short` , `char` , `int` , `long` , `float` , `double` или соответствующие типы в коробке:

```
- `a < b` tests if the value of `a` is less than the value of `b`.
- `a <= b` tests if the value of `a` is less than or equal to the value of `b`.
- `a > b` tests if the value of `a` is greater than the value of `b`.
- `a >= b` tests if the value of `a` is greater than or equal to the value of `b`.
```

Тип результата для этих операторов `boolean` во всех случаях.

Операторы отношения могут использоваться для сравнения чисел с разными типами. Например:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Операторы отношения могут использоваться, когда оба или оба числа являются экземплярами числовых типов в штучной упаковке. Например:

```
Integer i = 1;   // 1 is autoboxed to an Integer
Integer j = 2;   // 2 is autoboxed to an Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

Точное поведение суммируется следующим образом:

1. Если один из операндов является коробочным, он распаковывается.
2. Если любой из операндов теперь является `byte` , `short` или `char` , он продвигается до `int` .
3. Если типы операндов не совпадают, то операнд с «меньшим» типом продвигается к «более крупному» типу.
4. Сравнение выполняется по итоговым значениям `int` , `long` , `float` или `double` .

Вы должны быть осторожны с реляционными сравнениями, которые включают числа с плавающей запятой:

- Выражения, которые вычисляют числа с плавающей запятой, часто приводят к ошибкам округления из-за того, что представления с плавающей запятой компьютера имеют ограниченную точность.
- При сравнении целочисленного типа и типа с плавающей точкой преобразование целочисленного числа в плавающую точку также может приводить к ошибкам округления.

Наконец, Java немного поддерживает использование реляционных операторов с любыми типами, отличными от перечисленных выше. Например, вы *не можете* использовать эти операторы для сравнения строк, массивов чисел и т. Д.

Прочитайте операторы онлайн: https://riptutorial.com/ru/java/topic/176/операторы

глава 126: Операции с плавающей точкой Java

Вступление

Числа с плавающей запятой - это числа, которые имеют дробные части (обычно выраженные с десятичной точкой). В Java существует два примитивных типа для чисел с плавающей запятой, которые являются `float` (использует 4 байта) и `double` (использует 8 байтов). Эта страница документации предназначена для подробного описания операций с примерами, которые могут выполняться с плавающей точкой в Java.

Examples

Сравнение значений с плавающей запятой

Вы должны быть осторожны при сравнении значений с плавающей запятой (`float` или `double`) с использованием реляционных операторов: `==` `!=` , `<` И т. Д. Эти операторы дают результаты в соответствии с двоичными представлениями значений с плавающей запятой. Например:

```
public class CompareTest {
    public static void main(String[] args) {
        double oneThird = 1.0 / 3.0;
        double one = oneThird * 3;
        System.out.println(one == 1.0);      // prints "false"
    }
}
```

Расчет `oneThird` ввел крошечную ошибку округления, и когда мы умножаем `oneThird` на `3` мы получаем результат, немного отличающийся от `1.0` .

Эта проблема неточных представлений более резкая, когда мы пытаемся смешивать `double` и `float` в вычислениях. Например:

```
public class CompareTest2 {
    public static void main(String[] args) {
        float floatVal = 0.1f;
        double doubleVal = 0.1;
        double doubleValCopy = floatVal;

        System.out.println(floatVal);      // 0.1
        System.out.println(doubleVal);     // 0.1
        System.out.println(doubleValCopy); // 0.10000000149011612

        System.out.println(floatVal == doubleVal); // false
        System.out.println(doubleVal == doubleValCopy); // false
```

```
    }
}
```

Представления с плавающей запятой, используемые в Java для `float` и `double` типов, имеют ограниченное количество цифр точности. Для типа `float` точность составляет 23 двоичных цифры или около 8 десятичных цифр. Для `double` типа это 52 бит или около 15 десятичных цифр. Кроме того, некоторые арифметические операции будут приводить к ошибкам округления. Поэтому, когда программа сравнивает значения с плавающей запятой, стандартная практика определяет **приемлемую дельта** для сравнения. Если разница между двумя числами меньше дельта, они считаются равными. Например

```
if (Math.abs(v1 - v2) < delta)
```

Пример сравнения Delta:

```
public class DeltaCompareExample {

    private static boolean deltaCompare(double v1, double v2, double delta) {
        // return true iff the difference between v1 and v2 is less than delta
        return Math.abs(v1 - v2) < delta;
    }

    public static void main(String[] args) {
        double[] doubles = {1.0, 1.0001, 1.0000001, 1.000000001, 1.0000000000001};
        double[] deltas = {0.01, 0.00001, 0.0000001, 0.0000000001, 0};

        // loop through all of deltas initialized above
        for (int j = 0; j < deltas.length; j++) {
            double delta = deltas[j];
            System.out.println("delta: " + delta);

            // loop through all of the doubles initialized above
            for (int i = 0; i < doubles.length - 1; i++) {
                double d1 = doubles[i];
                double d2 = doubles[i + 1];
                boolean result = deltaCompare(d1, d2, delta);

                System.out.println("" + d1 + " == " + d2 + " ? " + result);

            }

            System.out.println();
        }
    }
}
```

Результат:

```
delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.0000001 ? true
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true
```

```
delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false
```

Также для сравнения `double` и `float` примитивных типов может использоваться статический метод `compare` соответствующего типа бокса. Например:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Наконец, определение того, какие дельта наиболее подходят для сравнения, может быть сложным. Общепринятый подход состоит в том, чтобы выбрать значения дельты, о которых говорят наши интуиции. Однако, если вы знаете масштаб и (истинную) точность входных значений и выполненные вычисления, может быть возможно получить математически обоснованные оценки точности результатов и, следовательно, для дельт. (Существует формальная ветвь математики, известная как численный анализ, которая раньше преподавалась ученым-вычислителям, которые занимались этим анализом).

OverFlow и UnderFlow

Тип данных с **плавающей запятой**

Тип данных float представляет собой 32-битную 32-битную IEEE 754 с плавающей точкой.

`Float` переполнение

Максимально возможное значение равно `3.4028235e+38` Когда оно превышает это значение, оно дает `Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

`Float` **UnderFlow**

Минимальное значение - 1.4e-45f, когда оно меньше этого значения, оно производит `0.0`

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

двойной тип данных

Двойным типом данных является `64-bit` плавающая точка IEEE 754 с двойной точностью.

`Double` **OverFlow**

Максимальное возможное значение: `1.7976931348623157e+308` Когда оно превышает это значение, оно создает `Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

`Double` **UnderFlow**

Минимальное значение - 4.9e-324, когда оно меньше этого значения, оно производит `0.0`

```
    double d = 4.8e-323;
    double result = d/1000;
    System.out.println(result); //0.0
```

Форматирование значений с плавающей запятой

Числа с плавающей запятой могут быть отформатированы как десятичное число, используя `String.format` с флагом `'f'`

```
    //Two digits in fracttional part are rounded
    String format1 = String.format("%.2f", 1.2399);
    System.out.println(format1); // "1.24"

    // three digits in fractional part are rounded
    String format2 = String.format("%.3f", 1.2399);
    System.out.println(format2); // "1.240"

    //rounded to two digits, filled with zero
    String format3 = String.format("%.2f", 1.2);
    System.out.println(format3); // returns "1.20"

    //rounder to two digits
    String format4 = String.format("%.2f", 3.19999);
```

```
    System.out.println(format4); // "3.20"
```

Числа с плавающей запятой могут быть отформатированы как десятичное число с использованием `DecimalFormat`

```
   // rounded with one digit fractional part
    String format = new DecimalFormat("0.#").format(4.3200);
    System.out.println(format); // 4.3

   // rounded with two digit fractional part
    String format = new DecimalFormat("0.##").format(1.2323000);
    System.out.println(format); //1.23

    // formatting floating numbers to decimal number
    double dv = 123456789;
    System.out.println(dv); // 1.23456789E8
    String format =  new DecimalFormat("0").format(dv);
    System.out.println(format); //123456789
```

Строгое соответствие спецификации IEEE

По умолчанию операции с плавающей запятой по `float` и `double` строго *не* соответствуют правилам спецификации IEEE 754. Выражению разрешено использовать расширения, специфичные для реализации, для диапазона этих значений; что позволяет им быть *более* точными, чем требуется.

`strictfp` отключает это поведение. Он применяется к классу, интерфейсу или методу и относится ко всему содержащемуся в нем, например к классам, интерфейсам, методам, конструкторам, инициализаторам переменных и т. Д. С `strictfp` промежуточные значения выражения с плавающей запятой *должны* быть в пределах установленное значение float или заданное двойное значение. Это приводит к тому, что результаты таких выражений будут точно такими, которые предсказывают спецификация IEEE 754.

Все константные выражения неявно строги, даже если они не входят в область `strictfp` .

Таким образом, `strictfp` имеет чистый эффект, иногда делая вычисления с `strictfp` углом зрения *менее* точными, а также может делать операции с плавающей точкой *медленнее* (поскольку ЦП теперь делает больше работы, чтобы гарантировать, что какая-либо собственная дополнительная точность не влияет на результат). Однако это также приводит к тому, что результаты будут одинаковыми на всех платформах. Поэтому он полезен в таких вещах, как научные программы, где воспроизводимость важнее скорости.

```
public class StrictFP { // No strictfp -> default lenient
    public strictfp float strict(float input) {
        return input * input / 3.4f; // Strictly adheres to the spec.
                                     // May be less accurate and may be slower.
    }

    public float lenient(float input) {
        return input * input / 3.4f; // Can sometimes be more accurate and faster,
```

```
                                    // but results may not be reproducable.
    }

    public static final strictfp class Ops { // strictfp affects all enclosed entities
        private StrictOps() {}

        public static div(double dividend, double divisor) { // implicitly strictfp
            return dividend / divisor;
        }
    }
}
```

Прочитайте Операции с плавающей точкой Java онлайн:
https://riptutorial.com/ru/java/topic/6167/операции-с-плавающей-точкой-java

глава 127: Ориентиры

Вступление

Написание тестов производительности в Java не так просто, как получение `System.currentTimeMillis()` в начале и в конце и вычисление разницы. Чтобы написать допустимые контрольные показатели производительности, следует использовать надлежащие инструменты.

Examples

Простой пример JMH

Одним из инструментов для написания надлежащих контрольных тестов является JMH . Предположим, мы хотим сравнить производительность поиска элемента в `HashSet` vs `TreeSet` .

Самый простой способ получить JHM в ваш проект - это использовать плагин maven и shade . Также вы можете увидеть `pom.xml` из примеров JHM .

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <finalName>/benchmarks</finalName>
                        <transformers>
                            <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                <mainClass>org.openjdk.jmh.Main</mainClass>
                            </transformer>
                        </transformers>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
```

```
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>1.18</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.18</version>
    </dependency>
</dependencies>
```

После этого вам нужно написать собственный класс тестов:

```
package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }

        stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
```

```
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testTreeSet(Blackhole blackhole) {
        blackhole.consume(treeSet.contains(stringToFind));
    }
}
```

Пожалуйста, имейте в виду этот `blackhole.consume()` , мы вернемся к нему позже. Также нам нужен основной класс для тестирования:

```
package benchmark;

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
                .include(CollectionFinderBenchmarkTest.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(options).run();
    }
}
```

И мы все настроены. Нам просто нужно запустить `mvn package` (он будет создавать `benchmarks.jar` в вашей `/target` папке) и запустить наш тестовый тест:

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

И после некоторых повторений разминки и расчета мы получим наши результаты:

```
# Run complete. Total time: 00:01:21

Benchmark                                    Mode  Cnt    Score    Error  Units
CollectionFinderBenchmarkTest.testHashSet    avgt   20    9.940 ±  0.270  ns/op
CollectionFinderBenchmarkTest.testTreeSet    avgt   20   98.858 ± 13.743  ns/op
```

Об этом `blackhole.consume()` . Если ваши вычисления не изменят состояние вашего приложения, java, скорее всего, просто проигнорирует его. Таким образом, чтобы избежать этого, вы можете либо заставить свои методы тестирования вернуть некоторую ценность, либо использовать объект `Blackhole` для его потребления.

Вы можете найти дополнительную информацию о написании надлежащих тестов в блоге Алексея Шипилева, в блоге Джейкоба Дженкова и в блоге java-performance: 1 , 2 .

Прочитайте Ориентиры онлайн: https://riptutorial.com/ru/java/topic/9514/ориентиры

глава 128: Основные управляющие структуры

замечания

Все управляющие структуры, если не указано иное, используют операторы блоков . Они обозначаются фигурными фигурными скобками `{}` .

Это отличается от обычных утверждений , которые *не* требуют фигурных скобок, но также имеют жесткую оговорку в том, что будет рассмотрена только строка, *непосредственно следующая* за предыдущим утверждением.

Таким образом, совершенно правильно написать любую из этих структур управления без фигурных скобок, пока только *одно* утверждение следует за началом, но оно ***сильно обескуражено*** , так как это может привести к ошибкам реализации или сломанному коду.

Пример:

```
// valid, but discouraged
Scanner scan = new Scanner(System.in);
int val = scan.nextInt();
if(val % 2 == 0)
    System.out.println("Val was even!");


// invalid; will not compile
// note the misleading indentation here
for(int i = 0; i < 10; i++)
    System.out.println(i);
    System.out.println("i is currently: " + i);
```

Examples

Если / Else If / Else Control

```
if (i < 2) {
  System.out.println("i is less than 2");
} else if (i > 2) {
  System.out.println("i is more than 2");
} else {
  System.out.println("i is not less than 2, and not more than 2");
}
```

Блок `if` будет работать только тогда, когда `i` равно 1 или меньше.

Условие `else if` проверяется только в том случае, если все условия перед ним (в

предыдущем, `else if` конструкторы и родительские, `if` конструкции) были проверены на `false` . В этом примере условие `else if` будет проверяться только в том случае, если `i` больше или равно 2.

Если его результат `true` , его блок запускается, а любое `else if` и `else` конструируется после того, как оно будет пропущено.

Если ни одно из условий `if` и `else if` не было проверено на `true` , блок `else` в конце будет запущен.

Для циклов

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

Эти три составляющих `for` цикла (разделенного `;`) являются описанием переменных / инициализации (здесь `int i = 0`), то условие (здесь `i < 100`), а инкремент утверждение (здесь `i++`). Объявление переменной выполняется один раз, как если бы оно помещалось только внутри `{` при первом запуске. Затем условие проверяется, если оно `true` тело цикла будет выполняться, если оно `false` цикл остановится. Предполагая, что цикл продолжается, тело выполнится, и, наконец, когда будет достигнуто значение `}` оператор инкремента будет выполняться непосредственно перед повторным проверкой условия.

Фигурные фигурные скобки необязательны (вы можете использовать одну строку с точкой с запятой), если цикл содержит только одно утверждение. Но всегда рекомендуется использовать фигурные скобки, чтобы избежать недоразумений и ошибок.

Компоненты цикла `for` являются необязательными. Если ваша бизнес-логика содержит одну из этих частей, вы можете опустить соответствующий компонент из цикла `for` .

```
int i = obj.getLastestValue(); // i value is fetched from a method

for (; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

Структура `for (;;) { function-body }` равна циклу `while (true)` .

`Nested For Loops`

Любой оператор цикла, имеющий другой оператор цикла внутри вложенного цикла. То же самое и для цикла с большим внутренним циклом называется «вложенным для цикла».

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
```

```
        }
        //Outer Loop Statements
    }
```

Вложенные для цикла могут быть продемонстрированы для печати чисел в форме треугольника.

```
for(int i=9;i>0;i--){//Outer Loop
    System.out.println();
    for(int k=i;k>0;k--){//Inner Loop -1
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++){//Inner Loop -2
        System.out.print(" "+j);
    }
 }
```

В то время как петли

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

В `while` цикл выполняется до тех пор , пока условие внутри скобок `true` . Это также называется структурой «pre-test loop», потому что условный оператор должен выполняться до того, как тело основного цикла будет выполняться каждый раз.

Кудрявые фигурные скобки необязательны, если цикл содержит только одно утверждение, но некоторые правила стиля кодирования предпочитают иметь скобки независимо.

do ... while Loop

Цикл `do...while` while отличается от других циклов тем, что он гарантированно выполняется **хотя бы один раз** . Он также называется структурой «посттест-цикл», потому что условный оператор выполняется после основного тела цикла.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

В этом примере цикл будет выполняться до тех пор, пока не будет напечатано число `100` (хотя условие равно `i < 100` а не `i <= 100`), поскольку условие цикла оценивается *после* выполнения цикла.

С гарантией хотя бы одного выполнения можно объявить переменные за пределами цикла и инициализировать их внутри.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

В этом контексте `theWord` определяется вне цикла, но поскольку он гарантированно имеет значение, основанное на его естественном потоке, `theWord` будет инициализирован.

Для каждого

Java SE 5

С помощью Java 5 и выше можно использовать для каждого цикла, также известные как расширенные для циклов:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
    System.out.println(string);
}
```

Для каждой петли можно использовать для итерации по массивам и реализации интерфейса `Iterable` , а позднее - классы Collections , такие как `List` или `Set` .

Переменная цикла может быть любого типа, которая может быть назначена из типа источника.

Переменная цикла для расширенного цикла для `Iterable<T>` или `T[]` может быть типа `S` , если

- `T extends S`
- как `T` и `S` являются примитивными типами и присваиваются без литья
- `S` является примитивным типом, и `T` может быть преобразован в тип, назначаемый `S` после преобразования с помощью распаковки.
- `T` является примитивным типом и может быть преобразован в `S` путем преобразования с помощью автоматического преобразования.

Примеры:

```
T elements = ...
for (S s : elements) {
}
```

T	S	Собирает
ИНТ []	долго	да
долго[]	ИНТ	нет
`Iterable<Byte>`	долго	да
`Iterable<String>`	CharSequence	да
`Iterable<CharSequence>`	строка	нет
ИНТ []	Долго	нет
ИНТ []	целое число	да

Если еще

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

Оператор `if` выполняет код условно в зависимости от результата условия в круглых скобках. Когда условие в скобках истинно, оно будет вводиться в блок оператора if, который определяется фигурными фигурными скобками типа `{` и `}` . открывающий кронштейн до закрытия скобки - это объем инструкции if.

Блок `else` является необязательным и может быть опущен. Он работает, если оператор `if` является `false` и не запускается, `if` оператор `if` истинен, потому что в этом случае выполняется оператор `if` .

См. Также: Тернар Если

Оператор switch

Оператор `switch` представляет собой многопроцессорную инструкцию Java. Он используется для замены длинных цепочек `if else if else` и делает их более читаемыми. Однако, в отличие от , `if` это заявления, один может не использовать неравенство; каждое значение должно быть конкретно определено.

В оператор `switch` есть три важных компонента:

- `case` : Это значение, которое оценивается для эквивалентности с аргументом инструкции `switch` .
- `default` : Это необязательное выражение catch-all, если ни одно из операторов `case` равно `true` .
- Резкое завершение заявления `case` ; обычно `break` : это необходимо для предотвращения нежелательной оценки дальнейших заявлений `case` .

За исключением `continue` , можно использовать любой оператор, который может вызвать резкое завершение инструкции . Это включает:

- `break`
- `return`
- `throw`

В приведенном ниже примере типичный оператор `switch` записывается с четырьмя возможными случаями, включая `default` .

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i is zero");
        break;
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is less than zero or greater than two");
}
```

Путем пропадания `break` или любого заявления, которое было бы внезапным завершением, мы можем использовать так называемые «провальные» случаи, которые оцениваются по нескольким значениям. Это можно использовать для создания диапазонов для того, чтобы значение было успешным против, но оно по-прежнему не столь гибко, как неравенство.

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("I'm equal or greater than one");
    case 2:
    case 3:
        System.out.println("I'm one, two, or three");
        break;
    default:
        System.out.println("I'm not either one, two, or three");
}
```

В случае `foo == 1` выход будет:

```
I'm equal or greater than one
I'm one, two, or three
```

В случае `foo == 3` вывод будет:

```
I'm one, two, or three
```

Java SE 5

Оператор switch также может использоваться с `enum` .

```
enum Option {
    BLUE_PILL,
    RED_PILL
}

public void takeOne(Option option) {
    switch(option) {
        case BLUE_PILL:
            System.out.println("Story ends, wake up, believe whatever you want.");
            break;
        case RED_PILL:
            System.out.println("I show you how deep the rabbit hole goes.");
            break;

    }
}
```

Java SE 7

Оператор `switch` также может использоваться с `String` s.

```
public void rhymingGame(String phrase) {
    switch (phrase) {
        case "apples and pears":
            System.out.println("Stairs");
            break;
        case "lorry":
            System.out.println("truck");
            break;
        default:
            System.out.println("Don't know any more");
    }
}
```

Тернарный оператор

Иногда вам нужно проверить условие и установить значение переменной.

Напр.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

Это можно легко записать в одной строке, так как

```
String name = A > B ? "Billy" : "Jimmy";
```

Значение переменной устанавливается в значение сразу после условия, если условие истинно. Если условие ложно, второе значение будет присвоено переменной.

Перерыв

Оператор `break` заканчивает цикл (например, `for` , `while`) или оценку оператора switch .

Loop:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
}
```

Цикл в примере будет работать вечно. Но когда `someCondition` равно `5` в какой-то момент исполнения, цикл заканчивается.

Если несколько циклов каскадируются, только самый внутренний цикл заканчивается использованием `break` .

Попробуйте ... Поймать ... Наконец

Для обработки Исключений используется структура управления `try { ... } catch ( ... ) { ... }` .

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
}
```

Это напечатает:

Неправильный ввод. «abc» не является допустимым целым числом.

После `catch` можно добавить предложение `finally` . Предложение `finally` всегда будет выполняться независимо от того, было ли исключено исключение.

```
try { ... } catch ( ... ) { ... } finally { ... }
```

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
} finally {
    System.out.println("This code will always be run, even if an exception is thrown");
}
```

Это напечатает:

Неправильный ввод. «abc» не является допустимым целым числом.
Этот код всегда будет запущен, даже если исключено исключение

Вложенный разрыв / продолжение

Можно `break` / `continue` внешний цикл с помощью операторов ярлыков:

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

Другого использования для ярлыков в Java нет.

Продолжить вывод в Java

Оператор continue используется, чтобы пропустить оставшиеся шаги в текущей итерации и начать с следующей итерации цикла. Элемент управления переходит из инструкции `continue` к значению шага (приращение или декремент), если таковые имеются.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};
```

```
    //john is not printed out
    for (String name : programmers) {
        if (name.equals("John"))
            continue;
        System.out.println(name);
    }
```

Оператор `continue` также может управлять переключением программы на значение шага (если оно есть) именованного цикла:

```
Outer: // The name of the outermost loop is kept here as 'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

Прочитайте Основные управляющие структуры онлайн:
https://riptutorial.com/ru/java/topic/118/основные-управляющие-структуры

глава 129: Отправка динамического метода

Вступление

Что такое диспетчер динамического метода?

Dynamic Method Dispatch - это процесс, в котором вызов переопределенного метода разрешается во время выполнения, а не во время компиляции. Когда переопределенный метод вызывается ссылкой, Java определяет, какая версия этого метода должна выполняться в зависимости от типа объекта, к которому он относится. Это также известно как полиморфизм времени выполнения.

Мы увидим это на примере.

замечания

- Динамическое связывание = Поздняя привязка
- Абстрактные классы не могут быть созданы, но они могут быть подклассифицированы (Base для дочернего класса)
- Абстрактным методом является метод, объявленный без реализации
- Абстрактный класс может содержать сочетание методов, объявленных с реализацией или без нее
- Когда абстрактный класс подклассифицирован, подкласс обычно предоставляет реализации для всех абстрактных методов в его родительском классе. Однако, если это не так, то подкласс также должен быть объявлен абстрактным
- Рассылка динамического метода - это механизм, с помощью которого вызов переопределенного метода разрешается во время выполнения. Так Java реализует полиморфизм времени выполнения.
- Upcasting: переход подтипа к супертипу, вверх к дереву наследования.
- Полиморфизм времени выполнения = динамический полиморфизм

Examples

Отправка динамического метода - пример кода

Аннотация Класс:

```
package base;

/*
Abstract classes cannot be instantiated, but they can be subclassed
*/
public abstract class ClsVirusScanner {
```

```
    //With One Abstract method
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
        System.out.println("Perform Virus Scanner Version Check");
    }

    protected void fnBootTimeScan(){
        System.out.println("Perform BootTime Scan");
    }
    protected void fnInternetSecutiry(){
        System.out.println("Scan for Internet Security");
    }

    protected void fnRealTimeScan(){
        System.out.println("Perform RealTime Scan");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("Detect Virus & Malware");
    }
}
```

Переопределение абстрактного метода в классе детей:

```
import base.ClsVirusScanner;

//All the 3 child classes inherits the base class ClsVirusScanner
//Child Class 1
class ClsPaidVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnCheckForUpdateVersion();
        super.fnBootTimeScan();
        super.fnInternetSecutiry();
        super.fnRealTimeScan();
        super.fnVirusMalwareScan();
    }
};    //ClsPaidVersion IS-A ClsVirusScanner
//Child Class 2

class ClsTrialVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnInternetSecutiry();
        super.fnVirusMalwareScan();
    }
};    //ClsTrialVersion IS-A ClsVirusScanner

//Child Class 3
class ClsFreeVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnVirusMalwareScan();
    }
};     //ClsTrialVersion IS-A ClsVirusScanner
```

Динамическое / позднее связывание приводит к отправке динамического

метода:

```
//Calling Class
public class ClsRunTheApplication {

    public static void main(String[] args) {

        final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";

        //Parent Refers Null
        ClsVirusScanner objVS=null;

        //String Cases Supported from Java SE 7
        switch (VIRUS_SCANNER_VERSION){
        case "FREE_VERSION":

            //Parent Refers Child Object 3
            //ClsFreeVersion IS-A ClsVirusScanner
            objVS = new ClsFreeVersion(); //Dynamic or Runtime Binding
            break;
        case "PAID_VERSION":

            //Parent Refers Child Object 1
            //ClsPaidVersion IS-A ClsVirusScanner
            objVS = new ClsPaidVersion(); //Dynamic or Runtime Binding
            break;
        case "TRIAL_VERSION":

            //Parent Refers Child Object 2
            objVS = new ClsTrialVersion(); //Dynamic or Runtime Binding
            break;
        }

        //Method fnStartScan() is the Version of ClsTrialVersion()
        objVS.fnStartScan();

    }
}
```

Результат:

```
Scan for Internet Security
Detect Virus & Malware
```

Ускорение:

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Прочитайте Отправка динамического метода онлайн: https://riptutorial.com/ru/java/topic/9204/отправка-динамического-метода

глава 130: Оценка XML XPath

замечания

Выражения XPath используются для навигации и выбора одного или нескольких узлов в документе дерева XML, например, для выбора определенного элемента или узла атрибута.

См. Эту рекомендацию W3C для справки на этом языке.

Examples

Оценка NodeList в XML-документе

Учитывая следующий XML-документ:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

Следующие извлекают все узлы `example` для тега Java (используйте этот метод, если только один раз оценивать XPath в XML. См. Другой пример, когда несколько вызовов XPath оцениваются в одном XML-файле.):

```
XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //Make new XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource,
```

```
XPathConstants.NODESET); //Evaluate the XPath
...
```

Разбор нескольких выражений XPath в одном XML

Используя тот же пример, что и **оценка NodeList в XML-документе** , вы можете эффективно использовать несколько вызовов XPath:

Учитывая следующий XML-документ:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

Вот как вы могли бы использовать XPath для оценки нескольких выражений в одном документе:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath
xPath.reset(); //Resets the xPath so it can be used again
NodeList androidExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc,
XPathConstants.NODESET); //Evaluate the XPath

...
```

Разбор одного выражения XPath несколько раз в XML

В этом случае вы хотите, чтобы выражение было скомпилировано перед оценками, поэтому каждый вызов для `evaluate` не `compile` одно и то же выражение. Простым синтаксисом будет:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Evaluate
the XPath from the already-compiled expression

NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Do it
again
```

В целом, два вызова `XPathExpression.evaluate()` будут намного эффективнее, чем два вызова `XPath.evaluate()` .

Прочитайте Оценка XML XPath онлайн: https://riptutorial.com/ru/java/topic/4148/оценка-xml-xpath

глава 131: Очереди и Deques

Examples

Использование PriorityQueue

`PriorityQueue` - это структура данных. Подобно `SortedSet` , `PriorityQueue` также сортирует свои элементы, основываясь на своих приоритетах. Прежде всего, элементы, имеющие более высокий приоритет. Тип `PriorityQueue` должен реализовывать `comparable` интерфейс или интерфейс `comparator` , методы которого определяют приоритеты элементов структуры данных.

```
//The type of the PriorityQueue is Integer.
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

//The elements are added to the PriorityQueue
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );

//The PriorityQueue sorts the elements by using compareTo method of the Integer Class
//The head of this queue is the least element with respect to the specified ordering
System.out.println( queue );  //The Output: [1, 2, 3, 9, 3, 8]
queue.remove();
System.out.println( queue );  //The Output: [2, 3, 3, 9, 8]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 3, 9]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 9]
queue.remove();
System.out.println( queue );  //The Output: [8, 9]
queue.remove();
System.out.println( queue );  //The Output: [9]
queue.remove();
System.out.println( queue );  //The Output: []
```

LinkedList как очередь FIFO

Класс `java.util.LinkedList` при реализации `java.util.List` является универсальной реализацией интерфейса `java.util.Queue` также работает с принципом FIFO (First In, First Out) .

В приведенном ниже примере с помощью метода `offer()` элементы вставляются в `LinkedList` . Эта операция вставки называется `enqueue` . В `while` петли ниже, элементы удаляются из `Queue` на основе FIFO. Эта операция называется `dequeue` .

```
Queue<String> queue = new LinkedList<String>();

queue.offer( "first element" );
queue.offer( "second element" );
queue.offer( "third element" );
```

```
queue.offer( "fourth. element" );
queue.offer( "fifth. element" );

while ( !queue.isEmpty() ) {
  System.out.println( queue.poll() );
}
```

Вывод этого кода

```
first element
second element
third element
fourth element
fifth element
```

Как видно на выходе, первый вставленный элемент «первый элемент» удаляется во-первых, «второй элемент» удаляется во втором месте и т. Д.

Стеки

Что такое стек?

В Java Stacks представляют собой структуру данных LIFO (Last In, First Out) для объектов.

API стека

Java содержит API стека со следующими методами:

```
Stack()            //Creates an empty Stack
isEmpty()          //Is the Stack Empty?            Return Type: Boolean
push(Item item)    //push an item onto the stack
pop()              //removes item from top of stack  Return Type: Item
size()             //returns # of items in stack     Return Type: Int
```

пример

```
import java.util.*;

public class StackExample {

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);

      st.push(10);
      System.out.println("10 was pushed to the stack");
      System.out.println("stack: " + st);
```

```
      st.push(15);
      System.out.println("15 was pushed to the stack");
      System.out.println("stack: " + st);

      st.push(80);
      System.out.println("80 was pushed to the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("80 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("15 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("10 was popped from the stack");
      System.out.println("stack: " + st);

     if(st.isEmpty())
      {
         System.out.println("empty stack");
      }
   }
}
```

Это возвращает:

```
stack: []
10 was pushed to the stack
stack: [10]
15 was pushed to the stack
stack: [10, 15]
80 was pushed to the stack
stack: [10, 15, 80]
80 was popped from the stack
stack: [10, 15]
15 was popped from the stack
stack: [10]
10 was popped from the stack
stack: []
empty stack
```

BlockingQueue

BlockingQueue - это интерфейс, который представляет собой очередь, которая блокируется, когда вы пытаетесь удалить из нее, и очередь пуста, или если вы пытаетесь вставить в нее элементы, а очередь уже заполнена. Поток, пытающийся удалить из пустой очереди, блокируется до тех пор, пока какой-либо другой поток не вставит элемент в очередь. Нить, пытающаяся присвоить элемент в полной очереди, блокируется до тех пор, пока какой-либо другой поток не освободит место в очереди, либо путем удаления одного или нескольких элементов, либо очистки всей очереди.

Методы BlockingQueue представлены в четырех формах с различными способами обработки операций, которые не могут быть удовлетворены немедленно, но могут быть удовлетворены в какой-то момент в будущем: один генерирует исключение, второй возвращает специальное значение (либо null, либо false, в зависимости от операция), третий блокирует текущую нить неопределенно до тех пор, пока операция не будет успешной, а четвертые блоки только для заданного максимального срока до отказа.

операция	Выбрасывает исключение	Специальная ценность	Блоки	Время вышло
Вставить	добавлять()	предложение (е)	ставить (е)	предложение (е, время, единица)
Удалить	Удалить()	опрос()	брать ()	опрос (время, единица)
исследовать	элемент()	PEEK ()	N / A	N / A

`BlockingQueue` может быть **ограниченным** или **неограниченным** . Ограниченный BlockingQueue - это тот, который инициализируется начальной загрузкой.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Любые вызовы метода put () будут заблокированы, если размер очереди равен начальной заданной емкости.

Неограниченная очередь - это то, которое инициализируется без емкости, фактически по умолчанию оно инициализируется Integer.MAX_VALUE.

Некоторые общие реализации `BlockingQueue` :

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Теперь давайте рассмотрим пример `ArrayBlockingQueue` :

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);
bQueue.put("This is entry 1");
System.out.println("Entry one done");
bQueue.put("This is entry 2");
System.out.println("Entry two done");
bQueue.put("This is entry 3");
System.out.println("Entry three done");
```

Это напечатает:

```
Entry one done
Entry two done
```

И поток будет заблокирован после второго выхода.

Интерфейс очереди

основы

`Queue` представляет собой набор для хранения элементов перед обработкой. Очереди обычно, но не обязательно, упорядочивают элементы в режиме FIFO (first-in-first-out).

Глава очереди - это элемент, который будет удален вызовом для удаления или опроса. В очереди FIFO все новые элементы вставляются в хвост очереди.

Интерфейс очереди

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Каждый метод `Queue` существует в двух формах:

- один генерирует исключение, если операция завершается с ошибкой;
- другой возвращает специальное значение, если операция завершается с ошибкой (либо `null` либо `false` зависимости от операции.

Тип операции	Исключение исключений	Возвращает специальное значение
Вставить	`add(e)`	`offer(e)`
Удалить	`remove()`	`poll()`
исследовать	`element()`	`peek()`

Deque

`Deque`

- это «двойная очередь», что означает, что элементы могут быть добавлены спереди или хвостом очереди. Очередь может добавлять элементы в хвост очереди.

`Deque` наследует интерфейс `Queue` что означает, что регулярные методы остаются, однако интерфейс Deque предлагает дополнительные методы, чтобы быть более гибкими с очередью. Дополнительные методы действительно говорят сами за себя, если вы знаете, как работает очередь, поскольку эти методы призваны повысить гибкость:

метод	Краткое описание
`getFirst()`	Получает первый элемент **главы** очереди, не удаляя ее.
`getLast()`	Получает первый элемент **хвоста** очереди, не удаляя его.
`addFirst(E e)`	Добавляет элемент в **голову** очереди
`addLast(E e)`	Добавляет элемент в **хвост** очереди
`removeFirst()`	Удаляет первый элемент в **голове** очереди
`removeLast()`	Удаляет первый элемент в **хвосте** очереди

Конечно, доступны те же варианты `offer` , `poll` и `peek` , однако они не работают с исключениями, а скорее со специальными значениями. Нет смысла показывать, что они здесь делают.

Добавление и доступ к элементам

Чтобы добавить элементы в хвост Deque, вы вызываете его метод `add()` . Вы также можете использовать `addFirst()` и `addLast()` , которые добавляют элементы в голову и хвост deque.

```
Deque<String> dequeA = new LinkedList<>();

dequeA.add("element 1");      //add element at tail
dequeA.addFirst("element 2"); //add element at head
dequeA.addLast("element 3");  //add element at tail
```

Вы можете заглянуть в элемент во главе очереди, не вытаскивая элемент из очереди. Это делается с помощью метода `element()` . Вы также можете использовать методы `getFirst()` и `getLast()` , которые возвращают первый и последний элемент в `Deque` . Вот как это выглядит:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Удаление элементов

Чтобы удалить элементы из deque, вы вызываете методы `remove()` , `removeFirst()` и `removeLast()` . Вот несколько примеров:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Прочитайте Очереди и Deques онлайн: https://riptutorial.com/ru/java/topic/7196/очереди-и-deques

глава 132: Ошибки Java - Nulls и NullPointerException

замечания

Значение `null` является значением по умолчанию для неинициализированного значения поля, тип которого является ссылочным типом.

Исключение `NullPointerException` (или NPE) - это исключение, которое возникает при попытке выполнить ненадлежащую операцию над ссылкой на `null` объект. К таким операциям относятся:

- вызов метода экземпляра на `null` целевом объекте,
- доступ к полю `null` целевого объекта,
- пытаясь проиндексировать `null` массив или получить доступ к его длине,
- используя ссылку на `null` объект в качестве мьютекса в `synchronized` блоке,
- литье ссылки на `null` объект,
- unboxing ссылку на `null` объект и
- бросая ссылку на `null` объект.

Наиболее распространенные первопричины для NPE:

- забыв инициализировать поле с эталонным типом,
- забывая инициализировать элементы массива ссылочного типа или
- не проверяя результаты определенных методов API, которые *указаны* как возвращающие `null` в определенных обстоятельствах.

Примеры обычно используемых методов, возвращающих значение `null` включают:

- Метод `get(key)` в API `Map` возвращает `null` если вы вызываете его с помощью ключа, который не имеет сопоставления.
- `getResource(path)` и `getResourceAsStream(path)` в API `ClassLoader` и `Class` API возвращают значение `null` если ресурс не может быть найден.
- Метод `get()` в `Reference` API возвращает `null` если сборщик мусора очистил ссылку.
- Различные методы `getXxxx` в API-интерфейсах Java EE возвращают значение `null` если вы `getXxxx` извлечь необязательный параметр запроса, атрибут сеанса или сеанса и т. Д.

Существуют стратегии избежания нежелательных NPE, таких как явное тестирование `null` или использование «Обозначения Йоды», но эти стратегии часто имеют нежелательный результат *скрытия* проблем в вашем коде, который действительно должен быть исправлен.

Examples

Pitfall - Ненужное использование примитивных оберток может привести к NullPointerExceptions

Иногда программисты, которые являются новыми Java, будут использовать примитивные типы и обертки взаимозаменяемо. Это может привести к проблемам. Рассмотрим этот пример:

```
public class MyRecord {
    public int a, b;
    public Integer c, d;
}

...
MyRecord record = new MyRecord();
record.a = 1;               // OK
record.b = record.b + 1;    // OK
record.c = 1;               // OK
record.d = record.d + 1;    // throws a NullPointerException
```

Наш класс `MyRecord` [1] использует инициализацию по умолчанию для инициализации значений в своих полях. Таким образом, когда мы записываем `new` запись, поля `a` и `b` будут установлены на ноль, а поля `c` и `d` будут установлены в `null` .

Когда мы пытаемся использовать инициализированные поля по умолчанию, мы видим, что поля `int` работают все время, но поля `Integer` работают в некоторых случаях, а не другие. В частности, в случае, когда не удается (с `d`), что происходит в том , что выражение на правой стороне пытается распаковывать с `null` ссылки, и это приводит к тому , что `NullPointerException` быть выброшен.

Есть несколько способов взглянуть на это:

- Если поля `c` и `d` должны быть примитивными оболочками, то либо мы не должны полагаться на инициализацию по умолчанию, либо мы должны тестировать значение `null` . Для прежнего это правильный подход, *если* для полей в `null` состоянии *не* существует определенного значения.
- Если поля не должны быть примитивными обертками, то ошибочно сделать их примитивными обертками. В дополнение к этой проблеме примитивные обертки имеют дополнительные накладные расходы по сравнению с примитивными типами.

Урок здесь заключается в том, чтобы не использовать примитивные типы оберток, если вам действительно не нужно.

---

1 - Этот класс не является примером хорошей практики кодирования. Например, у хорошо продуманного класса не было бы публичных полей. Однако это не относится к этому примеру.

Pitfall - использование null для представления пустого массива или коллекции

Некоторые программисты считают, что это хорошая идея, чтобы сэкономить место, используя `null` для представления пустого массива или коллекции. Хотя это правда, что вы можете сэкономить небольшое пространство, обратная сторона заключается в том, что он делает ваш код более сложным и более хрупким. Сравните эти две версии метода суммирования массива:

Первая версия - это то, как вы обычно кодируете метод:

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed
 * @return the sum
 **/
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```

Вторая версия - это то, как вам нужно закодировать метод, если вы привыкли использовать `null` для представления пустого массива.

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed, or null.
 * @return the sum, or zero if the array is null.
 **/
public int sum(int[] values) {
    int sum = 0;
    if (values != null) {
        for (int value : values) {
            sum += value;
        }
    }
    return sum;
}
```

Как вы можете видеть, код немного сложнее. Это напрямую связано с решением использовать `null` таким образом.

Теперь рассмотрим, используется ли этот массив, который может быть `null` во многих местах. В каждом месте, где вы его используете, вам нужно проверить, нужно ли вам проверять значение `null` . Если вы пропустите `null` тест, который должен быть там, вы рискуете `NullPointerException` . Следовательно, стратегия использования `null` в этом случае приводит к тому, что ваше приложение становится более хрупким; т.е. более уязвимы к последствиям ошибок программиста.

Урок здесь состоит в том, чтобы использовать пустые массивы и пустые списки, когда это то, что вы имеете в виду.

```
int[] values = new int[0];                      // always empty
List<Integer> list = new ArrayList();           // initially empty
List<Integer> list = Collections.emptyList();  // always empty
```

Недостаток пространства небольшой, и есть другие способы свести его к минимуму, если это стоит того.

Pitfall - «Создание хороших» неожиданных нулей

В StackOverflow мы часто видим такой код в ответах:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

Часто это сопровождается утверждением, которое является «лучшей практикой» для проверки `null` чтобы избежать исключения `NullPointerException` .

Это лучшая практика? Короче: Нет.

Есть некоторые основополагающие предположения, которые необходимо поставить под сомнение, прежде чем мы сможем сказать, если это хорошая идея сделать это в наших `joinStrings` :

Что значит для «a» или «b» быть нулевым?

Значение `String` может быть равно нулю или больше символов, поэтому у нас уже есть способ представления пустой строки. `null` означает что-то другое, чем `""` ? Если нет, то проблематично иметь два способа представления пустой строки.

Был ли null получен из неинициализированной переменной?

`null` может исходить из неинициализированного поля или неинициализированного элемента массива. Значение может быть неинициализировано по дизайну или случайно. Если это было случайно, это ошибка.

Указывает ли нуль значение «не знаю» или «отсутствует»?

Иногда `null` может иметь подлинный смысл; например, что реальное значение переменной неизвестно или недоступно или «необязательно». В Java 8 класс `Optional` предоставляет лучший способ выразить это.

Если это ошибка (или ошибка дизайна), мы должны «сделать хорошо»?

Одна из интерпретаций кода заключается в том, что мы «делаем хороший» неожиданный `null` , используя пустую строку на своем месте. Правильная ли стратегия? Было бы лучше позволить `NullPointerException` , а затем поймать исключение дальше по стеку и зарегистрировать его как ошибку?

Проблема с «хорошим достижением» заключается в том, что она может либо скрыть проблему, либо затруднить диагностику.

Является ли это эффективным / хорошим для качества кода?

Если подход «сделать хороший» используется последовательно, ваш код будет содержать множество «защитных» нулевых тестов. Это сделает его более длинным и трудным для чтения. Более того, все эти тесты и «делать добро» могут повлиять на производительность вашего приложения.

В итоге

Если значение `null` является значимым значением, то проверка `null` случая - правильный подход. Следствием является то, что если значение `null` имеет смысл, то это должно быть четко документировано в javadocs любых методов, которые принимают `null` значение или возвращают его.

В противном случае лучше рассмотреть неожиданный `null` как ошибку программирования, и пусть `NullPointerException` произойдет, чтобы разработчик узнал, что в коде есть проблема.

Pitfall - Возвращение null вместо исключения исключения

Некоторые Java-программисты имеют общее отвращение к выбросам или распространению исключений. Это приводит к следующему коду:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

Так в чем проблема?

Проблема в том, что `getReader` возвращает значение `null` в качестве специального значения, чтобы указать, что `Reader` не может быть открыт. Теперь возвращаемое значение нужно протестировать, чтобы проверить, не является ли оно значением `null` до его использования. Если тест не учитывается, результатом будет исключение `NullPointerException` .

Здесь есть три проблемы:

1. `IOException` было обнаружено слишком рано.
2. Структура этого кода означает, что существует риск утечки ресурса.
3. Затем был `null` потому что не было доступно «реальное» `Reader` .

На самом деле, предполагая, что исключение нужно было поймать раньше, было несколько альтернатив возврату `null` :

1. Можно было бы реализовать класс `NullReader` ; например, когда операции API ведут себя так, как если бы читатель уже находился в позиции «конец файла».
2. С Java 8 можно было бы объявить `getReader` как возвращающий `Optional<Reader>` .

Pitfall - не проверяет, не инициализирован ли поток ввода-вывода при его закрытии

Чтобы предотвратить утечку памяти, не следует забывать закрыть входной поток или выходной поток, чья работа выполнена. Обычно это делается с помощью заявления `try catch finally` без части `catch` :

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

Хотя приведенный выше код может выглядеть невинно, у него есть недостаток, который

может сделать отладку невозможной. Если строка, где `out` инициализирована (`out = new FileOutputStream(filename)`), генерирует исключение, тогда `out` будет `null` когда `out.close()` , что приводит к неприятному исключению `NullPointerException` !

Чтобы этого избежать, просто убедитесь, что поток не имеет `null` прежде чем пытаться его закрыть.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

Еще лучший подход - `try` -with-resources, так как он автоматически закрывает поток с вероятностью 0, чтобы выбросить NPE без необходимости блока `finally` .

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Pitfall - использование «нотации Yoda», чтобы избежать NullPointerException

Многие примеры кода, размещенные в StackOverflow, включают в себя следующие фрагменты:

```
if ("A".equals(someString)) {
    // do something
}
```

Это предотвращает или предотвращает возможное исключение `NullPointerException` в случае, если `someString` имеет значение `null` . Кроме того, можно утверждать, что

```
    "A".equals(someString)
```

лучше, чем:

```
    someString != null && someString.equals("A")
```

(Это более красноречиво, и в некоторых случаях это может быть более эффективным.

Однако, как мы утверждаем ниже, краткость может быть отрицательной.)

Тем не менее, реальная ловушка использует тест Йоды, **чтобы избежать `NullPointerExceptions` в качестве привычки.**

Когда вы пишете `"A".equals(someString)` вы на самом деле «делаете хорошо», когда `someString` имеет значение `null` . Но в качестве другого примера (Pitfall - «Создание хороших» неожиданных нулей) объясняет, что «создание хороших» `null` значений может быть вредным по целому ряду причин.

Это означает, что условия Йоды не являются «лучшей практикой» [1] . Если не ожидается `null` , лучше разрешить `NullPointerException` , чтобы вы могли получить отказ единичного теста (или отчет об ошибке). Это позволяет вам найти и исправить ошибку, которая вызвала появление неожиданного / нежелательного `null` .

Условия Yoda должны использоваться только в тех случаях, когда *ожидается* `null` потому что объект, который вы тестируете, исходит из API, который *документирован* как возвращающий `null` . И, возможно, лучше использовать один из менее привлекательных способов выражения теста, потому что это помогает выделить `null` тест тому, кто просматривает ваш код.

---

1 - Согласно Википедии : *«Лучшие методы кодирования - это набор неформальных правил, которые сообщество разработчиков программного обеспечения со временем узнало, что может помочь улучшить качество программного обеспечения».* , Использование нотации Yoda этого не достигает. Во многих ситуациях это делает код хуже.

Прочитайте Ошибки Java - Nulls и NullPointerException онлайн:
https://riptutorial.com/ru/java/topic/5680/ошибки-java---nulls-и-nullpointerexception

глава 133: Ошибки Java - потоки и параллелизм

Examples

Pitfall: неправильное использование wait () / notify ()

Методы `object.wait()` , `object.notify()` и `object.notifyAll()` предназначены для использования очень определенным образом. (см. http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

Проблема «потерянного оповещения»

Одна общая ошибка начинающего заключается в том, чтобы безоговорочно вызвать `object.wait()`

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

Причина, по которой это неверно, заключается в том, что в зависимости от какого-то другого потока можно вызвать `lock.notify()` или `lock.notifyAll()` , но ничто не гарантирует, что другой поток не сделал этот вызов *перед* потребительским потоком, называемым `lock.wait()` .

`lock.notify()` и `lock.notifyAll()` ничего не делать вообще , если какой -либо другой поток *уже* не дожидаясь уведомления. Поток, который вызывает `myConsumer()` в этом примере, будет зависать вечно, если слишком поздно поймать уведомление.

Ошибка «Недопустимое состояние монитора»

Если вы вызываете `wait()` или `notify()` на объект, не удерживая блокировку, тогда JVM будет `IllegalMonitorStateException` .

```
public void myConsumer() {
    lock.wait();      // throws exception
    consume();
}

public void myProducer() {
```

```
    produce();
    lock.notify();    // throws exception
}
```

(Конструкция `wait()` / `notify()` требует блокировки, потому что это необходимо для того, чтобы избежать системных условий гонки. Если бы можно было вызвать `wait()` или `notify()` без блокировки, тогда было бы невозможно реализовать основной пример использования этих примитивов: ожидание возникновения условия.)

Ожидание / уведомление слишком низкоуровневое

Лучший способ избежать проблем с `wait()` и `notify()` - не использовать их. Большинство проблем синхронизации можно решить, используя объекты синхронизации более высокого уровня (очереди, барьеры, семафоры и т. Д.), Которые доступны в пакете `java.utils.concurrent` .

Pitfall - расширение 'java.lang.Thread'

В javadoc для класса `Thread` показаны два способа определения и использования потока:

Использование пользовательского класса потоков:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Использование `Runnable` :

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeRun p = new PrimeRun(143);
```

```
new Thread(p).start();
```

(Источник: `java.lang.Thread` javadoc .)

Подход к пользовательскому потоку классов работает, но у него есть несколько проблем:

1. `PrimeThread` использовать `PrimeThread` в контексте, который использует классический пул потоков, исполнитель или инфраструктуру ForkJoin. (Это невозможно, потому что `PrimeThread` косвенно реализует `Runnable` , но использование пользовательского класса `Thread` как `Runnable` , безусловно, неудобно и может быть нецелесообразным ... в зависимости от других аспектов класса.)

2. В других методах есть больше возможностей для ошибок. Например, если вы объявили `PrimeThread.start()` без делегирования `Thread.start()` , вы получите «поток», который запускался в текущем потоке.

Подход к тому, чтобы логика потока в `Runnable` избегала этих проблем. Действительно, если вы используете анонимный класс (Java 1.1 onwards) для реализации `Runnable` результат будет более кратким и более читаемым, чем приведенные выше примеры.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}.start();
```

С выражением лямбда (Java 8 и далее) вышеупомянутый пример станет еще более элегантным:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Pitfall - Слишком много потоков делает приложение медленнее.

Многие люди, которые новичок в многопоточности, считают, что использование потоков автоматически делает приложение быстрее. На самом деле, это намного сложнее. Но одна вещь, которую мы можем с уверенностью сказать, заключается в том, что для любого компьютера существует ограничение на количество потоков, которые могут быть запущены одновременно:

- Компьютер имеет фиксированное количество *ядер* (или *гиперпотоков*).
- Для запуска Java-потока необходимо *назначить* ядро или гиперпоточность.
- Если есть больше запущенных потоков Java, чем (доступных) ядер / гиперпотоков,

некоторые из них должны ждать.

Это говорит о том, что простое создание все большего количества потоков Java *не может* заставить приложение работать быстрее и быстрее. Но есть и другие соображения:

- Для каждого потока требуется область памяти без кучи для стека потока. Типичный (по умолчанию) размер стека потоков составляет 512 Кбайт или 1 Мбайт. Если у вас есть значительное количество потоков, использование памяти может быть значительным.
- Каждый активный поток будет ссылаться на несколько объектов в куче. Это увеличивает рабочий набор *достижимых* объектов, что влияет на сбор мусора и на использование физической памяти.
- Накладные расходы на переключение между потоками нетривиальны. Обычно это приводит к переключению в пространство ядра ОС, чтобы принять решение о планировании потоков.
- Накладные расходы на синхронизацию потоков и межпоточную сигнализацию (например, wait (), notify () / notifyAll) *могут быть* значительными.

В зависимости от деталей вашего приложения эти факторы обычно означают, что для количества потоков существует «сладкое пятно». Помимо этого, добавление большего количества потоков обеспечивает минимальное улучшение производительности и может ухудшить производительность.

Если ваше приложение создается для каждой новой задачи, неожиданное увеличение рабочей нагрузки (например, высокая скорость запроса) может привести к катастрофическому поведению.

Лучший способ справиться с этим - использовать ограниченный пул потоков, размер которого вы можете контролировать (статически или динамически). Когда требуется слишком много работы, приложение должно поставить в очередь запросы. Если вы используете `ExecutorService` , он позаботится об управлении пулом потоков и очереди задач.

Pitfall - создание темы относительно дорого

Рассмотрим эти два микро-теста:

Первый тест просто создает, запускает и соединяет потоки. `Runnable` потока не работает.

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
```

```
                Thread t = new Thread(new Runnable() {
                        public void run() {
                }});
                t.start();
                t.join();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C
```

На типичном современном ПК под управлением Linux с 64-битным Java 8 u101 этот тест показывает среднее время, затрачиваемое на создание, запуск и объединение потоков между 33,6 и 33,9 микросекундами.

Второй контрольный показатель эквивалентен первому, но с использованием `ExecutorService` для отправки задач и `Future` для рандеву с окончанием задачи.

```
import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                    }
                });
                future.get();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ExecutorTest
6714.66053
```

```
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685
^C
```

Как вы можете видеть, средние значения составляют от 5,3 до 5,6 микросекунд.

Хотя фактическое время будет зависеть от множества факторов, разница между этими двумя результатами значительна. Очевидно, что быстрее использовать пул потоков, чтобы перерабатывать потоки, чем создавать новые потоки.

Pitfall: общие переменные требуют правильной синхронизации.

Рассмотрим этот пример:

```
public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (!stop) {
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        tt.stop = true;            // Tell child thread to stop.
    }
}
```

Цель этой программы состоит в том, чтобы запустить поток, позволить ему работать в течение 1000 миллисекунд, а затем заставить его остановиться, установив флаг `stop` .

Будет ли он работать по назначению?

Может быть да, может быть нет.

Приложение не обязательно останавливается, когда возвращается `main` метод. Если был создан другой поток, и этот поток не был помечен как поток демона, приложение

продолжит работу после окончания основного потока. В этом примере это означает, что приложение будет продолжать работать до тех пор, пока дочерний поток не закончится. Это должно `tt.stop` если для параметра `tt.stop` установлено значение `true` .

Но это на самом деле не совсем верно. На самом деле, ребенок нить остановится после того, как он *заметил* `stop` со значением `true` . Это произойдет? Может быть да, может быть нет.

Спецификация языка Java *гарантирует,* что чтение и запись в памяти в потоке видимы для этого потока в соответствии с порядком инструкций в исходном коде. Однако, в общем, это НЕ гарантируется, когда один поток пишет, а другой поток (впоследствии) читает. Чтобы получить гарантированную видимость, должна существовать цепочка *событий - до* отношений между записью и последующим чтением. В приведенном выше примере для обновления флажка `stop` нет такой цепочки, и поэтому не гарантируется, что дочерний поток увидит, что `stop` изменится на `true` .

(Примечание для авторов: должна быть отдельная Тема на модели памяти Java, чтобы войти в глубокие технические детали.)

Как мы исправим проблему?

В этом случае есть два простых способа убедиться, что обновление `stop` видимо:

1. Объявить `stop` чтобы она была `volatile` ; т.е.

   ```
   private volatile boolean stop = false;
   ```

   Для переменной `volatile` переменная JLS указывает, что между записью по одному потоку и последующим чтением вторым потоком *происходит* связь между событиями.

2. Используйте мьютекс для синхронизации следующим образом:

```
public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (true) {
            synchronize (this) {
                if (stop) {
                    break;
                }
            }
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
```

```
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        synchronize (tt) {
            tt.stop = true;        // Tell child thread to stop.
        }
    }
}
```

В дополнение к тому , чтобы есть взаимное исключение, то JLS указывает , что есть *происходит, прежде* , *чем* отношения между освобождающий мьютекс в одном потоке и получить тот же мьютекс во втором потоке.

Но не является ли атомарным присвоением?

Да, это!

Однако этот факт не означает, что эффекты обновления будут отображаться одновременно для всех потоков. Это будет гарантировать только правильная цепочка - *до* отношений.

Почему они это сделали?

Программисты, делающие многопоточное программирование в Java, впервые найдут модель памяти сложной задачей. Программы ведут себя неинтуитивно, потому что естественное ожидание состоит в том, что записи видны равномерно. Итак, почему дизайнеры Java проектируют модель памяти таким образом.

Это фактически сводится к компромиссу между производительностью и простотой использования (для программиста).

Современная компьютерная архитектура состоит из нескольких процессоров (ядер) с отдельными наборами регистров. Основная память доступна либо для всех процессоров, либо для групп процессоров. Еще одним свойством современного компьютерного оборудования является то, что доступ к регистрам, как правило, на порядок быстрее, чем доступ к основной памяти. По мере увеличения количества ядер, легко видеть, что чтение и запись в основную память может стать основным узким местом в системе.

Это несоответствие устраняется путем реализации одного или нескольких уровней кэширования памяти между процессорными ядрами и основной памятью. Каждый ядро доступа к ячейкам памяти через его кеш. Обычно чтение основной памяти происходит только в случае промаха в кеше, и запись основной памяти происходит только тогда, когда необходимо очистить строку кэша. Для приложения, в котором рабочий набор каждого ядра памяти помещается в его кеш, скорость ядра больше не ограничена основной скоростью / пропускной способностью памяти.

Но это дает нам новую проблему, когда несколько ядер считывают и записывают общие переменные. Последняя версия переменной может находиться в кеше одного ядра. Если это ядро не сбрасывает строку кэша в основную память, а другие ядра аннулируют свою кешированную копию старых версий, некоторые из них могут видеть устаревшие версии переменной. Но если кеши были сброшены в память каждый раз, когда есть запись в кеш («на всякий случай» был прочитан другим ядром), который будет излишне потреблять пропускную способность основной памяти.

Стандартное решение, используемое на уровне набора аппаратных команд, состоит в том, чтобы предоставить инструкции для недействительности кеша и прокрутки кэша и оставить его компилятору, чтобы решить, когда его использовать.

Возвращение на Java. модель памяти спроектирована так, что компиляторы Java не обязаны выдавать недействительность кэша и инструкции по записи, где они действительно не нужны. Предполагается, что программист будет использовать соответствующий механизм синхронизации (например, примитивные мьютексы, `volatile` , высокоуровневые классы параллелизма и т. Д.), Чтобы указать, что для этого нужна видимость памяти. В отсутствие отношения « *произойдет-до»* компиляторы Java могут *предположить,* что операции кеширования (или аналогичные) не требуются.

Это имеет значительные преимущества в производительности для многопоточных приложений, но недостатком является то, что писать правильные многопоточные приложения не так просто. Программист *не* должен понять , что он или она делает.

Почему я не могу воспроизвести это?

Существует ряд причин, по которым такие проблемы трудно воспроизвести:

1. Как объяснялось выше, следствие того, что проблема с видимостью памяти не связана с проблемами, *обычно* заключается в том, что ваше скомпилированное приложение неправильно обрабатывает кеширование памяти. Однако, как мы уже говорили выше, кэши памяти часто сбрасываются.

2. При изменении аппаратной платформы характеристики кэшей памяти могут измениться. Это может привести к разному поведению, если ваше приложение не синхронизируется правильно.

3. Вы можете наблюдая эффекты синхронизации *счастливой.* Например, если вы добавляете трассировки, их обычно происходит некоторая синхронизация, происходящая за кулисами в потоках ввода-вывода, которая вызывает сброс кеша. Поэтому добавление отпечатков *часто* приводит к тому, что приложение ведет себя по-разному.

4. Запуск приложения под отладчиком приводит к тому, что компилятор JIT компилируется по-разному. Точки останова и однократный шаг усугубляют это. Эти

эффекты часто изменят поведение приложения.

Эти вещи делают ошибки, которые из-за неадекватной синхронизации особенно трудно решить.

Прочитайте Ошибки Java - потоки и параллелизм онлайн:
https://riptutorial.com/ru/java/topic/5567/ошибки-java---потоки-и-параллелизм

глава 134: Ошибки Java - проблемы с производительностью

Вступление

В этом разделе описывается ряд «ошибок» (т. Е. Ошибок, создаваемых новичками java-программистов), которые относятся к производительности Java-приложений.

замечания

В этом разделе описываются некоторые «микро» методы кодирования Java, которые неэффективны. В большинстве случаев неэффективность относительно невелика, но по-прежнему стоит избегать их.

Examples

Pitfall - накладные расходы на создание сообщений журнала

Уровни журналов `TRACE` и `DEBUG` позволяют передавать подробные сведения о работе данного кода во время выполнения. Обычно рекомендуется устанавливать уровень журнала выше этих, однако необходимо соблюдать осторожность, чтобы эти утверждения не влияли на производительность даже при кажущемся «выключенном».

Рассмотрим этот оператор журнала:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Даже если для уровня журнала задано значение `INFO` , аргументы, переданные в `debug()` будут оцениваться при каждом выполнении строки. Это делает его излишне потребляющим по нескольким причинам:

- `String` конкатенация: несколько `String` экземпляров будут созданы
- `InetAddress` может даже выполнять поиск DNS.
- `veryLongParamArray` может быть очень длинным - создание String из него потребляет память, требует времени

Решение

Большинство систем ведения журналов предоставляют средства для создания

журнальных сообщений с использованием строк исправлений и ссылок на объекты. Сообщение журнала будет оцениваться только в том случае, если сообщение действительно зарегистрировано. Пример:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

Это работает очень хорошо, если все параметры могут быть преобразованы в строки с помощью String.valueOf (Object) . Если перевод сообщений журнала более сложный, уровень регистрации можно проверить перед протоколированием:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
}
```

Здесь `LOG.debug()` с дорогостоящим `Arrays.toString(Obect[])` обрабатывается только тогда, когда `DEBUG` фактически включен.

Pitfall - Конкатенация строк в цикле не масштабируется

В качестве иллюстрации рассмотрим следующий код:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Несчастливо этот код неэффективен, если список `words` длинный. Корень проблемы заключается в следующем:

```
message = message + " " + word;
```

Для каждой итерации цикла этот оператор создает новую строку `message` содержащую копию всех символов в исходной строке `message` с добавленными к ней дополнительными символами. Это создает много временных строк и много копирует.

Когда мы анализируем `joinWords` , предполагая, что существует N слов со средней длиной M, мы обнаруживаем, что создаются временные строки O (N), и в процессе будут скопированы символы O (MN^2). Компонент N^2 вызывает особую тревогу.

Рекомендуемый подход для этой проблемы [1] заключается в использовании `StringBuilder` вместо конкатенации строк следующим образом:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

Анализ `joinWords2` необходимо учитывать накладные расходы «взросление» в `StringBuilder` массива подложки , который содержит символы Строителя. Однако выясняется, что количество новых объектов, созданных, равно O (logN), а количество копируемых символов - O (MN). Последний включает символы, скопированные в окончательный вызов `toString()` .

(Возможно, это можно будет настроить дальше, создав `StringBuilder` с правильной способностью для начала. Однако общая сложность остается неизменной.)

Возвращаясь к исходному методу `joinWords` , выясняется, что критический оператор будет оптимизирован типичным компилятором Java примерно так:

```
  StringBuilder tmp = new StringBuilder();
  tmp.append(message).append(" ").append(word);
  message = tmp.toString();
```

Однако компилятор Java не будет «вытаскивать» `StringBuilder` из цикла, как это `joinWords2` вручную в коде для `joinWords2` .

Ссылка:

- «Является ли Java String« + »оператором в петле медленным?»

---

1 - В Java 8 и более поздних версиях класс `Joiner` может использоваться для решения этой конкретной проблемы. Тем не менее, это не то, о чем этот пример *действительно должен быть* .

Pitfall - использование «нового» для создания примитивных экземпляров оболочки неэффективно

Язык Java позволяет использовать `new` для создания экземпляров `Integer` , `Boolean` и т. Д., Но, как правило, это плохая идея. Лучше использовать autoboxing (Java 5 и более поздние версии) или метод `valueOf` .

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

Причина, по которой использование `new Integer(int)` явно является плохой идеей, заключается в том, что он создает новый объект (если только не оптимизирован JIT-компилятором). В отличие от этого, когда используется автобоксинг или явный вызов

`valueOf` , среда выполнения Java будет пытаться повторно использовать объект `Integer` из кэша ранее существовавших объектов. Каждый раз, когда время выполнения имеет «попадание в кеш», оно позволяет избежать создания объекта. Это также экономит память кучи и уменьшает накладные расходы GC, вызванные оттоком объектов.

Заметки:

1. В последних реализациях Java автобоксинг реализуется путем вызова `valueOf` , и есть кеши для `Boolean` , `Byte` , `Short` , `Integer` , `Long` и `Character` .
2. Поведение кэширования для интегральных типов обеспечивается спецификацией Java Language Specification.

Pitfall - вызов новой строки (String) неэффективен

Использование `new String(String)` для дублирования строки неэффективно и почти всегда не нужно.

- Строковые объекты неизменяемы, поэтому нет необходимости копировать их для защиты от изменений.
- В некоторых более старых версиях Java объекты `String` могут совместно использовать массивы поддержки с другими объектами `String` . В этих версиях возможно утечка памяти путем создания (малой) подстроки (большой) строки и ее сохранения. Однако, начиная с Java 7, массивы поддержки `String` не разделяются.

В отсутствие каких-либо ощутимых преимуществ вызов `new String(String)` просто расточительно:

- Выполнение копии занимает процессорное время.
- Копия использует больше памяти, что увеличивает заметную память приложения и / или увеличивает накладные расходы GC.
- Операции типа `equals(Object)` и `hashCode()` могут быть медленнее при копировании объектов String.

Pitfall - Calling System.gc () неэффективен

Это (почти всегда) плохая идея вызвать `System.gc()` .

Javadoc для метода `gc()` указывает следующее:

> «Вызов метода `gc` предполагает, что виртуальная машина Java тратит усилия на повторное использование неиспользуемых объектов, чтобы сделать память, которую они в настоящее время занимают, для быстрого повторного использования. Когда управление возвращается из вызова метода, виртуальная машина Java приложила все усилия для восстановления пространство от всех отброшенных объектов ».

Из этого можно выделить пару важных моментов:

1. Использование слова «предлагает», а не (скажем) «говорит» означает, что JVM может игнорировать это предложение. Поведение JVM по умолчанию (последние выпуски) заключается в том, чтобы следовать этому предложению, но это можно переопределить, установив `-XX:+DisableExplicitGC` при запуске JVM.

2. Фраза «лучшее усилие для освобождения пространства от всех отброшенных объектов» подразумевает, что вызов `gc` вызовет «полную» сборку мусора.

Итак, почему вызывает `System.gc()` плохую идею?

Во-первых, полная сборка мусора дорогая. Полный GC включает посещение и «маркировку» каждого объекта, который все еще доступен; т.е. каждый объект, который не является мусором. Если вы инициируете это, когда не нужно собирать мусор, GC делает большую работу за относительно небольшую выгоду.

Во-вторых, полная сборка мусора может нарушить свойства «локальности» объектов, которые не собираются. Объекты, которые распределены одним и тем же потоком примерно в одно и то же время, как правило, распределяются близко друг к другу в памяти. Это хорошо. Объекты, которые распределены одновременно, скорее всего, будут связаны; т.е. ссылаться друг на друга. Если ваше приложение использует эти ссылки, то вероятность того, что доступ к памяти будет быстрее из-за различных эффектов кэширования памяти и страницы. К сожалению, полная сборка мусора, как правило, перемещает объекты вокруг, так что объекты, которые когда-то были близки, теперь раздвинуты.

В-третьих, запуск полной сборки мусора может привести к приостановке вашего приложения до тех пор, пока сбор не будет завершен. Пока это происходит, ваше приложение будет невосприимчивым.

На самом деле, лучшая стратегия - позволить JVM решить, когда запускать GC и какую коллекцию запускать. Если вы не вмешиваетесь, JVM выберет время и тип сбора, который оптимизирует пропускную способность или минимизирует время паузы GC.

Сначала мы говорили «... (почти всегда) плохая идея ...». На самом деле есть несколько сценариев, где это *может* быть хорошей идеей:

1. Если вы выполняете единичный тест для некоторого кода, который чувствителен к сборке мусора (например, что-то включает финализаторы или слабые / мягкие / фантомные ссылки `System.gc()` может потребоваться вызов `System.gc()` .

2. В некоторых интерактивных приложениях могут быть определенные моменты времени, когда пользователю не будет делаться пауза в сборе мусора. Одним из примеров является игра, в которой есть естественные паузы в «игре»; например, при

загрузке нового уровня.

Pitfall - чрезмерное использование примитивных типов обертки неэффективно

Рассмотрим эти два фрагмента кода:

```
int a = 1000;
int b = a + 1;
```

а также

```
Integer a = 1000;
Integer b = a + 1;
```

Вопрос: Какая версия более эффективна?

Ответ: Две версии выглядят почти одинаково, но первая версия намного эффективнее второй.

Вторая версия использует представление для чисел, которые используют больше пространства, и опирается на автоматическое боксирование и автоматическое распаковывание за кулисами. Фактически вторая версия прямо эквивалентна следующему коду:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

Сравнивая это с другой версией, использующей `int` , есть четыре дополнительных вызова метода, когда используется `Integer` . В случае `valueOf` каждый вызов будет создавать и инициализировать новый объект `Integer` . Вся эта дополнительная работа по боксу и распаковке, вероятно, сделает вторую версию на порядок медленнее, чем первая.

В дополнение к этому, вторая версия выделяет объекты в куче в каждом вызове `valueOf` . Хотя использование пространства зависит от платформы, оно, вероятно, будет находиться в области 16 байт для каждого объекта `Integer` . Напротив, версия `int` нуждается в нулевом избытке кучи, предполагая, что `a` и `b` являются локальными переменными.

Еще одна важная причина, по которой примитивы быстрее, чем их эквивалент в коробке, - это то, как их соответствующие типы массивов выкладываются в памяти.

Если вы берете `int[]` и `Integer[]` в качестве примера, то в случае `int[]` *значения* `int` смежно располагаются в памяти. Но в случае `Integer[]` это не значения, которые выложены, а ссылки (указатели) на объекты `Integer` , которые, в свою очередь, содержат фактические значения `int` .

Помимо того, что это дополнительный уровень косвенности, это может быть большой танк, когда дело доходит до местоположения кеша при повторении значений. В случае `int[]` процессор может извлекать все значения в массиве в его кеш-память сразу, потому что они смежны в памяти. Но в случае `Integer[]` процессор потенциально должен выполнить дополнительную выборку памяти для каждого элемента, так как массив содержит только ссылки на фактические значения.

Короче говоря, использование примитивных типов обертки относительно дорого как в ресурсах ЦП, так и в памяти. Использование их излишне эффективно.

Pitfall - Итерация ключей карты может быть неэффективной

Следующий примерный код медленнее, чем нужно:

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

Это связано с тем, что для каждого ключа на карте требуется поиск по карте (метод `get()`). Этот поиск может быть неэффективным (в HashMap он влечет за собой вызов `hashCode` на ключ, а затем поиск правильного ведра во внутренних структурах данных, а иногда даже вызов `equals`). На большой карте это не может быть тривиальным накладным.

Правильный способ избежать этого - перебирать записи карты, которые подробно описаны в разделе « Коллекции»

Pitfall - использование параметра size () для проверки, является ли коллекция пустой, неэффективно.

Рамка коллекций Java предоставляет два связанных метода для всех объектов `Collection` :

- `size()` возвращает количество записей в `Collection` и
- Метод `isEmpty()` возвращает true, если (и только если) `Collection` пуста.

Оба метода можно использовать для проверки пустоты коллекции. Например:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

Хотя эти подходы выглядят одинаково, некоторые реализации коллекции не сохраняют размер. Для такой коллекции реализация `size()` должна вычислять размер каждый раз, когда он вызывается. Например:

- Простой класс связанного списка (но не `java.util.LinkedList`), возможно, потребуется пройти через список для подсчета элементов.
- Класс `ConcurrentHashMap` должен суммировать записи во всех «сегментах» карты.
- Для ленивой реализации коллекции может потребоваться реализовать всю коллекцию в памяти, чтобы подсчитать элементы.

Напротив, метод `isEmpty()` должен только проверять, есть ли *хотя бы один* элемент в коллекции. Это не связано с подсчетом элементов.

Хотя `size() == 0` не всегда менее эффективен, чем `isEmpty()` , для правильно реализованного `isEmpty()` немыслимо менее эффективно, чем `size() == 0` . Следовательно, `isEmpty()` является предпочтительным.

Pitfall - Эффективность связана с регулярными выражениями

Регулярное выражение является мощным инструментом (в Java и в других контекстах), но имеет некоторые недостатки. Одно из них, что регулярные выражения имеют тенденцию быть довольно дорогими.

Образцы шаблонов и совпадений следует использовать повторно

Рассмотрим следующий пример:

```
/**
 * Test if all strings in a list consist of English letters and numbers.
 * @param strings the list to be checked
 * @return 'true' if an only if all strings satisfy the criteria
 * @throws NullPointerException if 'strings' is 'null' or a 'null' element.
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

Этот код правильный, но он неэффективен. Проблема заключается в `matches(...)` вызова. Под капотом `s.matches("[A-Za-z0-9]*")` эквивалентно этому:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

что в свою очередь эквивалентно

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

`Pattern.compile("[A-Za-z0-9]*")` анализирует регулярное выражение, анализирует его и создает объект `Pattern` который содержит структуру данных, которая будет использоваться движком regex. Это нетривиальное вычисление. Затем объект `Matcher` создается, чтобы обернуть аргумент `s` . Наконец, мы вызываем `match()` чтобы выполнить фактическое сопоставление шаблонов.

Проблема в том, что эта работа повторяется для каждой итерации цикла. Решение состоит в том, чтобы реструктурировать код следующим образом:

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
    return true;
}
```

Обратите внимание, что javadoc для `Pattern` указывает:

> Экземпляры этого класса являются неизменными и безопасны для использования несколькими параллельными потоками. Экземпляры класса `Matcher` небезопасны для такого использования.

Не используйте match (), когда вы должны использовать find ()

Предположим, вы хотите проверить, содержит ли строка `s` три или более цифр подряд. Вы можете выразить это различными способами, в том числе:

```
  if (s.matches(".*[0-9]{3}.*")) {
      System.out.println("matches");
  }
```

или же

```
  if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
      System.out.println("matches");
  }
```

Первый из них более краткий, но он также, вероятно, будет менее эффективным. На первый взгляд первая версия попытается сопоставить всю строку с шаблоном. Кроме того, поскольку «. *» Является «жадным» шаблоном, совпадение шаблонов, вероятно, будет «нетерпеливо» пытаться до конца строки и возвращаться, пока не найдет совпадение.

Напротив, вторая версия будет искать слева направо и прекратит поиск, как только он найдет 3 цифры подряд.

Используйте более эффективные альтернативы регулярным выражениям

Регулярные выражения - это мощный инструмент, но они не должны быть вашим единственным инструментом. Многие задачи могут быть выполнены более эффективно другими способами. Например:

```
Pattern.compile("ABC").matcher(s).find()
```

делает то же самое, что:

```
s.contains("ABC")
```

за исключением того, что последний намного эффективнее. (Даже если вы можете амортизировать затраты на компиляцию регулярного выражения.)

Часто не-регулярное выражение является более сложным. Например, тест, выполняемый `allAlplanumeric matches()` вызывает более ранний `allAlplanumeric` метод, можно переписать как:

```
public boolean matches(String s) {
    for (char c : s) {
        if ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9')) {
             return false;
        }
    }
    return true;
}
```

Теперь это больше кода, чем использование `Matcher` , но он также будет значительно быстрее.

Катастрофический отход

(Это потенциально проблема со всеми реализациями регулярных выражений, но мы упомянем это здесь, потому что это ловушка для использования `Pattern` .)

Рассмотрим этот (надуманный) пример:

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
```

```
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

Первый вызов `println` быстро напечатает `true` . Второй будет печатать `false` . В конце концов. Действительно, если вы экспериментируете с приведенным выше кодом, вы увидите, что каждый раз, когда вы добавляете `A` перед `C` , время будет удвоенным.

Это поведение является примером *катастрофического отступления* . Механизм соответствия шаблонов, который реализует соответствие регулярному выражению, бесплодно пытается использовать все *возможные* способы, которыми *может* соответствовать шаблон.

Давайте посмотрим, что означает `(A+)+B` По-видимому, кажется, что «один или несколько символов `A` за которым следует значение `B` », но на самом деле он говорит одну или несколько групп, каждая из которых состоит из одного или нескольких символов `A` Так, например:

- «AB» соответствует только одному способу: '(A) B'
- «AAB» соответствует двум путям: «(AA) B» или «(A) (A) B`
- «AAAB» соответствует четырем путям: «(AAA) B» или «(AA) (A) B `or '(A)(AA)B` или «(A) (A) (A) B`
- и так далее

Другими словами, количество возможных совпадений равно 2 ^{N,} где N - количество символов `A`

Вышеприведенный пример явно надуман, но шаблоны, демонстрирующие такие характеристики (например, `O(2^N)` или `O(N^K)` для большого `K`) возникают часто, когда используются неосмотрительные регулярные выражения. Существуют некоторые стандартные средства защиты:

- Избегайте встраивания повторяющихся шаблонов в другие повторяющиеся шаблоны.
- Избегайте использования слишком много повторяющихся шаблонов.
- Повторно используйте повторное повторное следование.
- Не используйте регулярные выражения для сложных задач синтаксического анализа. (Вместо этого напишите правильный парсер.)

Наконец, остерегайтесь ситуаций, когда пользователь или клиент API могут снабжать строку регулярных выражений патологическими характеристиками. Это может привести к случайному или преднамеренному «отказу в обслуживании».

Рекомендации:

- Тег Regular Expressions , особенно http://www.riptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=20161001033913136116 3 и http://www.riptutorial.com/ регулярное выражение / тема / 259 / получение стартер-с-регулярные выражениями / 4527 / , когда вы-должны-не-потребительный регулярные

выражения # т = 20161001033959356491 3
- «Эффект регулярного выражения» Джеффа Этвуда.
- «Как убить Java с помощью регулярного выражения» Андреаса Хауфлера.

Pitfall - встроенные строки, чтобы вы могли использовать ==, - это плохая идея

Когда некоторые программисты видят этот совет:

> «Тестирование строк с использованием `==` неверно (если строки не интернированы)»

их первоначальная реакция заключается в интерновных строках, чтобы они могли использовать `==` . (В конце концов `==` быстрее, чем вызов `String.equals(...)` , не так ли.)

Это неправильный подход, с нескольких точек зрения:

хрупкость

Прежде всего, вы можете только безопасно использовать `==` если знаете, что *все* объекты `String` вы тестируете, были интернированы. JLS гарантирует, что строковые литералы в вашем исходном коде будут интернированы. Однако ни один из стандартных API Java SE не гарантирует возврата интернированных строк, кроме `String.intern(String)` . Если вы пропустили только один источник объектов `String` , которые не были интернированы, ваше приложение будет ненадежным. Эта ненадежность проявится как ложные негативы, а не исключения, которые могут затруднить обнаружение.

Затраты на использование «intern ()»

Под капотом интернирование выполняется путем сохранения хеш-таблицы, содержащей ранее интернированные объекты `String` . Используется какой-то слабый механизм ссылок, так что хеш-таблица интернирования не становится утечкой хранилища. Хотя хэш-таблица реализована в собственном коде (в отличие от `HashMap` , `HashTable` и т. Д.), `intern` вызовы по-прежнему относительно дорогостоящие с точки зрения используемого процессора и памяти.

Эта стоимость должна сравниваться с сохранением, которую мы собираемся получить, используя `==` вместо `equals` . На самом деле, мы не собираемся ломаться, даже если каждая интернированная строка сравнивается с другими строками «несколько раз».

(Помимо этого: несколько ситуаций, в которых интернирование стоит, как правило, сводятся к уменьшению печати в ноге памяти приложения, когда одни *и* те же строки повторяются много раз, *и* эти строки имеют долгий срок службы.)

Влияние на сбор мусора

Помимо прямых затрат на процессор и память, описанных выше, интернированные строки влияют на производительность сборщика мусора.

Для версий Java до Java 7 интерполяционные строки хранятся в пространстве «PermGen», который собирается нечасто. Если PermGen необходимо собрать, это (обычно) запускает полную сборку мусора. Если пространство PermGen заполняется полностью, JVM аварийно завершает работу, даже если в обычных пространствах кучи было свободное пространство.

В Java 7 пул строк был перемещен из «PermGen» в обычную кучу. Однако хеш-таблица по-прежнему будет долговечной структурой данных, которая заставит любые интернированные строки быть долговечными. (Даже если объекты интернированных строк были выделены в пространстве Эдена, они, скорее всего, будут продвигаться до их сбора.)

Таким образом, во всех случаях интернирование строки увеличивает продолжительность жизни по сравнению с обычной строкой. Это увеличит накладные расходы на сборку мусора на протяжении всей жизни JVM.

Вторая проблема заключается в том, что хеш-таблица должна использовать слабый механизм ссылок, чтобы предотвратить прерывание строки с утечкой памяти. Но такой механизм больше подходит для сборщика мусора.

Эти накладные расходы на сбор мусора трудно определить количественно, но нет сомнений в том, что они существуют. Если вы широко используете `intern` , они могут быть значительными.

Размер хэш-таблицы пула строк

Согласно этому источнику , начиная с Java 6, пул строк реализуется как хэш-таблица фиксированного размера с цепочками для обработки строк, которые хешируются в одном и том же ведре. В ранних выпусках Java 6 хэш-таблица имела (жесткий) постоянный размер. Параметр настройки (`-XX:StringTableSize`) был добавлен в качестве обновления в середине жизни на Java 6. Затем, в среднем обновлении до Java 7, размер пула по умолчанию был изменен с `1009` до `60013` .

Суть в том, что если вы намерены интенсивно использовать `intern` в своем коде, *рекомендуется* выбрать версию Java, где размер хеш-таблицы настраивается, и убедитесь, что вы правильно настроили его размер. В противном случае производительность `intern` может ухудшиться по мере увеличения пула.

Интернирование как потенциальный отказ в

обслуживании вектора

Алгоритм hashcode для строк хорошо известен. Если вы ставите строки, поставленные вредоносными пользователями или приложениями, это может быть использовано как часть атаки на отказ в обслуживании (DoS). Если агент злой агент устанавливает, что все строки, которые он предоставляет, имеют один и тот же хэш-код, это может привести к неуравновешенной хеш-таблице и производительности `O(N)` для `intern` ... где `N` - количество столкнутых строк.

(Есть более простые / более эффективные способы запуска DoS-атаки на службу. Однако этот вектор можно использовать, если целью DoS-атаки является нарушение безопасности или уклонение от DoS-защиты первой линии.)

Pitfall - Малые чтения / записи на небуферизованных потоках неэффективны

Рассмотрим следующий код для копирования одного файла в другой:

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]);
             OutputStream os = new FileOutputStream(args[1])) {
           int octet;
           while ((octet = is.read()) != -1) {
               os.write(octet);
           }
        }
    }
}
```

(Мы рассмотрели пропущенную проверку нормальных аргументов, сообщение об ошибках и т. Д., Поскольку они не имеют отношения к *точке* этого примера.)

Если вы скомпилируете вышеуказанный код и используете его для копирования огромного файла, вы заметите, что он очень медленный. Фактически, это будет, по крайней мере, на пару порядков медленнее, чем стандартные утилиты копирования файлов ОС.

(*Добавьте фактические измерения производительности здесь!*)

Основная причина того, что приведенный выше пример медленный (в случае большого файла) заключается в том, что он выполняет однобайтные чтения и однобайтные записи в небуферизованных байтовых потоках. Простым способом повышения производительности является обтекание потоков буферизованными потоками. Например:

```
import java.io.*;
```

```
public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new BufferedInputStream(
                    new FileInputStream(args[0]));
             OutputStream os = new BufferedOutputStream(
                    new FileOutputStream(args[1]))) {
           int octet;
           while ((octet = is.read()) != -1) {
               os.write(octet);
           }
        }
    }
}
```

Эти небольшие изменения улучшат скорость копирования данных, *по крайней мере,* на пару порядков, в зависимости от различных факторов, связанных с платформой. Буферизованные обертки потока заставляют данные считываться и записываться в больших кусках. В экземплярах оба буфера реализованы как массивы байтов.

- С `is` , данные считываются из файла в буфер за несколько килобайт за раз. Когда вызывается `read()` , реализация, как правило, возвращает байт из буфера. Он будет считываться только из основного входного потока, если буфер опустел.
- Поведение для `os` аналогично. Вызовы `os.write(int)` записывают одиночные байты в буфер. Данные записываются только в выходной поток при заполнении буфера или при сбросе или закрытии `os` .

Как насчет потоков, основанных на символах?

Как вам следует знать, Java I / O предоставляет различные API для чтения и записи двоичных и текстовых данных.

- `InputStream` и `OutputStream` являются базовыми API для потокового двоичного ввода-вывода
- `Reader` и `Writer` являются базовыми API-интерфейсами для потокового ввода-вывода.

Для ввода / вывода текста `BufferedReader` и `BufferedWriter` являются эквивалентами для `BufferedInputStream` и `BufferedOutputStream` .

Почему буферизованные потоки имеют большое значение?

Настоящая причина, по которой буферизованные потоки помогают повысить производительность, связана с тем, как приложение обращается к операционной системе:

- Java-метод в Java-приложении или вызовы собственных процедур в собственных

библиотеках времени выполнения JVM бывают быстрыми. Обычно они выполняют несколько инструкций машины и имеют минимальное влияние на производительность.

- Напротив, вызовы во время выполнения JVM для операционной системы выполняются не быстро. Они включают нечто вроде «syscall». Типичный шаблон для системного вызова выглядит следующим образом:

    1. Поместите аргументы syscall в регистры.
    2. Выполните команду ловушки SYSENTER.
    3. Обработчик ловушки переключается в привилегированное состояние и изменяет отображение виртуальной памяти. Затем он отправляет код для обработки конкретного системного вызова.
    4. Обработчик syscall проверяет аргументы, опасаясь, что ему не сообщается о доступе к памяти, которую пользовательский процесс не должен видеть.
    5. Выполняется конкретная работа в режиме syscall. В случае `read` syscall это может включать:
        1. проверяя, что есть данные для чтения в текущей позиции дескриптора файла
        2. вызывая обработчик файловой системы для извлечения требуемых данных с диска (или там, где он хранится) в буферный кеш,
        3. копирование данных из буферного кеша в JVM-адрес
        4. корректировка позиции дескриптора файловой позиции thstream
    6. Вернитесь из syscall. Это влечет за собой повторное изменение сопоставлений виртуальных машин и выключение привилегированного состояния.

Как вы можете себе представить, выполнение одного системного вызова может привести к тысячам машинных инструкций. Консервативно, *по крайней мере на* два порядка больше обычного вызова метода. (Вероятно, три или более.)

Учитывая это, причина, по которой буферизованные потоки имеет большое значение, заключается в том, что они резко сокращают количество системных вызовов. Вместо того, чтобы выполнять syscall для каждого вызова `read()` , буферизованный входной поток считывает большой объем данных в буфер по мере необходимости. Большинство вызовов `read()` для буферизованного потока выполняют некоторые простые проверки и возвращают `byte` который был прочитан ранее. Аналогичные рассуждения применяются в случае выходного потока, а также в случаях потока символов.

(Некоторые считают, что производительность буферизованного ввода-вывода происходит из-за несоответствия между размером запроса на чтение и размером блока диска, временной задержкой на диске и такими вещами. Фактически, современная ОС использует ряд стратегий для обеспечения того, чтобы *обычно* не нужно ждать диска. Это не настоящее объяснение.)

Буферизованные потоки всегда выигрывают?

Не всегда. Буферизованные потоки, безусловно, выигрывают, если ваше приложение собирается делать «маленькие» чтения или записи. Однако, если вашему приложению нужно выполнять большие чтения или записи в / из большого `byte[]` или `char[]` , то буферизованные потоки не дадут вам реальных преимуществ. Действительно, может быть даже небольшое (незначительное) исполнение.

Это самый быстрый способ скопировать файл на Java?

Нет, это не так. Когда вы используете API-интерфейсы, основанные на потоке Java, для копирования файла, вы берете на себя стоимость, по крайней мере, одной дополнительной копии данных, хранящейся в памяти. Этого можно избежать, если вы используете NIO `ByteBuffer` и API `Channel` . (*Добавьте ссылку на отдельный пример здесь.*)

Прочитайте Ошибки Java - проблемы с производительностью онлайн:
https://riptutorial.com/ru/java/topic/5455/ошибки-java---проблемы-с-производительностью

глава 135: пакеты

Вступление

пакет в java используется для группировки класса и интерфейсов. Это помогает разработчику избегать конфликтов, когда существует огромное количество классов. Если мы используем этот пакет для классов, мы можем создать класс / интерфейс с тем же именем в разных пакетах. Используя пакеты, мы можем импортировать кусок снова в другой класс. Там много *встроенных пакетов* в java как> 1.java.util> 2.java.lang> 3.java.io Мы можем определить наши собственные *пользовательские пакеты* .

замечания

Пакеты обеспечивают защиту доступа.

> оператор пакета должен быть первой строкой исходного кода. В одном исходном файле может быть только один пакет.

С помощью пакетов можно избежать конфликта между различными модулями.

Examples

Использование пакетов для создания классов с тем же именем

Первый тест. Класс:

```
package foo.bar

public class Test {

}
```

Также Test.class в другой упаковке

```
package foo.bar.baz

public class Test {

}
```

Вышеуказанное прекрасно, потому что два класса существуют в разных пакетах.

Использование защищенного пакета

В Java, если вы не предоставляете модификатор доступа, область по умолчанию для переменных является защищенным пакетом. Это означает, что классы могут обращаться к переменным других классов в одном пакете, как если бы эти переменные были общедоступными.

```
package foo.bar

public class ExampleClass {
    double exampleNumber;
    String exampleString;

    public ExampleClass() {
        exampleNumber = 3;
        exampleString = "Test String";
    }
    //No getters or setters
}

package foo.bar

public class AnotherClass {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Prints Example Number: 3
    System.out.println("Example String: " + clazz.exampleString);
    //Prints Example String: Test String
}
```

Этот метод не будет работать для класса в другом пакете:

```
package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Throws an exception
    System.out.println("Example String: " + clazz.exampleString);
    //Throws an exception
}
```

Прочитайте пакеты онлайн: https://riptutorial.com/ru/java/topic/8273/пакеты

глава 136: Параллельное программирование (темы)

Вступление

Параллельные вычисления - это одна из форм вычисления, при которой несколько вычислений выполняются одновременно, а не последовательно. Язык Java предназначен для поддержки параллельного программирования посредством использования потоков. Доступ к объектам и ресурсам осуществляется несколькими потоками; каждый поток может потенциально получить доступ к любому объекту в программе, и программист должен обеспечить, чтобы доступ для чтения и записи к объектам был правильно синхронизирован между потоками.

замечания

Связанная тема (ы) по StackOverflow:

- Атомные типы
- Исполнители, Исполнительные службы и пулы потоков
- Расширение `Thread` сравнению с реализацией `Runnable`

Examples

Базовая многопоточность

Если у вас много задач для выполнения, и все эти задачи не зависят от результата предыдущих, вы можете использовать **Multithreading** для вашего компьютера для выполнения всех этих задач одновременно с использованием большего количества процессоров, если ваш компьютер может. Это может **ускорить** выполнение вашей программы, если у вас есть большие независимые задачи.

```
class CountAndPrint implements Runnable {

    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ": " + i);
```

```
        }
    }

    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i <= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // Doing some others tasks in the main Thread
        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

Код метода run для различных экземпляров `CountAndPrint` будет выполняться в непредсказуемом порядке. Отрывок примера выполнения может выглядеть так:

```
Instance 4: 1
Instance 2: 1
Instance 4: 2
Instance 1: 1
Instance 1: 2
Main: 1
Instance 4: 3
Main: 2
Instance 3: 1
Instance 4: 4
...
```

Производитель-Потребитель

Простой пример решения проблемы производителя и потребителя. Обратите внимание, что для синхронизации используются классы JDK (`AtomicBoolean` и `BlockingQueue`), что уменьшает вероятность создания недопустимого решения. Проконсультируйтесь с Javadoc для различных типов BlockingQueue ; выбор другой реализации может кардинально изменить поведение этого примера (например, DelayQueue или Priority Queue).

```
public class Producer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Producer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int producedCount = 0;
        try {
            while (true) {
                producedCount++;
                //put throws an InterruptedException when the thread is interrupted
```

```
                queue.put(new ProducedData());
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            producedCount--;
            //re-interrupt the thread in case the interrupt flag is needeed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Produced " + producedCount + " objects");
    }
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Consumer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int consumedCount = 0;
        try {
            while (true) {
                //put throws an InterruptedException when the thread is interrupted
                ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // process data
                consumedCount++;
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            consumedCount--;
            //re-interrupt the thread in case the interrupt flag is needeed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Consumed " + consumedCount + " objects");
    }
}


public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
        // choice of queue determines the actual behavior: see various BlockingQueue
implementations

        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();

        Thread.sleep(1000);
        producer.interrupt();
        Thread.sleep(10);
        consumer.interrupt();
    }
```

```
}
```

Использование ThreadLocal

Полезным инструментом в Java Concurrency является `ThreadLocal` - это позволяет вам иметь переменную, которая будет уникальной для данного потока. Таким образом, если один и тот же код работает в разных потоках, эти исполнения не будут совместно использовать значение, но вместо этого каждый поток имеет свою собственную переменную, которая является *локальной для потока* .

Например, это часто используется для установления контекста (например, информации авторизации) обработки запроса в сервлете. Вы можете сделать что-то вроде этого:

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get returns the variable unique to this thread
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // save that context to our thread-local - other threads
                           // making this call don't overwrite ours
    try {
        // business logic
    } finally {
        contexts.remove(); // 'ensure' removal of thread-local variable
    }
}
```

Теперь вместо того, чтобы передавать `MyUserContext` в каждый отдельный метод, вы можете вместо этого использовать `MyServlet.getContext()` где он вам нужен. Теперь, конечно, это вводит переменную, которая должна быть документирована, но она поточно-безопасна, что устраняет множество недостатков для использования такой переменной с высокой степенью охвата.

Ключевым преимуществом здесь является то, что каждый поток имеет свою собственную локальную переменную потока в контейнере `contexts` . Пока вы используете его из определенной точки входа (например, требуя, чтобы каждый сервлет поддерживал свой контекст или, возможно, добавляя фильтр сервлета), вы можете полагаться на этот контекст, когда он вам нужен.

CountDownLatch

CountDownLatch

> Вспомогательное средство синхронизации, которое позволяет одному или нескольким потокам дождаться завершения набора операций в других потоках.

1. `CountDownLatch` инициализируется с заданным подсчетом.
2. Методы ожидания выполняются до тех пор, пока текущий счетчик не достигнет нуля из-за вызовов метода `countDown()` , после чего все ожидающие потоки освобождаются, и любые последующие вызовы ожидания возвращаются немедленно.
3. Это феномен с одним выстрелом - счетчик не может быть сброшен. Если вам нужна версия, которая сбрасывает счетчик, рассмотрите возможность использования `CyclicBarrier` .

Ключевые методы:

```
public void await() throws InterruptedException
```

Заставляет текущий поток ждать, пока защелка не будет отсчитываться до нуля, если поток не будет прерван.

```
public void countDown()
```

Уменьшает количество защелок, освобождая все ожидающие потоки, если счетчик достигает нуля.

Пример:

```
import java.util.concurrent.*;

class DoSomethingInAThread implements Runnable {
    CountDownLatch latch;
    public DoSomethingInAThread(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("Do some thing");
            latch.countDown();
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}

public class CountDownLatchDemo {
    public static void main(String[] args) {
        try {
            int numberOfThreads = 5;
            if (args.length < 1) {
                System.out.println("Usage: java CountDownLatchDemo numberOfThreads");
                return;
            }
            try {
                numberOfThreads = Integer.parseInt(args[0]);
            } catch(NumberFormatException ne) {

            }
            CountDownLatch latch = new CountDownLatch(numberOfThreads);
```

```
            for (int n = 0; n < numberOfThreads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("In Main thread after completion of " + numberOfThreads + "
threads");
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

выход:

```
java CountDownLatchDemo 5
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
In Main thread after completion of 5 threads
```

Объяснение:

1. `CountDownLatch` инициализируется счетчиком 5 в главной теме
2. Основной поток ожидает с помощью метода wait `await()` .
3. Создано пять экземпляров `DoSomethingInAThread` . Каждый экземпляр `countDown()` счетчик `countDown()` .
4. Когда счетчик станет нулевым, основной поток возобновится

синхронизация

В Java существует встроенный механизм блокировки на уровне языка: `synchronized` блок, который может использовать любой объект Java как встроенную блокировку (т.е. каждый объект Java может иметь связанный с ним монитор).

Внутренние блокировки обеспечивают атомарность групп операторов. Чтобы понять, что это значит для нас, давайте посмотрим на пример, когда `synchronized` полезна:

```
private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
                t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
```

```
    }
    executorService.shutdown();
}
```

В этом случае, если бы не `synchronized` блок, было бы много проблем параллелизма. Первый из них был бы с оператором приращения post (он сам по себе не является атомом), а вторым будет то, что мы будем наблюдать значение t после того, как произвольное количество других потоков получило бы возможность его модифицировать. Однако, поскольку мы приобрели встроенный замок, здесь не будет никаких условий гонки, и выход будет содержать цифры от 1 до 100 в их обычном порядке.

Встроенные замки в Java являются *мьютексы* (т.е. взаимного исполнения замков). Взаимное исполнение означает, что, если один поток приобрел блокировку, второй будет вынужден дождаться, когда первый из них освободит его, прежде чем он сможет получить блокировку для себя. Примечание. Операция, которая может поместить поток в состояние ожидания (сна), называется *блокировкой* . Таким образом, приобретение блокировки - это операция блокировки.

Внутренние блокировки на Java являются *реентерабельными* . Это означает, что если поток пытается получить блокировку, которой он уже владеет, он не будет блокироваться, и он успешно его приобретет. Например, следующий код *не* будет блокироваться при вызове:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Помимо `synchronized` блоков существуют также `synchronized` методы.

Следующие блоки кода практически эквивалентны (хотя байт-код кажется другим):

1. `synchronized` блок на `this` :

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. `synchronized` метод:

```
 public synchronized void foo() {
```

```
        doStuff();
    }
```

Аналогично для `static` методов это:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

имеет такой же эффект, как и этот:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Атомные операции

Атомная операция - это операция, которая выполняется «все сразу», без каких-либо шансов на то, что другие потоки будут наблюдать или изменять состояние во время выполнения атомной операции.

Давайте рассмотрим **ПЯТЫЙ ПРИМЕР** .

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            t++;
            System.out.println(MessageFormat.format("t: {0}", t));
        });
    }
    executorService.shutdown();
}
```

В этом случае есть два вопроса. Первая проблема заключается в том, что оператор post increment *не* является атомарным. Он состоит из нескольких операций: получите значение, добавьте 1 к значению, установите значение. Вот почему, если мы запустим этот пример, вероятно, мы не увидим в выходном файле `t: 100` - два потока могут одновременно получать значение, увеличивать его и устанавливать его: допустим, значение t равно 10, а два потоки увеличивают t. Оба потока устанавливают значение t равным 11, так как второй

поток наблюдает значение t до того, как первый поток завершил его увеличение.

Вторая проблема заключается в том, как мы наблюдаем t. Когда мы печатаем значение t, это значение может быть уже изменено другим потоком после операции приращения потока.

Чтобы исправить эти проблемы, мы будем использовать `java.util.concurrent.atomic.AtomicInteger` , который имеет много атомных операций для нас.

```
private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}
```

Метод `incrementAndGet` `AtomicInteger` атомарно увеличивает и возвращает новое значение, тем самым устраняя предыдущее состояние гонки. Обратите внимание, что в этом примере строки будут по-прежнему выходить из строя, потому что мы не прилагаем никаких усилий, чтобы упорядочить вызовы `println` и что это выходит за рамки этого примера, поскольку для этого потребуется синхронизация, и цель этого примера - показать, как использовать `AtomicInteger` для устранения условий гонки, касающихся состояния.

Создание базовой тупиковой системы

Тупик возникает, когда два конкурирующих действия ждут, пока другой закончит, и, таким образом, никогда не будет. В java есть один замок, связанный с каждым объектом. Чтобы избежать параллельной модификации, выполняемой несколькими потоками на одном объекте, мы можем использовать `synchronized` ключевое слово, но все идет по себестоимости. Использование `synchronized` ключевого слова ошибочно может привести к застрявшим системам, называемым системой с блокировкой.

Подумайте, что в 1 экземпляре работают 2 потока, Позволяет обрабатывать потоки как First и Second, и позволяет сказать, что у нас есть 2 ресурса R1 и R2. Сначала получает R1, а также R2 нуждается в его завершении, а Second приобретает R2 и нуждается в R1 для завершения.

так сказать, в момент времени t = 0,

Сначала R1, а второй - R2. теперь First ждет R2, а Second ждет R1. это ожидание неопределенное, и это приводит к тупиковой ситуации.

```
public class Example2 {

    public static void main(String[] args) throws InterruptedException {
        final DeadLock dl = new DeadLock();
        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                dl.methodA();
            }
        });

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                try {
                    dl.method2();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });
        t1.setName("First");
        t2.setName("Second");
        t1.start();
        t2.start();
    }
}

class DeadLock {

    Object mLock1 = new Object();
    Object mLock2 = new Object();


    public void methodA() {
        System.out.println("methodA wait for mLock1  " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("methodA mLock1 acquired   " +
Thread.currentThread().getName());
            try {
                Thread.sleep(100);
                method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
    public void method2() throws InterruptedException {
        System.out.println("method2 wait for mLock2  " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2  mLock2 acquired   " +
Thread.currentThread().getName());
            Thread.sleep(100);
            method3();
        }
```

```
    }
    public void method3() throws InterruptedException {
        System.out.println("method3  mLock1  "+ Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3   mLock1 acquired  " +
Thread.currentThread().getName());
        }
    }
}
```

Вывод этой программы:

```
methodA wait for mLock1  First
method2 wait for mLock2  Second
method2  mLock2 acquired   Second
methodA mLock1 acquired   First
method3  mLock1  Second
method2 wait for mLock2  First
```

Приостановка выполнения

`Thread.sleep` заставляет текущий поток приостанавливать выполнение в течение определенного периода. Это эффективное средство обеспечения времени процессора для других потоков приложения или других приложений, которые могут выполняться в компьютерной системе. В классе Thread есть два перегруженных метода `sleep` .

Тот, который указывает время сна на миллисекунду

```
public static void sleep(long millis) throws InterruptedException
```

Тот, который указывает время сна на наносекунду

```
public static void sleep(long millis, int nanos)
```

Приостановка выполнения в течение 1 секунды

```
Thread.sleep(1000);
```

Важно отметить, что это намек на планировщик ядра операционной системы. Это может быть не обязательно точным, и некоторые реализации даже не учитывают наносекундный параметр (возможно округление до ближайшей миллисекунды).

Рекомендуется заключить вызов `Thread.sleep` в try / catch и catch `InterruptedException` .

Визуализация барьеров чтения / записи при использовании синхронизированных / нестабильных

Поскольку мы знаем, что мы должны использовать `synchronized` ключевое слово, чтобы

выполнить выполнение метода или исключить блок. Но мало кто из нас может не знать об одном важном аспекте использования `synchronized` и `volatile` ключевого слова: *помимо создания атома кода, он также обеспечивает барьер чтения / записи* . Что это за барьер чтения / записи? Давайте обсудим это с помощью примера:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

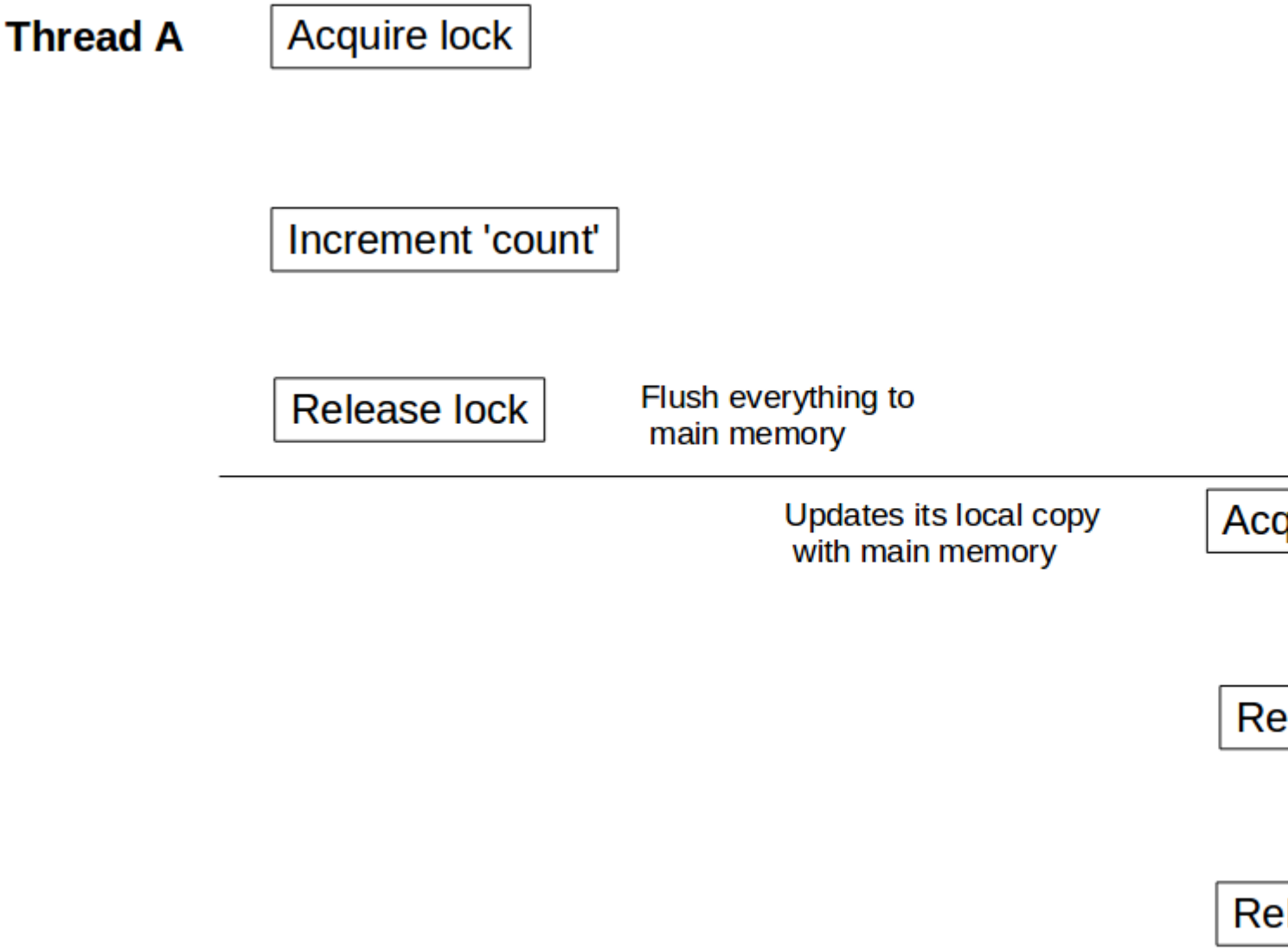
Предположим, что поток *A* сначала вызывает `incrementCount()` затем другой поток *B* вызывает `getCount()` . В этом случае нет гарантии, что B увидит обновленное значение `count` . Он все еще может `count` равным `10` , даже возможно, что он никогда не видит обновленную ценность `count` .

Чтобы понять это поведение, нам нужно понять, как модель памяти Java интегрируется с аппаратной архитектурой. В Java каждый поток имеет собственный поток стека. Этот стек содержит: стек вызовов метода и локальную переменную, созданные в этом потоке. В многоядерной системе вполне возможно, что два потока одновременно работают в отдельных ядрах. В таком сценарии возможно, что часть стека потока находится внутри регистра / кеша ядра. Если внутри потока, объект получает доступ с использованием `synchronized` (или `volatile`) ключевого слова, после `synchronized` блока этот поток синхронизирует его локальную копию этой переменной с основной памятью. Это создает барьер чтения / записи и гарантирует, что поток увидит последнее значение этого объекта.

Но в нашем случае, поскольку поток B не использовал синхронизированный доступ к `count` , он может ссылаться на значение `count` хранящегося в регистре, и никогда не может видеть обновления из потока A. Чтобы убедиться, что B видит последнее значение count, нам нужно сделать `getCount()` синхронизированы.

```
public synchronized Integer getCount() {
  return count;
}
```

Теперь , когда поток A делаются с обновлением `count` он разблокирует `Counter` экземпляр, в то же время создает барьер для записи и сбрасывает все изменения , сделанные внутри этого блока в основную память. Аналогично, когда поток B получает блокировку на одном экземпляре `Counter` , он входит в барьер чтения и считывает значение `count` из основной памяти и видит все обновления.



Тот же эффект видимости распространяется и на `volatile` чтение / запись. Все переменные, обновленные до записи в `volatile` будут сброшены в основную память, и все прочитанные после чтения `volatile` переменной будут записаны из основной памяти.

Создание экземпляра java.lang.Thread

Существует два основных подхода к созданию потока в Java. По сути, создание потока так же просто, как запись кода, который будет выполняться в нем. Эти два подхода отличаются тем, что вы определяете этот код.

В Java поток представлен объектом - экземпляром java.lang.Thread или его подкласса. Таким образом, первый подход заключается в создании этого подкласса и переопределении метода *run ()* .

Примечание . Я использую *Thread* для ссылки на класс java.lang.Thread и *поток* для ссылки на логическую концепцию потоков.

```
class MyThread extends Thread {
    @Override
    public void run() {
```

```
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Теперь, поскольку мы уже определили исполняемый код, поток можно создать просто так:

```
MyThread t = new MyThread();
```

Класс Thread также содержит конструктор, принимающий строку, которая будет использоваться в качестве имени потока. Это может быть особенно полезно при отладке программы с несколькими потоками.

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running! ");
        }
    }
}

MyThread t = new MyThread("Greeting Producer");
```

Второй подход - определить код с помощью java.lang.Runnable и его единственный метод *run ()* . Затем класс Thread позволяет выполнить этот метод в отдельном потоке. Для этого создайте поток, используя конструктор, принимающий экземпляр интерфейса Runnable .

```
Thread t = new Thread(aRunnable);
```

Это может быть очень мощным в сочетании с ссылками на lambdas или methods (только Java 8):

```
Thread t = new Thread(operator::hardWork);
```

Вы также можете указать имя потока.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Практически говоря, вы можете использовать оба подхода без забот. Однако общая мудрость говорит об использовании последнего.

Для каждого из четырех упомянутых конструкторов существует также альтернатива, принимающая экземпляр java.lang.ThreadGroup в качестве первого параметра.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

ThreadGroup представляет собой набор потоков. Вы можете добавить только тему на ThreadGroup с помощью Thread конструктор «s. ThreadGroup затем может быть использован для управления всем его Thread s вместе, а также резьба может получить информацию от своего ThreadGroup .

Итак, чтобы суманализировать, Thread может быть создан одним из этих публичных конструкторов:

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

Последний позволяет нам определить размер требуемого стека для нового потока.

Часто читаемость кода страдает при создании и настройке многих потоков с одинаковыми свойствами или из одного шаблона. Вот когда можно использовать java.util.concurrent.ThreadFactory . Этот интерфейс позволяет вам инкапсулировать процедуру создания потока через фабричный шаблон и его единственный метод *newThread (Runnable)* .

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Потоки прерывания потока / остановки

Каждый поток Java имеет флаг прерывания, который изначально ошибочен. Прерывание потока, по сути, не более чем установка этого флага в true. Код, выполняющийся на этом потоке, может иногда проверять флаг и действовать на него. Код также может полностью игнорировать его. Но почему каждый поток имеет такой флаг? В конце концов, наличие булевого флага в потоке - это то, что мы можем просто организовать, если и когда нам это нужно. Ну, есть методы, которые ведут себя особым образом, когда поток, в котором они работают, прерывается. Эти методы называются методами блокировки. Это методы, которые помещают поток в состояние WAITING или TIMED_WAITING. Когда поток находится в этом состоянии, прерывая его, будет выведено прерывание Exception на

прерванный поток, вместо того, чтобы флаг прерывания был установлен в true, и поток снова станет RUNNABLE. Код, который вызывает метод блокировки, вынужден иметь дело с InterruptedException, поскольку это проверенное исключение. Таким образом, и, следовательно, его имя, прерывание может иметь эффект прерывания WAIT, эффективно заканчивая его. Обратите внимание, что не все методы, которые каким-то образом ждут (например, блокирование IO), реагируют на прерывание таким образом, поскольку они не помещают поток в состояние ожидания. Наконец, поток, у которого установлен флаг прерывания, который вводит метод блокировки (т. Е. Пытается попасть в состояние ожидания), немедленно выдаст исключение InterruptedException, и флаг прерывания будет очищен.

Помимо этой механики, Java не назначает никакого специального семантического значения прерывания. Код может интерпретировать прерывание любым способом, который ему нравится. Но чаще всего прерывание используется для подачи сигнала в поток, который он должен прекратить работать в кратчайшие сроки. Но, как должно быть ясно из вышесказанного, для того, чтобы прекратить работу, отреагировать на это прерывание будет только код этого потока. Остановка потока - это сотрудничество. Когда поток прерывается, его код запуска может находиться на несколько уровней в стеке. Большая часть кода не вызывает метод блокировки и заканчивается достаточно своевременно, чтобы не задерживать остановку потока чрезмерно. Код, который должен в основном касаться реагирования на прерывание, - это код, который находится в задачах обработки цикла, пока их не осталось, или пока флаг не будет установлен, чтобы сигнализировать об этом, чтобы остановить этот цикл. Циклы, которые обрабатывают, возможно, бесконечные задачи (т. Е. Продолжают работать в принципе), должны проверять флаг прерывания, чтобы выйти из цикла. Для конечных циклов семантика может диктовать, что все задачи должны быть закончены до окончания или может быть уместно оставить некоторые необработанные задачи. Код, который вызывает методы блокировки, будет вынужден иметь дело с InterruptedException. Если это вообще возможно семантически, оно может просто распространять InterruptedException и объявлять его бросить. Таким образом, он становится методом блокировки в отношении своих вызывающих абонентов. Если он не может распространять исключение, он должен по крайней мере установить прерванный флаг, поэтому вызывающие выше столбцы также знают, что поток был прерван. В некоторых случаях метод должен продолжать ожидание независимо от InterruptedException, и в этом случае он должен задерживать установку прерванного флага до тех пор, пока он не будет завершен, это может включать настройку локальной переменной, которая должна быть проверена до выхода из метода затем прервите его поток.

Примеры :

Пример кода, который прекращает обработку задач при прерывании

```
class TaskHandler implements Runnable {
```

```
    private final BlockingQueue<Task> queue;

    TaskHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) { // check for interrupt flag, exit
loop when interrupted
            try {
                Task task = queue.take(); // blocking call, responsive to interruption
                handle(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // cannot throw InterruptedException (due
to Runnable interface restriction) so indicating interruption by setting the flag
            }
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Пример кода, который задерживает установку флага прерывания до полного завершения:

```
class MustFinishHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    MustFinishHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        boolean shouldInterrupt = false;

        while (true) {
            try {
                Task task = queue.take();
                if (task.isEndOfTasks()) {
                    if (shouldInterrupt) {
                        Thread.currentThread().interrupt();
                    }
                    return;
                }
                handle(task);
            } catch (InterruptedException e) {
                shouldInterrupt = true; // must finish, remember to set interrupt flag when
we're done
            }
        }
    }

    private void handle(Task task) {
        // actual handling
```

```
    }
}
```

Пример кода, который имеет фиксированный список задач, но может быть прекращен раньше, когда прерван

```
class GetAsFarAsPossible implements Runnable {

    private final List<Task> tasks = new ArrayList<>();

    @Override
    public void run() {
        for (Task task : tasks) {
            if (Thread.currentThread().isInterrupted()) {
                return;
            }
            handle(task);
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Несколько примеров производителей / потребителей с общей глобальной очередью

Ниже кода демонстрируется несколько программ Producer / Consumer. Как потоки Producer, так и Consumer затрагивают одну и ту же глобальную очередь.

```
import java.util.concurrent.*;
import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));

        pes.shutdown();
        ces.shutdown();
    }
}

/* Different producers produces a stream of integers continuously to a shared queue,
which is shared between all Producers and consumers */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
```

```
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // Producer produces a continuous stream of numbers for every 200 milli seconds
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:"+ threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}
/* Different consumers consume data from shared queue, which is shared by both producer and
consumer threads */
class Consumer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    public Consumer (BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.sharedQueue = sharedQueue;
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        // Consumer consumes numbers generated from Producer threads continuously
        while(true){
            try {
                int num = sharedQueue.take();
                System.out.println("Consumed: "+ num + ":by thread:"+threadNo);
            } catch (Exception err) {
               err.printStackTrace();
            }
        }
    }
}
```

выход:

```
Produced:69:by thread:2
Produced:553:by thread:1
Consumed: 69:by thread:1
Consumed: 553:by thread:2
Produced:41:by thread:2
Produced:796:by thread:1
Consumed: 41:by thread:1
Consumed: 796:by thread:2
Produced:728:by thread:2
Consumed: 728:by thread:1
```

и так далее .....

Объяснение:

1. `sharedQueue` , который является `LinkedBlockingQueue` , разделяется между всеми потоками Producer и Consumer.
2. Нити производителя производят одно целое число каждые 200 миллисекунд непрерывно и присоединяют его к `sharedQueue`
3. `Consumer` поток непрерывно потребляет целое число из `sharedQueue` .
4. Эта программа реализована без явных `synchronized` или `Lock` конструкций. BlockingQueue - это ключ к его достижению.

   Реализации BlockingQueue предназначены для использования в основном для очередей производителей-потребителей.

   Реализации BlockingQueue являются потокобезопасными. Все методы очередей осуществляют их эффекты атомарно, используя внутренние блокировки или другие формы контроля параллелизма.

Эксклюзивная запись / параллельный доступ для чтения

Иногда требуется, чтобы процесс одновременно записывал и читал одни и те же «данные».

Интерфейс `ReadWriteLock` и его реализация `ReentrantReadWriteLock` позволяют получить шаблон доступа, который можно описать следующим образом:

1. Может быть любое количество одновременных считывателей данных. Если есть хотя бы один доступ к читателю, доступ к записи невозможен.
2. Данные могут быть не более одного отдельного автора. Если есть доступ к записи, то ни один читатель не может получить доступ к данным.

Реализация может выглядеть так:

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Sample {

// Our lock. The constructor allows a "fairness" setting, which guarantees the chronology of
lock attributions.
protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

// This is a typical data that needs to be protected for concurrent access
protected static int data = 0;

/** This will write to the data, in an exclusive access */
public static void writeToData() {
    RW_LOCK.writeLock().lock();
    try {
        data++;
    } finally {
        RW_LOCK.writeLock().unlock();
```

```
    }
}

public static int readData() {
    RW_LOCK.readLock().lock();
    try {
        return data;
    } finally {
        RW_LOCK.readLock().unlock();
    }
}

}
```

ПРИМЕЧАНИЕ 1. Этот точный вариант использования имеет более чистое решение с использованием `AtomicInteger` , но здесь описывается шаблон доступа, который работает независимо от того, что данные здесь представляют собой целое число, которое как вариант Atomic.

ПРИМЕЧАНИЕ 2 : Блокировка считывающей части действительно необходима, хотя она может выглядеть не так, как у обычного читателя. Действительно, если вы не блокируете читателя, любое количество вещей может пойти не так, в том числе:

1. Запись примитивных значений не гарантируется атомарным на всех JVM, поэтому читатель мог видеть, например, только 32 бита из 64-битной записи, если `data` были 64-битным длинным типом
2. Видимость записи из потока, который не выполнял его, гарантируется JVM только в том случае, если мы устанавливаем *Happen Before relationship* между write и reads. Эта связь устанавливается, когда и читатели, и писатели используют свои соответствующие блокировки, но не иначе

Java SE 8

Если требуется более высокая производительность, при определенных типах использования существует более быстрый тип блокировки, называемый `StampedLock` , который, среди прочего, реализует оптимистичный режим блокировки. Этот замок работает совсем иначе, чем `ReadWriteLock` , и этот образец не может быть транспонирован.

Управляемый объект

Интерфейс `Runnable` определяет один метод, `run()` , предназначенный для содержания кода, выполняемого в потоке.

Объект `Runnable` передается конструктору `Thread` . И вызывается метод `start()` Thread.

пример

```
public class HelloRunnable implements Runnable {
```

```
    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

    public static void main(String[] args) {
        new Thread(new HelloRunnable()).start();
    }
}
```

Пример в Java8:

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Подпроцесс Runnable vs Thread

`Runnable` объекта более общее, потому что объект `Runnable` может подклассифицировать класс, отличный от `Thread` .

Подклассы `Thread` легче использовать в простых приложениях, но ограничены тем фактом, что ваш класс задачи должен быть потомком `Thread` .

Объект `Runnable` применим к API-интерфейсам управления потоками высокого уровня.

семафор

Семафор - это высокоуровневый синхронизатор, который поддерживает набор *разрешений,* которые могут быть получены и выпущены потоками. Семафор можно представить как счетчик *разрешений,* который будет уменьшаться при получении потока и увеличиваться при потоке. Если количество *разрешений* равно `0` когда поток пытается получить, то поток будет блокироваться до тех пор, пока разрешение не будет доступно (или пока поток не будет прерван).

Семафор инициализируется как:

```
Semaphore semaphore = new Semaphore(1); // The int value being the number of permits
```

Конструктор Семафор допускает дополнительный логический параметр для справедливости. Если установлено false, этот класс не дает никаких гарантий относительно порядка, в котором потоки приобретают разрешения. Когда справедливость установлена, семафор гарантирует, что потоки, вызывающие любой из методов получения, выбираются для получения разрешений в том порядке, в котором их обращение этих методов было обработано. Он объявляется следующим образом:

```
Semaphore semaphore = new Semaphore(1, true);
```

Теперь давайте рассмотрим пример из javadocs, где Семафор используется для контроля

доступа к пулу элементов. Семафор используется в этом примере для обеспечения функциональности блокировки, чтобы гарантировать, что всегда будут получаться элементы, получаемые при `getItem()` .

```
class Pool {
    /*
     * Note that this DOES NOT bound the amount that may be released!
     * This is only a starting value for the Semaphore and has no other
     * significant meaning UNLESS you enforce this inside of the
     * getNextAvailableItem() and markAsUnused() methods
     */
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

    /**
     * Obtains the next available item and reduces the permit count by 1.
     * If there are no items available, block.
     */
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    /**
     * Puts the item into the pool and add 1 permit.
     */
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
    }

    private Object getNextAvailableItem() {
        // Implementation
    }

    private boolean markAsUnused(Object o) {
        // Implementation
    }
}
```

Добавьте два массива `int`, используя Threadpool

В Threadpool есть очередь задач, каждая из которых будет выполнена на одной из этих потоков.

В следующем примере показано, как добавить две массивы `int` с помощью Threadpool.

Java SE 8

```
int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// Setup the ThreadPool:
```

```
// for each element in the array, submit a worker to the pool that adds elements
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker] );
}

// Wait for all Workers to finish:
try {
    // execute all submitted tasks
    pool.shutdown();
    // waits until all workers finish, or the timeout ends
    pool.awaitTermination(12, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    pool.shutdownNow(); //kill thread
}

System.out.println(Arrays.toString(result));
```

Заметки:

1. Этот пример является исключительно иллюстративным. На практике не будет никакого ускорения, если использовать потоки для этой задачи. Вероятно, замедление замедляется, поскольку накладные расходы на создание и планирование задач увеличивают время, затрачиваемое на выполнение задачи.

2. Если вы использовали Java 7 и более ранние версии, вы должны использовать анонимные классы вместо lambdas для выполнения задач.

Получить статус всех потоков, запущенных вашей программой, за исключением системных потоков

Фрагмент кода:

```
import java.util.Set;

public class ThreadStatus {
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++){
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("Thread :" + t + ":" + "state:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("Thread count started by Main thread:" + threadCount);
    }
}
```

```
class MyThread implements Runnable {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

Выход:

```
Thread :Thread[MyThread:1,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:3,5,main]:state:TIMED_WAITING
Thread :Thread[main,5,main]:state:RUNNABLE
Thread :Thread[MyThread:4,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:0,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:2,5,main]:state:TIMED_WAITING
Thread count started by Main thread:6
```

Объяснение:

`Thread.getAllStackTraces().keySet()` возвращает все `Thread` включая потоки приложений и потоки системы. Если вас интересует только статус Threads, запущенный вашим приложением, выполните итерацию набора `Thread` , проверив Thread Group определенного потока на ваш основной поток программы.

В отсутствии выше условия ThreadGroup программа возвращает статус ниже системных потоков:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Вызов и будущее

Хотя `Runnable` предоставляет средство для переноса кода, который должен выполняться в другом потоке, он имеет ограничение в том, что он не может вернуть результат выполнения. Единственный способ получить некоторое возвращаемое значение из выполнения `Runnable` - это присвоить результат переменной, доступной в области вне `Runnable` .

`Callable` была представлена на Java 5 как одноранговое средство `Runnable` . `Callable` по сути `Callable` и та же, за исключением того, что вместо `run` метод `call` . Метод `call` имеет дополнительную возможность возвращать результат, а также разрешено выдавать проверенные исключения.

Результат *отправки задания Callable* доступен для использования через будущее

`Future` можно рассматривать как контейнер, который содержит результат вычисления `Callable` . Вычисление вызываемого может продолжаться в другом потоке, и любая попытка использовать результат `Future` будет блокировать и будет возвращать результат только после его появления.

Интерфейс с возможностью вызова

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Будущее

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Использование примера Callable and Future:

```
public static void main(String[] args) throws Exception {
    ExecutorService es = Executors.newSingleThreadExecutor();

    System.out.println("Time At Task Submission : " + new Date());
    Future<String> result = es.submit(new ComplexCalculator());
    // the call to Future.get() blocks until the result is available.So we are in for about a
10 sec wait now
    System.out.println("Result of Complex Calculation is : " + result.get());
    System.out.println("Time At the Point of Printing the Result : " + new Date());
}
```

Наш Callable, который делает длительные вычисления

```
public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // just sleep for 10 secs to simulate a lengthy computation
        Thread.sleep(10000);
        System.out.println("Result after a lengthy 10sec calculation");
        return "Complex Result"; // the result
    }
}
```

Выход

```
Time At Task Submission : Thu Aug 04 15:05:15 EDT 2016
Result after a lengthy 10sec calculation
Result of Complex Calculation is : Complex Result
Time At the Point of Printing the Result : Thu Aug 04 15:05:25 EDT 2016
```

Другие операции, разрешенные на будущее

Хотя `get()` - это метод для извлечения фактического результата. Будущее имеет положение

- `get(long timeout, TimeUnit unit)` определяет максимальный период времени в течение текущего потока, который будет ждать результата;
- Для отмены отмены вызова задачи `cancel(mayInterruptIfRunning)` . Флаг `mayInterrupt` указывает, что задача должна быть прервана, если она была запущена и работает прямо сейчас;
- Чтобы проверить, завершена ли задача / завершена вызовом `isDone()` ;
- Чтобы проверить, отменена ли длительная задача, `isCancelled()` .

Замки как средства синхронизации

До внедрения параллельного пакета Java 5 потоки были более низкими. Введение этого пакета обеспечило несколько вспомогательных программных средств / конструкций, поддерживающих более высокий уровень.

Замки - это механизмы синхронизации потоков, которые по существу служат той же цели, что и синхронизированные блоки или ключевые слова.

Внутренняя блокировка

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Синхронизация с использованием замков

```
int count = 0; // shared among multiple threads

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // a non-atomic operation
    } finally {
        lockObj.unlock(); // sure to release the lock without fail
    }
}
```

Блокировки также имеют функциональные возможности, которые встроенная блокировка не предлагает, например, блокировка, но остается реагирующей на прерывание или пытается заблокировать, а не блокировать, когда не удается.

Блокировка, реагирующая на прерывание

```
class Locky {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
            try {
                lockObj.lockInterruptibly();
                ++count; // a non-atomic operation
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // stopping
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

Делайте только то, что можно заблокировать

```
public class Locky2 {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        boolean locked = lockObj.tryLock(); // returns true upon successful lock
        if (locked) {
            try {
                ++count; // a non-atomic operation
            } finally {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

Существует несколько вариантов блокировки. Для более подробной информации обратитесь к api docs здесь

Прочитайте Параллельное программирование (темы) онлайн:
https://riptutorial.com/ru/java/topic/121/параллельное-программирование--темы-

глава 137: Параллельное программирование с использованием структуры Fork / Join

Examples

Вилка / Присоединение задач в Java

Структура fork / join в Java идеально подходит для проблемы, которая может быть разделена на более мелкие части и решена параллельно. Основными шагами проблемы fork / join являются:

- Разделите проблему на несколько частей
- Решите каждую из частей параллельно друг другу
- Объедините каждый из подрешений в одно общее решение

ForkJoinTask - это интерфейс, который определяет такую проблему. Обычно ожидается, что вы подклассируете одну из своих абстрактных реализаций (обычно рекурсивную), а не реализуете интерфейс напрямую.

В этом примере мы собираемся суммировать набор целых чисел, делясь до тех пор, пока не получим размер партии не более десяти.

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // This constructor is only used internally as part of the dividing process
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // This is too big for a single batch, so we shall divide into two tasks
            int mid = (minInclusive + maxExclusive) / 2;
```

```
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // Submit the left hand task as a new task to the same ForkJoinPool
            leftTask.fork();

            // Run the right hand task on the same thread and get the result
            int rightResult = rightTask.compute();

            // Wait for the left hand task to complete and get its result
            int leftResult = leftTask.join();

            // And combine the result
            return leftResult + rightResult;
        } else {
            // This is fine for a single batch, so we will run it here and now
            int sum = 0;
            for (int i = minInclusive; i < maxExclusive; i++) {
                sum += numbers.get(i);
            }
            return sum;
        }
    }
}
```

Экземпляр этой задачи теперь можно передать экземпляру ForkJoinPool .

```
// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();
```

Прочитайте Параллельное программирование с использованием структуры Fork / Join онлайн: https://riptutorial.com/ru/java/topic/4245/параллельное-программирование-с-использованием-структуры-fork---join

глава 138: Параллельные коллекции

Вступление

Параллельная коллекция представляет собой [сборник] [1], который позволяет одновременно получать доступ к нескольким потокам. Различные потоки обычно могут проходить через содержимое коллекции и добавлять или удалять элементы. Сбор несет ответственность за то, чтобы сбор не стал коррумпированным. [1]: http://stackoverflow.com/documentation/java/90/collections#t=20161222193649729848 4

Examples

Тематические коллекции

По умолчанию различные типы коллекций не являются потокобезопасными.

Тем не менее, сделать коллекцию поточно-безопасной довольно простой.

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String,
String>());
```

Когда вы создаете поточно-безопасную коллекцию, вы никогда не должны обращаться к ней через исходную коллекцию, только через потокобезопасную оболочку.

Java SE 5

Начиная с Java 5, у `java.util.collections` есть несколько новых потокобезопасных коллекций, которым не нужны различные методы `Collections.synchronized` .

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Параллельные коллекции

Одновременные коллекции - это обобщение потокобезопасных коллекций, которые позволяют использовать более широкое использование в параллельной среде.

Хотя в потокобезопасных коллекциях есть безопасное добавление или удаление элементов из нескольких потоков, они не обязательно имеют безопасную итерацию в одном и том же контексте (возможно, не удастся выполнить итерацию через коллекцию в одном потоке, в то время как другая изменяет ее путем добавления / удаление элементов).

Здесь используются параллельные коллекции.

Поскольку итерация часто является базовой реализацией нескольких массовых методов в коллекциях, таких как `addAll` , `removeAll` , а также копирование коллекции (через конструктор или другие средства), сортировка, ... пример использования для параллельных коллекций на самом деле довольно большой.

Например, Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` - это поточно-безопасная и параллельная реализация `Lis` t, ее javadoc заявляет:

> Метод итератора стиля «моментальный снимок» использует ссылку на состояние массива в точке, в которой был создан итератор. Этот массив никогда не изменяется в течение жизни итератора, поэтому помехи невозможны, и итератору гарантировано не бросать ConcurrentModificationException.

Поэтому следующий код безопасен:

```
public class ThreadSafeAndConcurrent {

public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

public static void main(String[] args) throws InterruptedException {
    Thread modifier = new Thread(new ModifierRunnable());
    Thread iterator = new Thread(new IteratorRunnable());
    modifier.start();
    iterator.start();
    modifier.join();
    iterator.join();
}

public static final class ModifierRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 50000; i++) {
                LIST.add(i);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static final class IteratorRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                long total = 0;
                for(Integer inList : LIST) {
                    total += inList;
                }
                System.out.println(total);
            }
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }
}
}
```

Другая параллельная коллекция, посвященная итерации, - `ConcurrentLinkedQueue` , которая гласит:

> Итераторы слабо согласованы, возвращая элементы, отражающие состояние очереди, в какой-то момент или после создания итератора. Они не бросают java.util.ConcurrentModificationException и могут продолжаться одновременно с другими операциями. Элементы, содержащиеся в очереди с момента создания итератора, будут возвращены ровно один раз.

Нужно проверить javadocs, чтобы увидеть, является ли коллекция параллельной или нет. Атрибуты итератора, возвращаемые методом `iterator()` («fail fast», «слабо согласованный», ...), являются наиболее важным атрибутом для поиска.

Потоковые, но не параллельные примеры

В приведенном выше коде, изменяя объявление `LIST`

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Может (и статистически будет на большинстве современных многопроцессорных / основных архитектур) приведет к исключениям.

Синхронизированные коллекции из методов утилиты `Collections` являются потокобезопасными для добавления / удаления элементов, но не для итерации (если базовая коллекция не передается ей уже).

Вставка в ConcurrentHashMap

```
public class InsertIntoConcurrentHashMap
{

    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
```

```
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
         }

        else
        {
            //'value' reference is mapped to key = 1.
        }
    }
}
```

Прочитайте Параллельные коллекции онлайн: https://riptutorial.com/ru/java/topic/8363/параллельные-коллекции

глава 139: Перечисления

Вступление

Переменные Java (объявленные с использованием ключевого слова `enum`) - это сокращенный синтаксис для значительных количеств констант одного класса.

Синтаксис

- [public / protected / private] enum Enum_name {// Объявить новое перечисление.
- ENUM_CONSTANT_1 [, ENUM_CONSTANT_2 ...]; // Объявляем константы перечисления. Это должна быть первая строка внутри перечисления и должна быть разделена запятыми, с точкой с запятой в конце.
- ENUM_CONSTANT_1 (param) [, ENUM_CONSTANT_2 (param) ...]; // Объявлять константы enum с параметрами. Типы параметров должны соответствовать конструктору.
- ENUM_CONSTANT_1 {...} [, ENUM_CONSTANT_2 {...} ...]; // Объявлять константы enum с переопределенными методами. Это необходимо сделать, если перечисление содержит абстрактные методы; все такие методы должны быть реализованы.
- ENUM_CONSTANT.name () // Возвращает строку с именем константы перечисления.
- ENUM_CONSTANT.ordinal () // Возвращает порядковый номер этой константы перечисления, ее позицию в объявлении перечисления, где исходной константе присваивается порядковый номер нуля.
- Enum_name.values () // Возвращает новый массив (типа Enum_name []), содержащий каждую константу этого перечисления каждый раз, когда он вызывается.
- Enum_name.valueOf ("ENUM_CONSTANT") // Обратное к ENUM_CONSTANT.name () - возвращает константу перечисления с заданным именем.
- Enum.valueOf (Enum_name.class, "ENUM_CONSTANT") // Синоним предыдущей: Обратный ENUM_CONSTANT.name () - возвращает константу перечисления с заданным именем.

замечания

ограничения

Enums всегда расширяют `java.lang.Enum` , поэтому невозможно, чтобы enum расширил класс. Однако они могут реализовать множество интерфейсов.

Советы и хитрости

Из-за их специализированного представления есть более эффективные карты и наборы, которые можно использовать с перечислениями в качестве их ключей. Они часто будут работать быстрее, чем их неспециализированные коллеги.

Examples

Объявление и использование базового перечисления

Enum может считаться синтаксическим сахаром для закрытого класса, который создается только во время компиляции, чтобы определить набор констант.

Простые перечисления для перечисления разных сезонов будут объявлены следующим образом:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Хотя константы перечисления необязательно должны быть во всех шапках, это Java-соглашение, что имена констант полностью заглавные, со словами, разделенными символами подчеркивания.

Вы можете объявить Enum в собственном файле:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Но вы также можете объявить его в другом классе:

```
 public class Day {

    private Season season;

    public String getSeason() {
        return season.name();
    }

    public void setSeason(String season) {
        this.season = Season.valueOf(season);
    }
```

```
    /**
     * This enum is declared inside the Day.java file and
     * cannot be accessed outside because it's declared as private.
     */
    private enum Season {
        WINTER,
        SPRING,
        SUMMER,
        FALL
    }

}
```

Наконец, вы не можете объявить Enum внутри тела метода или конструктора:

```
public class Day {

    /**
     * Constructor
    */
    public Day() {
        // Illegal. Compilation error
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }
    }

    public void aSimpleMethod() {
        // Legal. You can declare a primitive (or an Object) inside a method. Compile!
        int primitiveInt = 42;

        // Illegal. Compilation error.
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }

        Season season = Season.SPRING;
    }

}
```

Дублирующие константы перечисления не допускаются:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Каждая константа enum является **`public`**, **`static`** и **`final`** по умолчанию. Поскольку каждая константа `static`, к ней можно получить доступ напрямую, используя имя перечисления.

Константы Enum могут передаваться как параметры метода:

```
public static void display(Season s) {
    System.out.println(s.name());  // name() is a built-in method that gets the exact name of
the enum constant
}

display(Season.WINTER);  // Prints out "WINTER"
```

Вы можете получить массив констант перечисления, используя метод `values()`. Значения гарантированно будут в порядке объявления в возвращаемом массиве:

```
Season[] seasons = Season.values();
```

Примечание. Этот метод выделяет новый массив значений каждый раз, когда он вызывается.

Чтобы перебрать константы перечисления:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

Вы можете использовать перечисления в инструкции `switch`:

```
public static void enumSwitchExample(Season s) {
    switch(s) {
        case WINTER:
            System.out.println("It's pretty cold");
            break;
        case SPRING:
            System.out.println("It's warming up");
            break;
        case SUMMER:
            System.out.println("It's pretty hot");
            break;
        case FALL:
            System.out.println("It's cooling down");
            break;
    }
}
```

Вы также можете сравнить константы enum, используя `==`:

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Другим способом сравнения констант перечисления является использование `equals()` как показано ниже, что считается плохой практикой, поскольку вы можете легко попасть в ловушки следующим образом:

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

Кроме того, хотя набор экземпляров в `enum` не может быть изменен во время выполнения, сами экземпляры не являются неотъемлемо неизменными, потому что, как и любой другой класс, `enum` может содержать изменяемые поля, как показано ниже.

```
public enum MutableExample {
    A,
    B;

    private int count = 0;

    public void increment() {
        count++;
    }

    public void print() {
        System.out.println("The count of " + name() + " is " + count);
    }
}

// Usage:
MutableExample.A.print();       // Outputs 0
MutableExample.A.increment();
MutableExample.A.print();       // Outputs 1 -- we've changed a field
MutableExample.B.print();       // Outputs 0 -- another instance remains unchanged
```

Однако хорошая практика состоит в том, чтобы сделать экземпляры `enum` неизменяемыми, т. Е. Когда у них либо нет каких-либо дополнительных полей, либо все такие поля отмечены как `final` и неизменяемы. Это гарантирует, что на протяжении всего срока службы `enum` не будет пропускать какую-либо память и что безопасно использовать его экземпляры во всех потоках.

Enums неявно реализуют `Serializable` и `Comparable` потому что класс `Enum` делает:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Перечисления с конструкторами

У `enum` не может быть публичного конструктора; однако частные конструкторы приемлемы (конструкторы для перечислений по умолчанию являются закрытыми по пакетам):

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // usual names for US coins
    // note that the above parentheses and the constructor arguments match
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

Рекомендуется сохранять все поля частными и предоставлять методы getter, так как существует конечное количество экземпляров для перечисления.

Если бы вы вместо этого выполняли `Enum` как `class` , это выглядело бы так:

```
public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);
    public static final Coin QUARTER = new Coin(25);

    private int value;

    private Coin(int value){
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

Константы континуума технически изменяемы, поэтому может быть добавлен сеттер для изменения внутренней структуры константы перечисления. Однако это считается очень плохой практикой, и его следует избегать.

Лучшая практика заключается в том, чтобы сделать поля Enum неизменяемыми, с `final` :

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }
```

```
    ...

}
```

Вы можете определить несколько конструкторов в одном и том же перечислении. Когда вы это сделаете, аргументы, которые вы передаете в объявлении enum, определяют, какой конструктор вызывается:

```
public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...

}
```

Примечание. Все не примитивные поля перечисления должны реализовывать `Serializable` потому что класс `Enum` делает.

Использование методов и статических блоков

Перечисление может содержать метод, как и любой класс. Чтобы увидеть, как это работает, мы объявим перечисление следующим образом:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Давайте получим метод, возвращающий перечисление в противоположном направлении:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    public Direction getOpposite(){
        switch (this){
            case NORTH:
                return SOUTH;
            case SOUTH:
                return NORTH;
            case WEST:
```

```
                return EAST;
            case EAST:
                return WEST;
            default: //This will never happen
                return null;
        }
    }
}
```

Это может быть дополнительно улучшено за счет использования полей и статических блоков инициализатора:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    private Direction opposite;

    public Direction getOpposite(){
        return opposite;
    }

    static {
        NORTH.opposite = SOUTH;
        SOUTH.opposite = NORTH;
        WEST.opposite = EAST;
        EAST.opposite = WEST;
    }
}
```

В этом примере противоположное направление сохраняется в поле частного экземпляра `opposite` , которое статически инициализируется при первом использовании `Direction` . В этом конкретном случае (поскольку `NORTH` ссылается на `SOUTH` и наоборот), мы не можем использовать Enums с конструкторами здесь (конструкторы `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST)` были бы более изящными и позволяли бы `opposite` будут объявлены `final` , но будут самореферентными и, следовательно, не разрешены).

Осуществляет интерфейс

Это `enum` , которое также является вызываемой функцией, которая проверяет входные данные `String` против предварительно скомпилированных шаблонов регулярных выражений.

```
import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }
```

```
    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}
```

Каждый член перечисления может также реализовать метод:

```
import java.util.function.Predicate;

enum Acceptor implements Predicate<String> {
    NULL {
        @Override
        public boolean test(String s) { return s == null; }
    },
    EMPTY {
        @Override
        public boolean test(String s) { return s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean test(String s) { return NULL.test(s) || EMPTY.test(s); }
    };
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Acceptor.NULL.test(null));  // true
        System.out.println(Acceptor.EMPTY.test(""));   // true
        System.out.println(Acceptor.NULL_OR_EMPTY.test(" ")); // false
    }
}
```

Образец полиморфизма Enum

Когда метод должен принимать «расширяемый» набор значений `enum` , программист может применять полиморфизм, как в обычном `class` путем создания интерфейса, который будет использоваться любым, где будут использоваться `enum` :

```
public interface ExtensibleEnum {
    String name();
}
```

Таким образом, любое `enum` помеченное (реализующим) интерфейс, может использоваться как параметр, позволяющий программисту создавать переменную сумму `enum` которая будет принята методом. Это может быть полезно, например, в API , где есть (неизменяемый) по

умолчанию `enum` и пользователь этих API , хотят «продлить» на `enum` с большим количеством значений.

Набор значений enum по умолчанию можно определить следующим образом:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Дополнительные значения затем могут быть определены следующим образом:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}
```

Пример, который показывает, как использовать перечисления - обратите внимание, как `printEnum()` принимает значения из обоих типов `enum` :

```
private void printEnum(ExtensibleEnum val) {
    System.out.println(val.name());
}

printEnum(DefaultValues.VALUE_ONE);    // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO);    // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR);  // VALUE_FOUR
```

Примечание. Этот шаблон не мешает вам переопределять значения перечисления, которые уже определены в одном перечислении, в другом перечислении. Тогда эти значения перечисления будут разными. Кроме того, невозможно использовать switch-on-enum, поскольку все, что у нас есть, это интерфейс, а не реальное `enum` .

Перечисления с абстрактными методами

Перечисления могут определять абстрактные методы, которые должен выполнять каждый член `enum` .

```
enum Action {
    DODGE {
        public boolean execute(Player player) {
            return player.isAttacking();
        }
    },
    ATTACK {
        public boolean execute(Player player) {
            return player.hasWeapon();
        }
    },
    JUMP {
        public boolean execute(Player player) {
            return player.getCoordinates().equals(new Coordinates(0, 0));
        }
```

```
    };

    public abstract boolean execute(Player player);
}
```

Это позволяет каждому члену перечисления определять свое поведение для данной операции без необходимости включать типы в метод в определении верхнего уровня.

Обратите внимание, что этот шаблон является короткой формой того, что обычно достигается с использованием полиморфизма и / или реализации интерфейсов.

Документирование перечислений

Не всегда имя `enum` достаточно понятно для понимания. Чтобы зарегистрировать `enum` , используйте стандартный javadoc:

```
/**
 * United States coins
 */
public enum Coins {

    /**
     * One-cent coin, commonly known as a penny,
     * is a unit of currency equaling one-hundredth
     * of a United States dollar
     */
    PENNY(1),

    /**
     * A nickel is a five-cent coin equaling
     * five-hundredth of a United States dollar
     */
    NICKEL(5),

    /**
     * The dime is a ten-cent coin refers to
     * one tenth of a United States dollar
     */
    DIME(10),

    /**
     * The quarter is a US coin worth 25 cents,
     * one-fourth of a United States dollar
     */
    QUARTER(25);

    private int value;

    Coins(int value){
        this.value = value;
    }

    public int getValue(){
        return value;
    }
}
```

Получение значений перечисления

Каждый класс перечисления содержит неявный статический метод с именем `values()` . Этот метод возвращает массив, содержащий все значения этого перечисления. Вы можете использовать этот метод для перебора значений. Однако важно отметить, что этот метод возвращает **новый** массив каждый раз, когда он вызывается.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
    * Print out all the values in this enum.
    */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}
```

Если вам нужен `Set` вы также можете использовать `EnumSet.allOf(Day.class)` .

Enum как параметр ограниченного типа

При написании класса с дженериками в java можно гарантировать, что параметр type является перечислением. Поскольку все перечисления расширяют класс `Enum` , может использоваться следующий синтаксис.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

В этом примере тип `T` *должен* быть перечислением.

Получить перечисление по имени

Скажем, у нас есть перечисление `DayOfWeek` :

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

Перечисление компилируется со встроенным статическим методом `valueOf()` который может использоваться для поиска константы по ее имени:

```
String dayName = DayOfWeek.SUNDAY.name();
```

```
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

Это также возможно с использованием динамического типа перечисления:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Оба эти метода `valueOf()` будут `IllegalArgumentException` если указанное перечисление не имеет константы с соответствующим именем.

Библиотека Guava предоставляет вспомогательный метод `Enums.getIfPresent()` который возвращает Guava `Optional` для устранения явной обработки исключений:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Реализовать шаблон Singleton с перечислением из одного элемента

Константы Enum создаются, когда перечисление ссылается в первый раз. Таким образом, это позволяет реализовать шаблон разработки программного обеспечения Singleton с перечислением из одного элемента.

```
public enum Attendant {

    INSTANCE;

    private Attendant() {
        // perform some initialization routine
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}


public class Main {

    public static void main(String... args) {
        Attendant.INSTANCE.sayHello();// instantiated at this point
    }
}
```

Согласно книге «Эффективная Ява» Джошуа Блоха, одноэлементное перечисление - лучший способ реализовать синглтон. Такой подход имеет следующие преимущества:

- безопасность потока

- гарантия единого экземпляра
- серийная сериализация

И, как показано в разделе, реализующем интерфейс, этот синглтон может также реализовывать один или несколько интерфейсов.

Enum со свойствами (полями)

В случае, если мы хотим использовать `enum` с дополнительной информацией, а не только как постоянные значения, и мы хотим иметь возможность сравнивать два перечисления.

Рассмотрим следующий пример:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    public boolean isGreaterThan(Coin other){
        return this.value > other.value;
    }

}
```

Здесь мы определили `Enum` называемый `Coin` который представляет его значение. С помощью метода `isGreaterThan` мы можем сравнить два `enum` s:

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

Преобразование перечисления в строку

Иногда вы хотите преобразовать свой enum в String, есть два способа сделать это.

Предположим, что мы имеем:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

Итак, как мы можем конвертировать что-то вроде `Fruit.APPLE` в `"APPLE"` ?

Преобразовать с помощью `name()`

`name()` - это внутренний метод в `enum` который возвращает представление `String` перечисления, возвращаемая `String` представляет ***точно,*** как было определено значение перечисления.

Например:

```
System.out.println(Fruit.BANANA.name());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Преобразование с помощью команды `toString()`

`toString()` *по умолчанию* переопределяется, чтобы иметь такое же поведение, как `name()`

Однако `toString()` скорее всего переопределяется *разработчиками,* чтобы сделать печать более удобной для пользователя `String`

> Не используйте `toString()` если вы хотите проверить свой код, `name()` для этого гораздо более стабильно. Используйте только `toString()` когда вы собираетесь выводить значение в журналы или stdout или что-то еще

По умолчанию:

```
System.out.println(Fruit.BANANA.toString());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Пример переопределения

```
System.out.println(Fruit.BANANA.toString());       // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Enum постоянное определенное тело

В `enum` можно определить конкретное поведение для конкретных постоянная `enum` , который переопределяет поведение по умолчанию `enum` , этот метод известен как *постоянная удельное тело.*

Предположим, что три ученика-пианиста - Джон, Бен и Люк - определены в `enum` имени `PianoClass` :

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

И в один прекрасный день приезжают двое других учеников - Рита и Том - с сексом (Женщина) и уровнем (Промежуточный), которые не соответствуют предыдущим:

```
    enum PianoClass2 {
    JOHN, BEN, LUKE, RITA, TOM;
        public String getSex() {
            return "Male"; // issue, Rita is a female
        }
        public String getLevel() {
            return "Beginner"; // issue, Tom is an intermediate student
        }
    }
```

так что просто добавление новых учеников в объявление констант, как указано ниже, неверно:

```
PianoClass2 tom = PianoClass2.TOM;
PianoClass2 rita = PianoClass2.RITA;
System.out.println(tom.getLevel()); // prints Beginner -> wrong Tom's not a beginner
System.out.println(rita.getSex()); // prints Male -> wrong Rita's not a male
```

Можно определить конкретное поведение для каждой из констант, Rita и Tom, которая переопределяет поведение `PianoClass2` умолчанию следующим образом:

```
enum PianoClass3 {
    JOHN, BEN, LUKE,
    RITA {
        @Override
        public String getSex() {
            return "Female";
        }
    },
    TOM {
        @Override
        public String getLevel() {
            return "Intermediate";
        }
    };
    public String getSex() {
        return "Male";
    }
    public String getLevel() {
        return "Beginner";
    }
}
```

и теперь уровень Тома и секс Риты такие, какими они должны быть:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Другой способ определить тело, специфичное для контента, - это использовать конструктор, например:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

и использование:

```
Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender());     // Male
System.out.println(john.getGender());    // Male
System.out.println(jane.getGender());    // Female
```

Нумерация нулевого экземпляра

```
enum Util {
    /* No instances */;

    public static int clamp(int min, int max, int i) {
        return Math.min(Math.max(i, min), max);
    }

    // other utility methods...
}
```

Так же, как `enum` может использоваться для одиночных игр (1 класс экземпляров), его можно использовать для служебных классов (0 классов экземпляров). Просто убедитесь, что завершили (пустой) список констант перечисления с помощью `;` ,

См. Вопрос Zune enum vs private constructors для предотвращения создания экземпляра для обсуждения pro и con по сравнению с частными конструкторами.

Перечисления со статическими полями

Если у вашего класса enum есть статические поля, имейте в виду, что они создаются **после** самих значений перечисления. Это означает, что следующий код приведет к `NullPointerException` :

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers = new HashMap<>();

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

Возможный способ исправить это:

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}
```

Не инициализируйте статическое поле:

```
enum Example {
    ONE(1), TWO(2);

    // after initialisisation integers is null!!
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
    // !!this may lead to null poiner exception!!
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}
```

initialisation:

- создавать значения перечисления
    - как побочный эффект putValue (), который инициализирует целые числа
- статические значения установлены
    - integers = null; // выполняется после перечислений, поэтому содержимое целых чисел теряется

Сравнить и Содержит для значений Enum

Перечисления содержат только константы и могут быть непосредственно сопоставлены с `==` . Итак, нужна только проверка ссылок, нет необходимости использовать метод `.equals` . Более того, если `.equals` используются неправильно, может вызвать `.equals` `NullPointerException` то время как это не относится к проверке `==` .

```
enum Day {
    GOOD, AVERAGE, WORST;
}

public class Test {

    public static void main(String[] args) {
        Day day = null;

        if (day.equals(Day.GOOD)) {//NullPointerException!
            System.out.println("Good Day!");
        }

        if (day == Day.GOOD) {//Always use == to compare enum
            System.out.println("Good Day!");
        }

    }
}
```

Чтобы группировать, дополнять, диапазон значений enum имеет класс `EnumSet` который содержит разные методы.

- `EnumSet#range` : для получения подмножества перечисления по диапазону, определяемому двумя конечными точками
- `EnumSet#of` : набор конкретных перечислений без какого-либо диапазона. Многократное перегружен `of` методов есть.
- `EnumSet#complementOf` : набор перечислений, который является дополнением к значениям перечисления, указанным в параметре метода

```
enum Page {
   A1, A2, A3, A4, A5, A6, A7, A8, A9, A10
}
```

```
public class Test {

  public static void main(String[] args) {
      EnumSet<Page> range = EnumSet.range(Page.A1, Page.A5);

      if (range.contains(Page.A4)) {
          System.out.println("Range contains A4");
      }

      EnumSet<Page> of = EnumSet.of(Page.A1, Page.A5, Page.A3);

      if (of.contains(Page.A1)) {
          System.out.println("Of contains A1");
      }
  }
}
```

Прочитайте Перечисления онлайн: https://riptutorial.com/ru/java/topic/155/перечисления

глава 140: Полиморфизм

Вступление

Полиморфизм является одним из основных концепций ООП (объектно-ориентированного программирования). Слово полиморфизма было получено из греческих слов «poly» и « morphs». Поли означает «много», а морфы означают «формы» (многие формы).

Существует два способа выполнения полиморфизма. **Перегрузка метода и переопределение метода** .

замечания

`Interfaces` - это еще один способ добиться полиморфизма в Java, помимо классового наследования. Интерфейсы определяют список методов, которые формируют API программы. Классы должны `implement interface` , переопределяя все его методы.

Examples

Перегрузка метода

Перегрузка метода , также известная как **перегрузка функций** , - это способность класса иметь несколько методов с тем же именем, если они отличаются друг от друга по количеству или типу аргументов.

Компилятор проверяет ***подпись*** метода для перегрузки метода.

Подпись метода состоит из трех вещей:

1. Имя метода
2. Количество параметров
3. Типы параметров

Если эти три одинаковы для любых двух методов в классе, то компилятор выдает **ошибку повторяющегося метода** .

Этот тип полиморфизма называется *статическим* или *компиляционным* полиморфизмом, потому что соответствующий метод, который должен быть вызван, определяется компилятором во время компиляции на основе списка аргументов.

```
class Polymorph {

    public int add(int a, int b){
        return a + b;
```

```
    }

    public int add(int a, int b, int c){
        return a + b + c;
    }

    public float add(float a, float b){
        return a + b;
    }

    public static void main(String... args){
        Polymorph poly = new Polymorph();
        int a = 1, b = 2, c = 3;
        float d = 1.5, e = 2.5;

        System.out.println(poly.add(a, b));
        System.out.println(poly.add(a, b, c));
        System.out.println(poly.add(d, e));
    }

}
```

Это приведет к:

```
2
6
4.000000
```

Перегруженные методы могут быть статическими или нестационарными. Это также не приводит к перегрузке метода.

```
public class Polymorph {

    private static void methodOverloaded()
    {
        //No argument, private static method
    }

    private int methodOverloaded(int i)
    {
        //One argument private non-static method
        return i;
    }

    static int methodOverloaded(double d)
    {
        //static Method
        return 0;
    }

    public void methodOverloaded(int i, double d)
    {
        //Public non-static Method
    }
}
```

Также, если вы измените тип возвращаемого метода, мы не сможем получить его как

перегрузку метода.

```
public class Polymorph {

void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Переопределение метода

Переопределение метода - это способность подтипов переопределять (переопределять) поведение их супертипов.

В Java это переводит в подклассы, переопределяя методы, определенные в суперклассе. В Java все непримитивные переменные являются фактически `references` , которые схожи с указателями на местоположение фактического объекта в памяти. `references` имеют только один тип, который является типом, с которым они были объявлены. Однако они могут указывать на объект либо их объявленного типа, либо любого из его подтипов.

Когда метод вызывается по `reference` , **вызывается** соответствующий **метод фактического объекта, на который указывают** .

```
class SuperType {
    public void sayHello(){
        System.out.println("Hello from SuperType");
    }

    public void sayBye(){
        System.out.println("Bye from SuperType");
    }
}

class SubType extends SuperType {
    // override the superclass method
    public void sayHello(){
        System.out.println("Hello from SubType");
    }
}

class Test {
    public static void main(String... args){
        SuperType superType = new SuperType();
        superType.sayHello(); // -> Hello from SuperType

        // make the reference point to an object of the subclass
        superType = new SubType();
        // behaviour is governed by the object, not by the reference
        superType.sayHello(); // -> Hello from SubType

        // non-overridden method is simply inherited
```

```
        superType.sayBye(); // -> Bye from SuperType
    }
}
```

Правила, которые следует учитывать

Чтобы переопределить метод в подклассе, метод переопределения (т. Е. Один в подклассе) **ДОЛЖЕН ИМЕТЬ** :

- то же имя
- тот же тип возврата в случае примитивов (подкласс разрешен для классов, это также называется ковариантными типами возврата).
- тот же тип и порядок параметров
- он может вызывать только те исключения, которые объявлены в предложении throws метода суперкласса или исключениях, которые являются подклассами объявленных исключений. Он также может выбрать НЕ выбрасывать какие-либо исключения. Имена типов параметров не имеют значения. Например, void methodX (int i) аналогичен void methodX (int k)
- Мы не можем переопределить конечные или статические методы. Единственное, что мы можем сделать, это изменить только тело метода.

Добавление поведения путем добавления классов без касания существующего кода

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class PolymorphismDemo {

    public static void main(String[] args) {
        List<FlyingMachine> machines = new ArrayList<FlyingMachine>();
        machines.add(new FlyingMachine());
        machines.add(new Jet());
        machines.add(new Helicopter());
        machines.add(new Jet());

        new MakeThingsFly().letTheMachinesFly(machines);
    }
}

class MakeThingsFly {
    public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class FlyingMachine {
    public void fly() {
```

```
        out.println("No implementation");
    }
}

class Jet extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start, taxi, fly");
    }

    public void bombardment() {
        out.println("Fire missile");
    }
}

class Helicopter extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start vertically, hover, fly");
    }
}
```

объяснение

a) Класс `MakeThingsFly` может работать со всем, что относится к типу `FlyingMachine` .

b) Метод `letTheMachinesFly` также работает без каких-либо изменений (!) при добавлении нового класса, например `PropellerPlane` :

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}
```

Это сила полиморфизма. Вы можете реализовать с ним открытый-закрытый принцип .

Виртуальные функции

Виртуальные методы - это методы в Java, которые нестатические и без ключевого слова Final впереди. Все методы по умолчанию являются виртуальными в Java. Виртуальные методы играют важную роль в полиморфизме, потому что классы детей в Java могут переопределять методы своих родительских классов, если переопределенная функция нестатическая и имеет одну и ту же подпись метода.

Однако существуют некоторые методы, которые не являются виртуальными. Например, если метод объявлен private или с ключевым словом final, то метод не является виртуальным.

Рассмотрим следующий измененный пример наследования с Virtual Methods из этого сообщения StackOverflow. Как виртуальные функции работают на C # и Java? :

```
public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");

    }
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Если мы вызываем класс B и вызываем hello () и boo (), мы получим «No» и «Say haha» в качестве выходного результата, потому что B переопределяет те же методы из A. Хотя приведенный выше пример почти такой же, как метод переопределения, важно понимать, что методы в классе A - это все по умолчанию Virtual.

Кроме того, мы можем реализовать виртуальные методы, используя ключевое слово abstract. Методы, объявленные с ключевым словом «abstract», не имеют определения метода, то есть тело метода еще не реализовано. Рассмотрим пример сверху, за исключением того, что метод boo () объявлен абстрактным:

```
public class A{
   public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Если мы будем вызывать boo () из B, выход будет по-прежнему «Say haha», так как B наследует абстрактный метод boo () и делает вывод boo () «Say haha».

Используемые источники и дальнейшие чтения:

Как виртуальные функции работают на C # и Java?

Ознакомьтесь с этим замечательным ответом, в котором содержится более полная информация о виртуальных функциях:

Можете ли вы писать виртуальные функции / методы на Java?

Полиморфизм и различные типы переопределения

Из учебника java

> Словарное определение полиморфизма относится к принципу в биологии, в котором организм или вид могут иметь много разных форм или стадий. Этот принцип также может быть применен к объектно-ориентированному программированию и языкам, таким как язык Java. ***Подклассы класса могут определять свое собственное уникальное поведение и совместно использовать одни и те же функциональные возможности родительского класса.***

Взгляните на этот пример, чтобы понять различные типы переопределения.

1. Базовый класс не обеспечивает реализацию, а подкласс должен переопределить полный метод - (аннотация)
2. Базовый класс обеспечивает реализацию по умолчанию, а подкласс может изменять поведение
3. `super.methodName()` добавляет расширение к реализации базового класса, вызывая `super.methodName()` как первый оператор
4. Базовый класс определяет структуру алгоритма (метод Template), а подкласс будет переопределять часть алгоритма

фрагмент кода:

```
import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean runGame = true;
    protected Player player1 = null;
    protected Player player2 = null;
    protected Player currentPlayer = null;

    public Game(){
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        currentPlayer = player1;
        initializeGame();
    }

    /* Type 1: Let subclass define own implementation. Base class defines abstract method to
force
```

```
        sub-classes to define implementation
    */

    protected abstract void initializeGame();

    /* Type 2: Sub-class can change the behaviour. If not, base class behaviour is applicable
*/
    protected void logTimeBetweenMoves(Player player){
        System.out.println("Base class: Move Duration: player.PlayerActTime -
player.MoveShownTime");
    }

    /* Type 3: Base class provides implementation. Sub-class can enhance base class
implementation by calling
        super.methodName() in first line of the child class method and specific implementation
later */
    protected void logGameStatistics(){
        System.out.println("Base class: logGameStatistics:");
    }
    /* Type 4: Template method: Structure of base class can't be changed but sub-class can
some part of behaviour */
    protected void runGame() throws Exception{
        System.out.println("Base class: Defining the flow for Game:");
        while (runGame) {
            /*
            1. Set current player
            2. Get Player Move
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }
    /* sub-part of the template method, which define child class behaviour */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }
    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }
    public void setNextPlayer(){
        if (currentPlayer == player1) {
            currentPlayer = player2;
        }else{
            currentPlayer = player1;
        }
    }
    public void run(){
        try{
            runGame();
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class Player{
```

```
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* Concrete Game implementation  */
class Chess extends Game{
    public Chess(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized Chess game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate Chess move:" + p.getName());
    }
    protected void logGameStatistics(){
        super.logGameStatistics();
        System.out.println("Child class: Add Chess specific logGameStatistics:");
    }
}
class TicTacToe extends Game{
    public TicTacToe(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized TicTacToe game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate TicTacToe move:" + p.getName());
    }
}

public class Polymorphism{
    public static void main(String args[]){
        try{

            Game game = new Chess();
            Thread t1 = new Thread(game);
            t1.start();
            Thread.sleep(1000);
            game.setRunGame(false);
            Thread.sleep(1000);

            game = new TicTacToe();
            Thread t2 = new Thread(game);
            t2.start();
            Thread.sleep(1000);
            game.setRunGame(false);

        }catch(Exception err){
            err.printStackTrace();
        }
    }
}
```

выход:

```
Child class: Initialized Chess game
Base class: Defining the flow for Game:
Child class: Validate Chess move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate Chess move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
Child class: Add Chess specific logGameStatistics:

Child class: Initialized TicTacToe game
Base class: Defining the flow for Game:
Child class: Validate TicTacToe move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate TicTacToe move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
```

Прочитайте Полиморфизм онлайн: https://riptutorial.com/ru/java/topic/980/полиморфизм

глава 141: предпочтения

Examples

Добавление прослушивателей событий

Есть два типа событий , испускаемых `Preferences` объекта: `PreferenceChangeEvent` и `NodeChangeEvent` .

PreferenceChangeEvent

`PreferenceChangeEvent` получает излучаемый объект `Properties` каждый раз, когда изменяется одна из пар ключ-значение узла. `PreferenceChangeEvent` s можно прослушать с помощью `PreferenceChangeListener` :

Java SE 8

```
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
```

Java SE 8

```
preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getNewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});
```

Этот слушатель не будет слушать измененные пары ключевых значений дочерних узлов.

NodeChangeEvent

Это событие будет запущено, когда дочерний узел узла « `Properties` будет добавлен или удален.

```
preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
```

```
    public void childRemoved(NodeChangeEvent evt) {
        Preferences removedChild = evt.getChild();
        Preferences parentOfRemovedChild = evt.getParent();
    }
});
```

Получение подустановок настроек

Объекты `Preferences` всегда представляют собой определенный узел в целом дереве `Preferences` , вроде этого:

```
/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        └── anotherapp
            ├── defaultFont=Helvetica
            ├── defaultSavePath=/home/matt/Documents
            └── exporting
                ├── defaultFormat=pdf
                └── openInBrowserAfterExport=false
```

Чтобы выбрать узел /com/mycompany/myapp :

1. По соглашению, основанному на пакете класса:

   ```
   package com.mycompany.myapp;

   // ...

   // Because this class is in the com.mycompany.myapp package, the node
   // /com/mycompany/myapp will be returned.
   Preferences myApp = Preferences.userNodeForPackage(getClass());
   ```

2. По относительной траектории:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   Использование относительного пути (путь, не начинающийся с /), приведет к тому, что путь будет разрешен относительно родительского узла, на котором он разрешен. Например, следующий пример вернет узел пути /one/two/three/com/mycompany/myapp :

   ```
   Preferences prefix = Preferences.userRoot().node("one/two/three");
   Preferences myAppWithPrefix = prefix.node("com/mycompany/myapp");
   // prefix          is /one/two/three
   // myAppWithPrefix is /one/two/three/com/mycompany/myapp
   ```

3. По абсолютной траектории:

   ```
   Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
   ```

   Использование абсолютного пути в корневом узле не будет отличаться от использования

относительного пути. Разница в том, что, если вызывается на подузле, путь будет разрешен относительно корневого узла.

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWitoutPrefix = prefix.node("/com/mycompany/myapp");
// prefix            is /one/two/three
// myAppWitoutPrefix is /com/mycompany/myapp
```

Координирование настроек доступа к нескольким экземплярам приложений

Все экземпляры Preferences всегда потокобезопасны в потоках одной виртуальной машины Java (JVM). Поскольку Preferences могут быть разделены между несколькими JVM, существуют специальные методы, связанные с синхронизацией изменений между виртуальными машинами.

Если у вас есть приложение, которое должно запускаться только в **одном экземпляре** , то **никакой внешней синхронизации** не требуется.

Если у вас есть приложение, которое выполняется в **нескольких экземплярах** в одной системе, и поэтому для доступа к JVM в системе требуется согласование Preferences , то метод **sync()** любого узла Preferences может использоваться для обеспечения того, чтобы изменения в узле Preferences видимые для других JVM в системе:

```
// Warning: don't use this if your application is intended
// to only run a single instance on a machine once
// (this is probably the case for most desktop applications)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // Deal with any errors while saving the preferences to the backing storage
    e.printStackTrace();
}
```

Экспорт предпочтений

Узлы Preferences можно экспортировать в XML-документ, представляющий этот узел. Полученное дерево XML можно снова импортировать. В результате XML-документ будет помнить, был ли он экспортирован из пользовательских или системных Preferences .

Чтобы экспортировать один узел, но **не его дочерние узлы** :

Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
```

```
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

Чтобы экспортировать один узел **с его дочерними узлами :**

Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

Импорт настроек

Узлы Preferences можно импортировать из XML-документа. Импорт предназначен для использования в сочетании с функциями экспорта Preferences , поскольку он создает соответствующие XML-документы.

Документы XML будут помнить, были ли они экспортированы из пользовательских или системных Preferences . Поэтому они могут быть импортированы в их соответствующие деревья Preferences снова, без необходимости выяснять или знать, откуда они пришли. Статическая функция автоматически узнает, был ли XML-документ экспортирован из пользовательских или системных Preferences и будет автоматически импортировать их в дерево, из которого они были экспортированы.

Java SE 7

```
try (InputStream is = ...) {
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
}
```

Java SE 7

```
InputStream is = null;
try {
    is = ...;
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}
```

Удаление прослушивателей событий

Слушатели событий могут быть удалены снова с любого узла « Properties , но для этого нужно сохранить экземпляр слушателя.

Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
```

```
    }
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

То же самое относится к NodeChangeListener .

Получение значений предпочтений

Значение узла Preferences может быть типа String , boolean , byte[] , double , float , int или long . Все вызовы должны предоставлять значение по умолчанию, если указанное значение отсутствует в узле « Preferences .

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);
```

Значения параметров настройки

Чтобы сохранить значение в узле « Preferences , используется один из методов putXXX() . Значение узла Preferences может быть типа String , boolean , byte[] , double , float , int или long .

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);
```

Использование настроек

Preferences можно использовать для хранения пользовательских настроек, которые отражают настройки личного приложения пользователя, например, их шрифт редактора, независимо от того, предпочитают ли они, чтобы приложение запускалось в полноэкранном режиме, проверяли ли они флажок «Не показывать это снова» и что как это.

```
public class ExitConfirmer {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirmer.class);
        boolean doShowDialog = preferences.getBoolean("showExitConfirmation", true); // true
is default value

        if (!doShowDialog) {
            return true;
```

```
        }

        //
        // Show a dialog here...
        //
        boolean exitWasConfirmed = ...; // whether the user clicked OK or Cancel
        boolean doNotShowAgain = ...; // get value from "Do not show again" checkbox

        if (exitWasConfirmed && doNotShowAgain) {
            // Exit was confirmed and the user chose that the dialog should not be shown again
            // Save these settings to the Preferences object so the dialog will not show again
next time
            preferences.putBoolean("showExitConfirmation", false);
        }

        return exitWasConfirmed;
    }

    public static void exit() {
        if (confirmExit()) {
            System.exit(0);
        }
    }
}
```

Прочитайте предпочтения онлайн: https://riptutorial.com/ru/java/topic/582/предпочтения

глава 142: Преобразование в строки и из них

Examples

Преобразование других типов данных в String

- Вы можете получить значение других примитивных типов данных как String, используя метод valueOf класса String.

  Например:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42”.
  ```

  Этот метод также перегружен для других типов данных, таких как float , double , boolean и even Object .

- Вы также можете получить любой другой объект (любой экземпляр любого класса) в виде строки, вызвав на него .toString . Для этого, чтобы дать полезный вывод, класс должен переопределить toString() . Большинство стандартных классов библиотеки Java, например Date и другие.

  Например:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Здесь stringifiedFoo содержит представление foo как String.

Вы также можете преобразовать любой тип номера в String с короткими обозначениями, как показано ниже.

```
int i = 10;
String str = i + "";
```

Или просто простой способ

```
String str = 10 + "";
```

Преобразование в / из байтов

Чтобы закодировать строку в массив байтов, вы можете просто использовать метод String#getBytes() , причем один из стандартных наборов символов доступен в любой среде исполнения Java:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

и для декодирования:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

вы можете еще больше упростить вызов, используя статический импорт:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

Для менее распространенных наборов символов вы можете указать набор символов со строкой:

```
byte[] bytes = "test".getBytes("UTF-8");
```

и наоборот:

```
String testString = new String (bytes, "UTF-8");
```

это означает, однако, что вы должны обрабатывать проверенное UnsupportedCharsetException .

Следующий вызов будет использовать набор символов по умолчанию. Набор символов по умолчанию является специфичным для платформы и обычно отличается между платформами Windows, Mac и Linux.

```
byte[] bytes = "test".getBytes();
```

и наоборот:

```
String testString = new String(bytes);
```

Обратите внимание, что недопустимые символы и байты могут быть заменены или пропущены этими методами. Для большего контроля - например, для проверки ввода - вам рекомендуется использовать CharsetEncoder и CharsetDecoder .

Кодирование / декодирование Base64

Иногда вы обнаружите необходимость кодирования двоичных данных в виде строки с кодом *base64* .

Для этого мы можем использовать класс DatatypeConverter из пакета javax.xml.bind :

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcnkgZGF0YQ=="

// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

В качестве альтернативы, мы можем использовать Base64 из общедоступного кодека Apache .

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);
```

```
// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

Если вы проверите эту программу, то увидите, что someBinaryData кодирует c29tZUJpbmFyeURhdGE= , очень c29tZUJpbmFyeURhdGE= объект String *-UTF-8* .

Java SE 8

Детали для этого же можно найти на Base64

```
// encode with padding
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// encode without padding
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// decode a String
byte [] barr = Base64.getDecoder().decode(encoded);
```

Ссылка

Разбор строк на числовое значение

Строка к примитивному числовому типу или числовому типу обертки:

Каждый числовой класс-оболочка предоставляет метод parseXxx который преобразует String в соответствующий примитивный тип. Следующий код преобразует String в int с Integer.parseInt метода Integer.parseInt :

```
String string = "59";
int primitive = Integer.parseInteger(string);
```

Чтобы преобразовать в String экземпляр класса числовой оболочки, вы можете либо использовать перегрузку классов- valueOf :

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

или полагаться на автоматический бокс (Java 5 и более поздние версии):

```
String string = "59";
Integer wrapper = Integer.parseInteger(string);  // 'int' result is autoboxed
```

Данная схема работает на byte , short , int , long , float и double и соответствующие классы - оболочек (Byte , Short , Integer , Long , Float и Double).

String to Integer с использованием radix:

```
String integerAsString = "0101"; // binary representation
int parseInt = Integer.parseInt(integerAsString,2);
Integer valueOfInteger = Integer.valueOf(integerAsString,2);
System.out.println(valueOfInteger); // prints 5
System.out.println(parseInt); // prints 5
```

Исключения

Исключенное исключение NumberFormatException будет выведено , если parseXxx(...) метод parseXxx(...) valueOf(String) или parseXxx(...) для строки, которая не является допустимым числовым представлением или представляет значение, выходящее за пределы допустимого диапазона.

Получение `String` из` InputStream`

String может быть прочитана из InputStream с использованием конструктора байтового массива.

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // supply the length of the string in bytes here
    input.read(bytes);
    return new String(bytes);
}
```

Это использует кодировку по умолчанию системы, хотя может быть задана альтернативная кодировка:

```
return new String(bytes, Charset.forName("UTF-8"));
```

Преобразование String в другие типы данных.

Вы можете преобразовать **числовую** строку в различные числовые типы Java следующим образом:

Строка для int:

```
String number = "12";
int num = Integer.parseInt(number);
```

Строка для плавания:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

Строка в два раза:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

Строка: boolean:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

Строка длинная:

```
String number = "47";
long num = Long.parseLong(number);
```

Строка для BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

Строка в BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Исключения:

Числовые преобразования выше будут NumberFormatException (непроверенное) NumberFormatException если вы попытаетесь проанализировать строку, которая не является соответствующим образом отформатированным номером, или вне диапазона для целевого типа. В разделе « Исключения» обсуждается, как бороться с такими исключениями.

Если вы хотите проверить, что вы можете разбирать строку, вы можете реализовать метод tryParse... следующим образом:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

Однако вызов этого метода tryParse... непосредственно перед разбором (возможно) является плохим. Было бы лучше просто вызвать метод parse... и обработать исключение.

Прочитайте Преобразование в строки и из них онлайн: https://riptutorial.com/ru/java/topic/6678/преобразование-в-строки-и-из-них

глава 143: Преобразование типа

Синтаксис

- TargetType target = (SourceType) источник;

Examples

Нецифровое примитивное литье

boolean тип не может быть отброшен в / из любого другого примитивного типа.

char может быть преобразован в / из любого числового типа, используя сопоставления кодовых точек, заданные Unicode. char представлен в памяти как неподписанное 16-битовое целочисленное значение (2 байта), поэтому отбрасывание до byte (1 байт) приведет к удалению 8 из этих битов (это безопасно для символов ASCII). Методы утилиты класса Character используют int (4 байта) для передачи в / из значений кодовой точки, но для хранения кодовой точки Unicode также достаточно short (2 байта).

```
int badInt   = (int)  true; // Compiler error: incompatible types

char char1   = (char)   65; // A
byte byte1   = (byte)  'A'; // 65
short short1 = (short) 'A'; // 65
int int1     = (int)   'A'; // 65

char char2   = (char) 8253; // ?
byte byte2   = (byte)  '?'; // 61 (truncated code-point into the ASCII range)
short short2 = (short) '?'; // 8253
int int2     = (int)   '?'; // 8253
```

Численное примитивное литье

Числовые примитивы можно отличить двумя способами. *Неявное* литье происходит, когда тип источника имеет меньший диапазон, чем целевой тип.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

Явное литье должно выполняться, когда тип источника имеет больший диапазон, чем целевой тип.

```
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

При отливке примитивов с плавающей запятой (float , double) и целых чисел число **округляется** .

Литье объектов

Как и в случае с примитивами, объекты могут быть указаны как явно, так и неявно.

Неявное кастинг происходит, когда тип источника расширяет или реализует целевой тип (отбрасывание на суперкласс или интерфейс).

Явное литье должно выполняться, когда тип источника расширяется или реализуется целевым типом (приведение к подтипу). Это может привести к исключению среды выполнения (ClassCastException), когда объект, который выполняется, не относится к типу цели (или подтипу цели).

```
Float floatVar = new Float(42.0f);
Number n = floatVar;                //Implicit (Float implements Number)
Float floatVar2 = (Float) n;        //Explicit
Double doubleVar = (Double) n;      //Throws exception (the object is not Double)
```

Основное числовое продвижение

```
 static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;      // Error: Cannot convert from int to char;
    // short1 = short1 + short2;   // Error: Cannot convert from int to short;
    int1 = char1 + char2;          // char is promoted to int.
    int1 = short1 + short2;        // short is promoted to int.
    int1 = char1 + short2;         // both char and short promoted to int.
    float1 = short1 + float2;      // short is promoted to float.
    int1 = int1 + int2;            // int is unchanged.
}
```

Тестирование, если объект может быть запущен с использованием instanceof

Java предоставляет оператору instanceof проверку, является ли объект определенным типом или подклассом этого типа. Затем программа может выбрать, чтобы бросить или не отбросить этот объект соответственно.

```
Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.
}
```

Прочитайте Преобразование типа онлайн: https://riptutorial.com/ru/java/topic/1392/преобразование-типа

глава 144: Примитивные типы данных

Вступление

8 примитивных типов данных byte , short , int , long , char , boolean , float и double - это типы, которые хранят большинство исходных числовых данных в Java-программах.

Синтаксис

- int aInt = 8; // Определяющая (числовая) часть этого объявления int называется литералом.
- int hexInt = 0x1a; // = 26; Вы можете определить литералы с шестнадцатеричными значениями с префиксом *0x* .
- int binInt = 0b11010; // = 26; Вы также можете определить бинарные литералы; с префиксом *0b* .
- long goodLong = 10000000000L; // По умолчанию целочисленные литералы имеют тип int. Добавив L в конец литерала, вы сообщаете компилятору, что литерал длинный. Без этого компилятор будет вызывать ошибку «Целое число слишком большое».
- double aDouble = 3,14; // Литералы с плавающей запятой по умолчанию имеют тип double.
- float aFloat = 3.14F; // По умолчанию этот литерал был бы двойным (и вызвал ошибку «Несовместимые типы»), но, добавив F, мы скажем компилятору, что это float.

замечания

Java имеет 8 *примитивных типов данных* , а именно: boolean , byte , short , char , int , long , float и double . (Все остальные типы являются *ссылочными* типами, включая все типы массивов и встроенные типы объектов / классы, которые имеют особое значение на языке Java, например String , Class и Throwable и его подклассы).

Результат всех операций (сложение, вычитание, умножение и т. Д.) В примитивном типе - это, по меньшей мере, int , поэтому добавление short в short вызывает int , как и добавление byte в byte , или char для char , Если вы хотите присвоить результат этого значения значению того же типа, вы должны его бросить. например

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

Неисполнение операции приведет к ошибке компиляции.

Это связано со следующей частью Java Language Spec, §2.11.1 :

> Компилятор кодирует нагрузки буквенных значений byte типов и short с помощью инструкций Java Virtual Machine, которые подписывают эти значения до значений типа int во время компиляции или времени выполнения. Загрузка значений букв типов boolean и char кодируется с использованием инструкций, которые нулевым образом расширяют литерал до значения типа int во время компиляции или времени выполнения. [..]. Таким образом, большинство операций над значениями фактических типов boolean , byte , char и short корректно выполняются инструкциями, действующими на значения вычислительного типа int .

Причина этого также указана в этом разделе:

> Учитывая **однобайтовый размер кода операции** Java Virtual Machine, типы кодирования в opcodes оказывают давление на дизайн своего набора команд. Если каждая типизированная команда поддерживает все типы данных во время выполнения Java Virtual Machine, будет больше инструкций, чем может быть представлено в byte . [...] Отдельные инструкции

могут использоваться для преобразования между неподдерживаемыми и поддерживаемыми типами данных по мере необходимости.

Examples

Int примитив

Примитивный тип данных, такой как int содержит значения непосредственно в переменной, которая его использует, между тем переменная, которая была объявлена с использованием Integer содержит ссылку на значение.

Согласно java API : «Класс Integer обертывает значение примитивного типа int в объекте. Объект типа Integer содержит одно поле, тип которого является int».

По умолчанию int представляет собой 32-разрядное целое число со знаком. Он может хранить минимальное значение -2^{31} и максимальное значение $2^{31} - 1$.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

Если вам нужно сохранить номер за пределами этого диапазона, следует использовать его в качестве long . Превышение диапазона значений int приводит к переполнению целых чисел, в результате чего значение, превышающее диапазон, будет добавлено к противоположному сайту диапазона (положительное становится отрицательным и наоборот). Значение равно ((value - MIN_VALUE) % RANGE) + MIN_VALUE или ((value + 2147483648) % 4294967296) - 2147483648

```
int demo = 2147483647; //maximum positive integer
System.out.println(demo); //prints 2147483647
demo = demo + 1; //leads to an integer overflow
System.out.println(demo); // prints -2147483648
```

Максимальное и минимальное значения int можно найти по адресу:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

Значение по умолчанию для int равно *0*

```
int defaultInt;    // defaultInt == 0
```

Короткий примитив

short является 16-разрядное целое число со знаком. Он имеет минимальное значение -2^{15} (-32,768), а максимальное значение составляет 2^{15} -1 (32 767)

```
short example = -48;
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

Максимальные и минимальные значения short можно найти по адресу:

```
short high = Short.MAX_VALUE;        // high == 32767
short low = Short.MIN_VALUE;         // low == -32768
```

Значение по умолчанию short равно *0*

```
short defaultShort;    // defaultShort == 0
```

Длинный примитив

По умолчанию long - это 64-разрядное целое число со знаком (в Java 8 оно может быть либо подписано, либо без знака). Подписано, оно может хранить минимальное значение -2^{63} и максимальное значение $2^{63} - 1$, а без знака оно может хранить минимальное значение 0 и максимальное значение $2^{64} - 1$

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 (2 ^ 39) is larger than the maximum value of an int (2^31 - 1),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

Максимальное и минимальное значения long можно найти по адресу:

```
long high = Long.MAX_VALUE;    // high == 9223372036854775807L
long low = Long.MIN_VALUE;     // low == -9223372036854775808L
```

Значение по умолчанию long равно *0L*

```
long defaultLong;    // defaultLong == 0L
```

Примечание: буква «L», добавленная в конце long литерала, нечувствительна к регистру, однако хорошей практикой является использование капитала, поскольку ее легче отличить от цифры один:

```
2L == 2l;            // true
```

Предупреждение: Java кэширует объекты Integer объектов из диапазона от -128 до 127. Объяснение здесь объясняется: https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

Следующие результаты можно найти:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;

System.out.println(val3 == val4); // false
```

Чтобы правильно сравнить значения 2 объекта Long, используйте следующий код (начиная с Java 1.7):

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

Сравнение примитива long long с Object long не приведет к ложному отрицанию, сравнив 2 объекта с.

Булевский примитив

boolean может сохранять одно из двух значений: true или false

```
boolean foo = true;
System.out.println("foo = " + foo);                // foo = true

boolean bar = false;
System.out.println("bar = " + bar);                // bar = false

boolean notFoo = !foo;
System.out.println("notFoo = " + notFoo);          // notFoo = false

boolean fooAndBar = foo && bar;
System.out.println("fooAndBar = " + fooAndBar);    // fooAndBar = false

boolean fooOrBar = foo || bar;
System.out.println("fooOrBar = " + fooOrBar);      // fooOrBar = true

boolean fooXorBar = foo ^ bar;
System.out.println("fooXorBar = " + fooXorBar);    // fooXorBar = true
```

Значение по умолчанию для boolean равно *false*

```
boolean defaultBoolean;    // defaultBoolean == false
```

Байт-примитив

byte представляет собой 8-разрядное целое число со знаком. Он может хранить минимальное значение -2^7 (-128), а максимальное значение $2^7 - 1$ (127)

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

Максимальное и минимальное значения byte можно найти по адресу:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

Значение по умолчанию byte равно *0*

```
byte defaultByte;    // defaultByte == 0
```

Поплавковый примитив

float - это 32-битное число с плавающей запятой IEEE 754 с одной точностью. По умолчанию десятичные знаки интерпретируются как двойные. Чтобы создать float , просто добавьте f в десятичный литерал.

```
double doubleExample = 0.5;      // without 'f' after digits = double
float floatExample = 0.5f;       // with 'f' after digits    = float

float myFloat = 92.7f;           // this is a float...
float positiveFloat = 89.3f;     // it can be positive,
float negativeFloat = -89.3f;    // or negative
float integerFloat = 43.0f;      // it can be a whole number (not an int)
float underZeroFloat = 0.0549f;  // it can be a fractional value less than 0
```

Поплавки обрабатывают пять общих арифметических операций: сложение, вычитание, умножение, деление и модуль.

Примечание. В результате ошибок с плавающей запятой могут незначительно отличаться. Некоторые результаты были округлены для ясности и удобочитаемости (т. Е. Напечатанный результат примера добавления был фактически 34.600002).

```
// addition
float result = 37.2f + -2.6f;  // result: 34.6

// subtraction
float result = 45.1f - 10.3f;    // result: 34.8

// multiplication
float result = 26.3f * 1.7f;   // result: 44.71

// division
float result = 37.1f / 4.8f;   // result: 7.729166

// modulus
float result = 37.1f % 4.8f;   // result: 3.4999971
```

Из-за того, что хранятся числа с плавающей точкой (т. Е. В двоичной форме), многие числа не имеют точного представления.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

Хотя использование float подходит для большинства приложений, для хранения точных представлений десятичных чисел (например, денежных сумм) или чисел, где требуется более высокая точность, не следует использовать float или double . Вместо этого следует использовать класс BigDecimal .

Значение по умолчанию для float равно *0.0f* .

```
float defaultFloat;    // defaultFloat == 0.0f
```

float точностью до ошибки составляет 1 из 10 миллионов.

Примечание: Float.POSITIVE_INFINITY , Float.NEGATIVE_INFINITY , Float.NaN - значения с float . NaN означает результаты операций, которые невозможно определить, например, деление 2 бесконечных значений. Кроме того, 0f и -0f различны, но == дает true:

```
float f1 = 0f;
float f2 = -0f;
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
```

```
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

Двойной примитив

double - это 64-битное число с плавающей запятой IEEE 754 с двойной точностью.

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3;    // 0.0012
```

Из-за того, что хранятся числа с плавающей запятой, многие номера не имеют точного представления.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

Хотя для большинства приложений используется double для хранения точных чисел, таких как валюта, не следует использовать ни float ни double . Вместо этого следует использовать класс BigDecimal

Значение по умолчанию double равно *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Примечание: Double.POSITIVE_INFINITY , Double.NEGATIVE_INFINITY , Double.NaN - double значения. NaN означает результаты операций, которые невозможно определить, например, деление 2 бесконечных значений. Кроме того, 0d и -0d различны, но == -0d true:

```
double d1 = 0d;
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

Первоначальный символ

char может хранить один 16-разрядный символ Юникода. Символьный литерал заключен в одинарные кавычки

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

Он имеет минимальное значение \u0000 (0 в десятичном представлении, также называемое *нулевым символом*) и максимальное значение \uffff (65,535).

Значением по умолчанию char является \u0000 .

```
char defaultChar;    // defaultChar == \u0000
```

Чтобы определить значение символа «char ' использовать escape-последовательность (символ, которому предшествует обратная косая черта):

```
char singleQuote = '\'';
```

Существуют также другие escape-последовательности:

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';
char singleQuote = '\'';
char doubleQuote = '\"'; // escaping redundant here; '"' would be the same; however still
allowed
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX represents the Unicode-value of the character you want to
display
```

Вы можете объявить char любого символа Юникода.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

Также можно добавить char . например, для повторения каждой строчной буквы, вы можете сделать следующее:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Представление отрицательного значения

Java и большинство других языков хранят отрицательные целые числа в представлении , обозначаемом *дополнением 2* .

Для уникального двоичного представления типа данных с использованием n бит значения кодируются следующим образом:

Наименее значимые n−1 биты сохраняют положительное целое число x в интегральном представлении. Наиболее значимое значение хранит бит с значением s . Значение, представленное этими битами, равно

$x - s * 2^{n-1}$

т.е. если самый старший бит равен 1, то значение, которое просто на 1 больше, чем число, которое вы могли бы представить с другими битами ($2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0 = 2^{n-1} - 1$), что дает уникальное двоичное представление для каждого значения от -2^{n-1} (s = 1; x = 0) до $2^{n-1} - 1$ (s = 0; x = $2^{n-1} - 1$).

Это также имеет хороший побочный эффект, что вы можете добавить двоичные представления, как если бы они были положительными двоичными числами:

$$v1 = x1 - s1 * 2^{n-1}$$
$$v2 = x2 - s2 * 2^{n-1}$$

s1	s2	переполнение x1 + x2	результат добавления
0	0	нет	x1 + x2 = v1 + v2

s1	s2	переполнение x1 + x2	результат добавления
0	0	да	слишком большой, чтобы быть представленным типом данных (переполнение)
0	1	нет	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1} = v1 + v2$
0	1	да	$(x1 + x2) \bmod 2^{n-1} = x1 + x2 - 2^{n-1} = v1 + v2$
1	0	*	см. выше (сменные слагаемые)
1	1	нет	слишком мал, чтобы быть представленным типом данных ($x1 + x2 - 2^{n} < -2^{n-1}$, underflow)
1	1	да	$(x1 + x2) \bmod 2^{n-1} - 2^{n-1} = (x1 + x2 - 2^{n-1}) - 2^{n-1} = (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1}) = v1 + v2$

Обратите внимание, что этот факт упрощает поиск двоичного представления аддитивного обратного (т. Е. Отрицательного значения):

Обратите внимание, что добавление побитового дополнения к числу приводит к тому, что все биты равны 1. Теперь добавьте 1, чтобы сделать переполнение значения, и вы получите нейтральный элемент 0 (все биты 0).

Таким образом, отрицательное значение числа i можно вычислить, используя (игнорируя возможную рекламу для int здесь)

```
(~i) + 1
```

Пример: принятие отрицательного значения 0 (byte):

Результатом отрицания 0 является 11111111 . Добавление 1 дает значение 100000000 (9 бит). Поскольку byte может хранить только 8 бит, самое левое значение усекается, а результат равен 00000000

оригинал	Процесс	Результат
0 (00000000)	сводить на нет	-0 (11111111)
11111111	Добавить 1 в двоичный	100000000
100000000	Усекать до 8 бит	00000000 (-0 равно 0)

Потребление памяти примитивов против бокс-примитивов

Примитивный	В штучной упаковке	Размер памяти примитива / в штучной упаковке
логический	логический	1 байт / 16 байт
байт	Байт	1 байт / 16 байт
короткая	короткий	2 байта / 16 байт
голец	голец	2 байта / 16 байт
ИНТ	целое число	4 байта / 16 байт
долго	Долго	8 байт / 16 байт
поплавок	терка	4 байта / 16 байт
двойной	двойной	8 байт / 16 байт

Объектам в штучной упаковке всегда требуется 8 байтов для управления типом и памятью, а так как размер объектов всегда кратен 8, для *всех* типов в штучной упаковке *требуется всего 16 байтов* . Кроме *того* , каждое использование объекта в штучной упаковке влечет за собой хранение ссылки, которая учитывает еще 4 или 8 байтов, в зависимости от параметров JVM и JVM.

В операциях с интенсивным использованием данных потребление памяти может существенно повлиять на производительность. Потребление памяти растет еще больше при использовании массивов: для массива float[5] требуется только 32 байта; тогда как Float[5] хранящий 5 различных ненулевых значений, потребует всего 112 байтов (на 64-разрядном без сжатых указателей, это увеличивается до 152 байт).

Вложенные кеширование значений

Косвенные накладные расходы в штучной упаковке могут быть в некоторой степени смягчены кэшами в штучной упаковке. Некоторые из типов в штучной упаковке реализуют кеш экземпляров. Например, по умолчанию класс Integer будет кэшировать экземпляры для представления чисел в диапазоне от -128 до +127 . Однако это не снижает дополнительные затраты, связанные с дополнительной памятью.

Если вы создаете экземпляр типа boxed либо с помощью autoboxing, либо путем вызова статического метода valueOf(primitive) , система времени выполнения попытается использовать кешированное значение. Если ваше приложение использует множество значений в кэше, то это может существенно снизить штраф за использование ящиков в коробке. Конечно, если вы создаете экземпляры экземпляров в штучной упаковке «вручную», лучше использовать valueOf вместо new . (new операция всегда создает новый экземпляр.) Если, однако, большинство ваших значений *не* находятся в кешированном диапазоне, быстрее можно вызвать new и сохранить поиск кеша.

Преобразование примитивов

В Java мы можем конвертировать между целыми значениями и значениями с плавающей запятой. Кроме того, поскольку каждый символ соответствует числу в кодировке Unicode, типы char могут быть преобразованы в типы и с целыми числами и с плавающей точкой. boolean - единственный примитивный тип данных, который не может быть преобразован в какой-либо другой примитивный тип данных.

Существует два типа конверсий: *расширение конверсии* и *сужение конверсии* .

Расширяющееся преобразование - это когда значение одного типа данных преобразуется в значение другого типа данных, который занимает больше битов, чем первый. В этом случае нет проблемы потери данных.

Соответственно, *сужение преобразования* - это когда значение одного типа данных преобразуется в значение другого типа данных, который занимает меньше битов, чем первый. В этом случае может произойти потеря данных.

Java автоматически *расширяет конверсии* . Но если вы хотите выполнить *сужение конверсии* (если вы уверены, что потеря данных не произойдет), вы можете заставить Java выполнить преобразование с использованием языковой конструкции, известной как cast .

Расширение конверсии:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Сужение конверсии:

```
double d = 18.96
int b = d;       // invalid conversion to int, will throw a compile-time error
int b = (int) d; // valid conversion to int, but result is truncated (gets rounded down)
                 // This is type-casting
                 // Now, b = 18
```

Примитивные типы

Таблица, показывающая диапазон размеров и значений всех примитивных типов:

тип данных	числовое представление	диапазон значений	значение по умолчанию
логический	н /	ложный и истинный	ложный
байт	8-разрядная подписка	-2^7 до $2^7 - 1$	0
		От -128 до +127	
короткая	16-разрядная подписка	-2^{15} - $2^{15} - 1$	0
		-32,768 до +32,767	
ИНТ	32-разрядная подписка	-2^{31} до $2^{31} - 1$	0
		-2,147,483,648 до +2,147,483,647	
долго	64-битная подпись	-2^{63} до $2^{63} - 1$	0L
		-9,223,372,036,854,775,808 - 9,223,372,036,854,775,807	
поплавок	32-битная с	1.401298464e-45 до 3.402823466e +	0.0f

тип данных	числовое представление	диапазон значений	значение по умолчанию
	плавающей запятой	38 (положительный или отрицательный)	
двойной	64-битная плавающая точка	4.94065645841246544e-324d до 1.79769313486231570e + 308d (положительный или отрицательный)	0.0d
голец	16-разрядный беззнаковый	0 до $2^{16} - 1$	0
		0 до 65535	

Заметки:

1. Спецификация языка Java предусматривает, что подписанные интегральные типы (byte через long) используют двоичное представление двухкомпонента, а типы с плавающей точкой используют стандартные двоичные представления с плавающей точкой IEE 754.
2. Java 8 и более поздние версии предоставляют методы для выполнения беззнаковых арифметических операций по int и long . Хотя эти методы позволяют программе *обрабатывать* значения соответствующих типов как неподписанные, типы остаются подписанными типами.
3. Наименьшая плавающая точка, показанная выше, является *субнормальной* ; т.е. они имеют меньшую точность, чем *нормальное* значение. Наименьшие нормальные числа: 1.175494351e-38 и 2.2250738585072014e-308
4. char обычно представляет собой *блок кода* Unicode / UTF-16.
5. Хотя boolean содержит только один бит информации, его размер в памяти варьируется в зависимости от реализации виртуальной машины Java (см. Boolean type).

Прочитайте Примитивные типы данных онлайн: https://riptutorial.com/ru/java/topic/148/примитивные-типы-данных

глава 145: Процесс

замечания

Обратите внимание, что API рекомендует, чтобы с версии 1.5 предпочтительный способ создания процесса - использовать ProcessBuilder.start() .

Другое важное замечание состоит в том, что значение выхода, созданное waitFor , зависит от выполняемой программы / скрипта. Например, коды выхода, созданные **calc.exe** , отличаются от **notepad.exe** .

Examples

Простой пример (версия Java <1.5)

Этот пример вызовет калькулятор окон. Важно заметить, что код выхода будет меняться в зависимости от вызываемой программы / скрипта.

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // Executes windows calculator
            Process p = Runtime.getRuntime().exec("calc.exe");

            // Wait for process until it terminates
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Использование класса ProcessBuilder

Класс ProcessBuilder упрощает отправку команды через командную строку. Все, что требуется, это список строк, которые составляют команды для ввода. Вы просто вызываете метод start () в экземпляре ProcessBuilder для выполнения команды.

Если у вас есть программа под названием Add.exe, которая принимает два аргумента и добавляет их, код выглядит примерно так:

```
List<String> cmds = new ArrayList<>();
cmds.add("Add.exe"); //the name of the application to be run
cmds.add("1"); //the first argument
cmds.add("5"); //the second argument

ProcessBuilder pb = new ProcessBuilder(cmds);

//Set the working directory of the ProcessBuilder so it can find the .exe
```

```
//Alternatively you can just pass in the absolute file path of the .exe
File myWorkingDirectory = new File(yourFilePathNameGoesHere);
pb.workingDirectory(myWorkingDirectory);

try {
    Process p = pb.start();
} catch (IOException e) {
    e.printStackTrace();
}
```

Некоторые вещи нужно иметь в виду:

- Массив команд должен быть массивом String
- Команды должны быть в порядке (в массиве), в котором они были бы, если бы вы совершили вызов программы в самой командной строке (то есть имя .exe не может идти после первого аргумента
- При настройке рабочего каталога вам необходимо передать объект File, а не только имя файла в виде строки

Блокировка против неблокирующих вызовов

В общем случае при вызове командной строки программа отправит команду и продолжит ее выполнение.

Однако вы можете дождаться завершения вызова вызываемой программы, прежде чем продолжить собственное выполнение (например, вызываемая программа будет записывать данные в файл, и вашей программе необходимо, чтобы для доступа к этим данным).

Это можно легко сделать, вызвав метод waitFor() из возвращаемого экземпляра Process .

Пример использования:

```
//code setting up the commands omitted for brevity...

ProcessBuilder pb = new ProcessBuilder(cmds);

try {
    Process p = pb.start();
    p.waitFor();
} catch (IOException e) {
    e.printStackTrack();
} catch (InterruptedException e) {
    e.printStackTrace();
}

//more lines of code here...
```

ch.vorburger.exec

Запуск внешних процессов из Java с использованием исходного API java.lang.ProcessBuilder напрямую может быть немного громоздким. Библиотека Apache Commons Exec упрощает работу. Библиотека ch.vorburger.exec далее распространяется на Commons Exec, чтобы сделать ее по-настоящему удобной:

```
 ManagedProcess proc = new ManagedProcessBuilder("path-to-your-executable-binary")
     .addArgument("arg1")
     .addArgument("arg2")
     .setWorkingDirectory(new File("/tmp"))
     .setDestroyOnShutdown(true)
     .setConsoleBufferMaxLines(7000)
     .build();
```

```
proc.start();
int status = proc.waitForExit();
int status = proc.waitForExitMaxMsOrDestroy(3000);
String output = proc.getConsole();

proc.startAndWaitForConsoleMessageMaxMs("started!", 7000);
// use service offered by external process...
proc.destroy();
```

Pitfall: Runtime.exec, Process и ProcessBuilder не понимают синтаксис оболочки

Runtime.exec(String ...) и Runtime.exec(String) позволяют выполнять команду как внешний процесс [1] . В первой версии вы указываете имя команды и аргументы команды как отдельные элементы массива строк, а среда выполнения Java запрашивает систему выполнения ОС для запуска внешней команды. Вторая версия обманчиво проста в использовании, но у нее есть некоторые подводные камни.

Прежде всего, это пример использования безопасного использования exec(String) :

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Пробелы в дорожках

Предположим, что мы обобщаем приведенный выше пример, чтобы мы могли создать произвольный каталог:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

Обычно это будет работать, но это не удастся, если dirPath (например) «/ home / user / My Documents». Проблема в том, что exec(String) разбивает строку на команду и аргументы, просто просматривая пробелы. Командная строка:

```
"mkdir /home/user/My Documents"
```

будут разделены на:

```
"mkdir", "/home/user/My", "Documents"
```

и это приведет к сбою команды mkdir, поскольку она ожидает один аргумент, а не два.

Столкнувшись с этим, некоторые программисты пытаются добавить кавычки вокруг имени пути. Это тоже не работает:

```
"mkdir \"/home/user/My Documents\""
```

будут разделены на:

```
"mkdir", "\"/home/user/My", "Documents\""
```

Дополнительные символы двойной кавычки, которые были добавлены в попытке «процитировать» пробелы, рассматриваются как любые другие символы без пробелов. В самом деле, все, что мы цитируем или избегаем пробелов, будет терпеть неудачу.

Способом решения этих конкретных проблем является использование перегрузки exec(String ...) .

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Это будет работать, если dirpath содержит пробельные символы, потому что эта перегрузка exec не пытается разделить аргументы. Строки передаются в системный вызов OS exec как есть.

Перенаправление, конвейеры и другой синтаксис оболочки

Предположим, что мы хотим перенаправить вход или выход внешней команды или запустить конвейер. Например:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

или же

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(В первом примере перечислены имена всех файлов Java в файловой системе, а второй печатает операторы package [2] в файлах Java в «исходном» дереве.)

Они не будут работать должным образом. В первом случае команда «find» будет запущена с «2> / dev / null» в качестве аргумента команды. Это не будет интерпретироваться как перенаправление. Во втором примере символ канала («|») и последующие за ним работы будут переданы команде «find».

Проблема здесь в том, что методы exec и ProcessBuilder не понимают никакого синтаксиса оболочки. Это включает в себя перенаправления, трубопроводы, расширение переменных, подтягивание и т. Д.

В нескольких случаях (например, простое перенаправление) вы можете легко достичь желаемого эффекта с помощью ProcessBuilder . Однако в целом это не так. Альтернативный подход - запустить командную строку в оболочке; например:

```
Process p = Runtime.exec("bash", "-c",
                         "find / -name *.java -print 2>/dev/null");
```

или же

```
Process p = Runtime.exec("bash", "-c",
                         "find source -name \\*.java | xargs grep package");
```

Но обратите внимание, что во втором примере нам нужно было избежать символа подстановки («*»), потому что мы хотим, чтобы шаблон был интерпретирован «find», а не оболочкой.

Команды встроенных команд не работают

Предположим, что следующие примеры не будут работать в системе с UNIX-подобной оболочкой:

```
Process p = Runtime.exec("cd", "/tmp");     // Change java app's home directory
```

или же

```
Process p = Runtime.exec("export", "NAME=value");  // Export NAME to the java app's
environment
```

Есть несколько причин, почему это не сработает:

1. Команды «cd» и «export» - это команды, встроенные в оболочку. Они не существуют как отдельные исполняемые файлы.

2. Для встроенных оболочек оболочки делать то, что они должны делать (например, изменить рабочий каталог, обновить среду), им необходимо изменить место, где это состояние находится. Для обычного приложения (включая приложение Java) состояние связано с процессом приложения. Так, например, дочерний процесс, который запускал команду «cd», не мог изменить рабочий каталог его родительского «java» процесса. Аналогично, один процесс exec не может изменить рабочий каталог для процесса, который следует за ним.

Это рассуждение применимо ко всем командам встроенной оболочки.

---

1 - Вы также можете использовать ProcessBuilder , но это не относится к точке этого примера.

2 - Это немного грубо и готово ... но еще раз, недостатки этого подхода не имеют отношения к примеру.

Прочитайте Процесс онлайн: https://riptutorial.com/ru/java/topic/4682/процесс

глава 146: Путь Класса

Вступление

Путь к классам перечисляет места, где среда выполнения Java должна искать классы и ресурсы. Путь классов также используется компилятором Java для поиска ранее скомпилированных и внешних зависимостей.

замечания

Загрузка класса Java

JVM (виртуальная машина Java) будет загружать классы как и когда требуются классы (это называется ленивой загрузкой). Местоположения классов, которые будут использоваться, указаны в трех местах:

1. Сначала загружаются те, которые требуются платформой Java, например, в библиотеке классов Java и ее зависимостях.
2. Далее загружаются классы расширения (т. jre/lib/ext/ Те, что указаны в jre/lib/ext/)
3. Затем загружаются пользовательские классы через путь к классам

Классы загружаются с использованием классов, которые являются подтипами java.lang.ClassLoader . Это описано более подробно в этой теме: Classloaders .

Classpath

Путь к классам – это параметр, используемый JVM или компилятором, который определяет расположение пользовательских классов и пакетов. Это можно установить в командной строке, как в большинстве этих примеров, или через переменную окружения (CLASSPATH)

Examples

Различные способы указания пути к классам

Существует три способа установки пути к классам.

1. Его можно установить с помощью переменной среды CLASSPATH :

```
set CLASSPATH=...          # Windows and csh
export CLASSPATH=...       # Unix ksh/bash
```

2. Его можно установить в командной строке следующим образом

```
java -classpath ...
javac -classpath ...
```

Обратите внимание, что параметр -classpath (или -cp) имеет приоритет над переменной среды CLASSPATH .

3. Путь классов для исполняемого файла JAR указывается с помощью элемента Class-Path в MANIFEST.MF :

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

Обратите внимание, что это применяется только тогда, когда файл JAR выполняется следующим образом:

```
java -jar some.jar ...
```

В этом режиме выполнения параметр -classpath и переменная среды CLASSPATH будут игнорироваться, даже если в JAR-файле нет элемента Class-Path .

Если classpath не указан, то java -jar по умолчанию - это выбранный JAR-файл при использовании java -jar или текущего каталога в противном случае.

Связанные с:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Добавление всех JAR в каталог к пути к классам

Если вы хотите добавить все JAR-файлы в каталог в путь к классам, вы можете сделать это в сжатом виде с помощью синтаксиса подстановочных знаков класса; например:

```
someFolder/*
```

Это сообщает JVM добавлять все JAR и ZIP-файлы в каталог someFolder в someFolder к классам. Этот синтаксис может быть использован в -cp аргумента, в CLASSPATH переменной окружения, или Class-Path атрибут в явном file.See виде исполняемого JAR - файла *Настройка пути* к *классам: Путь класса Wild Cards* для примеров и предостережений.

Заметки:

1. Классовые маски классов были впервые введены в Java 6. Ранние версии Java не рассматривают «*» как подстановочный знак.
2. Вы не можете вводить другие символы до или после « *»; например, «someFolder / .jar» не является подстановочным знаком.
3. Подстановочный знак соответствует только файлам с суффиксом «.jar» или «.JAR». Файлы ZIP игнорируются, как и файлы JAR с разными суффиксами.
4. Подстановочный знак соответствует только JAR-файлам в самом каталоге, а не в его подкаталогах.
5. Когда группа файлов JAR сопоставляется с подстановочной записью, их относительный порядок в пути к классам не указан.

Синтаксис пути к классу

Путь к классам - это последовательность записей, которые являются именами путей каталогов, JAR или файловыми файлами ZIP или спецификациями подстановки JAR / ZIP.

- Для пути -classpath указанного в командной строке (например, -classpath) или в качестве переменной среды, записи должны быть разделены ; (точки с запятой) в Windows, или : (двоеточие) символов на других платформах (Linux, UNIX, MacOSX и т. д.).

- Для элемента Class-Path в файле MANIFEST.MF файла JAR используйте одно пространство для разделения записей.

Иногда необходимо вставить пробел в элемент pathpath

- Когда путь класса указан в командной строке, это просто вопрос использования соответствующего цитирования оболочки. Например:

  ```
  export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
  ```

  (Детали могут зависеть от используемой командной оболочки.)

- Когда путь класса указан в файле JAR файла «MANIFEST.MF», необходимо использовать кодировку

URL.

```
Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
```

Динамический путь класса

Иногда просто добавить все JAR из папки недостаточно, например, когда у вас есть собственный код и вам нужно выбрать подмножество JAR. В этом случае вам понадобятся два метода main() . Первый построит загрузчик классов, а затем использует этот загрузчик классов для вызова второго main() .

Вот пример, который выбирает правильный JAR для SWT для вашей платформы, добавляет все JAR-приложения вашего приложения, а затем вызывает реальный метод main() : создание кросс-платформенного приложения Java SWT

Загрузка ресурса из пути к классам

Может быть полезно загрузить ресурс (изображение, текстовый файл, свойства, KeyStore, ...), который упакован внутри JAR. Для этой цели мы можем использовать Class и ClassLoader s.

Предположим, что мы имеем следующую структуру проекта:

```
program.jar
|
\-com
  \-project
    |
    |-file.txt
    \-Test.class
```

И мы хотим получить доступ к содержимому file.txt из класса Test . Мы можем сделать это, запросив загрузчик классов:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

Используя загрузчик классов, мы должны указать полный путь нашего ресурса (каждый пакет).

Или, альтернативно, мы можем напрямую спросить объект класса Test

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Используя объект класса, путь относится к самому классу. Наш Test.class находящийся в пакете com.project , так же, как file.txt , нам не нужно указывать какой-либо путь вообще.

Однако мы можем использовать абсолютные пути от объекта класса, например:

```
 is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Сопоставление имен классов с именами путей

Стандартная инструментальная цепочка Java (и сторонние инструменты, предназначенные для взаимодействия с ними) имеют определенные правила для сопоставления имен классов с именами файлов и других ресурсов, которые их представляют.

Отображения таковы:

- Для классов в пакете по умолчанию пути являются простыми именами файлов.
- Для классов в именованном пакете компоненты имени пакета отображаются в каталогах.
- Для названных вложенных и внутренних классов компонент имени файла формируется путем

объединения имен классов с символом $.
- Для анонимных внутренних классов вместо имен используются числа.

Это проиллюстрировано в следующей таблице:

Classname	Исходный путь	Путь к файлу класса
SomeClass	SomeClass.java	SomeClass.class
com.example.SomeClass	com/example/SomeClass.java	com/example/SomeClass.class
SomeClass.Inner	(в SomeClass.java)	SomeClass$Inner.class
SomeClass anon внутренние классы	(в SomeClass.java)	SomeClass$1.class , SomeClass$2.class и т. Д.

Что означает classpath: как работают поисковые запросы

Цель classpath – указать JVM, где можно найти классы и другие ресурсы. Значение classpath и процесс поиска переплетаются.

Путь к классам – это форма пути поиска, которая определяет последовательность *местоположений* для поиска ресурсов. В стандартном classpath эти места представляют собой либо каталог в файловой системе хоста, файл JAR, либо ZIP-файл. В каждом случае местоположение является корнем *пространства имен,* которое будет искать.

Стандартная процедура поиска класса в пути к классам выглядит следующим образом:

1. Сопоставьте имя класса с относительным именем класса RP . Отображение имен классов для имен файлов классов описано в другом месте.
2. Для каждой записи E в пути к классам:
    - Если запись является файловой системой:
        - Решите RP относительно E чтобы дать абсолютный путь AP .
        - Проверьте, является ли AP для существующего файла.
        - Если да, загрузите класс из этого файла
    - Если запись является JAR или ZIP-файлом:
        - Поиск RP в индексе файла JAR / ZIP.
        - Если соответствующая запись JAR / ZIP-файла существует, загрузите класс из этой записи.

Процедура поиска ресурса в пути к классам зависит от того, является ли путь ресурса абсолютным или относительным. Для абсолютного пути ресурса процедура выполняется так же, как указано выше. Для пути относительного ресурса, разрешенного с использованием Class.getResource или Class.getResourceAsStream , путь к пакету классов добавляется перед поиском.

(Обратите внимание, что это процедуры, выполняемые стандартными загрузчиками классов Java. Пользовательский загрузчик классов может выполнять поиск по-разному.)

Путь к бутстрапу

Обычные загрузчики классов Java ищут классы сначала в пути класса bootstrap, прежде чем проверять расширения и путь класса приложения. По умолчанию путь bootstrap classpath состоит из файла «rt.jar» и некоторых других важных файлов JAR, которые поставляются установкой JRE. Они обеспечивают все классы стандартной стандартной библиотеки классов Java SE, а также различные «внутренние» классы реализации.

При нормальных обстоятельствах вам не нужно беспокоиться об этом. По умолчанию команды java ,

javac и т. Д. Будут использовать соответствующие версии библиотек времени исполнения.

Очень редко необходимо переопределить нормальное поведение среды выполнения Java, используя альтернативную версию класса в стандартных библиотеках. Например, вы можете столкнуться с ошибкой «show stopper» в библиотеках времени выполнения, с которыми вы не можете работать обычными способами. В такой ситуации можно создать JAR-файл, содержащий измененный класс, а затем добавить его в путь класса bootstrap, который запускает JVM.

Команда java предоставляет следующие опции -X для изменения пути класса bootstrap:

- -Xbootclasspath:<path> заменяет текущий путь к пути загрузки указанным путем.
- -Xbootclasspath/a:<path> добавляет предоставленный путь к текущему пути к загрузке.
- -Xbootclasspath/p:<path> добавляет предоставленный путь к текущему пути к загрузке.

Обратите внимание, что при использовании параметров bootclasspath для замены или переопределения класса Java (etcetera) вы технически модифицируете Java. При распространении кода *могут возникнуть* последствия для лицензирования. (Обратитесь к положениям и условиям бинарной лицензии Java ... и проконсультируйтесь с адвокатом.)

Прочитайте Путь Класса онлайн: https://riptutorial.com/ru/java/topic/3720/путь-класса

Синтаксис

- `javap [опции] <classes>`

параметры

название	Описание
`<classes>`	Список классов для демонтажа. Может быть в формате `package1.package2.Classname` , *или в* формате `package1/package2/Classname` . **Не** включайте `.class` расширение.
`-help` , `--help` , `-?`	Распечатайте это сообщение об использовании
`-version`	Информация о версии
`-v` , `-verbose`	Распечатать дополнительную информацию
`-l`	Номер строки печати и таблицы локальных переменных
`-public`	Показать только публичные классы и участники
`-protected`	Показать защищенные / общедоступные классы и участники
`-package`	Показать пакеты / защищенные / общедоступные классы и участники (по умолчанию)
`-p` , `-private`	Показать все классы и участники
`-c`	Разберите код
`-s`	Печать внутренних подписей типа
`-sysinfo`	Показывать системную информацию (путь, размер, дата, хеш MD5) обрабатываемого класса
`-constants`	Показать конечные константы
`-classpath <path>`	Укажите, где найти файлы классов пользователя
`-cp <path>`	Укажите, где найти файлы классов пользователя
`-bootclasspath <path>`	Переопределить расположение файлов классов начальной загрузки

Examples

Просмотр байт-кода с помощью javap

Если вы хотите увидеть сгенерированный байт-код для Java-программы, вы можете использовать предоставленную команду javap для ее просмотра.

Предполагая, что у нас есть следующий исходный файл Java:

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Service;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

@Service
public class HelloWorldService {

    public void sayHello() {
        System.out.println("Hello, World!");
    }

    private Object[] pvtMethod(List<String> strings) {
        return new Object[]{strings};
    }

    protected String tryCatchResources(String filename) throws IOException {
        try (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            byte[] bytes = new byte[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
            e.printStackTrace();
            throw e;
        }
    }

    void stuff() {
        System.out.println("stuff");
    }
}
```

После компиляции исходного файла наиболее простое использование:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Что дает результат

```
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
  public com.stackoverflow.documentation.HelloWorldService();
  public void sayHello();
  protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
  void stuff();
}
```

В этом перечислены все не частные методы в классе, но это не особенно полезно для большинства

целей. Следующая команда намного полезнее:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Что дает результат:

```
Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
  Last modified Jul 22, 2016; size 2167 bytes
  MD5 checksum 6e33b5c292ead21701906353b7f06330
  Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
    #1 = Methodref          #5.#60         // java/lang/Object."<init>":()V
    #2 = Fieldref           #61.#62        // java/lang/System.out:Ljava/io/PrintStream;
    #3 = String             #63            // Hello, World!
    #4 = Methodref          #64.#65        // java/io/PrintStream.println:(Ljava/lang/String;)V
    #5 = Class              #66            // java/lang/Object
    #6 = Methodref          #5.#67         // java/lang/Object.getClass:()Ljava/lang/Class;
    #7 = Methodref          #68.#69        //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
    #8 = Methodref          #70.#71        // java/io/InputStream.read:([B)I
    #9 = Class              #72            // java/lang/String
   #10 = Methodref          #9.#73         // java/lang/String."<init>":([BII)V
   #11 = Methodref          #70.#74        // java/io/InputStream.close:()V
   #12 = Class              #75            // java/lang/Throwable
   #13 = Methodref          #12.#76        //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
   #14 = Class              #77            // java/io/IOException
   #15 = Class              #78            // java/lang/RuntimeException
   #16 = Methodref          #79.#80        // java/lang/Exception.printStackTrace:()V
   #17 = String             #55            // stuff
   #18 = Class              #81            // com/stackoverflow/documentation/HelloWorldService
   #19 = Utf8               <init>
   #20 = Utf8               ()V
   #21 = Utf8               Code
   #22 = Utf8               LineNumberTable
   #23 = Utf8               LocalVariableTable
   #24 = Utf8               this
   #25 = Utf8               Lcom/stackoverflow/documentation/HelloWorldService;
   #26 = Utf8               sayHello
   #27 = Utf8               pvtMethod
   #28 = Utf8               (Ljava/util/List;)[Ljava/lang/Object;
   #29 = Utf8               strings
   #30 = Utf8               Ljava/util/List;
   #31 = Utf8               LocalVariableTypeTable
   #32 = Utf8               Ljava/util/List<Ljava/lang/String;>;
   #33 = Utf8               Signature
   #34 = Utf8               (Ljava/util/List<Ljava/lang/String;>;)[Ljava/lang/Object;
   #35 = Utf8               tryCatchResources
   #36 = Utf8               (Ljava/lang/String;)Ljava/lang/String;
   #37 = Utf8               bytes
   #38 = Utf8               [B
   #39 = Utf8               read
   #40 = Utf8               I
   #41 = Utf8               inputStream
   #42 = Utf8               Ljava/io/InputStream;
   #43 = Utf8               e
```

```
   #44 = Utf8               Ljava/lang/Exception;
   #45 = Utf8               filename
   #46 = Utf8               Ljava/lang/String;
   #47 = Utf8               StackMapTable
   #48 = Class              #81            // com/stackoverflow/documentation/HelloWorldService
   #49 = Class              #72            // java/lang/String
   #50 = Class              #82            // java/io/InputStream
   #51 = Class              #75            // java/lang/Throwable
   #52 = Class              #38            // "[B"
   #53 = Class              #83            // java/lang/Exception
   #54 = Utf8               Exceptions
   #55 = Utf8               stuff
   #56 = Utf8               SourceFile
   #57 = Utf8               HelloWorldService.java
   #58 = Utf8               RuntimeVisibleAnnotations
   #59 = Utf8               Lorg/springframework/stereotype/Service;
   #60 = NameAndType        #19:#20        // "<init>":()V
   #61 = Class              #84            // java/lang/System
   #62 = NameAndType        #85:#86        // out:Ljava/io/PrintStream;
   #63 = Utf8               Hello, World!
   #64 = Class              #87            // java/io/PrintStream
   #65 = NameAndType        #88:#89        // println:(Ljava/lang/String;)V
   #66 = Utf8               java/lang/Object
   #67 = NameAndType        #90:#91        // getClass:()Ljava/lang/Class;
   #68 = Class              #92            // java/lang/Class
   #69 = NameAndType        #93:#94        //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
   #70 = Class              #82            // java/io/InputStream
   #71 = NameAndType        #39:#95        // read:([B)I
   #72 = Utf8               java/lang/String
   #73 = NameAndType        #19:#96        // "<init>":([BII)V
   #74 = NameAndType        #97:#20        // close:()V
   #75 = Utf8               java/lang/Throwable
   #76 = NameAndType        #98:#99        // addSuppressed:(Ljava/lang/Throwable;)V
   #77 = Utf8               java/io/IOException
   #78 = Utf8               java/lang/RuntimeException
   #79 = Class              #83            // java/lang/Exception
   #80 = NameAndType        #100:#20       // printStackTrace:()V
   #81 = Utf8               com/stackoverflow/documentation/HelloWorldService
   #82 = Utf8               java/io/InputStream
   #83 = Utf8               java/lang/Exception
   #84 = Utf8               java/lang/System
   #85 = Utf8               out
   #86 = Utf8               Ljava/io/PrintStream;
   #87 = Utf8               java/io/PrintStream
   #88 = Utf8               println
   #89 = Utf8               (Ljava/lang/String;)V
   #90 = Utf8               getClass
   #91 = Utf8               ()Ljava/lang/Class;
   #92 = Utf8               java/lang/Class
   #93 = Utf8               getResourceAsStream
   #94 = Utf8               (Ljava/lang/String;)Ljava/io/InputStream;
   #95 = Utf8               ([B)I
   #96 = Utf8               ([BII)V
   #97 = Utf8               close
   #98 = Utf8               addSuppressed
   #99 = Utf8               (Ljava/lang/Throwable;)V
  #100 = Utf8               printStackTrace
{
  public com.stackoverflow.documentation.HelloWorldService();
    descriptor: ()V
```

```
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 10: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

  public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String Hello, World!
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 13: 0
        line 14: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

  private java.lang.Object[] pvtMethod(java.util.List<java.lang.String>);
    descriptor: (Ljava/util/List;)[Ljava/lang/Object;
    flags: ACC_PRIVATE
    Code:
      stack=4, locals=2, args_size=2
         0: iconst_1
         1: anewarray     #5                  // class java/lang/Object
         4: dup
         5: iconst_0
         6: aload_1
         7: aastore
         8: areturn
      LineNumberTable:
        line 17: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
            0       9     1 strings   Ljava/util/List;
      LocalVariableTypeTable:
        Start  Length  Slot  Name   Signature
            0       9     1 strings   Ljava/util/List<Ljava/lang/String;>;
    Signature: #34                          //
(Ljava/util/List<Ljava/lang/String;>;)[Ljava/lang/Object;

  protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
    descriptor: (Ljava/lang/String;)Ljava/lang/String;
    flags: ACC_PROTECTED
    Code:
      stack=5, locals=10, args_size=2
         0: aload_0
         1: invokevirtual #6                  // Method
```

```
java/lang/Object.getClass:()Ljava/lang/Class;
         4: aload_1
         5: invokevirtual #7                  // Method
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
         8: astore_2
         9: aconst_null
        10: astore_3
        11: sipush        8192
        14: newarray       byte
        16: astore        4
        18: aload_2
        19: aload         4
        21: invokevirtual #8                  // Method java/io/InputStream.read:([B)I
        24: istore        5
        26: new           #9                  // class java/lang/String
        29: dup
        30: aload         4
        32: iconst_0
        33: iload         5
        35: invokespecial #10                 // Method java/lang/String."<init>":([BII)V
        38: astore        6
        40: aload_2
        41: ifnull        70
        44: aload_3
        45: ifnull        66
        48: aload_2
        49: invokevirtual #11                 // Method java/io/InputStream.close:()V
        52: goto          70
        55: astore        7
        57: aload_3
        58: aload         7
        60: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
        63: goto          70
        66: aload_2
        67: invokevirtual #11                 // Method java/io/InputStream.close:()V
        70: aload         6
        72: areturn
        73: astore        4
        75: aload         4
        77: astore_3
        78: aload         4
        80: athrow
        81: astore        8
        83: aload_2
        84: ifnull        113
        87: aload_3
        88: ifnull        109
        91: aload_2
        92: invokevirtual #11                 // Method java/io/InputStream.close:()V
        95: goto          113
        98: astore        9
       100: aload_3
       101: aload         9
       103: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
       106: goto          113
       109: aload_2
       110: invokevirtual #11                 // Method java/io/InputStream.close:()V
       113: aload         8
       115: athrow
```

```
       116: astore_2
       117: aload_2
       118: invokevirtual #16                 // Method
java/lang/Exception.printStackTrace:()V
       121: aload_2
       122: athrow
      Exception table:
         from    to  target type
            48    52    55   Class java/lang/Throwable
            11    40    73   Class java/lang/Throwable
            11    40    81   any
            91    95    98   Class java/lang/Throwable
            73    83    81   any
             0    70   116   Class java/io/IOException
             0    70   116   Class java/lang/RuntimeException
            73   116   116   Class java/io/IOException
            73   116   116   Class java/lang/RuntimeException
      LineNumberTable:
        line 21: 0
        line 22: 11
        line 23: 18
        line 24: 26
        line 25: 40
        line 21: 73
        line 25: 81
        line 26: 117
        line 27: 121
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
           18      55     4 bytes   [B
           26      47     5  read   I
            9     107     2 inputStream   Ljava/io/InputStream;
          117       6     2     e   Ljava/lang/Exception;
            0     123     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
            0     123     1 filename   Ljava/lang/String;
      StackMapTable: number_of_entries = 9
        frame_type = 255 /* full_frame */
          offset_delta = 55
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, class "[B", int, class
java/lang/String ]
          stack = [ class java/lang/Throwable ]
        frame_type = 10 /* same */
        frame_type = 3 /* same */
        frame_type = 255 /* full_frame */
          offset_delta = 2
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable ]
          stack = [ class java/lang/Throwable ]
        frame_type = 71 /* same_locals_1_stack_item */
          stack = [ class java/lang/Throwable ]
        frame_type = 255 /* full_frame */
          offset_delta = 16
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, top, top, top, top,
class java/lang/Throwable ]
          stack = [ class java/lang/Throwable ]
        frame_type = 10 /* same */
        frame_type = 3 /* same */
        frame_type = 255 /* full_frame */
          offset_delta = 2
```

```
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String ]
          stack = [ class java/lang/Exception ]
    Exceptions:
      throws java.io.IOException

  void stuff();
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #17                 // String stuff
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 32: 0
        line 33: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
}
SourceFile: "HelloWorldService.java"
RuntimeVisibleAnnotations:
  0: #59()
```

Прочитайте Разборка и декомпиляция онлайн: https://riptutorial.com/ru/java/topic/2318/разборка-и-декомпиляция

глава 148: Развертывание Java

Вступление

Существует множество технологий для «упаковки» Java-приложений, веб-приложений и т. Д. Для развертывания на платформе, на которой они будут работать. Они варьируются от простой библиотеки или исполняемых файлов JAR , WAR и EAR до инсталляторов и автономных исполняемых файлов.

замечания

На самом фундаментальном уровне программа Java может быть развернута путем копирования скомпилированного класса (например, файла «.class») или дерева каталогов, содержащего скомпилированные классы. Однако Java обычно развертывается одним из следующих способов:

- Копируя JAR-файл или коллекцию JAR-файлов в систему, в которой они будут выполнены; например, с помощью javac .
- Копирование или загрузка WAR, EAR или аналогичного файла в «контейнер сервлетов» или «сервер приложений».
- Запустив какой-то установщик приложений, который автоматизирует вышеуказанное. Установщик также может установить встроенную JRE.
- Поместив файлы JAR для приложения на веб-сервер, чтобы они могли быть запущены с использованием Java WebStart.

Пример создания JAR, WAR и EAR файлов суммирует различные способы создания этих файлов.

Существует множество открытых и коммерческих генераторов «генератор установки» и «генератор EXE» для Java. Аналогичным образом, существуют инструменты для обфускации файлов классов Java (чтобы сделать сложную реверсивную инженерию) и для добавления проверки лицензии во время выполнения. Все они недоступны для документации «Язык программирования Java».

Examples

Создание исполняемого JAR из командной строки

Чтобы создать банку, вам нужен один или несколько файлов классов. Это должно иметь основной метод, если он должен выполняться двойным щелчком.

В этом примере мы будем использовать:

```
import javax.swing.*;
import java.awt.Container;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

Он был назван HelloWorld.java

Затем мы хотим скомпилировать эту программу.

Вы можете использовать любую программу, которую хотите сделать. Для запуска из командной строки см. Документацию по компиляции и запуску вашей первой java-программы.

Когда у вас есть HelloWorld.class, создайте новую папку и вызовите ее, как хотите.

Сделайте еще один файл с именем manifest.txt и вставьте в него

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Поместите его в ту же папку с HelloWorld.class
Используйте командную строку для создания вашей текущей cd C:\Your\Folder\Path\Here (cd C:\Your\Folder\Path\Here on windows).

Используйте Terminal и смените каталог в каталог (cd /Users/user/Documents/Java/jarfolder на Mac) в вашей папке

Когда это будет сделано, введите jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class и нажмите enter. Это делает файл jar (в папке с вашим манифестом и HelloWorld.class), используя указанные .class файлы и называемые HelloWorld.jar. См. Раздел «Синтаксис» для получения информации об опциях (например, -m и -v).
После этих шагов перейдите в свой каталог с файлом манифеста, и вы должны найти HelloWorld.jar Нажатие на него должно отображать Hello, World в текстовом поле.

Создание файлов JAR, WAR и EAR

Типы файлов JAR, WAR и EAR являются в основном ZIP-файлами с «манифестным» файлом и (для WAR и EAR-файлов) конкретной внутренней структурой каталога / файла.

Рекомендуемым способом создания этих файлов является использование специального инструмента Java, который «понимает» требования к соответствующим типам файлов. Если вы не используете инструмент построения, тогда IDE «экспорт» - это следующий вариант, который нужно попробовать.

(*Редакционное примечание: описания того, как создавать эти файлы, лучше всего размещать в документации для соответствующих инструментов. Поместите их там. Пожалуйста, покажите некоторые сдержанности и НЕ ПОЛУЧИТЕ их в этом примере!*)

Создание файлов JAR и WAR с использованием Maven

Создание JAR или WAR с использованием Maven - это просто вопрос ввода правильного элемента <packaging> в файл POM; например,

```
<packaging>jar</packaging>
```

или же

```
<packaging>war</packaging>
```

Больше подробностей. Maven может быть настроен на создание «исполняемых» JAR-файлов путем добавления необходимой информации о классе входа и внешних зависимостях в качестве свойств плагина для плагина maven jar. Существует даже плагин для создания файлов «uberJAR», которые объединяют приложение и его зависимости в один JAR-файл.

Дополнительную информацию см. В документации Maven (http://www.riptutorial.com/topic/898).

Создание файлов JAR, WAR и EAR с использованием Ant

Инструмент сборки Ant имеет отдельные «задачи» для построения JAR, WAR и EAR. Для получения дополнительной информации обратитесь к документации Ant (http://www.riptutorial.com/topic/4223

).

Создание файлов JAR, WAR и EAR с использованием среды IDE

Три наиболее популярных Java IDE имеют встроенную поддержку для создания файлов развертывания. Функциональность часто описывается как «экспорт».

- Eclipse - http://www.riptutorial.com/topic/1143
- NetBeans - http://www.riptutorial.com/topic/5438
- Intellij-IDEA - экспорт

Создание файлов JAR, WAR и EAR с использованием команды `jar` .

Также возможно создать эти файлы «вручную», используя команду jar . Это просто вопрос сборки дерева файлов с правильными файлами компонентов в нужном месте, создания файла манифеста и запуска jar для создания файла JAR.

Пожалуйста, обратитесь к команде jar Topic (Создание и изменение файлов JAR) для получения дополнительной информации

Введение в Java Web Start

Учебники Oracle Java Tutorials обобщают Web Start следующим образом:

> Программное обеспечение Java Web Start обеспечивает возможность запуска полнофункциональных приложений одним щелчком мыши. Пользователи могут загружать и запускать приложения, такие как полная электронная таблица или клиент интернет-чата, без длительных процедур установки.

Другими преимуществами Java Web Start являются поддержка подписанного кода и явное объявление зависимостей платформы, а также поддержка кэширования кода и развертывания обновлений приложений.

Java Web Start также называется JavaWS и JAWS. Основными источниками информации являются:

- Учебники Java - Урок: Java Web Start
- Руководство по началу работы с Java Web
- Часто задаваемые вопросы по Java Web Start
- Спецификация JNLP
- javax.jnlp API javax.jnlp
- Сайт разработчика Java Web Start

Предпосылки

На высоком уровне Web Start работает, распространяя приложения Java, упакованные как файлы JAR с удаленного веб-сервера. Предпосылки:

- Предварительно существующая установка Java (JRE или JDK) на целевой машине, на которой должно выполняться приложение. Требуется Java 1.2.2 или выше:
    - Начиная с версии Java 5.0 поддержка Web Start включена в JRE / JDK.
    - Для более ранних выпусков поддержка Web Start устанавливается отдельно.
    - Инфраструктура Web Start включает в себя некоторые Javascript, которые могут быть включены в веб-страницу, чтобы помочь пользователю установить необходимое программное обеспечение.
- Веб-сервер, на котором размещается программное обеспечение, должен быть доступен для целевой машины.
- Если пользователь запустит приложение Web Start, используя ссылку на веб-странице, то:
    - им нужен совместимый веб-браузер, и

- для современных (безопасных) браузеров им нужно сказать, как сообщить браузеру о возможности запуска Java ... без ущерба для безопасности веб-браузера.

Пример файла JNLP

Следующий пример предназначен для иллюстрации основных функций JNLP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
    href="demo_webstart.jnlp">
    <information>
        <title>Demo</title>
        <vendor>The Example.com Team</vendor>
    </information>
    <resources>
        <!-- Application Resources -->
        <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
        <jar href="Demo.jar" main="true"/>
    </resources>
    <application-desc
         name="Demo Application"
         main-class="com.example.jwsdemo.Main"
         width="300"
         height="300">
     </application-desc>
     <update check="background"/>
</jnlp>
```

Как вы можете видеть, файл JNLP на основе XML, и вся информация содержится в <jnlp> .

- Атрибут spec предоставляет версию спецификации JNPL, которой соответствует этот файл.
- Атрибут codebase дает базовый URL для разрешения относительных URL-адресов href в остальной части файла.
- Атрибут href дает окончательный URL для этого файла JNLP.
- Элемент <information> содержит метаданные приложения, включая его название, авторы, описание и справочный сайт.
- Элемент <resources> описывает зависимости для приложения, включая требуемую версию Java, платформу ОС и файлы JAR.
- Элемент <application-desc> (или <applet-desc>) предоставляет информацию, необходимую для запуска приложения.

Настройка веб-сервера

Веб-сервер должен быть настроен на использование application/x-java-jnlp-file как application/x-java-jnlp-file MIMEtype для .jnlp .

Файл JNLP и файлы JAR приложения должны быть установлены на веб-сервере, чтобы они были доступны с использованием URL-адресов, указанных в файле JNLP.

Включение запуска через веб-страницу

Если приложение должно быть запущено через веб-ссылку, страница, содержащая ссылку, должна быть создана на веб-сервере.

- Если вы можете предположить, что Java Web Start уже установлен на компьютере пользователя, веб-страница просто должна содержать ссылку для запуска приложения. Например.

  ```
  <a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
  ```

- В противном случае на странице также должны быть указаны некоторые сценарии, чтобы определить тип браузера, который пользователь использует, и запросить загрузку и установку

требуемой версии Java.

ПРИМЕЧАНИЕ. Неплохая идея побудить пользователей поощрять установку Java таким образом или даже включить Java в своих веб-браузерах, чтобы запуск веб-страницы JNLP работал.

Запуск приложений Web Start из командной строки

Инструкции для запуска приложения Web Start из командной строки просты. Предполагая, что у пользователя установлена Java 5.0 JRE или JDK, просто нужно запустить это:

```
$ javaws <url>
```

где <url> - это URL-адрес для файла JNLP на удаленном сервере.

Создание UberJAR для приложения и его зависимостей

Общее требование для приложения Java - это то, что можно развернуть, копируя один файл. Для простых приложений, которые зависят только от стандартных библиотек классов Java SE, это требование выполняется путем создания JAR-файла, содержащего все (скомпилированные) классы приложений.

Все это не так просто, если приложение зависит от сторонних библиотек. Если вы просто поместите JAR-файлы зависимостей в JAR приложения, стандартный загрузчик классов Java не сможет найти классы библиотеки, и ваше приложение не запустится. Вместо этого вам нужно создать один JAR-файл, который содержит классы приложений и связанные ресурсы вместе с классами и ресурсами зависимостей. Они должны быть организованы как единое пространство имен для поиска загрузчика классов.

Такой JAR-файл часто упоминается как UberJAR.

Создание UberJAR с помощью команды «jar»

Процедура создания UberJAR прямолинейна. (Я буду использовать команды Linux для простоты. Команды должны быть идентичными для Mac OS и аналогичны для Windows.)

1. Создайте временный каталог и смените каталог на него.

   ```
   $ mkdir tempDir
   $ cd tempDir
   ```

2. Для каждого зависимого JAR-файла *в обратном порядке*, который должен появиться в пути к классам приложения, используется команда jar для распаковки JAR во временный каталог.

   ```
   $ jar -xf <path/to/file.jar>
   ```

   Выполнение этого для нескольких файлов JAR будет *перекрывать* содержимое JAR.

3. Скопируйте классы приложений из дерева сборки во временный каталог

   ```
   $ cp -r path/to/classes .
   ```

4. Создайте UberJAR из содержимого временного каталога:

   ```
   $ jar -cf ../myApplication.jar
   ```

   Если вы создаете исполняемый JAR-файл, включите соответствующий MANIFEST.MF, как описано здесь.

Создание UberJAR с использованием Maven

Если ваш проект построен с использованием Maven, вы можете создать его для создания UberJAR с использованием плагинов «maven-assembly» или «maven-shade». Подробнее см. Тему Ассамблеи Maven (в документации Maven).

Преимущества и недостатки UberJARs

Некоторые из преимуществ UberJARs очевидны:

- UberJAR легко распространять.
- Вы не можете разбить библиотечные зависимости для UberJAR, поскольку библиотеки являются автономными.

Кроме того, если вы используете соответствующий инструмент для создания UberJAR, у вас будет возможность исключить классы библиотек, которые не используются из файла JAR. Однако это обычно делается путем статического анализа классов. Если ваше приложение использует рефлексию, обработку аннотации и аналогичные методы, вы должны быть осторожны, чтобы классы не были исключены неверно.

UberJARs также имеют некоторые недостатки:

- Если у вас много UberJAR с одинаковыми зависимостями, то каждый из них будет содержать копии зависимостей.
- Некоторые библиотеки с открытым исходным кодом имеют лицензии , которые *могут* препятствовать [1] их использование в качестве UberJAR.

---

1 - Некоторые библиотеки библиотек с открытым исходным кодом позволяют использовать библиотеку только конечного пользователя, которая может заменить одну версию библиотеки на другую. UberJAR могут затруднить замену зависимостей версий.

Прочитайте Развертывание Java онлайн: https://riptutorial.com/ru/java/topic/6840/развертывание-java

глава 149: Разделение строки на части с фиксированной длиной

замечания

Цель здесь состоит в том, чтобы не потерять контент, поэтому регулярное выражение не должно потреблять (сопоставлять) любой вход. Скорее, он должен совпадать *между* последним символом предыдущего целевого ввода и первым символом следующего целевого ввода. например, для 8-символьных подстрок, нам нужно сломать вход вверх (т.е. совпадение) в местах, отмеченных ниже:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
              ^               ^               ^
```

Игнорируйте пробелы на входе, которые должны отображаться *между* позициями символов.

Examples

Разбить цепочку на подстроки всей известной длины

Хитрость заключается в использовании внешнего вида с регулярным выражением \G , что означает «конец предыдущего совпадения»:

```
String[] parts = str.split("(?<=\\G.{8})");
```

Регулярное выражение соответствует 8 символам после окончания последнего совпадения. Так как в этом случае совпадение имеет нулевую ширину, мы могли бы просто сказать «8 символов после последнего совпадения».

Удобно, \G инициализируется для начала ввода, поэтому он также работает для первой части ввода.

Разбить строку на подстроки всей переменной длины

То же, что и пример известной длины, но вставляем длину в регулярное выражение:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Прочитайте Разделение строки на части с фиксированной длиной онлайн:
https://riptutorial.com/ru/java/topic/5613/разделение-строки-на-части-с-фиксированной-длиной

глава 150: Реализации Java-плагинов

замечания

Если вы используете систему IDE и / или сборки, гораздо проще настроить такой проект. Вы создаете основной модуль приложения, затем модуль API, затем создаете модуль плагина и делаете его зависимым от модуля API или обоих. Затем вы настраиваете, где должны быть помещены артефакты проекта - в нашем случае скомпилированные плагины могут быть отправлены прямо в каталог « plugins», что позволяет избежать ручного перемещения.

Examples

Использование URLClassLoader

Существует несколько способов реализации плагиновой системы для Java-приложения. Одним из самых простых является использование *URLClassLoader* . В следующем примере будет задействован бит кода JavaFX.

Предположим, что у нас есть модуль основного приложения. Предполагается, что этот модуль загружает плагины в форме Jars из папки «plugins». Исходный код:

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //arbitrary directory
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //a container to show the visual info later
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Затем мы создаем интерфейс, который будет представлять собой плагин.

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

Мы хотим загрузить классы, реализующие этот интерфейс, поэтому сначала нам нужно отфильтровать файлы с расширением «.jar»:

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

Если есть какие-либо файлы, нам необходимо создать коллекции URL-адресов и имен классов:

```
    if(files!=null && files.length>0)
    {
        ArrayList<String> classes=new ArrayList<>();
        ArrayList<URL> urls=new ArrayList<>(files.length);
        for(File file:files)
        {
            JarFile jar=new JarFile(file);
            jar.stream().forEach(jarEntry -> {
                if(jarEntry.getName().endsWith(".class"))
                {
                    classes.add(jarEntry.getName());
                }
            });
            URL url=file.toURI().toURL();
            urls.add(url);
        }

    }
```

Давайте добавим статический HashSet к *MainApplication*, который будет содержать загруженные плагины:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Затем мы создаем экземпляр *URLClassLoader* и *повторяем* имена классов, *создавая* экземпляры классов, которые реализуют интерфейс *Plugin* :

```
URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
//transforming to binary name
        Class[] interfaces=cls.getInterfaces();
        for(Class intface:interfaces)
        {
            if(intface.equals(Plugin.class)) //checking presence of Plugin interface
            {
                Plugin plugin=(Plugin) cls.newInstance(); //instantiating the Plugin
                plugins.add(plugin);
                break;
            }
        }
    }
    catch (Exception e){e.printStackTrace();}
});
```

Затем мы можем вызвать методы плагина, например, для их инициализации:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

Окончательный код *MainApplication* :

```
package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
        VBox loadedPlugins=new VBox(6);
        loadedPlugins.setAlignment(Pos.CENTER);
        if(files!=null && files.length>0)
        {
            ArrayList<String> classes=new ArrayList<>();
            ArrayList<URL> urls=new ArrayList<>(files.length);
            for(File file:files)
            {
                JarFile jar=new JarFile(file);
                jar.stream().forEach(jarEntry -> {
                    if(jarEntry.getName().endsWith(".class"))
                    {
                        classes.add(jarEntry.getName());
                    }
                });
                URL url=file.toURI().toURL();
                urls.add(url);
            }
            URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new
URL[urls.size()]));
            classes.forEach(className->{
                try
                {
                    Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
                    Class[] interfaces=cls.getInterfaces();
                    for(Class intface:interfaces)
                    {
                        if(intface.equals(Plugin.class))
                        {
                            Plugin plugin=(Plugin) cls.newInstance();
                            plugins.add(plugin);
                            break;
                        }
                    }
                }
                catch (Exception e){e.printStackTrace();}
            });
            if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded
plugins:"));
            plugins.forEach(plugin -> {
                plugin.initialize();
                loadedPlugins.getChildren().add(new Label(plugin.name()));
            });
        }
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
```

```
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Давайте создадим два плагина. Очевидно, что источник плагина должен быть в отдельном модуле.

```
package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Второй плагин:

```
package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //overrided to show user's home directory
    {
        System.out.println("User home directory: "+System.getProperty("user.home"));
    }
}
```

Эти плагины должны быть упакованы в стандартные Jars - этот процесс зависит от вашей среды разработки или других инструментов.

Когда Jars будут помещены в «плагины» напрямую, *MainApplication* обнаружит их и создаст соответствующие классы.

Прочитайте Реализации Java-плагинов онлайн: https://riptutorial.com/ru/java/topic/7160/реализации-java-плагинов

глава 151: Регулярные выражения

Вступление

Регулярное выражение представляет собой специальную последовательность символов, которая помогает в сопоставлении или поиске других строк или наборов строк с использованием специализированного синтаксиса, содержащегося в шаблоне. Java поддерживает регулярное использование выражений через пакет java.util.regex . Эта тема должна представить и помочь разработчикам понять больше с примерами того, как регулярные выражения должны использоваться в Java.

Синтаксис

- Pattern patternName = Pattern.compile (regex);
- Matcher matcherName = patternName.matcher (textToSearch);
- matcherName.matches () // Возвращает true, если textToSearch точно соответствует регулярному выражению
- matcherName.find () // Ищет через textToSearch для первого экземпляра подстроки, соответствующей регулярному выражению. Последующие вызовы будут искать оставшуюся часть строки.
- matcherName.group (groupNum) // Возвращает подстроку внутри группы захвата
- matcherName.group (groupName) // Возвращает подстроку внутри указанной группы захвата (Java 7+)

замечания

импорт

Вам нужно будет добавить следующий импорт, прежде чем вы сможете использовать Regex:

```
import java.util.regex.Matcher
import java.util.regex.Pattern
```

Ловушки

В java обратная косая черта сбрасывается с двойной обратной косой чертой, поэтому обратная косая черта в строке регулярного выражения должна вводиться как двойной обратный слэш. Если вам нужно избежать двойной обратной косой черты (чтобы соответствовать одной обратной косой чертой с регулярным выражением, вам нужно ввести ее в виде четырехкратной обратной косой черты.

Обозначенные важные символы

символ	Описание
*	Совпадение предыдущего символа или подвыражения 0 или более раз
+	Совпадение предыдущего символа или подвыражения 1 или более раз
?	Соответствует предыдущему символу или подвыражению 0 или 1 раз

дальнейшее чтение

Тема регулярного выражения содержит больше информации о регулярных выражениях.

Examples

Использование групп захвата

Если вам нужно извлечь часть строки из входной строки, мы можем использовать **группы захвата** regex.

В этом примере мы начнем с простого регулярного выражения номера телефона:

```
\d{3}-\d{3}-\d{4}
```

Если скобки добавлены в регулярное выражение, каждый набор круглых скобок считается *группой захвата* . В этом случае мы используем так называемые нумерованные группы захвата:

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Прежде чем мы сможем использовать его в Java, мы не должны забывать следовать правилам строк, избегая обратных косых черт, в результате получим следующий шаблон:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

Сначала нам нужно скомпилировать шаблон регулярного выражения, чтобы создать Pattern а затем нам нужен Matcher для соответствия нашей входной строке с шаблоном:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Далее, Matcher должен найти первую подпоследовательность, которая соответствует регулярному выражению:

```
phoneMatcher.find();
```

Теперь, используя групповой метод, мы можем извлечь данные из строки:

```
String number = phoneMatcher.group(0); //"800-555-1234" (Group 0 is everything the regex
matched)
String aCode = phoneMatcher.group(1); //"800"
String threeDigit = phoneMatcher.group(2); //"555"
String fourDigit = phoneMatcher.group(3); //"1234"
```

Примечание: Matcher.group() может использоваться вместо Matcher.group(0) .

Java SE 7

Java 7 представила названные группы захвата. Именованные группы захвата функционируют так же, как и нумерованные группы захвата (но с именем вместо числа), хотя есть небольшие изменения синтаксиса. Использование названных групп захвата улучшает читаемость.

Мы можем изменить приведенный выше код для использования названных групп:

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^-----------------^ ^-----^ ^-----^
AreaCode            Group 2 Group 3
```

Чтобы получить содержимое «AreaCode», мы можем вместо этого использовать:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Использование регулярных выражений с пользовательским поведением путем компиляции шаблона с флагами

Pattern можно скомпилировать с помощью флагов, если регулярное выражение используется как литерал String , используйте встроенные модификаторы:

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
pattern.matcher("FOO\n").matches(); // Is true.

/* Had the regex not been compiled case insensitively and singlelined,
 * it would fail because FOO does not match /foo/ and \n (newline)
 * does not match /./.
 */

Pattern anotherPattern = Pattern.compile("(?si)foo");
anotherPattern.matcher("FOO\n").matches(); // Is true.

"foOt".replaceAll("(?si)foo", "ca"); // Returns "cat".
```

Персонажи побега

В общем-то

Чтобы использовать регулярные выражения определенных символов (?+| т. Д.) В их буквальном значении, их нужно избегать. В обычном регулярном выражении это выполняется обратным слэшем \ . Однако, поскольку это имеет особое значение в Java-строках, вам нужно использовать двойную обратную косую черту \\ .

Эти два примера не будут работать:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

Этот пример работает

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

Разделение строки с разделителями труб

Это не возвращает ожидаемый результат:

```
"a|b".split ("|"); // [a, |, b]
```

Это возвращает ожидаемый результат:

```
"a|b".split ("\\|"); // [a, b]
```

Сбрасывание обратной косой черты \

Это даст ошибку:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

Это работает:

```
"\\".matches("\\\\"); // true
```

Согласование с литералом регулярного выражения.

Если вам нужно сопоставить символы, которые являются частью синтаксиса регулярных выражений, вы можете пометить все или часть шаблона как литерал регулярного выражения.

\Q обозначает начало литерала регулярного выражения. \E обозначает конец литерала регулярного выражения.

```
// the following throws a PatternSyntaxException because of the un-closed bracket
"[123".matches("[123");

// wrapping the bracket in \Q and \E allows the pattern to match as you would expect.
"[123".matches("\\Q[\\E123"); // returns true
```

Более простой способ сделать это без необходимости запоминать escape-последовательности \Q и \E - использовать Pattern.quote()

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Не соответствует заданной строке

Чтобы сопоставить то, что *не* содержит заданную строку, можно использовать отрицательный просмотр:

Синтаксис Regex: (?!string-to-not-match)

Пример:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Выход:

```
[popcorn] nope!
[unicorn] matched!
```

Соответствие обратной косой черты

Если вы хотите совместить обратную косую черту в своем регулярном выражении, вам придется ее избежать.

Обратная косая черта - это символ escape в регулярных выражениях. Вы можете использовать '\\', чтобы ссылаться на одну обратную косую черту в регулярном выражении.

Однако обратная косая черта *также* является символом escape в строках строки Java. Для того, чтобы регулярное выражение из строки буквальным, вы должны бежать каждый из *его* обратной косой черты. В строковом литерале «\\\\» можно использовать для создания регулярного выражения с «\\», которое, в свою очередь, может соответствовать «\».

Например, рассмотрите соответствующие строки, такие как «C: \ dir \ myfile.txt». Регулярное выражение ([A-Za-z]):\\(.*) Будет совпадать и предоставить букву диска в качестве группы захвата. Обратите внимание на удвоенную обратную косую черту.

Чтобы выразить этот шаблон в строковом литерале Java, каждая обратная косая черта в регулярном

выражении должна быть экранирована.

```
    String path = "C:\\dir\\myfile.txt";
    System.out.println( "Local path: " + path ); // "C:\dir\myfile.txt"

    String regex = "([A-Za-z]):\\\\.*"; // Four to match one
    System.out.println("Regex:      " + regex ); // "([A-Za-z]):\\(.*)"

    Pattern pattern = Pattern.compile( regex );
    Matcher matcher = pattern.matcher( path );
    if ( matcher.matches()) {
        System.out.println( "This path is on drive " + matcher.group( 1 ) + ":.");
        // This path is on drive C:.
    }
```

Если вы хотите совместить *две* обратные косые черты, вы обнаружите, что используете восьмеричную строку, чтобы представить четыре в регулярном выражении, чтобы соответствовать двум.

```
    String path = "\\\\myhost\\share\\myfile.txt";
    System.out.println( "UNC path: " + path ); // \\myhost\share\myfile.txt"

    String regex = "\\\\\\\\(.+?)\\\\(.*)"; // Eight to match two
    System.out.println("Regex:    " + regex ); // \\\\(.+?)\\(.*)

    Pattern pattern = Pattern.compile( regex );
    Matcher matcher = pattern.matcher( path );

    if ( matcher.matches()) {
        System.out.println( "This path is on host '" + matcher.group( 1 ) + "'.");
        // This path is on host 'myhost'.
    }
```

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/java/topic/135/регулярные-выражения

глава 152: Рекурсия

Вступление

Рекурсия происходит, когда метод вызывает себя. Такой метод называется **рекурсивным** . Рекурсивный метод может быть более кратким, чем эквивалентный нерекурсивный подход. Однако для глубокой рекурсии иногда итерационное решение может потреблять меньше пространства стека в потоке.

В этом разделе приведены примеры рекурсии на Java.

замечания

Проектирование рекурсивного метода

При разработке рекурсивного метода помните, что вам нужно:

- **Базовый вариант.** Это определит, когда ваша рекурсия остановится и выведет результат. Базовый случай в факториальном примере:

```
if (n <= 1) {
    return 1;
}
```

- **Рекурсивный вызов.** В этом заявлении вы повторно вызываете метод с измененным параметром. Рекурсивный вызов в приведенном выше факториальном примере:

```
else {
    return n * factorial(n - 1);
}
```

Выход

В этом примере вы вычисляете n-й факторный номер. Первыми факториалами являются:

0! = 1

1! = 1

2! = 1 x 2 = 2

3! = 1 x 2 x 3 = 6

4! = 1 x 2 x 3 x 4 = 24

...

Удаление Java и Tail-call

Текущие компиляторы Java (вплоть до Java 9) не выполняют исключение хвостового вызова. Это может повлиять на производительность рекурсивных алгоритмов, и если рекурсия достаточно глубокая, это может привести к сбоям StackOverflowError ; см. Глубокая рекурсия проблематична в Java

Examples

Основная идея рекурсии

Что такое рекурсия:

В общем, рекурсия - это когда функция вызывает себя, прямо или косвенно. Например:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Условия применения рекурсии к задаче:

Существуют две предпосылки для использования рекурсивных функций для решения конкретной проблемы:

1. Должно быть базовое условие проблемы, которое будет конечной точкой для рекурсии. Когда рекурсивная функция достигает базового условия, она не делает дальнейших (более глубоких) рекурсивных вызовов.

2. Каждый уровень рекурсии должен пытаться решить меньшую проблему. Таким образом, рекурсивная функция делит проблему на более мелкие и мелкие части. Предполагая, что проблема конечна, это обеспечит завершение рекурсии.

В Java есть третье предварительное условие: не нужно слишком глубоко задумываться о решении проблемы; см. Глубокая рекурсия проблематична в Java

пример

Следующая функция вычисляет факториалы с использованием рекурсии. Обратите внимание, как метод factorial вызывает себя внутри функции. Каждый раз, когда он называет себя, он уменьшает параметр n на 1. Когда n достигает 1 (базовое условие), функция не будет углубляться.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Это не практический способ вычисления факториалов в Java, поскольку он не учитывает переполнение целого числа или переполнение стека вызовов (например, исключения StackOverflowError) при больших значениях n .

Вычисление N-го числа Фибоначчи

Следующий метод вычисляет N-й номер Фибоначчи с использованием рекурсии.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

Метод реализует базовый случай (n <= 2) и рекурсивный случай (n> 2). Это иллюстрирует использование рекурсии для вычисления рекурсивного отношения.

Однако, хотя этот пример является иллюстративным, он также неэффективен: каждый отдельный экземпляр метода дважды вызовет эту функцию, что приведет к экспоненциальному росту числа раз, когда функция называется ростом N. Вышеуказанная функция O (2^N), но эквивалентное итерационное решение имеет сложность O (N). Кроме того, существует выражение «закрытая форма», которое может быть оценено с помощью умножения с плавающей запятой O (N).

Вычисление суммы целых чисел от 1 до N

Следующий метод вычисляет сумму целых чисел от 0 до N, используя рекурсию.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

Этот метод O (N) и может быть сведен к простому циклу с использованием оптимизации хвостового вызова. На самом деле существует *замкнутое* выражение *формы*, которое вычисляет сумму в операциях O(1) .

Вычисление N-й степени числа

Следующий метод вычисляет значение num увеличенное до степени exp используя рекурсию:

```
public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}
```

Это иллюстрирует принципы, упомянутые выше: рекурсивный метод реализует базовый случай (два случая, n = 0 и n = 1), который завершает рекурсию, и рекурсивный случай, который вызывает метод снова. Этот метод O (N) и может быть сведен к простому циклу с использованием оптимизации хвостового вызова.

Обратить строку с помощью рекурсии

Ниже приведен рекурсивный код для изменения строки

```
/**
 * Just a snippet to explain the idea of recursion
 *
 **/

public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
        System.out.println(reverse(string)); //prints dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

Перемещение структуры данных дерева с рекурсией

Рассмотрим класс Node, содержащий 3 элемента данных, левый указатель на ребенка и правый дочерний указатель, как показано ниже.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}
```

Мы можем пересечь дерево, построенное путем соединения нескольких объектов класса Node, как показано ниже, обход называется обходным деревом в порядке.

```
public static void inOrderTraversal(Node root) {
        if (root != null) {
            inOrderTraversal(root.left); // traverse left sub tree
            System.out.print(root.data + " "); // traverse current node
            inOrderTraversal(root.right); // traverse right sub tree
        }
    }
```

Как показано выше, используя **рекурсию**, мы можем пересечь структуру **древовидных данных** без использования какой-либо другой структуры данных, которая невозможна при **итеративном** подходе.

Типы рекурсии

Рекурсия может быть классифицирована как **Head Recursion** или **Tail Recursion** , в зависимости от места размещения рекурсивного метода.

При **рекурсии головы** рекурсивный вызов, когда это происходит, предшествует другой обработке в функции (думайте, что это происходит сверху или в голове функции).

В **хвостовой рекурсии** это противоположность – обработка происходит до рекурсивного вызова. Выбор между двумя рекурсивными стилями может показаться произвольным, но выбор может иметь значение.

Функция с путём с единственным рекурсивным вызовом в начале пути использует так называемую рекурсию головы. Факториальная функция предыдущего экспоната использует рекурсию головы. Первое, что он делает, когда оно определяет, что требуется рекурсия, – это вызвать себя с декрементированным параметром. Функция с единственным рекурсивным вызовом в конце пути использует хвостовую рекурсию.

```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);

   tail(n-1);
}
```

```
public void head(int n)
   {
     if(n == 0)
        return;
     else
        head(n-1);

      System.out.println(n);
    }
```

Если рекурсивный вызов встречается в конце метода, он называется tail recursion . Рекурсия хвоста similar to a loop . method executes all the statements before jumping into the next recursive call .

Если рекурсивный вызов происходит в beginning of a method, it is called a head recursion . method saves the state before jumping into the next recursive call .

Ссылка: Разница между рекурсией головы и хвоста

StackOverflowError & recursion to loop

Если рекурсивный вызов «слишком глубокий», это приводит к возникновению StackOverflowError . Java выделяет новый кадр для каждого вызова метода в стеке потока. Однако пространство стека каждого потока ограничено. Слишком много кадров в стеке приводит к переполнению стека (SO).

пример

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

Вызов этого метода с большими параметрами (например, recursion(50000) вероятно, приведет к переполнению стека. Точное значение зависит от размера стека потоков, что, в свою очередь, зависит от конструкции потока, параметров командной строки, таких как -Xss или размер по умолчанию для JVM.

Временное решение

Рекурсия может быть преобразована в цикл путем хранения данных для каждого рекурсивного вызова в структуре данных. Эта структура данных может храниться в куче, а не в стеке потоков.

В общем случае данные, необходимые для восстановления состояния вызова метода, могут храниться в стеке, а цикл while может использоваться для «имитации» рекурсивных вызовов. Данные, которые могут потребоваться, включают:

- объект, к которому был вызван метод (только методы экземпляра)
- параметры метода
- локальные переменные
- текущая позиция в исполнении или метод

пример

Следующий класс позволяет рекурсивно печатать древовидную структуру с заданной глубиной.

```
public class Node {

    public int data;
    public Node left;
    public Node right;

    public Node(int data) {
        this(data, null, null);
    }

    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    public void print(final int maxDepth) {
        if (maxDepth <= 0) {
            System.out.print("(...)");
        } else {
            System.out.print("(");
            if (left != null) {
                left.print(maxDepth-1);
```

```
            }
            System.out.print(data);
            if (right != null) {
                right.print(maxDepth-1);
            }
            System.out.print(")");
        }
    }

}
```

например

```
Node n = new Node(10, new Node(20, new Node(50), new Node(1)), new Node(30, new Node(42),
null));
n.print(2);
System.out.println();
```

Печать

```
(((...)20(...))10((...)30))
```

Это можно преобразовать в следующий цикл:

```
public class Frame {

    public final Node node;

    // 0: before printing anything
    // 1: before printing data
    // 2: before printing ")"
    public int state = 0;
    public final int maxDepth;

    public Frame(Node node, int maxDepth) {
        this.node = node;
        this.maxDepth = maxDepth;
    }

}
```

```
List<Frame> stack = new ArrayList<>();
stack.add(new Frame(n, 2)); // first frame = initial call

while (!stack.isEmpty()) {
    // get topmost stack element
    int index = stack.size() - 1;
    Frame frame = stack.get(index); // get topmost frame
    if (frame.maxDepth <= 0) {
        // termial case (too deep)
        System.out.print("(...)");
        stack.remove(index); // drop frame
    } else {
        switch (frame.state) {
            case 0:
                frame.state++;

                // do everything done before the first recursive call
                System.out.print("(");
```

```
                if (frame.node.left != null) {
                    // add new frame (recursive call to left and stop)
                    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
                    break;
                }
            case 1:
                frame.state++;

                // do everything done before the second recursive call
                System.out.print(frame.node.data);
                if (frame.node.right != null) {
                    // add new frame (recursive call to right and stop)
                    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
                    break;
                }
            case 2:
                // do everything after the second recursive call & drop frame
                System.out.print(")");
                stack.remove(index);
        }
    }
}
System.out.println();
```

Примечание. Это всего лишь пример общего подхода. Часто вы можете найти гораздо лучший способ представления кадра и / или хранения данных кадра.

Глубокая рекурсия проблематична в Java

Рассмотрим следующий наивный метод для добавления двух положительных чисел с помощью рекурсии:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

Это алгоритмически правильно, но это имеет серьезную проблему. Если вы вызовете add с большим a , он выйдет из строя с помощью StackOverflowError на любой версии Java до (по крайней мере) Java 9.

В типичном языке функционального программирования (и многих других языках) компилятор оптимизирует хвостовую рекурсию . Компилятор заметил бы, что вызов для add (в помеченной строке) является хвостовым вызовом и будет эффективно переписывать рекурсию как цикл. Это преобразование называется устранением хвостового вызова.

Однако компиляторы Java поколения текущего поколения не выполняют исключение хвостового вызова. (Это не простой надзор. Для этого существуют существенные технические причины, см. Ниже.) Вместо этого каждый рекурсивный вызов add вызывает выделение нового кадра в стеке потока. Например, если вы вызываете add(1000, 1) , для ответа 1001 потребуется 1000 рекурсивных вызовов.

Проблема в том, что размер потока потоков Java фиксируется при создании потока. (Сюда входит «основной» поток в однопоточной программе.) Если слишком много кадров стека распределены, стек будет переполняться. JVM обнаружит это и выбросит StackOverflowError .

Один из способов борьбы с этим - просто использовать больший стек. Есть JVM варианты , которые контролируют по умолчанию размера стека, и вы можете также указать размер стека в качестве Thread параметра конструктора. К сожалению, это только «отбрасывает» переполнение стека. Если вам нужно выполнить вычисления, для которых требуется еще больший стек, возвращается

StackOverflowError .

Реальное решение состоит в том, чтобы идентифицировать рекурсивные алгоритмы, где вероятна глубокая рекурсия, и *вручную* выполнить оптимизацию хвостового вызова на уровне исходного кода. Например, наш метод add можно переписать следующим образом:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(Очевидно, что есть лучшие способы добавить два целых числа. Вышеупомянутое просто иллюстрирует эффект устранения ручного хвостового вызова.)

Почему исключение хвостового вызова не реализовано на Java (пока)

Существует ряд причин, по которым добавление исключения хвоста в Java нелегко. Например:

- Некоторый код может полагаться на StackOverflowError для (например) размещения привязки по размеру вычислительной проблемы.
- Менеджеры безопасности Sandbox часто полагаются на анализ стека вызовов при принятии решения о том, разрешать ли непривилегированный код выполнять привилегированное действие.

Как объясняет Джон Роуз в «Хвостах звонков на ВМ» :

> *«Эффекты удаления фрейма стека вызывающего абонента видны некоторым API-интерфейсам, особенно проверкам контроля доступа и трассировке стека. Как будто вызывающий вызывающий вызывал непосредственно вызываемый вызывающий. Любые привилегии, которыми обладает вызывающий, отбрасываются после того, как управление передано в Однако связь и доступность метода вызываемого метода вычисляются до передачи управления и учитывают вызывающего звонка вызывающего абонента ».*

Другими словами, устранение хвостового вызова может привести к тому, что метод управления доступом ошибочно полагает, что защищенный от безопасности API был вызван доверенным кодом.

Прочитайте Рекурсия онлайн: https://riptutorial.com/ru/java/topic/914/рекурсия

глава 153: Ресурсы (на пути к классам)

Вступление

Java позволяет извлекать файловые ресурсы, хранящиеся внутри JAR, вместе с скомпилированными классами. В этом разделе основное внимание уделяется загрузке этих ресурсов и обеспечению их доступности для вашего кода.

замечания

Ресурс - это файловые данные с именем типа пути, которое находится в пути к классам. Наиболее частое использование ресурсов - это объединение изображений приложений, звуков и данных только для чтения (таких как конфигурация по умолчанию).

Доступ к ресурсам можно получить с помощью методов ClassLoader.getResource и ClassLoader.getResourceAsStream . Наиболее распространенным случаем является наличие ресурсов, размещенных в том же пакете, что и класс, который их читает; методы Class.getResource и Class.getResourceAsStream служат этому обычному варианту использования.

Единственное различие между методом getResource и методом getResourceAsStream заключается в том, что первый возвращает URL-адрес, а второй открывает этот URL-адрес и возвращает InputStream.

Методы ClassLoader принимают имя типа пути как аргумент и ищут каждое местоположение в пути класса ClassLoader для записи, соответствующей этому имени.

- Если местоположение класса является файлом .jar, запись в jar с указанным именем считается совпадением.
- Если расположение пути к классам является каталогом, относительный файл в этом каталоге с указанным именем считается совпадением.

Имя ресурса похоже на часть пути относительного URL. На *всех платформах* он использует косые черты (/) в качестве разделителей каталогов. Он не должен начинаться с косой черты.

Соответствующие методы класса аналогичны, за исключением:

- Имя ресурса может начинаться с косой черты, и в этом случае начальная косая черта удаляется, а остальная часть имени передается соответствующему методу ClassLoader.
- Если имя ресурса не начинается с косой черты, оно рассматривается как относительное к классу, вызываемому методом getResource или getResourceAsStream. Фактическое имя ресурса становится *package* / *name* , где *package* - это имя пакета, к которому принадлежит класс, причем каждый период заменяется косой чертой, а *имя* - исходным аргументом, заданным для метода.

Например:

```
package com.example;

public class ExampleApplication {
    public void readImage()
    throws IOException {

        URL imageURL = ExampleApplication.class.getResource("icon.png");

        // The above statement is identical to:
        // ClassLoader loader = ExampleApplication.class.getClassLoader();
        // URL imageURL = loader.getResource("com/example/icon.png");

        Image image = ImageIO.read(imageURL);
    }
}
```

Ресурсы должны размещаться в именованных пакетах, а не в корневом каталоге .jar, по той же причине классы помещаются в пакеты: Чтобы предотвратить столкновение между несколькими поставщиками. Например, если несколько файлов .jar находятся в пути к классам, и более чем один из них содержит запись config.properties в своем корне, вызовы методов getResource или getResourceAsStream возвращают параметры config.properties из того, что .jar указан первым в путь к классам. Это не предсказуемое поведение в средах, где порядок пути к классам не находится под непосредственным контролем приложения, например Java EE.

Все методы getResource и getResourceAsStream возвращают значение null если указанный ресурс не существует. Поскольку ресурсы должны быть добавлены в приложение во время сборки, их местоположения должны быть известны при написании кода; отказ найти ресурс во время выполнения обычно является результатом ошибки программиста.

Ресурсы доступны только для чтения. Невозможно написать ресурс. Разработчики новичка часто ошибаются, полагая, что, поскольку ресурс является отдельным физическим файлом при разработке в среде IDE (например, Eclipse), безопасно рассматривать его как отдельный физический файл в общем случае. Однако это неверно; приложения почти всегда распространяются как архивы, такие как файлы .jar или .war, и в таких случаях ресурс не будет отдельным файлом и не будет доступен для записи. (Метод getFile для класса URL-адресов *не* является обходным путем для этого, несмотря на его имя, он просто возвращает часть пути URL-адреса, которая отнюдь не гарантирует, что это действительное имя файла).

Не существует безопасного способа перечислить ресурсы во время выполнения. Опять же, поскольку разработчики несут ответственность за добавление файлов ресурсов в приложение во время сборки, разработчики должны уже знать их пути. Хотя есть обходные пути, они не надежны и в конечном итоге потерпят неудачу.

Examples

Загрузка изображения с ресурса

Чтобы загрузить связанное изображение:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Загрузка конфигурации по умолчанию

Чтобы прочитать свойства конфигурации по умолчанию:

```
package com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
            ExampleApplication.class.getResourceAsStream("config.properties")) {

            defaults.load(defaultsStream);
        }

        return defaults;
    }
```

```
}
```

Загрузка одноименного ресурса из нескольких JAR

Ресурс с одним и тем же путем и именем может существовать в более чем одном JAR-файле в пути к классам. Обычными случаями являются ресурсы, следующие за конвенцией, или которые являются частью спецификации упаковки. Примерами таких ресурсов являются

- META-INF / MANIFEST.MF
- META-INF / beans.xml (CDI Spec)
- Свойства ServiceLoader, содержащие поставщиков реализации

Чтобы получить доступ ко *всем* этим ресурсам в разных банках, нужно использовать ClassLoader, у которого есть метод для этого. Возвращаемое Enumeration можно удобно преобразовать в List с помощью функции «Коллекции».

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-
INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Поиск и чтение ресурсов с помощью загрузчика классов

Загрузка ресурсов в Java включает следующие шаги:

1. Поиск Class или ClassLoader , который найдет ресурс.
2. Поиск ресурса.
3. Получение байтового потока для ресурса.
4. Чтение и обработка байтового потока.
5. Закрытие байтового потока.

Последние три шага обычно выполняются путем передачи URL-адреса библиотечному методу или конструктору для загрузки ресурса. В этом случае вы обычно будете использовать метод getResource . Также возможно считывать данные ресурсов в коде приложения. В этом случае вы обычно будете использовать getResourceAsStream .

Абсолютные и относительные пути ресурсов

Ресурсы , которые могут быть загружены из пути к классам, обозначены *пути.* Синтаксис пути похож на путь файла UNIX / Linux. Он состоит из простых имен, разделенных символами прямой косой черты (/). *Относительный путь* начинается с имени, а *абсолютный путь* начинается с разделителя.

Как описывают примеры Classpath, путь к классам JVM определяет пространство имен путем наложения пространств имен каталогов и JAR или ZIP-файлов в пути к классам. Когда абсолютный путь разрешен, он загрузчик классов интерпретирует исходный / как смысл корня пространства имен. Напротив, относительный путь *может быть* разрешен относительно любой «папки» в пространстве имен. Используемая папка будет зависеть от объекта, который используется для разрешения пути.

Получение класса или загрузчика классов

Ресурс может быть расположен с использованием объекта Class или объекта ClassLoader . Объект Class может разрешать относительные пути, поэтому вы обычно используете один из них, если у вас есть (класс) относительный ресурс. Существует множество способов получить объект Class . Например:

- *Литерал класса* даст вам объект Class для любого класса, который вы можете назвать в исходном коде Java; например String.class предоставляет объект Class для типа String .
- Object.getClass() предоставит вам объект Class для типа od любого объекта; например,

"hello".getClass() - это еще один способ получить Class типа String .

- Метод Class.forName(String) будет (при необходимости) динамически загружать класс и возвращать его объект Class ; например Class.forName("java.lang.String") .

Объект ClassLoader обычно получается путем вызова getClassLoader() объекта Class . Также можно получить загрузчик классов по умолчанию JVM, используя статический ClassLoader.getSystemClassLoader() .

Методы get

Если у вас есть экземпляр Class или ClassLoader , вы можете найти ресурс, используя один из следующих способов:

методы	Описание
ClassLoader.getResource(path) ClassLoader.getResources(path)	Возвращает URL-адрес, который представляет местоположение ресурса с заданным путем.
ClassLoader.getResources(path) Class.getResources(path)	Возвращает Enumeration<URL> указывающее URL-адреса, которые можно использовать для поиска ресурса foo.bar ; увидеть ниже.
ClassLoader.getResourceAsStream(path) Class.getResourceStream(path)	Возвращает InputStream из которого вы можете прочитать содержимое ресурса foo.bar в виде последовательности байтов.

Заметки:

- Основное различие между версиями методов ClassLoader и Class заключается в том, как интерпретируются относительные пути.
    - Методы Class разрешают относительный путь в папке, соответствующей пакету классов.
    - Методы ClassLoader обрабатывают относительные пути, как если бы они были абсолютными; т.е. разрешить их в «корневой папке» пространства имен classpath.
- Если запрошенный ресурс (или ресурсы) не может быть найден, methods return getResource и getResourceAsStream methods return null , and the methods return an empty getResources methods return an empty перечисление.
- URL.toStream() URL-адреса будут URL.toStream() с помощью URL.toStream() . Они могут быть file: URL-адресами или другими обычными URL-адресами, но если ресурс находится в JAR-файле, они будут jar: URL-адресами, которые идентифицируют JAR-файл и определенный ресурс внутри него.
- Если ваш код использует метод getResourceAsStream (или URL.toStream()) для получения InputStream , он отвечает за закрытие объекта потока. Невозможность закрыть поток может привести к утечке ресурсов.

Прочитайте Ресурсы (на пути к классам) онлайн: https://riptutorial.com/ru/java/topic/2433/ресурсы--на-пути-к-классам-

глава 154: Розетки

Вступление

Сокет является одной конечной точкой двусторонней линии связи между двумя программами, запущенными в сети.

Examples

Чтение из сокета

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
) {
    //Use the socket
}
```

Прочитайте Розетки онлайн: https://riptutorial.com/ru/java/topic/9918/розетки

глава 155: Свободный интерфейс

замечания

цели

Основная цель Свободного интерфейса - повышенная читаемость.

При использовании для построения объектов выбор, доступный для вызывающего, может быть четко и принудительно осуществлен с помощью проверок времени компиляции. Например, рассмотрим следующее дерево опций, представляющее шаги по пути для создания некоторого сложного объекта:

```
A -> B
  -> C -> D -> Done
       -> E -> Done
       -> F -> Done.
       -> G -> H -> I -> Done.
```

Строитель, используя свободный интерфейс, позволит вызывающему пользователю легко увидеть, какие опции доступны на каждом этапе. Например, **A -> B** возможен, но **A -> C** не является и приведет к ошибке времени компиляции.

Examples

Истина - Свободное тестирование

Из раздела «Как использовать правду» http://google.github.io/truth/

```
String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();
```

Свободный стиль программирования

В беглом стиле программирования вы возвращаете this из беглого (сеттера) методов, которые ничего не вернут в ненадежном стиле программирования.

Это позволяет вам связывать различные вызовы методов, которые делают ваш код короче и легче обрабатывать для разработчиков.

Рассмотрим этот непрозрачный код:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
```

```
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    Person person = new Person();
    person.setFirstName("John");
    person.setLastName("Doe");
    System.out.println(person.whoAreYou());
  }
}
```

Поскольку методы setter ничего не возвращают, нам нужно 4 инструкции в main методе, чтобы создать экземпляр Person с некоторыми данными и распечатать его. С быстрым стилем этот код можно изменить на:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public Person withFirstName(String firstName) {
    this.firstName = firstName;
    return this;
  }

  public String getLastName() {
    return lastName;
  }

  public Person withLastName(String lastName) {
    this.lastName = lastName;
    return this;
  }

  public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    System.out.println(new Person().withFirstName("John")
      .withLastName("Doe").whoAreYou());
  }
}
```

Идея состоит в том, чтобы всегда возвращать некоторый объект, чтобы создать построение цепочки вызовов метода и использовать имена методов, которые отражают естественную речь. Этот свободный стиль делает код более удобочитаемым.

Прочитайте Свободный интерфейс онлайн: https://riptutorial.com/ru/java/topic/5090/свободный-интерфейс

глава 156: Сериализация

Вступление

Java предоставляет механизм, называемый сериализации объектов, где объект может быть представлен как последовательность байтов, которая включает в себя данные объекта, а также информацию о типе объекта и типах данных, хранящихся в объекте.

После того, как сериализованный объект был записан в файл, он может быть прочитан из файла и десериализован, то есть информация о типе и байты, которые представляют объект и его данные, могут использоваться для воссоздания объекта в памяти.

Examples

Базовая сериализация в Java

Что такое сериализация

Сериализация – это процесс преобразования состояния объекта (включая его ссылки) в последовательность байтов, а также процесс перестройки этих байтов в живой объект в будущем. Сериализация используется, когда вы хотите сохранить объект. Он также используется Java RMI для передачи объектов между JVM, либо как аргументы в вызове метода от клиента к серверу, либо как возвращаемые значения из вызова метода, либо как исключения, создаваемые удаленными методами. В общем случае сериализация используется, когда мы хотим, чтобы объект существовал за время существования JVM.

java.io.Serializable – это интерфейс маркера (не имеет тела). Он просто используется, чтобы «маркировать» классы Java как сериализуемые.

Сериализация runtime связывает каждый сериализуемый класс с номером версии, называемым serialVersionUID , который используется во время де- сериализации для проверки того, что отправитель и получатель сериализованного объекта загружают классы для этого объекта, которые совместимы с сериализацией. Если получатель загрузил класс для объекта с другим serialVersionUID чем класс соответствующего класса отправителя, то десериализация приведет к InvalidClassException . Сериализуемый класс может объявить свой собственный serialVersionUID явно, объявив поле с именем serialVersionUID которое должно быть static, final, и long :

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

Как сделать класс подходящим для сериализации

Чтобы сохранить объект, соответствующий класс должен реализовать интерфейс java.io.Serializable .

```
import java.io.Serializable;

public class SerialClass implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date currentTime;

    public SerialClass() {
        currentTime = Calendar.getInstance().getTime();
    }

    public Date getCurrentTime() {
        return currentTime;
    }
}
```

Как написать объект в файл

Теперь нам нужно записать этот объект в файловую систему. Для этой цели мы используем java.io.ObjectOutputStream .

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //We will write this object to file system.
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //Write byte stream to file system.
            out.close();
        } catch(IOException ex){
            ex.printStackTrace();
        }
     }
 }
```

Как воссоздать объект из его сериализованного состояния

Сохраненный объект может быть прочитан из файловой системы позже, используя java.io.ObjectInputStream как показано ниже:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.java.lang.ClassNotFoundException;

 public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        } catch(IOException ex){
            ex.printStackTrace();
        } catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        // print out restored time
        System.out.println("Restored time: " + time.getTime());
     }
 }
```

Сериализованный класс находится в двоичной форме. Отмена десериализации может быть проблематичной, если определение класса изменяется: подробности см. В главе «Версии Serialized Objects» спецификации Java Serialization .

Сериализация объекта сериализует весь граф объектов, который является корнем, и работает корректно в присутствии циклических графов. Метод reset() предоставляется, чтобы заставить ObjectOutputStream забыть об объектах, которые уже были сериализованы.

Переходные поля - Сериализация

Сериализация с помощью Gson

Сериализация с Gson проста и выведет правильный JSON.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}
```

(Сериализация)

```
//Skills
List<String> skills = new LinkedList<String>();
skills.add("leadership");
skills.add("Java Experience");

//Employe
Employe obj = new Employe();
obj.setFirstName("Christian");
obj.setLastName("Lusardi");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//Serialization process
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"Christian","lastName":"Lusardi","age":25,"salary":10000,"skills":["leadership","Java
Experience"]}
```

Обратите внимание, что вы не можете сериализовать объекты с круговыми ссылками, поскольку это приведет к бесконечной рекурсии.

(Десериализация)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Сериализация с помощью Jackson 2

Ниже приведена реализация, демонстрирующая, как объект может быть сериализован в соответствующую строку JSON.

```
class Test {

    private int idx;
    private String name;

    public int getIdx() {
        return idx;
    }
```

```
    public void setIdx(int idx) {
        this.idx = idx;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Сериализация:

```
Test test = new Test();
test.setIdx(1);
test.setName("abc");

ObjectMapper mapper = new ObjectMapper();

String jsonString;
try {
    jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(test);
    System.out.println(jsonString);
} catch (JsonProcessingException ex) {
    // Handle Exception
}
```

Выход:

```
{
  "idx" : 1,
  "name" : "abc"
}
```

Вы можете опустить Default Pretty Printer, если вам это не нужно.

Используемая здесь зависимость выглядит следующим образом:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

Пользовательская сериализация

В этом примере мы хотим создать класс, который будет генерировать и выводить на консоль случайное число между двумя целыми числами, которые передаются в качестве аргументов во время инициализации.

```
    public class SimpleRangeRandom implements Runnable {
    private int min;
    private int max;

    private Thread thread;
```

```
    public SimpleRangeRandom(int min, int max){
        this.min = min;
        this.max = max;
        thread = new Thread(this);
        thread.start();
    }

    @Override
 private void WriteObject(ObjectOutputStreamout) throws IO Exception;
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
    public void run() {
        while(true) {
            Random rand = new Random();
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    }
```

Теперь, если мы хотим сделать этот класс Serializable, возникнут некоторые проблемы. Thread является одним из определенных классов системного уровня, которые не являются Serializable. Поэтому нам нужно объявить поток как **переходный** . Сделав это, мы сможем сериализовать объекты этого класса, но у нас все еще будет проблема. Как вы можете видеть в конструкторе, мы устанавливаем минимальные и максимальные значения нашего рандомизатора, после чего начинаем поток, который отвечает за создание и печать случайного значения. Таким образом, при восстановлении сохраненного объекта вызовом ***readObject ()*** конструктор не будет запускаться снова, поскольку нет создания нового объекта. В этом случае нам нужно разработать **пользовательскую сериализацию** , предоставив два метода внутри класса. Эти методы:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Таким образом, добавив нашу реализацию в ***readObject ()***, мы можем инициировать и запускать наш поток:

```
class RangeRandom implements Serializable, Runnable {

private int min;
private int max;

private transient Thread thread;
//transient should be any field that either cannot be serialized e.g Thread or any field you
do not want serialized

public RangeRandom(int min, int max){
    this.min = min;
    this.max = max;
    thread = new Thread(this);
    thread.start();
}

@Override
public void run() {
    while(true) {
        Random rand = new Random();
```

```
        System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    thread = new Thread(this);
    thread.start();
}
}
```

Вот основной пример нашего примера:

```
public class Main {
public static void main(String[] args) {
    System.out.println("Hello");
    RangeRandom rangeRandom = new RangeRandom(1,10);

    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try
    {
        fos = new FileOutputStream("test");
        out = new ObjectOutputStream(fos);
        out.writeObject(rangeRandom);
        out.close();
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }


    RangeRandom rangeRandom2 = null;
       FileInputStream fis = null;
       ObjectInputStream in = null;
       try
       {
             fis = new FileInputStream("test");
             in = new ObjectInputStream(fis);
           rangeRandom2 = (RangeRandom)in.readObject();
             in.close();
           }
       catch(IOException ex)
       {
             ex.printStackTrace();
           }
       catch(ClassNotFoundException ex)
       {
             ex.printStackTrace();
           }
```

```
}
}
```

Если вы запустите main, вы увидите, что для каждого экземпляра RangeRandom выполняется два потока, и это связано с тем, что метод ***Thread.start ()*** теперь находится как в конструкторе, так и в ***readObject ()*** .

Версии и serialVersionUID

Когда вы реализуете интерфейс java.io.Serializable для создания сериализуемого класса, компилятор ищет static final поле с именем serialVersionUID типа long . Если класс не указал это поле явно, то компилятор создаст одно такое поле и присвоит ему значение, которое выходит из зависимого от реализации вычисления serialVersionUID . Это вычисление зависит от различных аспектов класса, и оно следует за спецификациями сериализации объектов, данными Sun. Но, во всех реализациях компилятора не гарантируется одинаковое значение.

Это значение используется для проверки совместимости классов в отношении сериализации, и это выполняется при де-сериализации сохраненного объекта. Последовательность выполнения Serialization проверяет, что serialVersionUID считанный из де-сериализованных данных, и serialVersionUID объявленные в классе, точно такие же. Если это не так, это вызывает InvalidClassException .

Настоятельно рекомендуется явно объявить и инициализировать статическое окончательное поле типа long и с именем «serialVersionUID» во всех ваших классах, которые вы хотите сделать Serializable, вместо того, чтобы полагаться на вычисление значения по умолчанию для этого поля, даже если вы не собираетесь использование управление версиями. **Вычисление «serialVersionUID» чрезвычайно чувствительно и может варьироваться от одной реализации компилятора к другой, и, следовательно, вы можете получить InvalidClassException даже для того же класса только потому, что вы использовали разные реализации компилятора на отправителе и на концах приемника процесса сериализации.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

Пока serialVersionUID тот же, Java Serialization может обрабатывать разные версии класса. Совместимые и несовместимые изменения;

Совместимые изменения

- **Добавление полей:** когда восстанавливаемый класс имеет поле, которое не встречается в потоке, это поле в объекте будет инициализировано значением по умолчанию для его типа. Если требуется инициализация, специфичная для класса, класс может предоставить метод readObject, который может инициализировать поле для значений небезопасности.
- **Добавление классов:** поток будет содержать иерархию типов каждого объекта в потоке. Сравнение этой иерархии в потоке с текущим классом может обнаруживать дополнительные классы. Поскольку в потоке, из которого инициализируется объект, нет информации, поля класса будут инициализированы значениями по умолчанию.
- **Удаление классов:** сравнение иерархии классов в потоке с потоком текущего класса может обнаружить, что класс был удален. В этом случае поля и объекты, соответствующие этому классу, считываются из потока. Примитивные поля отбрасываются, но объекты, на которые ссылается удаленный класс, создаются, так как они могут быть переданы позже в потоке. Они будут собирать мусор, когда поток собирается или сбрасывается мусором.
- **Добавление методов writeObject / readObject:** если версия, читающая поток, имеет эти методы, тогда readObject, как обычно, ожидается, чтобы прочитать необходимые данные, записанные в поток по сериализации по умолчанию. Он должен сначала вызвать defaultReadObject перед чтением любых дополнительных данных. Метод writeObject, как обычно, должен вызвать функцию defaultWriteObject для записи необходимых данных, а затем может записывать дополнительные данные.

- **Добавление java.io.Serializable:** это эквивалентно добавлению типов. В потоке для этого класса не будет значений, поэтому его поля будут инициализированы значениями по умолчанию. Поддержка подклассификации несериализуемых классов требует, чтобы супертип класса имел конструктор no-arg, и сам класс будет инициализирован значениями по умолчанию. Если конструктор no-arg недоступен, генерируется исключение InvalidClassException.
- **Изменение доступа к полю:** модификаторы доступа public, package, protected и private не влияют на возможность сериализации присвоить значения полям.
- **Изменение поля от статического до нестатического или переходного к нетрансходящему:** если полагаться на сериализацию по умолчанию для вычисления сериализуемых полей, это изменение эквивалентно добавлению поля в класс. Новое поле будет записано в поток, но предыдущие классы будут игнорировать значение, поскольку сериализация не будет присваивать значения статическим или переходным полям.

Несовместимые изменения

- **Удаление полей:** если поле удалено в классе, то записанный поток не будет содержать его значения. Когда поток считывается более ранним классом, значение поля будет установлено на значение по умолчанию, потому что в потоке не доступно значение. Однако это значение по умолчанию может отрицательно повлиять на способность более ранней версии выполнять свой контракт.
- **Перемещение классов вверх или вниз по иерархии:** это невозможно, так как данные в потоке отображаются в неправильной последовательности.
- **Изменение нестатического поля в статическом или нетрансловом поле на переходный период:** если полагаться на сериализацию по умолчанию, это изменение эквивалентно удалению поля из класса. Эта версия класса не будет записывать эти данные в поток, поэтому она не будет доступна для чтения более ранними версиями класса. Как и при удалении поля, поле более ранней версии будет инициализировано значением по умолчанию, которое может привести к сбою класса неожиданным образом.
- **Изменение объявленного типа примитивного поля:** каждая версия класса записывает данные с объявленным типом. Более ранние версии класса, пытающиеся прочитать поле, будут терпеть неудачу, потому что тип данных в потоке не соответствует типу поля.
- Изменение метода writeObject или readObject, чтобы он больше не записывал или не читал данные поля по умолчанию или не менял его так, чтобы он пытался записать его или прочитать, когда предыдущая версия не выполнялась. Данные поля по умолчанию должны последовательно отображаться или не отображаться в потоке.
- Изменение класса от Serializable до Externalizable или наоборот – это несовместимое изменение, поскольку поток будет содержать данные, которые несовместимы с реализацией доступного класса.
- Изменение класса из типа, отличного от enum, до типа перечисления или наоборот, поскольку поток будет содержать данные, которые несовместимы с реализацией доступного класса.
- Удаление Serializable или Externalizable является несовместимым изменением, поскольку после написания он больше не будет предоставлять поля, необходимые для более старых версий класса.
- Добавление метода writeReplace или readResolve к классу несовместимо, если поведение приведет к созданию объекта, который несовместим с любой более старой версией класса.

Пользовательская десериализация JSON с Джексоном

Мы используем API для отдыха как формат JSON, а затем отключаем его до POJO. Джексон's org.codehaus.jackson.map.ObjectMapper «просто работает» из коробки, и мы действительно ничего не делаем в большинстве случаев. Но иногда нам нужен настраиваемый десериализатор для выполнения наших собственных потребностей, и этот учебник поможет вам в создании собственного пользовательского десериализатора.

Допустим, у нас есть следующие сущности.

```
public class User {
    private Long id;
    private String name;
    private String email;
```

```
    //getter setter are omitted for clarity
}
```

А также

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

Давайте сначала сериализуем / маршалируем объект.

```
User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();
```

final String json = objectMapper.writeValueAsString (программа); System.out.println (JSON);

Вышеприведенный код будет содержать следующие JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Теперь можно сделать обратное очень легко. Если у нас есть этот JSON, мы можем развязать объект программы с помощью ObjectMapper следующим образом:

Скажем так, это не настоящий случай, у нас будет другой JSON из API, который не соответствует нашему классу Program .

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
}
```

Посмотрите на строку JSON, вы можете видеть, у нее есть другое поле, которое является owenerId.

Теперь, если вы хотите сериализовать этот JSON, как мы это делали ранее, у вас будут исключения.

Есть два способа избежать исключений и сделать это сериализованным -

Игнорировать неизвестные поля

Игнорируйте onwerId . Добавьте в класс программы следующую аннотацию

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Напишите пользовательский десериализатор

Но есть случаи, когда вам действительно нужно это поле owerId . Предположим, вы хотите связать его как идентификатор класса User .

В таком случае вам нужно написать собственный десериализатор,

Как вы можете видеть, сначала вам нужно получить доступ к JsonNode из JonsParser . А затем вы можете легко извлекать информацию из JsonNode с помощью метода get() . и вы должны убедиться в имени поля. Это должно быть точное имя, ошибка орфографии приведет к исключениям.

И , наконец, вы должны зарегистрировать свой ProgramDeserializer в ObjectMapper .

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"ownerId\":1,\"contents\":\"Some
contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

Кроме того, вы можете использовать аннотацию для регистрации десериализатора напрямую -

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Прочитайте Сериализация онлайн: https://riptutorial.com/ru/java/topic/767/сериализация

Синтаксис

- новый Socket («localhost», 1234); // Подключается к серверу по адресу «localhost» и порту 1234
- новый SocketServer («localhost», 1234); // Создает сервер сокетов, который может прослушивать новые сокеты по адресу localhost и порту 1234
- socketServer.accept (); // Принимает новый объект Socket, который может использоваться для связи с клиентом

Examples

Основные клиентские и серверные коммуникации с использованием разъема

Сервер: запуск и ожидание входящих соединений

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Сервер: обработка клиентов

Мы будем обрабатывать каждый клиент в отдельном потоке, чтобы сразу несколько клиентов могли взаимодействовать с сервером. Этот метод работает отлично, пока количество клиентов невелико (<< 1000 клиентов, в зависимости от архитектуры ОС и ожидаемой загрузки каждого потока).

```
new Thread(() -> {
    // Get the socket's InputStream, to read bytes from the socket
    InputStream in = clientSocket.getInputStream();
    // wrap the InputStream in a reader so you can read a String instead of bytes
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(in, StandardCharsets.UTF_8));
    // Read text from the socket and print line by line
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
    }).start();
```

Клиент: подключитесь к серверу и отправьте сообщение

```
// 127.0.0.1 is the address of the server (this is the localhost address; i.e.
// the address of our own machine)
// 1234 is the port that the server will be listening on
Socket socket = new Socket("127.0.0.1", 1234);

// Write a string into the socket, and flush the buffer
OutputStream outStream = socket.getOutputStream();
```

```
PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

Закрытие разъемов и исключение обработки

В приведенных выше примерах были некоторые вещи, которые упрощали их чтение.

1. Так же, как файлы и другие внешние ресурсы, важно сообщить ОС, когда мы с ними закончим. Когда мы закончим сокет, вызовите socket.close() чтобы правильно закрыть его.

2. Сокеты обрабатывают операции ввода / вывода (ввода / вывода), которые зависят от множества внешних факторов. Например, что, если другая сторона внезапно отключится? Что делать, если есть сетевая ошибка? Эти вещи находятся вне нашего контроля. Вот почему многие операции сокета могут вызывать исключения, особенно IOException .

Таким образом, более полный код для клиента будет примерно таким:

```
// "try-with-resources" will close the socket once we leave its scope
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //Handle the error
}
```

Базовый сервер и клиент - полные примеры

Сервер:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // Wait for a client connection.
                Socket clientSocket = serverSocket.accept();

                // Create and start a thread to handle the new client
                new Thread(() -> {
                    try {
                        // Get the socket's InputStream, to read bytes
                        // from the socket
                        InputStream in = clientSocket.getInputStream();
                        // wrap the InputStream in a reader so you can
                        // read a String instead of bytes
                        BufferedReader reader = new BufferedReader(
                                new InputStreamReader(in, StandardCharsets.UTF_8));
                        // Read from the socket and print line by line
                        String line;
```

```
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // This finally block ensures the socket is closed.
                        // A try-with-resources block cannot be used because
                        // the socket is passed into a thread, so it isn't
                        // created and closed in the same block
                        try {
                            clientSocket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Клиент:

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // We'll reach this code once we've connected to the server

            // Write a string into the socket, and flush the buffer
            OutputStream outStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                    new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
            writer.println("Hello world!");
            writer.flush();
        } catch (IOException e) {
            // Exception should be handled.
            e.printStackTrace();
        }
    }
}
```

Загрузка TrustStore и KeyStore из InputStream

```
public class TrustLoader {

    public static void main(String args[]) {
```

```
        try {
                //Gets the inputstream of a a trust store file under ssl/rpgrenadesClient.jks
                //This path refers to the ssl folder in the jar file, in a jar file in the
same directory
                //as this jar file, or a different directory in the same directory as the jar
file
                InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
                //Both trustStores and keyStores are represented by the KeyStore object
                KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
                //The password for the trustStore
                char[] trustStorePassword = "password".toCharArray();
                //This loads the trust store into the object
                trustStore.load(stream, trustStorePassword);

                //This is defining the SSLContext so the trust store will be used
                //Getting default SSLContext to edit.
                SSLContext context = SSLContext.getInstance("SSL");
                //TrustMangers hold trust stores, more than one can be added
                TrustManagerFactory factory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                //Adds the truststore to the factory
                factory.init(trustStore);
                //This is passed to the SSLContext init method
                TrustManager[] managers = factory.getTrustManagers();
                context.init(null, managers, null);
                //Sets our new SSLContext to be used.
                SSLContext.setDefault(context);
            } catch (KeyStoreException | IOException | NoSuchAlgorithmException
                    | CertificateException | KeyManagementException ex) {
                //Handle error
                ex.printStackTrace();
            }

    }
}
```

Intiating KeyStore работает одинаково, за исключением замены любого слова Trust в имени объекта с помощью Key . Кроме того, массив KeyManager[] должен быть передан первому аргументу SSLContext.init . Это SSLContext.init(keyMangers, trustMangers, null)

Пример сокета - чтение веб-страницы с использованием простого сокета

```
import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//We don't handle Exceptions in
this example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //Send a basic HTTP header
```

```
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead  = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
          sb.append(buffer, 0, charsRead);
        }
    }
}
```

Вы должны получить ответ, который начинается с HTTP/1.1 200 OK , что указывает на обычный HTTP-ответ, за которым следует остальная часть HTTP-заголовка, а затем необработанная веб-страница в форме HTML.

Обратите внимание, что readFully() важен для предотвращения преждевременного исключения EOF. В последней строке веб-страницы может отсутствовать возврат, чтобы сигнализировать о конце строки, а затем readLine() будет жаловаться, поэтому нужно прочитать ее вручную или использовать служебные методы из Apache commons-io IOUtils

Этот пример предназначен для простой демонстрации подключения к существующему ресурсу с помощью сокета, это не практический способ доступа к веб-страницам. Если вам нужно получить доступ к веб-странице с помощью Java, лучше использовать существующую клиентскую библиотеку HTTP, такую как HTTP-клиент Apache или HTTP-клиент Google

Основная клиентская / серверная связь с использованием UDP (датаграмма)

Client.java

```
import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}
```

В этом случае мы передаем адрес сервера через аргумент (args[0]). Порт, который мы используем, - 4160.

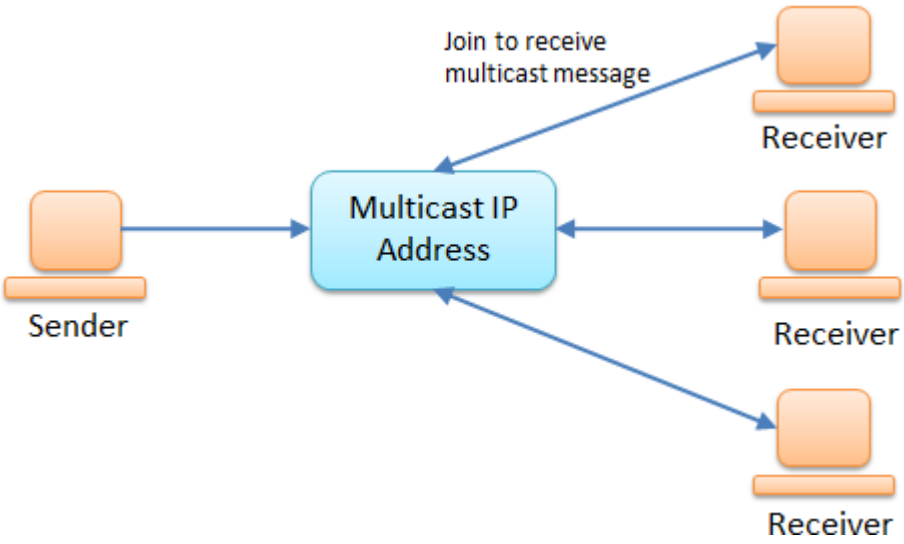
Server.java

```
import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}
```

На стороне сервера объявите DatagramSocket в том же порту, на который мы отправили наше сообщение (4160), и дождитесь ответа.

Multicasting

Многоадресная рассылка - это тип Datagram Socket. В отличие от обычных дейтаграмм, многоадресная рассылка не обрабатывает каждый клиент отдельно, а отправляет его на один IP-адрес, и все подписанные клиенты получат сообщение.



Пример кода для серверной части:

```
public class Server {

    private DatagramSocket serverSocket;

    private String ip;

    private int port;

    public Server(String ip, int port) throws SocketException, IOException{
        this.ip = ip;
        this.port = port;
        // socket used to send
        serverSocket = new DatagramSocket();
    }

    public void send() throws IOException{
        // make datagram packet
```

```
        byte[] message = ("Multicasting...").getBytes();
        DatagramPacket packet = new DatagramPacket(message, message.length,
            InetAddress.getByName(ip), port);
        // send packet
        serverSocket.send(packet);
    }

    public void close(){
        serverSocket.close();
    }
}
```

Пример кода для клиентской стороны:

```
public class Client {

    private MulticastSocket socket;

    public Client(String ip, int port) throws IOException {

        // important that this is a multicast socket
        socket = new MulticastSocket(port);

        // join by ip
        socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // make datagram packet to recieve
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // recieve the packet
        socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}
```

Код для запуска сервера:

```
public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
        server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Код для запуска клиента:

```
public static void main(String[] args) {
    try {
```

```
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Client client = new Client(ip, port);
        client.printMessage();
        client.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Сначала запустите клиент: клиент должен подписаться на IP-адрес, прежде чем он сможет начать получать какие-либо пакеты. Если вы запустите сервер и вызовите метод send() , а затем создайте клиент (& call printMessage()). Ничего не произойдет, потому что клиент подключился после отправки сообщения.

Временное отключение проверки SSL (для целей тестирования)

Иногда в среде разработки или тестирования цепочка сертификатов SSL, возможно, не была полностью установлена (пока).

Чтобы продолжить разработку и тестирование, вы можете отключить проверку SSL программно, установив «доверительный» доверительный менеджер:

```
try {
   // Create a trust manager that does not validate certificate chains
   TrustManager[] trustAllCerts = new TrustManager[] {
      new X509TrustManager() {
       public X509Certificate[] getAcceptedIssuers() {
           return null;
       }
       public void checkClientTrusted(X509Certificate[] certs, String authType) {
       }
       public void checkServerTrusted(X509Certificate[] certs, String authType) {
       }
      }
   };

   // Install the all-trusting trust manager
   SSLContext sc = SSLContext.getInstance("SSL");
   sc.init(null, trustAllCerts, new java.security.SecureRandom());
   HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

   // Create all-trusting host name verifier
   HostnameVerifier allHostsValid = new HostnameVerifier() {
       public boolean verify(String hostname, SSLSession session) {
           return true;
       }
   };

   // Install the all-trusting host verifier
   HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

Загрузка файла с использованием канала

Если файл уже существует, он будет перезаписан!

```
String fileName     = "file.zip";                  // name of the file
String urlToGetFrom = "http://www.mywebsite.com/"; // URL to get it from
String pathToSaveTo = "C:\\Users\\user\\";         // where to put it

//If the file already exists, it will be overwritten!

//Opening OutputStream to the destination file
try (ReadableByteChannel rbc =
      Channels.newChannel(new URL(urlToGetFrom + fileName).openStream()) ) {
    try ( FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel(); ) {
      channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* Output directory not found */ }
    catch (IOException e)           { /* File IO error */ }
}
catch (MalformedURLException e)     { /* URL is malformed */ }
catch (IOException e)               { /* IO error connecting to website */ }
```

Заметки

- Не оставляйте блоки захвата пустыми!
- В случае ошибки проверьте, существует ли удаленный файл
- Это операция блокировки, может занять много времени с большими файлами

Прочитайте сетей онлайн: https://riptutorial.com/ru/java/topic/149/сетей

глава 158: сканер

Синтаксис

- Сканер сканера = новый сканер (источник источника);
- Сканер сканера = новый сканер (System.in);

параметры

параметр	подробности
Источник	Источник может быть либо одним из String, File или любого типа InputStream

замечания

Класс Scanner был введен в Java 5. Метод reset() был добавлен в Java 6, а в Java 7 было добавлено несколько новых конструкторов для взаимодействия с (тогда) новым интерфейсом Path .

Examples

Чтение системного входа с помощью сканера

```
Scanner scanner = new Scanner(System.in); //Scanner obj to read System input
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // reading one line of input
    if (input.matches("\\s+"))         // if it matches spaces/tabs, stop reading
        break;
    inputTaken += input + " ";
}
System.out.println(inputTaken);
```

Объект сканера инициализируется для чтения ввода с клавиатуры. Таким образом, для нижнего входа от брелка он будет выводить вывод как Reading from keyboard

```
Reading
from
keyboard
  //space
```

Чтение ввода файла с помощью сканера

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("Names.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
} finally {
    if (scanner != null)
```

```
        scanner.close();
}
```

Здесь объект Scanner создается путем передачи объекта File содержащего имя текстового файла в качестве входного. Этот текстовый файл будет открыт объектом File и будет считываться с помощью объекта сканера в следующих строках. scanner.hasNext() проверит, есть ли следующая строка данных в текстовом файле. Объединение , что с в while цикла позволит перебирать все строки данных в Names.txt файл. Для извлечения самих данных мы можем использовать такие методы, как nextLine() , nextInt() , nextBoolean() и т. Д. В приведенном выше примере используется scanner.nextLine() . nextLine() относится к следующей строке в текстовом файле, и объединение ее со scanner объектом позволяет распечатать содержимое строки. Чтобы закрыть объект сканера, вы должны использовать .close() .

Используя попытку с ресурсами (начиная с Java 7), вышеупомянутый код можно написать элегантно, как показано ниже.

```
try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}
```

Прочитайте весь ввод в виде строки с помощью сканера

Вы можете использовать Scanner для чтения всего текста на входе в виде строки, используя \Z (весь ввод) в качестве разделителя. Например, это можно использовать для чтения всего текста в текстовом файле в одной строке:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Помните, что вам придется закрыть сканер, а также поймать IoException это может вызвать, как описано в примере Чтение файла с помощью Scanner .

Использование пользовательских разделителей

Вы можете использовать пользовательские разделители (регулярные выражения) со Сканером, с .useDelimiter(",") , чтобы определить, как считывается ввод. Это работает аналогично String.split(...) . Например, вы можете использовать Scanner для чтения из списка значений, разделенных запятыми, в строке String:

```
Scanner scanner = null;
try{
    scanner = new Scanner("i,like,unicorns").useDelimiter(",");;
    while(scanner.hasNext()){
        System.out.println(scanner.next());
    }
}catch(Exception e){
    e.printStackTrace();
}finally{
    if (scanner != null)
        scanner.close();
}
```

Это позволит вам читать каждый элемент на входе индивидуально. Обратите внимание, что вы **не** должны использовать это для анализа данных CSV, вместо этого используйте правильную библиотеку парсеров CSV, см. Парсер CSV для Java для других возможностей.

Общая схема, которая чаще всего задается в задачах

Ниже приведен пример того, как правильно использовать класс java.util.Scanner для интерактивного чтения пользовательского ввода из System.in (иногда называемого stdin , особенно на языках C, C ++ и других языках, а также в Unix и Linux). Он идиоматически демонстрирует наиболее распространенные вещи, которые требуются.

```
package com.stackoverflow.scanner;

import javax.annotation.Nonnull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/])\\d{2}\\1\\d{2}"); //
http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands
to exit %s", EXIT_COMMANDS);
    }

    /**
     * Using exceptions to control execution flow is always bad.
     * That is why this is encapsulated in a method, this is done this
     * way specifically so as not to introduce any external libraries
     * so that this is a completely self contained example.
     * @param s possible url
     * @return true if s represents a valid url, false otherwise
     */
    private static boolean isValidURL(@Nonnull final String s)
    {
        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@Nonnull final String format, @Nonnull final Object... args)
    {
        System.out.println(format(format, args));
    }

    public static void main(final String[] args)
    {
        final Scanner sis = new Scanner(System.in);
```

```
        output(HELP_MESSAGE);
        while (sis.hasNext())
        {
            if (sis.hasNextInt())
            {
                final int next = sis.nextInt();
                output("You entered an Integer = %d", next);
            }
            else if (sis.hasNextLong())
            {
                final long next = sis.nextLong();
                output("You entered a Long = %d", next);
            }
            else if (sis.hasNextDouble())
            {
                final double next = sis.nextDouble();
                output("You entered a Double = %f", next);
            }
            else if (sis.hasNext("\\d+"))
            {
                final BigInteger next = sis.nextBigInteger();
                output("You entered a BigInteger = %s", next);
            }
            else if (sis.hasNextBoolean())
            {
                final boolean next = sis.nextBoolean();
                output("You entered a Boolean representation = %s", next);
            }
            else if (sis.hasNext(DATE_PATTERN))
            {
                final String next = sis.next(DATE_PATTERN);
                output("You entered a Date representation = %s", next);
            }
            else // unclassified
            {
                final String next = sis.next();
                if (isValidURL(next))
                {
                    output("You entered a valid URL = %s", next);
                }
                else
                {
                    if (EXIT_COMMANDS.contains(next))
                    {
                        output("Exit command %s issued, exiting!", next);
                        break;
                    }
                    else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                    else { output("You entered an unclassified String = %s", next); }
                }
            }
        }
        /*
           This will close the underlying Readable, in this case System.in, and free those
resources.
           You will not be to read from System.in anymore after this you call .close().
           If you wanted to use System.in for something else, then don't close the Scanner.
        */
        sis.close();
        System.exit(0);
    }
```

```
}
```

Чтение int из командной строки

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

Если вы хотите прочитать int из командной строки, просто используйте этот фрагмент. Прежде всего, вам нужно создать объект Scanner, который будет прослушивать System.in, который по умолчанию является командной строкой, когда вы запускаете программу из командной строки. После этого с помощью объекта Scanner вы читаете первый int, который пользователь переходит в командную строку и сохраняет его в номере переменной. Теперь вы можете делать все, что захотите, с сохраненным int.

Аккуратно закрыть сканер

может случиться, что вы используете сканер с параметром System.in как для конструктора, тогда вам нужно знать, что закрытие сканера закроет InputStream, давая в следующем, чтобы каждый попытался прочитать ввод на этом (или любой другой объект сканера) будет вызывать java.util.NoSuchElementException или java.lang.IllegalStateException

пример:

```
    Scanner sc1 = new Scanner(System.in);
    Scanner sc2 = new Scanner(System.in);
    int x1 = sc1.nextInt();
    sc1.close();
    // java.util.NoSuchElementException
    int x2 = sc2.nextInt();
    // java.lang.IllegalStateException
    x2 = sc1.nextInt();
```

Прочитайте сканер онлайн: https://riptutorial.com/ru/java/topic/551/сканер

глава 159: Служба печати Java

Вступление

API службы печати Java предоставляет функциональные возможности для обнаружения служб печати и отправки запросов на печать для них.

Он включает в себя расширяемые атрибуты печати на основе стандартных атрибутов, указанных в протоколе IP Printing (IPP) 1.1, из спецификации IETF, RFC 2911 .

Examples

Обнаружение доступных служб печати

Чтобы обнаружить все доступные службы печати, мы можем использовать класс PrintServiceLookup . Давайте посмотрим, как:

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (Printservice printService : allPrintServices) {
            System.out.println("Print service name: " + printService.getName());
        }
    }

}
```

Эта программа, выполняемая в среде Windows, напечатает что-то вроде этого:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Обнаружение службы печати по умолчанию

Чтобы обнаружить службу печати по умолчанию, мы можем использовать класс PrintServiceLookup . Посмотрим, как:

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
```

```
        System.out.println("Default print service name: " + defaultPrintService.getName());
    }

}
```

Создание задания печати из службы печати

Задание печати - это запрос на печать чего-либо в конкретной службе печати. Он состоит, в основном, путем:

- данные, которые будут напечатаны (см. « Создание документа, который будет напечатан»)
- набор атрибутов

После выбора правильного экземпляра службы печати мы можем запросить создание задания на печать:

```
DocPrintJob printJob = printService.createPrintJob();
```

Интерфейс DocPrintJob предоставляет нам метод print :

```
printJob.print(doc, pras);
```

doc аргумент Doc : данные, которые будут напечатаны.

И pras аргумент является PrintRequestAttributeSet интерфейс: набор PrintRequestAttribute . Являются примерами атрибутов запроса печати:

- количество копий (1, 2 и т. д.),
- ориентация (портрет или пейзаж)
- хромкость (монохромный, цветной)
- качество (черновик, нормальный, высокий)
- стороны (односторонние, двухсторонние и т. д.)
- и так далее...

Метод печати может PrintException .

Создание документа, который будет распечатан

Doc - это интерфейс, а API службы печати Java предоставляет простую реализацию под названием SimpleDoc .

Каждый экземпляр Doc в основном состоит из двух аспектов:

- сам контент данных печати (электронная почта, изображение, документ и т. д.),
- формат данных печати, называемый DocFlavor (тип MIME + класс представления).

Перед созданием объекта Doc нам нужно загрузить наш документ откуда-то. В этом примере мы загрузим конкретный файл с диска:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

Итак, теперь нам нужно выбрать DocFlavor который соответствует нашему контенту. Класс DocFlavor содержит кучу констант для представления наиболее обычных типов данных. INPUT_STREAM.PDF :

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Теперь мы можем создать новый экземпляр SimpleDoc :

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

Объект doc теперь можно отправить в запрос на задание на печать (см. « Создание задания печати из службы печати»).

Определение атрибутов запроса печати

Иногда нам нужно определить некоторые аспекты запроса на печать. Мы будем называть их *атрибутом* .

Являются примерами атрибутов запроса печати:

- количество копий (1, 2 и т. д.),
- ориентация (портрет или пейзаж)
- хромкость (монохромный, цветной)
- качество (черновик, нормальный, высокий)
- стороны (односторонние, двухсторонние и т. д.)
- и так далее...

Прежде чем выбрать один из них и какое значение у каждого будет иметь, сначала нам нужно создать набор атрибутов:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Теперь мы можем добавить их. Вот некоторые примеры:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

Объект pras теперь можно отправить на запрос задания на печать (см. « Создание задания печати из службы печати»).

Прослушивание статуса запроса на печать

Для большинства клиентов печати чрезвычайно полезно знать, закончилось или не выполнено задание на печать.

API службы печати Java предоставляет некоторые функции для получения информации об этих сценариях. Все, что нам нужно сделать, это:

- обеспечить реализацию интерфейса PrintJobListener и
- зарегистрируйте эту реализацию на задании печати.

Когда состояние задания печати изменится, мы будем уведомлены. Мы можем сделать что угодно, например:

- обновить пользовательский интерфейс,
- начать другой бизнес-процесс,
- записывать что-то в базе данных,
- или просто запишите его.

В приведенном ниже примере мы записываем каждое изменение статуса задания на печать:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobListener;

public class LoggerPrintJobListener implements PrintJobListener {

    // Your favorite Logger class goes here!
```

```
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);


    public void printDataTransferCompleted(PrintJobEvent pje) {
        LOG.info("Print data transfer completed ;) ");
    }

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }

    public void printJobCanceled(PrintJobEvent pje) {
        LOG.info("Print job canceled :| ");
    }

    public void printJobNoMoreEvents(PrintJobEvent pje) {
        LOG.info("No more events to the job ");
    }

    public void printJobRequiresAttention(PrintJobEvent pje) {
        LOG.info("Print job requires attention :O ");
    }
}
```

Наконец, мы можем добавить нашу реализацию приемника задания печати на задание печати перед самим запросом на печать следующим образом:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

Аргумент *PrintJobEvent pje*

Обратите внимание, что каждый метод имеет аргумент PrintJobEvent pje . Мы не используем его в этом примере для простоты, но вы можете использовать его для изучения состояния. Например:

```
pje.getPrintJob().getAttributes();
```

PrintJobAttributeSet экземпляр объекта PrintJobAttributeSet и вы можете запускать их по-каждому.

Другой способ достичь той же цели

Другим вариантом достижения той же цели является расширение класса PrintJobAdapter , как сказано в названии, является адаптером для PrintJobListener . Внедряя интерфейс, мы обязательно должны реализовать их все. Преимуществом этого метода является то, что нам нужно переопределить только те методы, которые мы хотим. Давайте посмотрим, как это работает:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {
```

```
    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }
}
```

Обратите внимание, что мы переопределяем только некоторые конкретные методы.

Как и в примере реализации интерфейса PrintJobListener , мы добавляем слушателя к заданию на печать перед отправкой его на печать:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Прочитайте Служба печати Java онлайн: https://riptutorial.com/ru/java/topic/10178/служба-печати-java

глава 160: Создание изображений программно

замечания

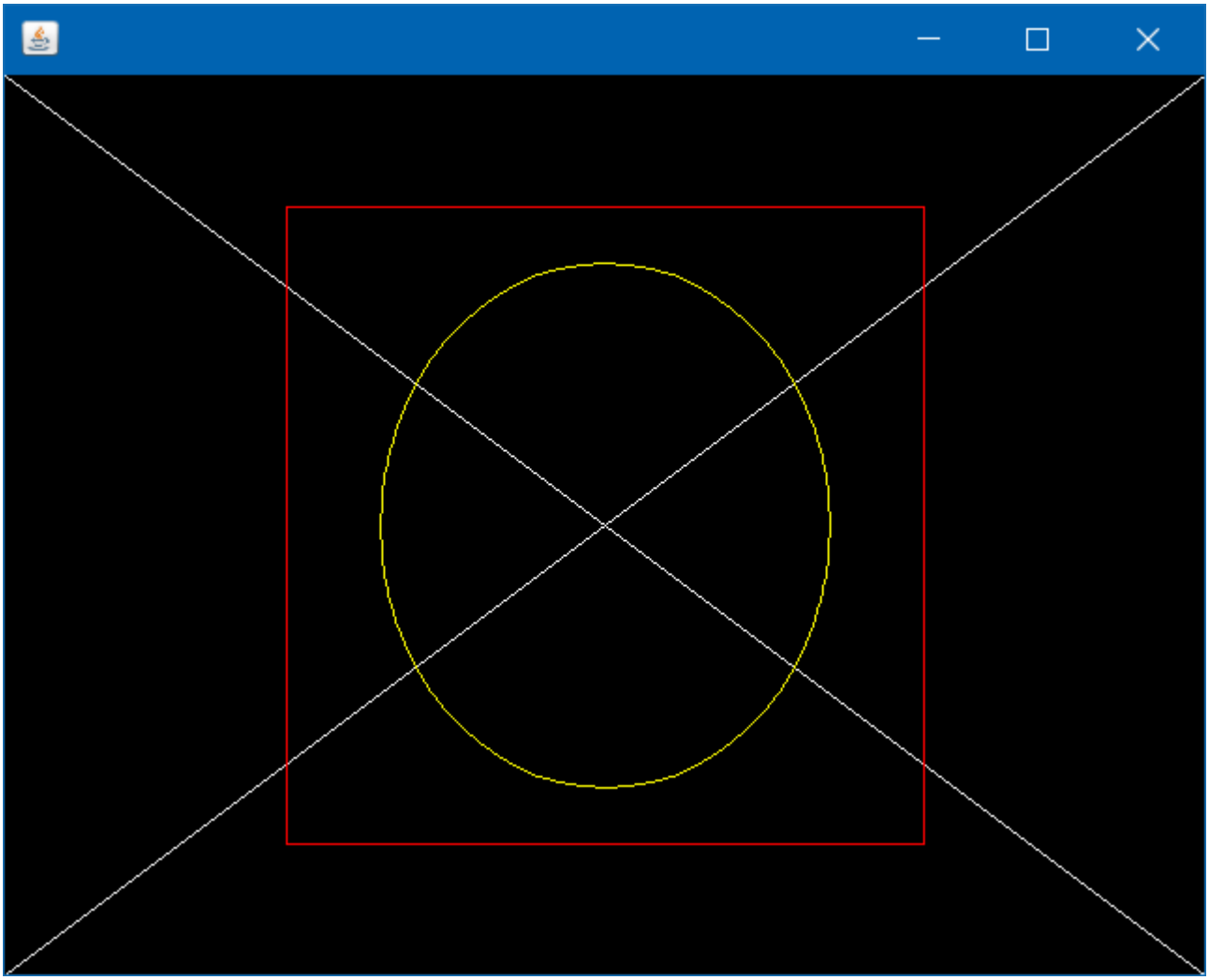
`BufferedImage.getGraphics()` всегда возвращает `Graphics2D` .

Использование `VolatileImage` может значительно улучшить скорость операций рисования, но также имеет свои недостатки: его содержимое может быть потеряно в любой момент, и их, возможно, придется перерисовать с нуля.

Examples

Создание простого образа программно и отображение его

```
class ImageCreationExample {

    static Image createSampleImage() {
        // instantiate a new BufferedImage (subclass of Image) instance
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);

        //draw something on the image
        paintOnImage(img);

        return img;
    }

    static void paintOnImage(BufferedImage img) {
        // get a drawable Graphics2D (subclass of Graphics) object
        Graphics2D g2d = (Graphics2D) img.getGraphics();

        // some sample drawing
        g2d.setColor(Color.BLACK);
        g2d.fillRect(0, 0, 640, 480);
        g2d.setColor(Color.WHITE);
        g2d.drawLine(0, 0, 640, 480);
        g2d.drawLine(0, 480, 640, 0);
        g2d.setColor(Color.YELLOW);
        g2d.drawOval(200, 100, 240, 280);
        g2d.setColor(Color.RED);
        g2d.drawRect(150, 70, 340, 340);

        // drawing on images can be very memory-consuming
        // so it's better to free resources early
        // it's not necessary, though
        g2d.dispose();
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Image img = createSampleImage();
        ImageIcon icon = new ImageIcon(img);
        frame.add(new JLabel(icon));
        frame.pack();
        frame.setVisible(true);
    }
}
```



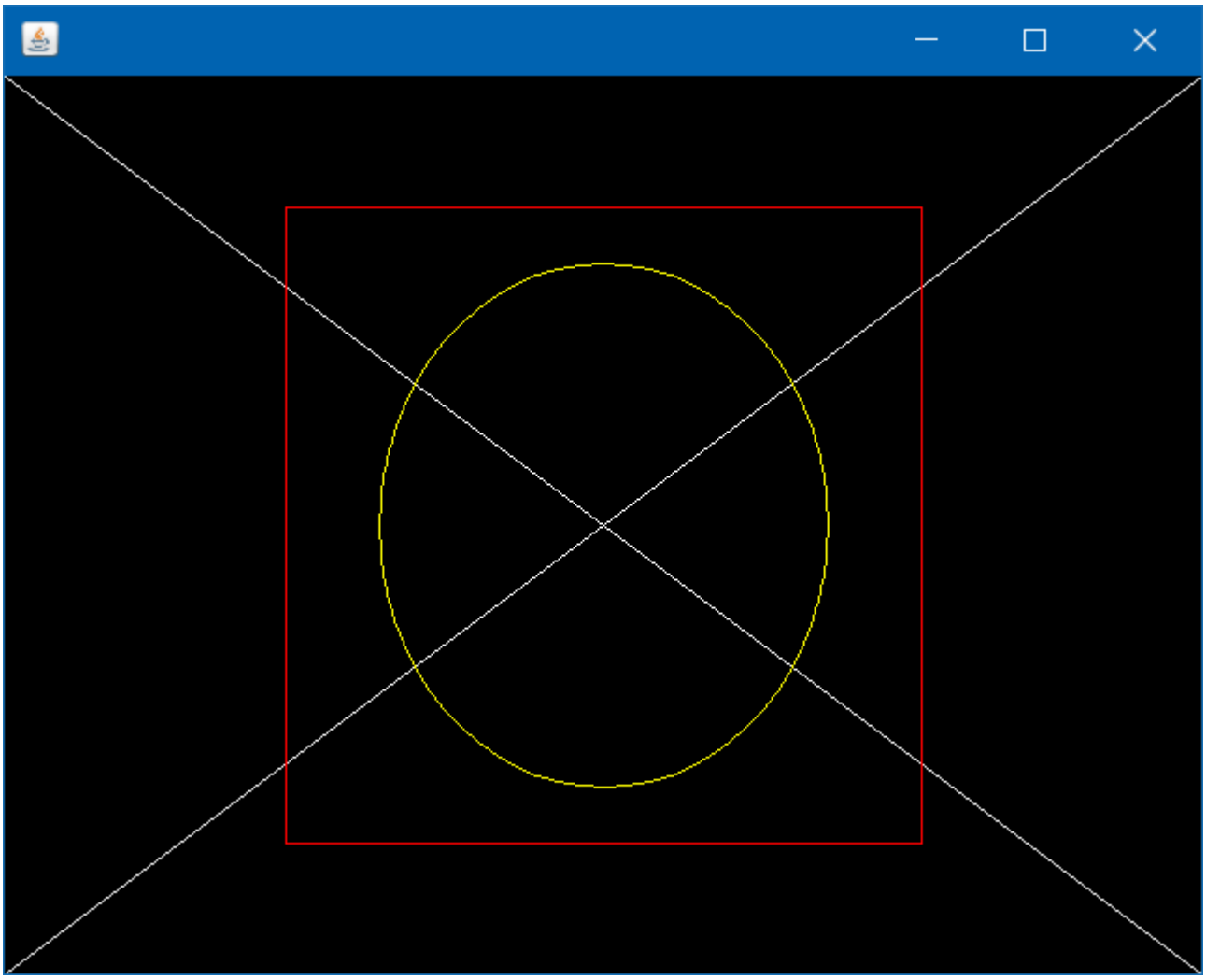
Сохранить изображение на диск

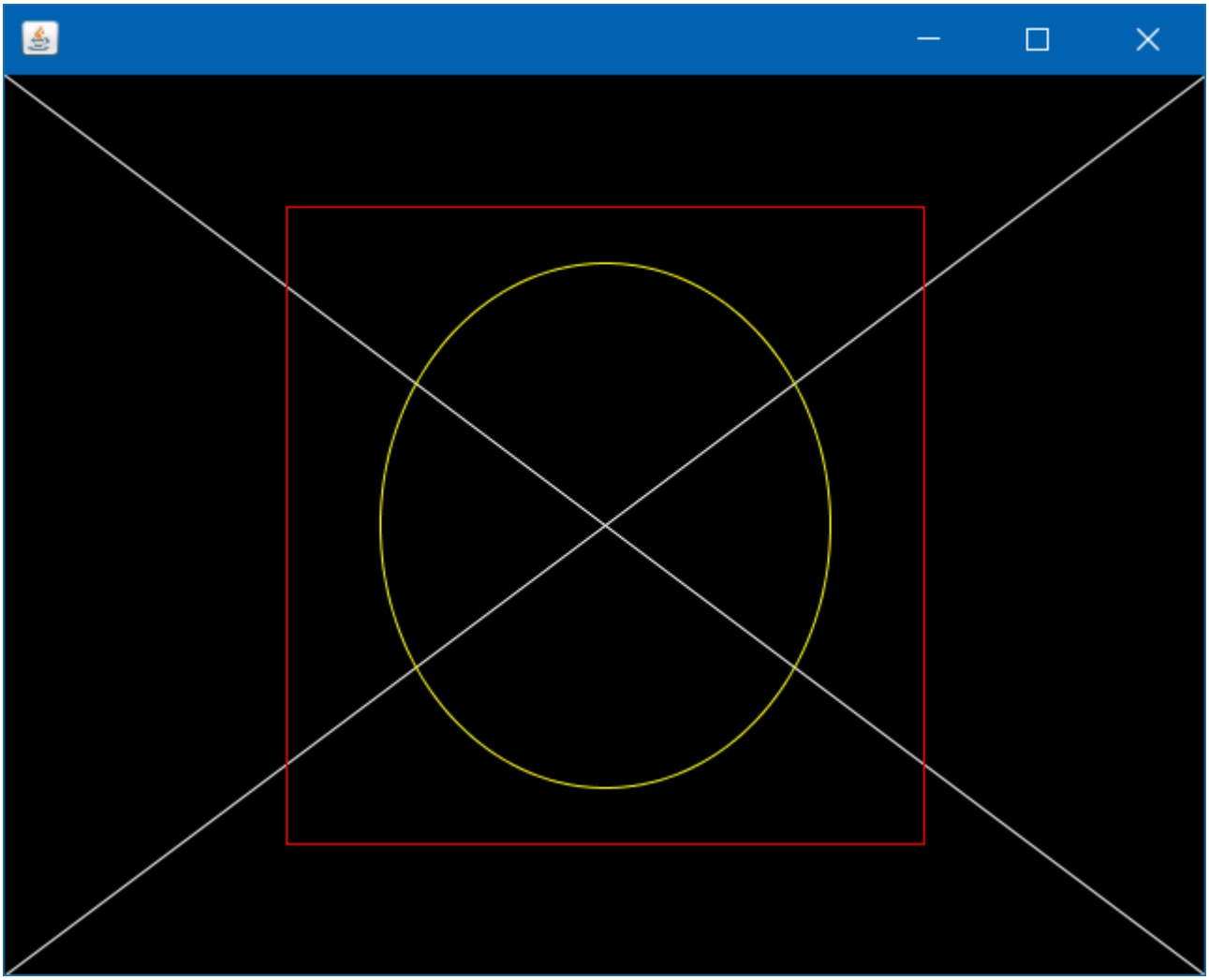
```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it"
example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

Определение качества рендеринга изображений

```
static void setupQualityHigh(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
    // many other RenderingHints KEY/VALUE pairs to specify
}


static void setupQualityLow(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);
}
```

Сравнение рендеринга качества КАЧЕСТВА и SPEED изображения образца:





Создание образа с помощью класса BufferedImage

```
int width = 256; //in pixels
int height = 256; //in pixels
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
//BufferedImage.TYPE_4BYTE_ABGR - store RGB color and visibility (alpha), see javadoc for more
info

Graphics g = image.createGraphics();

//draw whatever you like, like you would in a drawComponent(Graphics g) method in an UI
application
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //dispose graphics objects when they are no longer needed

//now image has programmatically generated content, you can use it in graphics.drawImage() to
draw it somewhere else
//or just simply save it to a file
ImageIO.write(image, "png", new File("myimage.png"));
```

Выход:

Редактирование и повторное использование изображения с помощью BufferedImage

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //read existing file

//modify it
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//now create a new image
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//and draw the old one on it, 16 times
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //add some nice border
g.dispose(); //and done

ImageIO.write(cats, "png", new File("cats.png"));
```

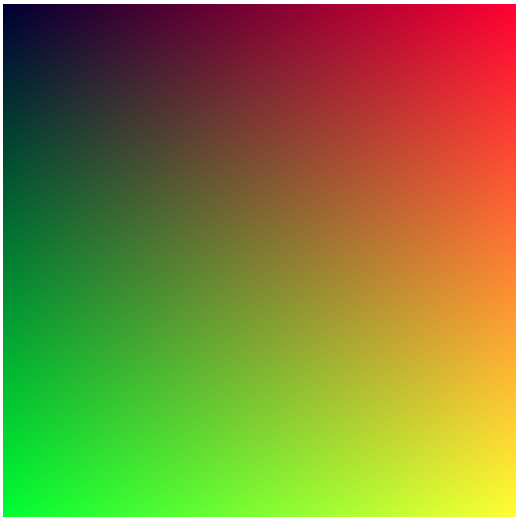
Оригинальный файл кота:

Произведенный файл:

Настройка цвета отдельного пикселя в BufferedImage

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

Выход:



Как масштабировать BufferedImage

```
 /**
 * Resizes an image using a Graphics2D object backed by a BufferedImage.
 * @param srcImg - source image to scale
 * @param w - desired width
```

```
 * @param h - desired height
 * @return - the new resized image
 */
private BufferedImage getScaledImage(Image srcImg, int w, int h){

    //Create a new image with good size that contains or might contain arbitrary alpha values
between and including 0.0 and 1.0.
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);

    //Create a device-independant object to draw the resized image
    Graphics2D g2 = resizedImg.createGraphics();

    //This could be changed, Cf. http://stackoverflow.com/documentation/java/5482/creating-
images-programmatically/19498/specifying-image-rendering-quality
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //Finally draw the source image in the Graphics2D with the desired size.
    g2.drawImage(srcImg, 0, 0, w, h, null);

    //Disposes of this graphics context and releases any system resources that it is using
    g2.dispose();

    //Return the image used to create the Graphics2D
    return resizedImg;
}
```

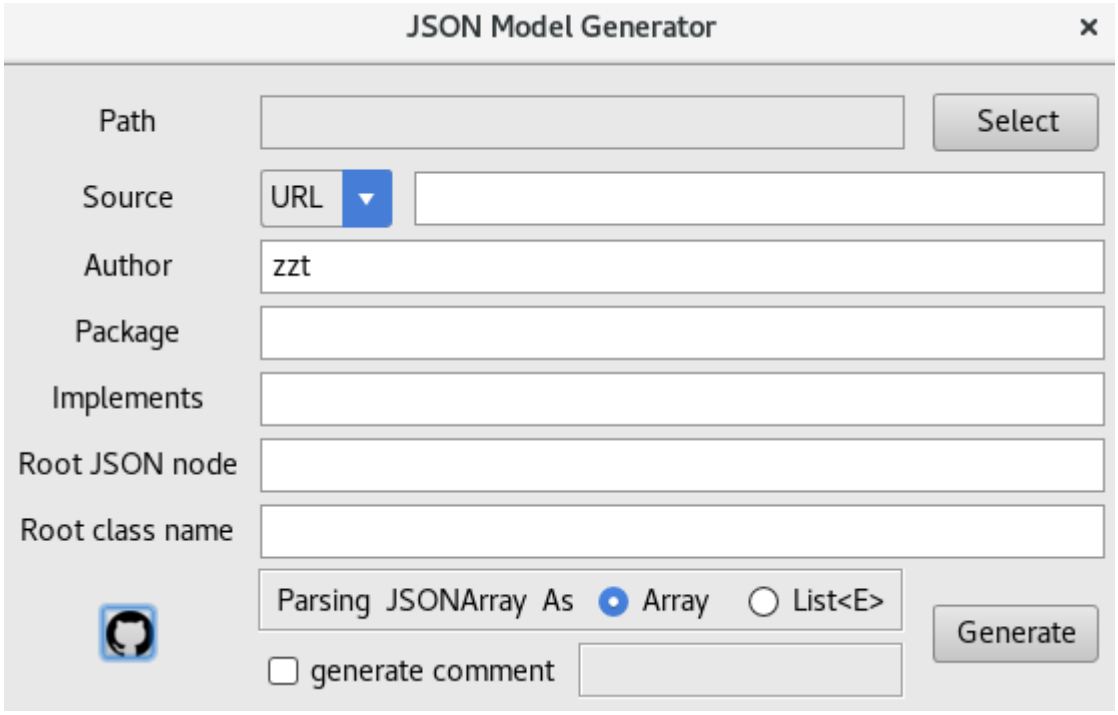
Прочитайте Создание изображений программно онлайн: https://riptutorial.com/ru/java/topic/5482/создание-изображений-программно

глава 161: Создание кода Java

Examples

Создание POJO от JSON

- Установите плагин JSON Model Genrator Intellij, выполнив поиск в настройках Intellij.
- Запустите плагин из раздела «Инструменты»
- Введите поле пользовательского интерфейса, как показано ниже («Путь», «Источник», «Пакет»):



- Нажмите кнопку «Создать», и все будет сделано.

Прочитайте Создание кода Java онлайн: https://riptutorial.com/ru/java/topic/9400/создание-кода-java

глава 162: Сокеты Java

Вступление

Сокеты - это низкоуровневый сетевой интерфейс, который помогает в создании соединения между двумя программами, главным образом клиентами, которые могут работать или не работать на одном компьютере.

Программирование сокетов - одна из наиболее широко используемых сетевых концепций.

замечания

Существует два типа трафика интернет-протокола:
1. Протокол TCP-Transmission Control 2. UDP - протокол дейтаграмм пользователей

TCP - это протокол, ориентированный на соединение.
UDP - протокол без установления соединения.

TCP подходит для приложений, требующих высокой надежности, а время передачи относительно менее критично.

UDP подходит для приложений, которым требуется быстрая и эффективная передача, например игр. Безгражданность UDP также полезна для серверов, которые отвечают на небольшие запросы от огромного числа клиентов.

В более простых словах -
Используйте TCP, когда вы не можете позволить себе потерять данные, и когда время для отправки и получения данных не имеет значения. Используйте UDP, когда вы не можете позволить себе потерять время и когда потеря данных не имеет значения.

Существует абсолютная гарантия того, что переданные данные остаются нетронутыми и поступают в том же порядке, в каком они были отправлены в случае TCP.
в то время как нет никакой гарантии, что отправленные сообщения или пакеты достигнут вообще в UDP.

Examples

Простой сервер эхо-ответа TCP

Наш сервер эхо-ответа TCP будет отдельным потоком. Это просто, как начало. Он будет просто отсылать все, что вы отправляете, но в заглавной форме.

```
public class CAPECHOServer extends Thread{

    // This class implements server sockets. A server socket waits for requests to come
    // in over the network only when it is allowed through the local firewall
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // Create a new Server on specified port.
            serverSocket = new ServerSocket(port);
            // SoTimeout is basiacally the socket timeout.
            // timeout is the time until socket timeout in milliseconds
            serverSocket.setSoTimeout(timeout);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

```
    @Override
    public void run(){
        try {
            // We want the server to continuously accept connections
            while(!Thread.interrupted()){

            }
            // Close the server once done.
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Теперь принимать соединения. Давайте обновим метод run.

```
@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // Log with the port number and machine ip
            Logger.getLogger((this.getClass().getName())).log(Level.INFO, "Listening for
Clients at {0} on {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept();  // Accept client conncetion
            // Now get DataInputStream and DataOutputStreams
            DataInputStream istream = new DataInputStream(client.getInputStream()); // From
client's input stream
            DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
            // Important Note
            /*
                The server's input is the client's output
                The client's input is the server's output
            */
            // Send a welcome message
            ostream.writeUTF("Welcome!");

            // Close the connection
            istream.close();
            ostream.close();
            client.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Close the server once done

    try {
        serverSocket.close();
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Теперь, если вы можете открыть telnet и попробовать подключиться, вы увидите приветственное сообщение.

Вы должны подключиться к указанному порту и IP-адресу.

Вы должны увидеть результат, подобный этому:

```
Welcome!

Connection to host lost.
```

Ну, связь была потеряна, потому что мы ее прекратили. Иногда нам приходилось программировать наш собственный клиент TCP. В этом случае нам нужен клиент для запроса ввода от пользователя и отправки его по сети, получения заглавного ввода.

Если сервер сначала отправляет данные, клиент сначала должен прочитать данные.

```
public class CAPECHOClient extends Thread{

Socket server;
Scanner key; // Scanner for input

    public CAPECHOClient(String ip, int port){
        try {
            server = new Socket(ip, port);
            key = new Scanner(System.in);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        DataInputStream istream = null;
        DataOutputStream ostream = null;
        try {
            istream = new DataInputStream(server.getInputStream()); // Familiar lines
            ostream = new DataOutputStream(server.getOutputStream());
            System.out.println(istream.readUTF());  // Print what the server sends
            System.out.print(">");
            String tosend = key.nextLine();
            ostream.writeUTF(tosend);   // Send whatever the user typed to the server
            System.out.println(istream.readUTF());  // Finally read what the server sends
before exiting.
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                istream.close();
                ostream.close();
                server.close();
            } catch (IOException ex) {
                Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Теперь обновите сервер

```
ostream.writeUTF("Welcome!");

String inString = istream.readUTF();    // Read what the user sent
String outString = inString.toUpperCase();  // Change it to caps
ostream.writeUTF(outString);
```

```
// Close the connection
istream.close();
```

И теперь запустите сервер и клиент, у вас должен быть выход, похожий на этот

```
Welcome!
>
```

Прочитайте Сокеты Java онлайн: https://riptutorial.com/ru/java/topic/9923/сокеты-java

глава 163: Списки

Вступление

Список представляет собой *упорядоченный* набор значений. В Java списки являются частью Framework коллекций Java . Списки реализуют интерфейс java.util.List , который расширяет java.util.Collection .

Синтаксис

- ls.add (элемент E); // Добавляет элемент
- ls.remove (элемент E); // Удаляет элемент
- for (элемент E: ls) {} // Итерации по каждому элементу
- ls.toArray (новая строка [ls.length]); // Преобразует список строк в массив строк
- ls.get (индекс int); // Возвращает элемент по указанному индексу.
- ls.set (индекс int, элемент E); // Заменяет элемент в указанной позиции.
- ls.isEmpty (); // Возвращает true, если массив не содержит элементов, иначе он возвращает false.
- ls.indexOf (Object o); // Возвращает индекс первого местоположения указанного элемента o или, если его нет, возвращает -1.
- ls.lastIndexOf (Object o); // Возвращает индекс последнего местоположения указанного элемента o или, если его нет, возвращает -1.
- ls.size (); // Возвращает количество элементов в Списке.

замечания

Список - это объект, в котором хранится упорядоченный набор значений. «Упорядочено» означает, что значения хранятся в определенном порядке - на первом месте появляется один элемент, второй - второй и т. Д. Отдельные значения обычно называются «элементами». Списки Java обычно предоставляют следующие возможности:

- Списки могут содержать ноль или несколько элементов.
- Списки могут содержать повторяющиеся значения. Другими словами, элемент можно вставлять в список более одного раза.
- Списки хранят свои элементы в определенном порядке, что означает, что на первом приходит один элемент, следующий - и т. Д.
- Каждый элемент имеет *индекс*, указывающий его позицию в списке. Первый элемент имеет индекс 0, следующий - индекс 1 и т. Д.
- Списки позволяют вставлять элементы в начале, в конце или в любом индексе в списке.
- Тестирование того, содержит ли список определенное значение, обычно означает проверку каждого элемента в списке. Это означает, что время выполнения этой проверки - O (n) , пропорциональное размеру списка.

Добавление значения в список в какой-то момент, кроме конца, приведет к перемещению всех следующих элементов «вниз» или «вправо». Другими словами, добавление элемента с индексом *n* перемещает элемент, который раньше имел индекс *n* для индекса *n + 1* , и так далее. Например:

```
List<String> list = new ArrayList<>();
list.add("world");
System.out.println(list.indexOf("world"));      // Prints "0"
// Inserting a new value at index 0 moves "world" to index 1
list.add(0, "Hello");
System.out.println(list.indexOf("world"));      // Prints "1"
System.out.println(list.indexOf("Hello"));      // Prints "0"
```

Examples

Сортировка общего списка

Класс Collections предлагает два стандартных статических метода для сортировки списка:

- sort(List<T> list) применимый к спискам, где T extends Comparable<? super T> , и
- sort(List<T> list, Comparator<? super T> c) применимый к спискам любого типа.

Применение первого требует внесения изменений в класс сортируемых элементов списка, что не всегда возможно. Это также может быть нежелательным, поскольку, несмотря на то, что он обеспечивает сортировку по умолчанию, другие заказы сортировки могут потребоваться в разных обстоятельствах, или сортировка - это просто одна задача.

У нас есть задача сортировки объектов, которые являются экземплярами следующего класса:

```
public class User {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }
}
```

Чтобы использовать Collections.sort(List<User> list) нам необходимо изменить класс User для реализации интерфейса Comparable . Например

```
public class User implements Comparable<User> {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }

    @Override
    /** The natural ordering for 'User' objects is by the 'id' field. */
    public int compareTo(User o) {
        return id.compareTo(o.id);
    }
}
```

(За исключением: многие стандартные классы Java , такие как String , Long , Integer реализовать Comparable интерфейс Это делает списки тех элементов , отсортированных по умолчанию, и упрощает реализацию. compare или compareTo в других классах.)

В приведенной выше модификации мы можем легко отсортировать список объектов User на основе *естественного упорядочения* классов. (В этом случае мы определили, что это упорядочение на основе значений id). Например:

```
List<User> users = Lists.newArrayList(
    new User(33L, "A"),
```

```
    new User(25L, "B"),
    new User(28L, ""));
Collections.sort(users);

System.out.print(users);
// [B:25, C:28, A:33]
```

Однако предположим, что мы хотели отсортировать объекты User по name а не по id . В качестве альтернативы предположим, что мы не смогли изменить класс, чтобы он реализовал Comparable .

Здесь метод sort с аргументом Comparator полезен:

```
Collections.sort(users, new Comparator<User>() {
    @Override
    /* Order two 'User' objects based on their names. */
    public int compare(User left, User right) {
        return left.username.compareTo(right.username);
    }
});
System.out.print(users);
// [A:33, B:25, C:28]
```

Java SE 8

В Java 8 вы можете использовать *лямбда* вместо анонимного класса. Последний сводится к однострочному:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Кроме того, там Java 8 добавляет метод sort по умолчанию в интерфейсе List , что упрощает сортировку еще больше.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Создание списка

Предоставление вашего списка типа

Для создания списка вам нужен тип (любой класс, например String). Это тип вашего List . List будет хранить объекты только указанного типа. Например:

```
List<String> strings;
```

Может хранить "string1" , "hello world!" , "goodbye" и т. д., но он не может хранить 9.2 , однако:

```
List<Double> doubles;
```

Может хранить 9.2 , но не "hello world!" ,

Инициализация вашего списка

Если вы попытаетесь добавить что-то в список выше, вы получите исключение NullPointerException, потому что strings и doubles равные **нулю** !

Существует два способа инициализации списка:

Вариант 1: используйте класс, который реализует List

List - это интерфейс, который означает, что у него нет конструктора, а скорее методов, которые

класс должен переопределить. ArrayList является наиболее часто используемым List , хотя LinkedList также распространен. Итак, мы инициализируем наш список следующим образом:

```
List<String> strings = new ArrayList<String>();
```

или же

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

Начиная с Java SE 7, вы можете использовать *алмазный оператор* :

```
List<String> strings = new ArrayList<>();
```

или же

```
List<String> strings = new LinkedList<>();
```

Вариант 2: используйте класс Collections

Класс Collections предоставляет два полезных метода для создания списков без переменной List :

- emptyList() : возвращает пустой список.
- singletonList(T) : создает список типов T и добавляет указанный элемент.

И метод, который использует существующий List для заполнения данных в:

- addAll(L, T...) : добавляет все указанные элементы в список, переданный в качестве первого параметра.

Примеры:

```
import java.util.List;
import java.util.Collections;

List<Integer> l = Collections.emptyList();
List<Integer> l1 = Collections.singletonList(42);
Collections.addAll(l1, 1, 2, 3);
```

Операции с позиционным доступом

API-интерфейс списка содержит восемь методов для операций позиционного доступа:

- add(T type)
  - add(int index, T type)
  - remove(Object o)
  - remove(int index)
  - get(int index)
  - set(int index, E element)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)

Итак, если у нас есть Список:

```
List<String> strings = new ArrayList<String>();
```

И мы хотели добавить строки «Привет мир!». и "Прощай мир!" к этому мы будем делать это как таковое:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

И наш список будет содержать два элемента. Теперь скажем, мы хотели добавить «Program start!». в **начале** списка. Мы сделали бы это следующим образом:

```
strings.add(0, "Program starting!");
```

ПРИМЕЧАНИЕ. Первый элемент равен 0.

Теперь, если мы хотим удалить «До свидания мир!» мы можем сделать это следующим образом:

```
strings.remove("Goodbye world!");
```

И если бы мы хотели удалить первую строку (которая в этом случае была бы «Program start!», Мы могли бы сделать это следующим образом:

```
strings.remove(0);
```

Замечания:

1. Добавление и удаление элементов списка изменяет список, и это может привести к исключению ConcurrentModificationException если список повторяется одновременно.
2. Добавление и удаление элементов может быть O(1) или O(N) зависимости от класса списка, используемого метода и того, добавляете / удаляете элемент в начале, в конце или в середине списка.

Чтобы получить элемент списка в указанной позиции, вы можете использовать E get(int index); метод API-интерфейса списка. Например:

```
strings.get(0);
```

вернет первый элемент списка.

Вы можете заменить любой элемент в указанной позиции, используя set(int index, E element); , Например:

```
strings.set(0,"This is a replacement");
```

Это установит строку «Это замена» в качестве первого элемента списка.

Примечание. Метод установки перезапишет элемент в позиции 0. Он не добавит новую строку в позицию 0 и перетащит старую в позицию 1.

int indexOf(Object o); возвращает позицию первого вхождения объекта, переданного в качестве аргумента. Если в списке нет вхождений объекта, тогда возвращается значение -1. В продолжение предыдущего примера, если вы вызываете:

```
strings.indexOf("This is a replacement")
```

ожидается, что 0 будет возвращено, поскольку мы устанавливаем строку «Это замена» в позиции 0 нашего списка. В случае, если в списке имеется более одного вхождения, когда int indexOf(Object o); называется, то, как упоминалось, будет возвращен индекс первого вхождения. int lastIndexOf(Object o) вы можете получить индекс последнего вхождения в списке. Поэтому, если добавить еще одну «Это замена»:

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

На этот раз 1 будет возвращен, а не 0;

Итерирование элементов в списке

Например, скажем, что у нас есть List of type String, который содержит четыре элемента: «hello», «how», «are», «you?»?

Лучший способ перебора каждого элемента - использовать для каждого цикла:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Что бы напечатать:

```
hello,
how
are
you?
```

Чтобы распечатать их все в одной строке, вы можете использовать StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Будет печать:

```
hello, how are you?
```

Кроме того, вы можете использовать индексацию элементов (как описано в разделе «Доступ к элементу в i-м индексе от ArrayList»), чтобы перебрать список. Предупреждение: этот подход неэффективен для связанных списков.

Удаление элементов из списка B, которые присутствуют в списке A

Давайте предположим , что у вас есть 2 Списки A и B, и вы хотите , чтобы удалить из **B** все элементы , которые вы имеете в **A** метод в данном случае является

```
 List.removeAll(Collection c);
```

#Пример:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));
    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);

    numbersB.removeAll(numbersA);
```

```
    System.out.println("B cleared: " + numbersB);
    }
```

это напечатает

```
A: [1, 3, 4, 7, 5, 2]

B: [13, 32, 533, 3, 4, 2]

B: [13, 32, 533]
```

Поиск общих элементов между двумя списками

Предположим, у вас есть два списка: A и B, и вам нужно найти элементы, которые существуют в обоих списках.

Вы можете сделать это, просто вызвав метод List.retainAll() .

Пример:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);

}
```

Преобразование списка целых чисел в список строк

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
    .collect(Collectors.toList());
```

То есть:

1. Создайте поток из списка
2. Сопоставьте каждый элемент с помощью Object::toString
3. Соберите значения String в List с помощью Collectors.toList()

Создание, добавление и удаление элемента из массива ArrayList

ArrayList является одной из встроенных структур данных в Java. Это динамический массив (где размер структуры данных, который не требуется сначала объявлять) для хранения элементов (объектов).

Он расширяет класс AbstractList и реализует интерфейс List . ArrayList может содержать повторяющиеся элементы, где он поддерживает порядок вставки. Следует отметить, что класс ArrayList не синхронизирован, поэтому следует проявлять осторожность при работе с параллелизмом

с ArrayList . ArrayList допускает произвольный доступ, потому что массив работает на основе индекса. Манипуляция медленнее в ArrayList из-за изменения, которое часто возникает, когда элемент удаляется из списка массивов.

ArrayList может быть создан следующим образом:

```
List<T> myArrayList = new ArrayList<>();
```

Где T (Generics) - это тип, который будет храниться внутри ArrayList .

Тип ArrayList может быть любым объектом. Тип не может быть примитивным типом (вместо этого используйте их *классы-оболочки*).

Чтобы добавить элемент в ArrayList , используйте метод add() :

```
myArrayList.add(element);
```

Или добавить элемент к определенному индексу:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

Чтобы удалить элемент из ArrayList , используйте метод remove() :

```
myArrayList.remove(element);
```

Или удалить элемент из определенного индекса:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Замена на месте элемента списка

Этот пример касается замены элемента List , гарантируя, что элемент замены находится в том же положении, что и заменяемый элемент.

Это можно сделать, используя следующие методы:

- set (индекс int, тип T)
- int indexOf (тип T)

Рассмотрим ArrayList содержащий элементы «Начало программы!», «Привет, мир!». и "Прощай мир!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Если нам известен индекс элемента, который мы хотим заменить, мы можем просто использовать set следующим образом:

```
strings.set(1, "Hi world");
```

Если мы не знаем индекс, мы можем сначала его найти. Например:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

Заметки:

1. Операция set не вызовет ConcurrentModificationException .
2. Операция set выполняется быстро (O(1)) для ArrayList но медленная (O(N)) для LinkedList .
3. Поиск indexOf в ArrayList или LinkedList медленный (O(N)).

Создание списка не подлежащих регистрации

Класс Collections позволяет сделать список немодифицируемым:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

Если вам нужен немодифицируемый список с одним элементом, который вы можете использовать:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Перемещение объектов в списке

Класс Collections позволяет перемещать объекты в списке с помощью различных методов (ls - это List):

Перемещение списка:

```
Collections.reverse(ls);
```

Поворот позиций элементов в списке

Для метода rotate требуется целочисленный аргумент. Это то, сколько мест перемещать по линии. Ниже приведен пример:

```
List<String> ls = new ArrayList<String>();
ls.add(" how");
ls.add(" are");
ls.add(" you?");
ls.add("hello,");
Collections.rotate(ls, 1);

for(String line : ls) System.out.print(line);
System.out.println();
```

Это будет печатать «привет, как дела?»

Перемешивание элементов в списке

Используя тот же список выше, мы можем перетасовать элементы в списке:

```
Collections.shuffle(ls);
```

Мы также можем дать ему объект java.util.Random, который он использует для случайного размещения объектов в точках:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

Классы, реализующие список - за и против

Интерфейс List реализуется различными классами. Каждый из них имеет свой собственный путь для реализации этого с различными стратегиями и предоставления различных плюсов и минусов.

Классы, реализующие List

Это все public классы в Java SE 8, которые реализуют интерфейс java.util.List :

1. **Абстрактные классы:**
   - AbstractList
   - AbstractSequentialList
2. **Бетонные классы:**
   - ArrayList
   - AttributeList
   - CopyOnWriteArrayList
   - LinkedList
   - RoleList
   - RoleUnresolvedList
   - стек
   - Вектор

Плюсы и минусы каждой реализации с точки зрения временной сложности

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList - это реализация интерфейса списка с изменяемым размером. Сохраняя список в массив, *ArrayList* предоставляет методы (в дополнение к методам, реализующим интерфейс *List*) для управления размером массива.

Инициализировать ArrayList из Integer с размером 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- PROS:

Операции size, isEmpty, ***get*** , ***set*** , iterator и listIterator выполняются в постоянное время. Таким образом, получение и установка каждого элемента списка имеют одинаковую *стоимость* :

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- CONS:

При реализации с массивом (статическая структура) добавление элементов по размеру массива имеет большую стоимость из-за того, что для всего массива необходимо выполнить новое распределение. Однако из документации :

> Операция add выполняется в режиме амортизированного постоянного времени, то есть добавление n элементов требует O (n) времени

Для удаления элемента требуется время O (n).

AttributeList

Прибытие

CopyOnWriteArrayList

Прибытие

LinkedList

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList реализуется двусвязным списком связанной структуры данных, которая состоит из набора последовательно связанных записей, называемых узлами.

Iitialize LinkedList из Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS:

Добавление или удаление элемента в начале списка или до конца имеет постоянное время.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONS: Из документации :

> Операции, которые индексируются в список, будут перемещаться по списку от начала или до конца, в зависимости от того, что ближе к указанному индексу.

Операции, такие как:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

RoleList

Прибытие

RoleUnresolvedList

Прибытие

стек

Прибытие

Вектор

Прибытие

Прочитайте Списки онлайн: https://riptutorial.com/ru/java/topic/2989/списки

глава 164: Список против SET

Вступление

Каковы различия между List и Set collection на верхнем уровне и как выбрать, когда использовать List в java и когда использовать Set in Java

Examples

Список против набора

```
import java.util.ArrayList;
```

import java.util.HashSet; import java.util.List; import java.util.Set;

public class SetAndListExample {public static void main (String [] args) {System.out.println (" Пример списка ....."); Список List = новый ArrayList (); list.add ("1"); list.add ("2"); list.add ("3"); list.add ("4"); list.add ("1");

```
    for (String temp : list){
        System.out.println(temp);
    }

    System.out.println("Set example .....");
    Set<String> set = new HashSet<String>();
    set.add("1");
    set.add("2");
    set.add("3");
    set.add("4");
    set.add("1");
    set.add("2");
    set.add("5");

    for (String temp : set){
        System.out.println(temp);
    }
}
```

}

Пример списка вывода ..... 1 2 3 4 1 Пример установки 3 2 10 5 4

Прочитайте Список против SET онлайн: https://riptutorial.com/ru/java/topic/10125/список-против-set

глава 165: Сравнение C ++

Вступление

Java и C ++ - это схожие языки. Этот раздел служит в качестве краткого справочного руководства для разработчиков Java и C ++.

замечания

Классы, определенные в других конструкциях

Определено в другом классе

C ++

Вложенный класс [ref] (требуется ссылка на класс включения)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Джава

[нестатический] Вложенный класс (он же внутренний класс или класс участника)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Статически определено в другом классе

C ++

Статический вложенный класс

```
class Outer {
   class Inner {
      ...
   };
};
```

Джава

Статический вложенный класс (aka Static Member Class) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
```

```
        ...
    }
}
```

Определено в рамках метода

(например, обработка событий)

C ++

Местный класс [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

См. Также Лямбда-выражения

Джава

Местный класс [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

Переопределение и перегрузка

Следующие пункты Overriding vs Overloading применяются как для C ++, так и для Java:

- Переопределенный метод имеет то же имя и аргументы, что и его базовый метод.
- Перегруженный метод имеет одно и то же имя, но разные аргументы и не полагается на наследование.
- Два метода с тем же именем и аргументами, но с другим типом возврата являются незаконными. См. Связанные вопросы Stackoverflow о «перегрузке с помощью другого типа возврата в Java» - вопрос 1 ; вопрос 2

Полиморфизм

Полиморфизм - это способность объектов разных классов, связанных с наследованием, реагировать по-разному на один и тот же вызов метода. Вот пример:

- базовый класс Форма с областью как абстрактный метод
- два производных класса, квадрат и круг, методы области реализации
- Отображаются опорные точки формы к квадрату и области.

В C ++ полиморфизм активируется виртуальными методами. В Java методы по умолчанию являются виртуальными.

Порядок строительства / уничтожения

Очистка объектов

В C ++ рекомендуется объявить деструктор виртуальным, чтобы гарантировать, что деструктор подкласса будет вызываться, если указатель базового класса будет удален.

В Java метод finalize аналогичен деструктору в C ++; однако финализаторы непредсказуемы (они полагаются на GC). Лучшая практика - используйте метод «закрыть» для явной очистки.

```
protected void close() {
    try {
       // do subclass cleanup
    }
    finally {
       isClosed = true;
       super.close();
    }
}

protected void finalize() {
    try {
       if(!isClosed) close();
    }
    finally {
       super.finalize();
    }
}
```

Абстрактные методы и классы

концепция	C ++	Джава
Абстрактный метод объявлен без реализации	чистый виртуальный метод virtual void eat(void) = 0;	абстрактный метод abstract void draw();
Абстрактный класс не может быть	не может быть создан; имеет по крайней мере 1 чистый виртуальный метод class AB {public: virtual void f() = 0;};	не может быть создан; могут иметь не абстрактные методы abstract class GraphicObject {}
Интерфейс нет полей экземпляра	нет «интерфейса», но может имитировать интерфейс Java с возможностями абстрактного класса	очень похож на абстрактный класс, но 1) поддерживает множественное наследование; 2) никаких полей экземпляра interface TestInterface {}

Модификаторы доступности

Модификатор	C ++	Джава
Общественность - доступна всем	*нет специальных заметок*	*нет специальных заметок*
Защищено - доступно по подклассам	также доступны друзьям	также доступны в одном пакете

Модификатор	C ++	Джава
Частный - доступный для участников	также доступны друзьям	*нет специальных заметок*
дефолт	class default - частный; struct default - общедоступный	доступный для всех классов в одном пакете
Другой	Друг - способ предоставить доступ к частным и защищенным членам без наследования (см. Ниже)	

Пример использования C ++

```
class Node {
  private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

Проблема ужасного алмаза

> Проблема с алмазом - это двусмысленность, возникающая, когда два класса B и C наследуют от A, а класс D наследуется как от B, так и от C. Если в A существует метод, который B и C переопределены, а D не переопределяет его, тогда какая версия метода наследует D: B или C? (из Википедии)

Хотя C ++ всегда была восприимчива к проблеме алмаза, Java была восприимчивой до Java 8. Первоначально Java не поддерживала множественное наследование, но с появлением методов интерфейса по умолчанию классы Java не могут наследовать «реализацию» из более чем одного класса ,

Класс java.lang.Object

В Java все классы наследуют, неявно или явно, из класса Object. Любая ссылка Java может быть перенесена в тип объекта.

У C ++ нет сопоставимого класса «Объект».

Контейнеры Java и C ++

Коллекции Java являются символами C ++ Containers.

Блок-схема Java Collections

Блок-схема контейнеров C ++

Целочисленные типы

Биты	Min	Максимум	Тип C ++ (на LLP64 или LP64)	Тип Java
8	-2 (8-1) = -128	2 (8-1) -1 = 127	голец	байт
8	0	2 (8) -1 = 255	unsigned char	-
16	-2 (16-1) = -32,768	2 (16-1) -1 = 32 767	короткая	короткая

Биты	Min	Максимум	Тип C ++ (на LLP64 или LP64)	Тип Java
16	0 (\ u0000)	2 (16) -1 = 65,535 (\ uFFFF)	беззнаковый короткий	char (без знака)
32	-2 (32-1) = -2,147 млрд.	2 (32-1) -1 = 2,147 млрд.	ИНТ	ИНТ
32	0	2 (32) -1 = 4,295 млрд	unsigned int	-
64	-2 (64-1)	2 (16-1) -1	долго*	долго долго
64	0	2 (16) -1	unsigned long * unsigned long long	-

* API Win64 - всего 32 бит

Много больше типов C ++

Examples

Участники статического класса

Статические члены имеют класс, а не объект

Пример C ++

```
// define in header
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Пример Java

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Классы, определенные в других конструкциях

Определено в другом классе

C ++

Вложенный класс [ref] (требуется ссылка на класс включения)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Джава

[нестатический] Вложенный класс (он же внутренний класс или класс участника)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Статически определено в другом классе

C ++

Статический вложенный класс

```
class Outer {
   class Inner {
      ...
   };
};
```

Джава

Статический вложенный класс (aka Static Member Class) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Определено в рамках метода

(например, обработка событий)

C ++

Местный класс [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Джава

Местный класс [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

Передача по значению и пересылка по ссылке

Многие утверждают, что Java ТОЛЬКО ПОКЛОННАЯ, но она более тонкая, чем та. Сравните следующие примеры на C ++ и Java, чтобы увидеть множество разновидностей pass-by-value (aka copy) и pass-by-reference (aka alias).

Пример C ++ (полный код)

```
  // passes a COPY of the object
  static void passByCopy(PassIt obj) {
     obj.i = 22;  // only a "local" change
  }

  // passes a pointer
  static void passByPointer(PassIt* ptr) {
     ptr->i = 33;
     ptr = 0; // better to use nullptr instead if '0'
  }

  // passes an alias (aka reference)
  static void passByAlias(PassIt& ref) {
     ref.i = 44;
  }

  // This is an old-school way of doing it.
  // Check out std::swap for the best way to do this
  static void swap(PassIt** pptr1, PassIt** pptr2) {
     PassIt* tmp = *pptr1;
     *pptr1 = *pptr2;
     *pptr2 = tmp;
  }
```

Пример Java (полный код)

```
   // passes a copy of the variable
   // NOTE: in java only primitives are pass-by-copy
   public static void passByCopy(int copy) {
```

```
      copy = 33;  // only a "local" change
   }

   // No such thing as pointers in Java
   /*
   public static void passByPointer(PassIt *ptr) {
      ptr->i = 33;
      ptr = 0; // better to use nullptr instead if '0'
   }
   */

   // passes an alias (aka reference)
   public static void passByAlias(PassIt ref) {
      ref.i = 44;
   }

   // passes aliases (aka references),
   // but need to do "manual", potentially expensive copies
   public static void swap(PassIt ref1, PassIt ref2) {
      PassIt tmp = new PassIt(ref1);
      ref1.copy(ref2);
      ref2.copy(tmp);
   }
```

Наследование и композиция

C ++ и Java являются объектно-ориентированными языками, поэтому следующая диаграмма применима к обоим.

Изгой понижающее приведение

Остерегайтесь использования «downcasting» - Downcasting отличает иерархию наследования от базового класса до подкласса (т. Е. Напротив полиморфизма). В общем, используйте полиморфизм и переопределение вместо экземпляра & downcasting.

Пример C ++

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Пример Java

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Абстрактные методы и классы

Абстрактный метод

объявлен без реализации

C ++

чистый виртуальный метод

```
virtual void eat(void) = 0;
```

Джава

абстрактный метод

```
abstract void draw();
```

Абстрактный класс

не может быть

C ++

не может быть создан; имеет по крайней мере 1 чистый виртуальный метод

```
class AB {public: virtual void f() = 0;};
```

Джава

не может быть создан; могут иметь не абстрактные методы

```
abstract class GraphicObject {}
```

Интерфейс

нет полей экземпляра

C ++

ничего похожего на Java

Джава

очень похож на абстрактный класс, но 1) поддерживает множественное наследование; 2) никаких полей экземпляра

```
interface TestInterface {}
```

Прочитайте Сравнение C ++ онлайн: https://riptutorial.com/ru/java/topic/10849/сравнение-c-plusplus

глава 166: Сравнительный и компаратор

Синтаксис

- Открытый класс MyClass реализует Comparable <MyClass >
- Открытый класс MyComparator реализует Comparator <SomeOtherClass >
- public int compareTo (другие MyClass)
- public int compare (SomeOtherClass o1, SomeOtherClass o2)

замечания

При реализации метода compareTo(..) который зависит от double , **не** выполняйте следующее:

```
public int comareTo(MyClass other) {
    return (int)(doubleField - other.doubleField); //THIS IS BAD
}
```

Усечение, вызванное действием (int) приведет к тому, что метод иногда неверно возвращает 0 вместо положительного или отрицательного числа и может таким образом привести к ошибкам сравнения и сортировки.

Вместо этого простейшей правильной реализацией является использование Double.compare , как такового:

```
public int comareTo(MyClass other) {
    return Double.compare(doubleField,other.doubleField); //THIS IS GOOD
}
```

Неоднородная версия Comparable<T> , просто Comparable , существовала со времен Java 1.2 . Помимо взаимодействия с устаревшим кодом, всегда лучше реализовать общую версию Comparable<T> , так как она не требует кастинга при сравнении.

Для класса очень стандартно сопоставимо с самим собой, как в:

```
public class A implements Comparable<A>
```

Хотя можно отказаться от этой парадигмы, будьте осторожны при этом.

Comparator<T> все еще может использоваться для экземпляров класса, если этот класс реализует Comparable<T> . В этом случае будет использоваться логика Comparator ; естественный порядок, указанный в реализации Comparable будет проигнорирован.

Examples

Сортировка списка с использованием Comparable или компаратором

Предположим, что мы работаем над классом, представляющим Личность, по имени и фамилии. Мы создали базовый класс для этого и реализовали правильные методы equals и hashCode .

```
public class Person {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull
```

```
    public Person(String firstName, String lastName){
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

Теперь мы хотели бы отсортировать список объектов Person по их имени, например, в следующем сценарии:

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //This currently won't work.
}
```

К сожалению, как отмечено, вышесказанное в настоящее время не будет компилироваться. Collections.sort(..) знает только, как сортировать список, если элементы в этом списке сопоставимы или задан пользовательский метод сравнения.

Если вас попросили отсортировать следующий список: 1,3,5,4,2 , у вас не возникло бы проблемы с ответом 1,2,3,4,5 . Это связано с тем, что целые (как в Java, так и математически) имеют *естественное упорядочение* , стандартное стандартное сравнение сравнения по умолчанию. Чтобы дать нашему классу Person естественный порядок, мы реализуем Comparable<Person> , который требует реализации метода compareTo(Person p):

```
public class Person implements Comparable<Person> {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName) {
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }
```

```
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }

    @Override
    public int compareTo(Person other) {
        // If this' lastName and other's lastName are not comparably equivalent,
        // Compare this to other by comparing their last names.
        // Otherwise, compare this to other by comparing their first names
        int lastNameCompare = lastName.compareTo(other.lastName);
        if (lastNameCompare != 0) {
            return lastNameCompare;
        } else {
            return firstName.compareTo(other.firstName);
        }
    }
}
```

Теперь основной метод будет функционировать правильно

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Now functions correctly

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Если, однако, вы либо не хотите, либо не можете изменить класс Person , вы можете предоставить пользовательский Comparator<T> который обрабатывает сравнение любых двух объектов Person . Если вас попросили отсортировать следующий список: circle, square, rectangle, triangle, hexagon вы не могли бы, но если бы вас попросили отсортировать этот список *по количеству углов* , вы могли бы. Именно поэтому предоставление компаратора инструктирует Java, как сравнивать два нормально не сопоставимых объекта.

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Компараторы также могут быть созданы / использованы как анонимный внутренний класс

```
//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.

    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //Anonymous Class
    Collections.sort(people, new Comparator<Person>() { //Legal
        public int compare(Person p1, Person p2) {
            //Method code...
        }
    });
}
```

Java SE 8

Комбинированные на основе Lambda выражения

Начиная с Java 8, компараторы также могут быть выражены в виде лямбда-выражений

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
```

```
        //Method code....
    });
```

Методы сравнения по умолчанию

Кроме того, существуют интересные методы по умолчанию для интерфейса компаратора для построения компараторов: следующее построение сравнительного сравнения по lastName а затем firstName .

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Обращаясь к порядку компаратора

Любой компаратор также может быть легко изменен с помощью метода reversedMethod который изменит восходящий порядок на нисходящий.

Сравнение и сравнение методов

Интерфейс Comparable<T> требует одного метода:

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

И интерфейс Comparator<T> требует одного метода:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

Эти два метода делают одно и то же, с одной незначительной разницей: compareTo сравнивает this с other , тогда как compare сравнивает t1 с t2 , не заботясь об this .

Помимо этой разницы, оба метода имеют схожие требования. В частности (для сравнения), сравнивает этот объект с указанным объектом для заказа. Возвращает отрицательное целое число, ноль или положительное целое число, так как этот объект меньше, равен или больше указанного объекта.
Таким образом, для сравнения a и b :

- Если a < b , a.compareTo(b) и compare(a,b) должны возвращать отрицательное целое число, а b.compareTo(a) и compare(b,a) следует возвращать положительное целое число
- Если a > b , a.compareTo(b) и compare(a,b) должны возвращать положительное целое число, а b.compareTo(a) и compare(b,a) следует возвращать отрицательное целое число
- Если a равно b для сравнения, все сравнения должны возвращать 0 .

Естественная (сопоставимая) и явная (сравнительная) сортировка

Существует два метода Collections.sort() :

- Один, который принимает List<T> в качестве параметра, где T должен реализовать Comparable и переопределить метод compareTo() который определяет порядок сортировки.
- Один, который принимает список и компаратор в качестве аргументов, где компаратор определяет порядок сортировки.

Во-первых, вот класс Person, который реализует Comparable:

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public int compareTo(Person o) {
        return this.getAge() - o.getAge();
    }
    @Override
    public String toString() {
        return this.getAge()+"-"+this.getName();
    }

}
```

Вот как вы могли бы использовать вышеприведенный класс для сортировки списка в естественном порядке его элементов, определяемого переопределением метода compareTo() :

```
//-- usage
List<Person> pList = new ArrayList<Person>();
            Person p = new Person();
            p.setName("A");
            p.setAge(10);
            pList.add(p);
            p = new Person();
            p.setName("Z");
            p.setAge(20);
            pList.add(p);
            p = new Person();
            p.setName("D");
            p.setAge(30);
            pList.add(p);

            //-- natural sorting i.e comes with object implementation, by age
            Collections.sort(pList);

            System.out.println(pList);
```

Вот как вы можете использовать анонимный встроенный компаратор для сортировки списка, который не реализует Comparable, или в этом случае, для сортировки списка в порядке, отличном от естественного упорядочения:

```
            //-- explicit sorting, define sort on another property here goes with name
            Collections.sort(pList, new Comparator<Person>() {

                @Override
                public int compare(Person o1, Person o2) {
```

```
                return o1.getName().compareTo(o2.getName());
            }
        });
        System.out.println(pList);
```

Сортировка записей в карте

Начиная с Java 8 на интерфейсе Map.Entry существуют методы по Map.Entry позволяющие сортировать итерации карт.

Java SE 8

```
Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human ressources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// Output the smallest departement in terms of number of employees
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
    .forEach(System.out::println);   // outputs : executives=10
```

Конечно, они также могут использоваться вне потока api:

Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Создание компаратора с использованием метода сравнения

```
Comparator.comparing(Person::getName)
```

Это создает компаратор для класса Person который использует имя этого человека в качестве источника сравнения. Также можно использовать версию метода для сравнения long, int и double. Например:

```
Comparator.comparingInt(Person::getAge)
```

Обратный порядок

Чтобы создать компаратор, который накладывает обратный порядок, используйте метод reverse reversed() :

```
Comparator.comparing(Person::getName).reversed()
```

Цепь компараторов

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

Это создаст компаратор, который сравнивает с фамилией, затем сравнивает его с именем. Вы можете объединить столько компараторов, сколько захотите.

Прочитайте Сравнительный и компаратор онлайн: https://riptutorial.com/ru/java/topic/3137/сравнительный-и-компаратор

глава 167: Ссылки на объекты

замечания

Это должно помочь вам понять «Исключение Null Pointer» - один получает один из них, потому что ссылка на объект имеет значение null, но программный код ожидает, что программа что-то использует в этой ссылке на объект. Тем не менее, это заслуживает своей темы ...

Examples

Ссылки на объекты как параметры метода

В этом разделе объясняется концепция *ссылки* на *объект* ; он ориентирован на людей, которые новичок в программировании на Java. Вы уже должны быть знакомы с некоторыми терминами и значениями: определение класса, основной метод, экземпляр объекта и вызов методов «на» объекта и передача параметров методам.

```
public class Person {

  private String name;

  public void setName(String name) { this.name = name; }

  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);

    System.out.println(person.getName() + " " + i);
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }
}
```

Чтобы быть полностью компетентным в программировании на Java, вы должны уметь объяснить этот пример кому-то еще с головы. Его концепции имеют фундаментальное значение для понимания того, как работает Java.

Как вы можете видеть, у нас есть main задача, которая создает объект для person переменной и вызывает метод для установки поля name в этом объекте на "Bob" . Затем он вызывает другой метод и передает person как один из двух параметров; другой параметр представляет собой целочисленную переменную, установленную в 5.

Метод , называемый задает name значение на пройденный объекта к «Линде», и устанавливает целочисленную переменную передается до 99, а затем возвращается.

Так что будет печататься?

```
Linda 5
```

Итак, почему внесение изменений в person вступает в силу в main , но изменение, внесенное в целое число, не так ли?

Когда вызов выполняется, основной метод передает *ссылку объекта* для person методу setPersonName ; любое изменение, которое setAnotherName делает для этого объекта, является частью этого объекта, и поэтому эти изменения остаются частью этого объекта при возврате метода.

Другой способ сказать одно и то же: person указывает на объект (хранится в куче, если вам интересно). Любое изменение метода делает для этого объекта «на этом объекте» и не зависит от того, активен или вернулся метод внесения изменения. Когда метод возвращается, любые изменения, внесенные в объект, все еще сохраняются на этом объекте.

Сравните это с целым числом, которое передается. Поскольку это *примитивный* int (а не экземпляр объекта Integer), он передается «по значению», то есть его значение предоставляется методу, а не указателю на исходное целое число, переданное в. Метод может изменить его для метода собственных целей, но это не влияет на переменную, используемую при вызове метода.

В Java все примитивы передаются по значению. Объекты передаются по ссылке, что означает, что указатель на объект передается как параметр любым методам, которые их принимают.

Еще одна очевидная вещь: это означает, что вызываемый метод не может создать *новый* объект и вернуть его в качестве одного из параметров. Единственный способ для метода вернуть объект, который создается, прямо или косвенно, вызовом метода, является возвращаемым значением из метода. Давайте сначала посмотрим, как это не сработает, и как это будет работать.

Давайте добавим еще один способ к нашему маленькому примеру:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

И, вернувшись в main , под вызовом setAnotherName , давайте перейдем к этому методу и другому вызову println:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Теперь программа распечатает:

```
Linda 5
Linda
```

Что случилось с объектом, в котором был Джордж? Ну, параметр, который был передан, был указателем на Линду; когда метод getAnotherObjectNot создал новый объект, он заменил ссылку на объект Linda ссылкой на объект George. Объект Линда все еще существует (в куче), main метод все равно может получить к нему доступ, но метод getAnotherObjectNot после этого не сможет ничего с ним сделать, потому что он не имеет к нему ссылки. Похоже, что автор кода, предназначенный для метода, создавал новый объект и передавал его обратно, но если это так, это не сработало.

Если это то, что хотел сделать автор, ему нужно было бы вернуть вновь созданный объект из метода, примерно так:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Затем назовите его так:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

И весь выпуск программы теперь будет:

```
Linda 5
Linda
Mary
```

Вот и вся программа с двумя дополнениями:

```
public class Person {
  private String name;

  public void setName(String name) { this.name = name; }
  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);
    System.out.println(person.getName() + " " + i);

    getAnotherObjectNot(person);
    System.out.println(person.getName());

    Person person;
    person = getAnotherObject();
    System.out.println(person.getName());
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }

  private static void getAnotherObjectNot(Person person) {
    person = new Person();
    person.setMyName("George");
  }

  private static person getAnotherObject() {
    Person person = new Person();
    person.setMyName("Mary");
    return person;
  }
}
```

Прочитайте Ссылки на объекты онлайн: https://riptutorial.com/ru/java/topic/5454/ссылки-на-объекты

глава 168: Стандарт официального кода Oracle

Вступление

Официальное руководство Oracle по языку программирования Java является стандартом, за которым следуют разработчики Oracle, и рекомендуется следовать за любым другим разработчиком Java. Он охватывает имена файлов, организацию файлов, отступы, комментарии, декларации, заявления, пробелы, соглашения об именах, методы программирования и включает пример кода.

замечания

- Приведенные выше примеры строго следуют новому официальному руководству по стилю от Oracle. Иными словами, это *не* субъективно составлено авторами этой страницы.
- Официальное руководство стиля было тщательно написано, чтобы быть обратно совместимым с оригинальным руководством по стилю и большей частью кода в дикой природе.
- Официальный стиль руководства был равный обзор в числе прочих, Брайан Гетц (Java Language архитектор) и Марк Рейнхольд (главный архитектор платформы Java).
- Примеры не являются нормативными; Хотя они намереваются проиллюстрировать правильный способ форматирования кода, могут быть и другие способы правильного форматирования кода. Это общий принцип: может быть несколько способов форматирования кода, все придерживаясь официальных рекомендаций.

Examples

Соглашения об именах

Названия пакетов

- Имена пакетов должны быть в нижнем регистре без символов подчеркивания или других специальных символов.
- Имена пакетов начинаются с перевернутой части полномочий веб-адреса компании разработчика. За этой частью может следовать подструктура пакета, зависящая от структуры проекта / программы.
- Не используйте множественную форму. Следуйте стандарту стандартного API, который использует, например, java.lang.annotation а не java.lang.annotations .
- **Примеры:** com.yourcompany.widget.button , com.yourcompany.core.api

Имена классов, интерфейсов и имен

- Имена классов и перечислений обычно должны быть существительными.
- Имена интерфейсов обычно должны быть существительными или прилагательными, заканчивающимися ... способными.
- Используйте смешанный регистр с первой буквой в каждом слове в верхнем регистре (например, CamelCase).
- Сопоставьте регулярное выражение ^[AZ][a-zA-Z0-9]*$.
- Используйте целые слова и избегайте использования сокращений, если аббревиатура более широко используется, чем длинная форма.
- Отформатируйте аббревиатуру как слово, если оно является частью более длинного имени класса.
- **Примеры:** ArrayList , BigInteger , ArrayIndexOutOfBoundsException , Iterable .

Имена методов

Имена методов обычно должны быть глаголами или другими описаниями действий

- Они должны соответствовать регулярному выражению ^[az][a-zA-Z0-9]*$.

- Используйте смешанный футляр с первой буквой в нижнем регистре.
- **Примеры:** toString , hashCode

переменные

Имена переменных должны быть в смешанном случае с первой буквой в нижнем регистре

- Сопоставьте регулярное выражение ^[az][a-zA-Z0-9]*$
- Дальнейшая рекомендация: Переменные
- **Примеры:** elements , currentIndex

Переменные типа

Для простых случаев, когда задействовано несколько переменных типа, используйте одну букву верхнего регистра.

- Сопоставьте регулярное выражение ^[AZ][0-9]?$
- Если одна буква более описательная, чем другая (например, K и V для ключей и значений на картах или R для возвращаемого типа функции), используйте это, в противном случае используйте T
- Для сложных случаев, когда переменные однобуквенного типа запутываются, используйте более длинные имена, написанные всеми прописными буквами, и используйте подчеркивание (_) для разделения слов.
- **Примеры:** T , V , SRC_VERTEX

Константы

Константы (static final поля, содержимое которых неизменно, по языковым правилам или по соглашению) должны быть названы всеми прописными буквами и подчеркиванием (_) для разделения слов.

- Сопоставьте регулярное выражение ^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$
- **Примеры:** BUFFER_SIZE , MAX_LEVEL

Другие рекомендации по именованию

- Избегайте методов скрытия / теневого копирования, переменных и переменных типа во внешних областях.
- Пусть многословность имени соотносится с размером области. (Например, используйте описательные имена для полей больших классов и краткие имена для локальных короткоживущих переменных).
- При присвоении имен статическим членам, пусть идентификатор будет самоописательным, если вы считаете, что они будут статически импортированы.
- Дальнейшее чтение: раздел именования (в официальном руководстве по стилю Java)

Источник: рекомендации стиля Java от Oracle

Исходные файлы Java

- Все строки должны быть завершены символом линии (LF, ASCII значение 10), а не, например, CR или CR + LF.
- В конце строки может отсутствовать конечное пробел.
- Имя исходного файла должно совпадать с именем класса, которое оно содержит, за которым следует расширение .java , даже для файлов, которые содержат только закрытый класс пакета. Это не относится к файлам, которые не содержат объявлений классов, например package-info.java .

Специальные символы

- Помимо LF единственным допустимым символом пробела является Space (значение ASCII 32). Обратите внимание, что это означает, что другие символы пробела (в, например, строковые и символьные литералы) должны быть записаны в экранированной форме.
- \' , \" , \\ , \t , \b , \r , \f и \n должны быть предпочтительнее соответствующих восьмиугольных (например, \047) или Unicode (например, \u0027) экранированных символов.
- Если есть необходимость идти против вышеуказанных правил для тестирования, тест должен *генерировать* требуемый ввод программно.

Объявление пакета

```
package com.example.my.package;
```

Объявление пакета не должно быть обернуто линией, независимо от того, превышает ли она максимальную максимальную длину строки.

Импортные заявления

```
// First java/javax packages
import java.util.ArrayList;
import javax.tools.JavaCompiler;

// Then third party libraries
import com.fasterxml.jackson.annotation.JsonProperty;

// Then project imports
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// Then static imports (in the same order as above)
import static java.util.stream.Collectors.toList;
```

- Операторы импорта должны быть отсортированы ...
    - ... прежде всего нестатические / статические с нестационарным импортом.
    - ... вторично по происхождению пакета согласно следующему порядку
        - java-пакеты
        - пакеты javax
        - внешние пакеты (например, org.xml)
        - внутренние пакеты (например, com.sun)
    - ... третичный по идентификатору пакета и класса в лексикографическом порядке
- Операторы импорта не должны быть обернуты линией, независимо от того, превышает ли она максимальную максимальную длину строки.
- Не должно быть неиспользованного импорта.

Импорт подстановок

- Импорт подстановочных знаков вообще не должен использоваться.
- При импорте большого количества тесно связанных классов (например, внедрение посетителя над деревом с десятками различных классов «узлов») может использоваться импорт подстановочных знаков.
- В любом случае следует использовать не более одного подстановочного импорта на файл.

Структура классов

Порядок учеников

Члены класса должны быть заказаны следующим образом:

1. Поля (в порядке публичных, защищенных и частных)
2. Конструкторы
3. Фабричные методы
4. Другие методы (в порядке публичных, защищенных и частных)

Приобретение полей и методов в основном с помощью их модификаторов доступа или идентификатора не требуется.

Вот пример этого порядка:

```
class Example {

    private int i;

    Example(int i) {
        this.i = i;
    }

    static Example getExample(int i) {
        return new Example(i);
    }

    @Override
    public String toString() {
        return "An example [" + i + "]";
    }

}
```

Группирование участников

- Связанные поля должны быть сгруппированы вместе.
- Вложенный тип может быть объявлен непосредственно перед его первым использованием; иначе он должен быть объявлен перед полями.
- Конструкторы и перегруженные методы должны быть сгруппированы по функциональности и упорядочены с увеличением arity. Это означает, что делегирование между этими конструкциями идет вниз в коде.
- Конструкторы должны быть сгруппированы вместе между другими членами.
- Перегруженные варианты метода должны быть сгруппированы вместе между другими членами.

Модификаторы

```
class ExampleClass {
    // Access modifiers first (don't do for instance "static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

interface ExampleInterface {
    // Avoid 'public' and 'abstract' since they are implicit
    void sayHello();
}
```

- Модификаторы должны идти в следующем порядке

    - Модификатор доступа (public / private / protected)
    - abstract
    - static
    - final
    - transient
    - volatile
    - default
    - synchronized
    - native
    - strictfp

- Модификаторы не должны выписываться, если они неявные. Например, методы интерфейса не должны быть объявлены public или abstract , а вложенные перечисления и интерфейсы не должны быть объявлены статическими.

- Параметры метода и локальные переменные не должны объявляться final если они не улучшают читаемость или не документируют фактическое дизайнерское решение.

- Поля должны быть объявлены final если нет веской причины сделать их изменчивыми.

вдавливание

- Уровень отступов – **четыре пробела** .
- Для отступов могут использоваться только пробелы. **Нет вкладок.**
- Пустые строки не должны иметь отступ. (Это подразумевает правило без пробелов в пробеле).
- строки case должны иметь отступы с четырьмя пробелами, а выражения в этом случае должны иметь отступы с четырьмя пробелами.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Обратитесь к инструкциям об обложке для рекомендаций относительно путей продолжения отступа.

Заявления об упаковке

- Исходный код и комментарии обычно не должны превышать 80 символов в строке и редко, если когда-либо превышать 100 символов в строке, включая отступ.

  Предельный характер должен оцениваться в каждом конкретном случае. В действительности имеет значение семантическая «плотность» и читаемость строки. Создание линий безвозвратно долгим образом затрудняет их чтение; аналогичным образом, делая «героические попытки» поместить их в 80 столбцов, также может затруднить их чтение. Гибкость, изложенная здесь, направлена на то, чтобы позволить разработчикам избежать этих крайностей, а не максимизировать использование монитора.

- URL-адреса или примеры команд не должны быть обернуты.

```
// Ok even though it might exceed max line width when indented.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));

// Wrapping preferable
```

```
String pretty = Stream.of(args)
                      .map(Argument::prettyPrint)
                      .collectors(joining(", "));

// Too strict interpretation of max line width. Readability suffers.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(
                targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(
                targetContainerType);

// Should be wrapped even though it fits within the character limit
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(", "));
```

- Обертка на более высоком синтаксическом уровне предпочтительнее обертывания на более низком синтаксическом уровне.
- В строке должно быть не более 1 строки.
- Строка продолжения должна быть отступом одним из следующих четырех способов:
    - **Вариант 1** : с 8 дополнительными пробелами относительно отступов предыдущей строки.
    - **Вариант 2** : с 8 дополнительными пробелами относительно начального столбца обернутого выражения.
    - **Вариант 3** : Согласовано с предыдущим выражением смежности (если ясно, что это линия продолжения)
    - **Вариант 4** : Согласован с предыдущим вызовом метода в закодированном выражении.

Объявления об упаковке

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
    …
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt, long aLong,
        double aDouble, long aLong) {
    …
}

int someMethod(String aString,
               List<Map<Integer, StringBuffer>> aListOfMaps,
               Map<String, String> aMap)
        throws IllegalArgumentException {
    …
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt)
                throws IllegalArgumentException {
    …
}
```

- Объявления метода могут быть отформатированы путем перечисления аргументов по вертикали или новой строки и +8 дополнительных пробелов

- Если предложение throws необходимо обернуть, поставьте разрыв строки перед предложением throws и убедитесь, что он выделяется из списка аргументов, либо отступом +8 относительно объявления функции, либо +8 относительно предыдущей строки.

Условные выражения

- Если линия приближается к максимальному пределу символа, всегда рассматривайте разбиение на несколько операторов / выражений вместо того, чтобы обертывать линию.
- Перерыв перед операторами.
- Перерыв перед. в цепных вызовах метода.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Пробелы

Вертикальные пробелы

- Для разделения ...
    - Объявление пакета
    - Объявления классов
    - Конструкторы
    - методы
    - Статические инициализаторы
    - Инициализаторы экземпляра
- ... и может использоваться для разделения логических групп
    - операторы импорта
    - поля
    - заявления
- Несколько последовательных пустых строк следует использовать только для разделения групп связанных элементов, а не как стандартного межсимвольного интервала.

Горизонтальные пробелы

- Необходимо использовать одно пространство ...
    - Чтобы отделить ключевые слова от соседних открывающих или закрывающих скобок и скобок
    - До и после всех двоичных операторов и операторов, таких как символы, такие как стрелки в лямбда-выражениях и двоеточие в расширенном для циклов (но не до двоеточия метки)
    - После // запускает комментарий.
    - После запятой разделяя аргументы и точки с запятой, разделяющие части цикла for.
    - После закрытия круглой скобки.
- В объявлениях переменных не рекомендуется выравнивать типы и переменные.

Объявления переменных

- Одна переменная для каждого объявления (и не более одного объявления в строке)

- Квадратные скобки массивов должны быть в типе (String[] args), а не в переменной (String args[]).
- Объявите локальную переменную прямо перед ее первым использованием и инициализируйте ее как можно ближе к объявлению.

Аннотации

Аннотации декларации должны быть помещены в отдельную строку из аннотации объявления.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

Тем не менее, несколько или короткие аннотации, аннотирующие однострочный метод, могут быть помещены в ту же строку, что и метод, если он улучшает читаемость. Например, можно написать:

```
@Nullable String getName() { return name; }
```

В целях согласованности и удобочитаемости все аннотации должны быть помещены в одну строку или каждая аннотация должна быть помещена в отдельную строку.

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

Лямбда-выражения

```
Runnable r = () -> System.out.println("Hello World");

Supplier<String> c = () -> "Hello World";

// Collection::contains is a simple unary method and its behavior is
// clear from the context. A method reference is preferred here.
appendFilter(goodStrings::contains);
```

```
// A lambda expression is easier to understand than just tempMap::put in this case
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- Экспрессия lambdas предпочтительнее одноблочных блоков лямбда.
- Ссылки на методы обычно должны быть предпочтительнее, чем лямбда-выражения.
- Для ссылок на метод привязанных экземпляров или методов с arity больше единицы выражение лямбда может быть легче понять и, следовательно, предпочтительнее. Особенно, если поведение метода не ясно из контекста.
- Типы параметров следует опустить, если они не улучшают читаемость.
- Если выражение лямбда простирается более чем на несколько строк, подумайте о создании метода.

Резервные скобки

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Резервные скобки для группировки (т.е. скобки, которые не влияют на оценку) могут использоваться, если они улучшают читаемость.
- Резервные скобки для группировки обычно должны быть исключены из более коротких выражений, включающих общие операторы, но включаться в более длинные выражения или выражения, содержащие операторов, приоритет и ассоциативность которых неясны без круглых скобок. Тернарные выражения с нетривиальными условиями принадлежат последним.
- Все выражения, следующие за ключевым словом return не должны быть окружены скобками.

литералы

```
long l = 5432L;
int i = 0x123 + 0xABC;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d;  // or 1 / 5432.0
double d2 = 0x1.3p2;
```

- long литералы должны использовать суффикс буквы L верхнего регистра.
- Шестнадцатеричные литералы должны использовать буквы верхнего регистра A - F
- Все остальные числовые префиксы, инфиксы и суффиксы должны использовать строчные буквы.

фигурные скобки

```
class Example {
    void method(boolean error) {
        if (error) {
            Log.error("Error occurred!");
            System.out.println("Error!");
        } else { // Use braces since the other block uses braces.
            System.out.println("No error");
        }
    }
}
```

- Открытие брекетов должно быть помещено в конец текущей строки, а не по отдельной линии.

- Перед закрывающей скобкой должна появиться новая строка, если блок не пуст (см. Краткие формы ниже)
- Скобки рекомендуются даже там, где язык делает их необязательными, например, однострочные и т. Д.
    - Если блок охватывает более одной строки (включая комментарии), он должен иметь фигурные скобки.
    - Если один из блоков в операторе if / else имеет фигурные скобки, другой блок тоже должен быть.
    - Если блок приходит последним в закрывающем блоке, он должен иметь фигурные скобки.
- Ключевое слово else , catch и while do…while циклы идут по той же строке, что и закрывающая скобка предыдущего блока.

Краткие формы

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

Вышеуказанные рекомендации призваны улучшить единообразие (и, таким образом, повысить узнаваемость / удобочитаемость). В некоторых случаях «короткие формы», которые отличаются от приведенных выше рекомендаций, являются столь же читаемыми и могут использоваться вместо этого. К таким случаям относятся, например, простые перечисления enum и тривиальные методы и лямбда-выражения.

Прочитайте Стандарт официального кода Oracle онлайн: https://riptutorial.com/ru/java/topic/2697/стандарт-официального-кода-oracle

глава 169: Строковый токенизатор

Вступление

Класс java.util.StringTokenizer позволяет разбить строку на токены. Это простой способ разбить строку.

Набор разделителей (символы, разделяющие токены) могут быть указаны либо во время создания, либо на основе каждого токена.

Examples

StringTokenizer Разделить пространство

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("apple ball cat dog"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

Выход:

```
яблоко

мяч

кошка

собака
```

StringTokenizer Split by comma ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Выход:

```
яблоко

шарик кот

собака
```

Прочитайте Строковый токенизатор онлайн: https://riptutorial.com/ru/java/topic/10563/строковый-токенизатор

глава 170: Струны

Вступление

Строки (java.lang.String) - это фрагменты текста, хранящиеся в вашей программе. Строки **не** являются примитивным типом данных на Java , однако они очень распространены в программах Java.

В Java строки являются неизменными, что означает, что они не могут быть изменены. (Нажмите здесь, чтобы получить более подробное объяснение неизменности.)

замечания

Поскольку строки Java неизменяемы , все методы, которые управляют String , **возвращают новый объект String** . Они не меняют исходную String . Это включает в себя методы подстроки и замещения, которые программисты на C и C ++ ожидали бы изменить целевой объект String .

Используйте StringBuilder вместо String если вы хотите объединить более двух объектов String , значения которых не могут быть определены во время компиляции. Этот метод более эффективен, чем создание новых объектов String и их объединение, поскольку StringBuilder изменен.

StringBuffer также может использоваться для конкатенации объектов String . Однако этот класс менее эффективен, потому что он предназначен для обеспечения потокобезопасности и получает мьютекс перед каждой операцией. Поскольку вы почти никогда не нуждаетесь в защите потоков при конкатенации строк, лучше всего использовать StringBuilder .

Если вы можете выразить конкатенацию строк как одно выражение, тогда лучше использовать оператор + . Компилятор Java преобразует выражение, содержащее + конкатенации, в эффективную последовательность операций, используя либо String.concat(...) либо StringBuilder . Совет по использованию StringBuilder явно применяется только тогда, когда конкатенация включает в себя несколько выражений.

Не храните конфиденциальную информацию в строках. Если кто-то может получить дамп памяти вашего запущенного приложения, тогда они смогут найти все существующие объекты String и прочитать их содержимое. Сюда входят объекты String , недоступные и ожидающие сбора мусора. Если это вызывает беспокойство, вам нужно будет стереть конфиденциальные строковые данные, как только вы закончите с этим. Вы не можете сделать это с объектами String поскольку они неизменяемы. Поэтому рекомендуется использовать объекты char[] для хранения конфиденциальных символьных данных и уничтожить их (например, перезаписать их символами '\000'), когда вы закончите.

Все экземпляры String создаются в куче, даже экземпляры, соответствующие строковым литералам. Особенность струнных литералов в том, что JVM гарантирует, что все литералы, которые являются равными (т.е. состоят из одних и тех же символов), представлены одним объектом String (это поведение указано в JLS). Это реализовано загрузчиками классов JVM. Когда загрузчик классов загружает класс, он сканирует строковые литералы, которые используются в определении класса, каждый раз, когда он видит один, он проверяет, есть ли уже запись в пуле строк для этого литерала (используя литерал как ключ) , Если уже есть запись для литерала, используется ссылка на экземпляр String хранящийся как пара для этого литерала. В противном случае создается новый экземпляр String и ссылка на экземпляр хранится для литерала (используется как ключ) в пуле строк. (Также см. Интернирование строк).

Пул строк хранится в куче Java и подчиняется обычной сборке мусора.

Java SE 7

В версиях Java до Java 7 пул строк состоялся в особой части кучи, известной как «PermGen». Эта часть собиралась лишь иногда.

Java SE 7

В Java 7 пул строк был удален из «PermGen».

Обратите внимание, что строковые литералы неявно достижимы из любого метода, который их использует. Это означает, что соответствующие объекты String могут быть собраны только при сборке мусора, если сам код является сборкой мусора.

До Java 8 объекты String реализованы как массив символов UTF-16 (2 байта на символ). В Java 9 предлагается реализовать String как массив байтов с полем флага кодирования, чтобы отметить, что строка кодируется как байты (LATIN-1) или символы (UTF-16).

Examples

Сравнение строк

Для сравнения строк равенства, следует использовать строковый объект в equals или equalsIgnoreCase методы.

Например, следующий фрагмент будет определять, равны ли два экземпляра String для всех символов:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
}
```

Демо-версия

Этот пример будет сравнивать их, независимо от их случая:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Демо-версия

Обратите внимание: equalsIgnoreCase не позволяет указать Locale . Например, если вы сравниваете два слова "Taki" и "TAKI" на английском языке, они равны; однако на турецком языке они разные (на турецком, нижний регистр I - ı). Для таких случаев преобразование обеих строк в нижний регистр (или в верхний регистр) с помощью Locale а затем сравнение с equals - это решение.

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); //prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
                   secondString.toLowerCase(locale))); //prints false
```

Демо-версия

Не используйте оператор == для сравнения строк

Если вы не можете гарантировать, что все строки были интернированы (см. Ниже), вы **не должны** использовать операторы == или != Для сравнения строк. Эти операторы фактически проверяют ссылки,

и поскольку несколько объектов String могут представлять одну и ту же строку, это может привести к неправильному ответу.

Вместо этого используйте метод String.equals(Object) , который будет сравнивать объекты String на основе их значений. Подробное объяснение см. В Pitfall: использование == для сравнения строк .

Сравнение строк в инструкции switch

Java SE 7

Начиная с Java 1.7, можно сравнить переменную String с литералами в инструкции switch . Убедитесь, что String не имеет значения null, иначе он всегда будет генерировать NullPointerException . Значения сравниваются с использованием String.equals , т.е. чувствительны к регистру.

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //the code goes here
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

Демо-версия

Сравнение строк с постоянными значениями

При сравнении значения String с константой вы можете поместить постоянное значение в левой части equals чтобы убедиться, что вы не получите NullPointerException если другая String равна null .

```
"baz".equals(foo)
```

В то время как foo.equals("baz") выкинет foo.equals("baz") NullPointerException если foo имеет значение null , "baz".equals(foo) будет оцениваться как false .

Java SE 7

Более читаемой альтернативой является использование Objects.equals() , которая выполняет нулевую проверку обоих параметров: Objects.equals(foo, "baz") .

(Примечание: Это спорно , как к тому, что лучше избегать NullPointerExceptions в целом, или пусть произойдет , а затем устранить причину, см здесь и здесь , конечно, назвав стратегию избегания «лучшая практика» не является оправданной.) .

Строковые упорядочения

Класс String реализует Comparable<String> с методом String.compareTo (как описано в начале этого примера). Это делает естественным упорядочение String объектов с учетом регистра. Класс String предоставляет константу Comparator<String> называемую CASE_INSENSITIVE_ORDER подходящую для сортировки без CASE_INSENSITIVE_ORDER регистра.

Сравнение с интернированными строками

Спецификация языка Java (JLS 3.10.6) гласит следующее:

> Более того, строковый литерал всегда ссылается на один и тот же экземпляр класса String . Это связано с тем, что строковые литералы, или, в более общем смысле, строки, являющиеся значениями константных выражений, *интернированы*, чтобы обмениваться уникальными экземплярами, используя метод String.intern ".

Это означает, что безопасно сравнивать ссылки на два строковых *литерала*, используя == . Более того, то же самое верно для ссылок на объекты String , которые были созданы с использованием String.intern() .

Например:

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
if (strObj.equals(str)) {
    System.out.println("The strings are equal");
}

// The two string references do not point to the same object
if (strObj != str) {
    System.out.println("The strings are not the same object");
}

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("The interned string and the literal are the same object");
}
```

За кулисами механизм интернирования поддерживает хеш-таблицу, содержащую все интернированные строки, которые все еще *доступны* . Когда вы вызываете intern() в String , метод ищет объект в хеш-таблице:

- Если строка найдена, то это значение возвращается как интернированная строка.
- В противном случае копия строки добавляется в хэш-таблицу, и эта строка возвращается как интернированная строка.

Можно использовать интернирование, чтобы строки могли сравниваться с помощью == . Однако есть серьезные проблемы с этим; см. Pitfall. Интернированные строки, чтобы вы могли использовать ==, - это плохая идея для деталей. Это не рекомендуется в большинстве случаев.

Изменение случая символов внутри строки

Тип String предоставляет два метода преобразования строк между верхним регистром и нижним регистром:

- toUpperCase для преобразования всех символов в верхний регистр
- toLowerCase для преобразования всех символов в нижний регистр

Эти методы возвращают преобразованные строки в виде новых экземпляров String : исходные объекты String не изменяются, поскольку String неизменна в Java. См. Это больше для неизменяемости: неизменность строк в Java

```
String string = "This is a Random String";
```

```
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);   // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

Этими методами не затрагиваются неалфавитные символы, такие как цифры и знаки препинания. Обратите внимание, что эти методы также могут неправильно обрабатывать определенные символы Unicode при определенных условиях.

Примечание . Эти методы *чувствительны к локали* и могут давать неожиданные результаты, если они используются в строках, которые предназначены для интерпретации независимо от языка. Примерами являются идентификаторы языка программирования, ключи протокола и теги HTML .

Например, "TITLE".toLowerCase() в турецком языке возвращает « tıtle », где ı (\u0131) является ı (\u0131) LATIN SMALL LETTER DOTLESS I. Чтобы получить правильные результаты для нечувствительных к регистру строк, перейдите в Locale.ROOT как параметр к соответствующему методу преобразования случая (например, toLowerCase(Locale.ROOT) или toUpperCase(Locale.ROOT)).

Хотя использование Locale.ENGLISH также верно для большинства случаев, **инвариантным языком** является Locale.ROOT .

Подробный список символов Юникода, требующих специальной оболочки, можно найти на веб-сайте Консорциума Юникода .

Изменение случая определенного символа в строке ASCII:

Чтобы изменить случай конкретного символа строки ASCII, можно использовать следующий алгоритм:

шаги:

1. Объявите строку.
2. Введите строку.
3. Преобразуйте строку в массив символов.
4. Введите символ, который нужно искать.
5. Поиск символа в массиве символов.
6. Если найден, проверьте, имеет ли символ строчный или верхний регистр.
    - Если в верхнем регистре добавьте 32 к коду ASCII символа.
    - Если в нижнем регистре вычесть 32 из ASCII-кода символа.
7. Измените исходный символ из массива символов.
8. Преобразуйте массив символов в строку.

Вуаля, Дело о персонаже изменено.

Примером кода для алгоритма является:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
        } else if (d >= 'A' && d <= 'Z') {
            d += 32;
```

```
        }
        a[i] = d;
        break;
    }
}
s = String.valueOf(a);
System.out.println(s);
```

Поиск строки в другой строке

Чтобы проверить, содержится ли конкретная строка a в String.contains() b или нет, мы можем использовать метод String.contains() со следующим синтаксисом:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

Метод String.contains() может использоваться для проверки наличия CharSequence в String. Метод ищет струнный a в строке b в регистрозависимой образом.

```
String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";

System.out.println(str1.contains(str2)); //prints true
System.out.println(str1.contains(str3)); //prints false
```

Живая демонстрация на Ideone

Чтобы найти точную позицию, в которой строка начинается с другой строки, используйте String.indexOf() :

```
String s = "this is a long sentence";
int i = s.indexOf('i');    // the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z');    // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

Живая демонстрация на Ideone

Метод String.indexOf() возвращает первый индекс char или String в другой String . Метод возвращает -1 если он не найден.

Примечание . Метод String.indexOf() чувствителен к регистру.

Пример поиска, игнорирующий случай:

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);                                 // -1
str1.toLowerCase().contains(str2.toLowerCase());  // true
str1.toLowerCase().indexOf(str2.toLowerCase());   // 6
```

Живая демонстрация на Ideone

Получение длины строки

Чтобы получить длину объекта String , вызовите на нем метод length() . Длина равна количеству кодовых единиц UTF-16 (символов) в строке.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Живая демонстрация на Ideone

char в строке является значение UTF-16. Кодовые страницы Unicode, значения которых составляют ≥ 0x1000 (например, большинство emojis), используют две позиции char. Чтобы подсчитать количество кодовых точек Unicode в String, независимо от того, соответствует ли каждый код в значении char UTF-16, вы можете использовать метод codePointCount :

```
int length = str.codePointCount(0, str.length());
```

Вы также можете использовать Stream of codepoints, начиная с Java 8:

```
int length = str.codePoints().count();
```

Подстроки

```
String s = "this is an example";
String a = s.substring(11); // a will hold the string starting at character 11 until the end
("example")
String b = s.substring(5, 10); // b will hold the string starting at character 5 and ending
right before character 10 ("is an")
String b = s.substring(5, b.length()-3); // b will hold the string starting at character 5
ending right before b' s lenght is out of 3  ("is an exam")
```

Подстроки могут также применяться к фрагменту и добавлять / заменять символ в его исходную строку. Например, вы столкнулись с китайской датой, содержащей китайские иероглифы, но хотите сохранить ее в виде строкового формата даты.

```
String datestring = "2015□11□17□";
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

Метод подстроки извлекает часть String . При наличии одного параметра параметр является началом, а кусок продолжается до конца String . При задании двух параметров первым параметром является начальный символ, а второй параметр - индекс символа сразу после конца (символ в индексе не включен). Легкий способ проверки заключается в том, что вычитание первого параметра из второго должно приводить к ожидаемой длине строки.

Java SE 7

В версиях JDK <7u6 метод substring создает экземпляр String который имеет один и тот же базовый char[] в качестве исходной String и имеет внутренние поля offset и count заданные для начала и длины результата. Такое совместное использование может привести к утечкам памяти, что может быть предотвращено вызовом new String(s.substring(...)) для принудительного создания копии, после чего char[] может быть собран в мусор.

Java SE 7

Из JDK 7u6 метод substring всегда копирует весь базовый массив char[] , делая сложность линейной по сравнению с предыдущей константой, но гарантируя отсутствие утечек памяти в одно и то же время.

Получение n-го символа в строке

```
String str = "My String";
```

```
System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // Last character "g"
```

Чтобы получить n-й символ в строке, просто вызовите charAt(n) в String , где n - это индекс символа, который вы хотите получить

ПРИМЕЧАНИЕ: индекс n начинается с 0 , поэтому первый элемент находится при n = 0.

Независимый от платформы новый разделитель строк

Поскольку новый разделитель строк варьируется от платформы к платформе (например, \n в Unix-подобных системах или \r\n в Windows), часто необходимо иметь независимый от платформы способ доступа к нему. В Java он может быть получен из системного свойства:

```
System.getProperty("line.separator")
```

Java SE 7

Поскольку новый разделитель строк так часто необходим, из Java 7 по методу быстрого доступа, возвращающему точно такой же результат, как и код выше, доступен:

```
System.lineSeparator()
```

Примечание . Поскольку маловероятно, что новый разделитель строк изменяется во время выполнения программы, рекомендуется хранить его в статической конечной переменной, а не извлекать его из системного свойства каждый раз, когда это необходимо.

При использовании String.format используйте %n вместо \n или '\ r \ n' для вывода независимого от платформы нового разделителя строк.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Добавление метода toString () для настраиваемых объектов

Предположим, вы определили следующий класс Person :

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Если вы создаете экземпляр нового объекта Person :

```
Person person = new Person(25, "John");
```

и позже в вашем коде вы используете следующий оператор для печати объекта:

```
System.out.println(person.toString());
```

Живая демонстрация на Ideone

вы получите результат, похожий на следующий:

```
Person@7ab89d
```

Это результат реализации метода toString() определенного в классе Object , суперклассе Person . Документация объекта Object.toString() гласит:

> Метод toString для класса Object возвращает строку, состоящую из имени класса, объектом которого является экземпляр, символа at-sign `@ 'и шестизначного шестнадцатеричного представления хеш-кода объекта. Другими словами, этот метод возвращает строку, равную значению:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Таким образом, для значимого вывода вам придется **переопределить** метод toString() :

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Теперь выход будет:

```
My name is John and my age is 25
```

Вы также можете написать

```
System.out.println(person);
```

Живая демонстрация на Ideone

На самом деле println() неявно вызывает метод toString для объекта.

Разделение строк

Вы можете разделить String на конкретный разделительный символ или регулярное выражение , вы можете использовать метод String.split() который имеет следующую подпись:

```
public String[] split(String regex)
```

Обратите внимание, что разделительный символ или регулярное выражение удаляется из результирующего массива String.

Пример с использованием разделительного символа:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Пример с использованием регулярного выражения:

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

Вы даже можете разделить String литерал:

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Предупреждение . Не забывайте, что параметр всегда рассматривается как регулярное выражение.

```
"aaa.bbb".split("."); // This returns an empty array
```

В предыдущем примере . рассматривается как подстановочный знак регулярного выражения, который соответствует любому символу, и поскольку каждый символ является разделителем, результатом является пустой массив.

Разделение на основе разделителя, который является метасимволом регулярного выражения

Следующие символы считаются специальными (иначе метасимволами) в регулярном выражении

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

Чтобы разбить строку на основе одного из указанных разделителей, вам нужно либо *сбежать* с помощью \\ либо использовать Pattern.quote() :

- Использование Pattern.quote() :

  ```
  String s = "a|b|c";
  String regex = Pattern.quote("|");
  String[] arr = s.split(regex);
  ```

- Выход из специальных символов:

  ```
  String s = "a|b|c";
  String[] arr = s.split("\\|");
  ```

Сплит удаляет пустые значения

split(delimiter) по умолчанию удаляет конечные пустые строки из массива результатов. Чтобы отключить этот механизм, нам нужно использовать перегруженную версию split(delimiter, limit) с ограничением, установленным на отрицательное значение, например

```
String[] split = data.split("\\|", -1);
```

split(regex) внутренне возвращает результат split(regex, 0) .

Предельный параметр управляет количеством применений шаблона и, следовательно, влияет на длину результирующего массива.
Если предел *n* больше нуля, шаблон будет применен не более *n - 1* раз, длина массива будет не больше *n* , а последняя запись массива будет содержать все входные данные за пределами последнего сопоставленного разделителя.
Если *n* отрицательно, то шаблон будет применяться столько раз, сколько возможно, и массив может иметь любую длину.
Если *n* равно нулю, шаблон будет применяться столько раз, сколько возможно, массив может иметь любую длину, а конечные пустые строки будут отброшены.

Разделение с помощью StringTokenizer

Помимо метода split() Строки также можно разделить с помощью StringTokenizer .

StringTokenizer является еще более ограничивающим, чем String.split() , а также немного сложнее в использовании. Он по существу предназначен для вытаскивания токенов, ограниченных фиксированным набором символов (заданных как String). Каждый символ будет действовать как

разделитель. Из-за этого ограничения он примерно в два раза быстрее, чем String.split() .

Набор символов по умолчанию - это пустые пространства (\t\n\r\f). Следующий пример будет печатать каждое слово отдельно.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Это напечатает:

```
the
lazy
fox
jumped
over
the
brown
fence
```

Вы можете использовать разные наборы символов для разделения.

```
String str = "jumped over";
// In this case character `u` and `e` will be used as delimiters
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Это напечатает:

```
j
mp
d ov
r
```

Объединение строк с разделителем

Java SE 8

Массив строк можно объединить с помощью статического метода String.join() :

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Точно так же существует перегруженный String.join() для Iterable s.

Чтобы иметь мелкозернистый контроль над присоединением, вы можете использовать класс StringJoiner :

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
    // The last two arguments are optional,
    // they define prefix and suffix for the result string
```

```
sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // Prints "[foo, bar, foobar]"
```

Чтобы присоединиться к потоку строк, вы можете использовать сборщик :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

Здесь также можно указать префикс и суффикс :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Реверсивные струны

Есть несколько способов изменить строку, чтобы сделать ее обратно.

1. StringBuilder / StringBuffer:

```
 String code = "code";
 System.out.println(code);

 StringBuilder sb = new StringBuilder(code);
 code = sb.reverse().toString();

 System.out.println(code);
```

2. Char array:

```
String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++,
mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));
```

Подсчет вхождений подстроки или символа в строке

Метод countMatches из org.apache.commons.lang3.StringUtils обычно используется для подсчета вхождения подстроки или символа в String :

```
import org.apache.commons.lang3.StringUtils;

String text = "One fish, two fish, red fish, blue fish";
```

```
// count occurrences of a substring
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// count occurrences of a char
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

В противном случае для того же, что и для стандартных Java API, вы можете использовать регулярные выражения:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "One fish, two fish, red fish, blue fish";
System.out.println(countStringInString("fish", text)); // prints 4
System.out.println(countStringInString(",", text)); // prints 3


public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
      stringOccurrences++;
    }
    return stringOccurrences;
}
```

Конкатенация строк и StringBuilders

Конкатенацию строк можно выполнить с помощью оператора + . Например:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Обычно реализация компилятора будет выполнять вышеупомянутую конкатенацию с использованием методов, связанных с StringBuilder под капотом. При компиляции код будет выглядеть примерно так:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

StringBuilder есть несколько перегруженных методов для добавления разных типов, например, для добавления int вместо String . Например, реализация может конвертировать:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

к следующему:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

Вышеприведенные примеры иллюстрируют простую операцию конкатенации, которая эффективно выполняется в одном месте в коде. Конкатенация включает в себя один экземпляр StringBuilder . В некоторых случаях конкатенация осуществляется кумулятивным образом, например, в цикле:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

В таких случаях оптимизация компилятора обычно не применяется, и каждая итерация создаст новый объект StringBuilder . Это можно оптимизировать, явно преобразуя код для использования одного StringBuilder :

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
return result.toString();
```

StringBuilder будет инициализирован пустым пространством всего 16 символов. Если вы заранее знаете, что будете строить большие строки, может быть полезно инициализировать его с достаточным размером заранее, так что внутренний буфер не нужно изменять:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

Если вы производите много строк, рекомендуется повторно использовать StringBuilder s:

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outputfile.write(buf.toString());
}
```

Если (и только если) несколько потоков записываются в один и тот же буфер, используйте StringBuffer , который является synchronized версией StringBuilder . Но поскольку, как правило, только один поток записывает в буфер, обычно быстрее использовать StringBuilder без синхронизации.

Использование метода concat ():

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

Это возвращает новую строку, которая является string1 с добавлением string2 к ней в конце. Вы также можете использовать метод concat () со строковыми литералами, как в:

```
"My name is ".concat("Buyya");
```

Замена частей строк

Два способа заменить: регулярным выражением или точным совпадением.

Примечание: исходный объект String не изменится, возвращаемое значение содержит измененную

строку.

Полное совпадение

Замените одиночный символ другим символом:

```
String replace(char oldChar, char newChar)
```

> Возвращает новую строку в результате замены всех вхождений oldChar в этой строке с помощью newChar.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Результат:

```
WoWcorn
```

Замените последовательность символов другой последовательностью символов:

```
String replace(CharSequence target, CharSequence replacement)
```

> Заменяет каждую подстроку этой строки, которая соответствует буквенной целевой последовательности с указанной последовательностью замены литерала.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Результат:

```
metallica petallica et al.
```

Regex

Примечание : группировка использует символ $ для ссылки на группы, например $1 .

Заменить все совпадения:

```
String replaceAll(String regex, String replacement)
```

> Заменяет каждую подстроку этой строки, которая соответствует данному регулярному выражению с указанной заменой.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Результат:

```
spiral metallica petallica et al.
```

Заменить только первое совпадение:

```
String replaceFirst(String regex, String replacement)
```

> Заменяет первую подстроку этой строки, которая соответствует данному регулярному

выражению с указанной заменой

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Результат:

```
spiral metallica petal et al.
```

Удаление пробелов с начала и конца строки

Метод trim() возвращает новую строку с удаленным пробелом и конечным пробелом.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

Если вы trim строку, которая не имеет пробелов для удаления, вам будет возвращен тот же экземпляр String.

Обратите внимание, что метод trim() имеет свое собственное понятие пробела , которое отличается от понятия, используемого методом Character.isWhitespace() :

- Все управляющие символы ASCII с кодами U+0000 до U+0020 считаются простыми и удаляются с помощью trim() . Это включает в себя U+0020 'SPACE' , U+0009 'CHARACTER TABULATION' , U+000A 'LINE FEED' и U+000D 'CARRIAGE RETURN' , но также символы, такие как U+0007 'BELL' .
- U+00A0 'NO-BREAK SPACE' Unicode, такие как U+00A0 'NO-BREAK SPACE' или U+2003 'EM SPACE' , *не* распознаются trim() .

Струнный пул и хранилище кучи

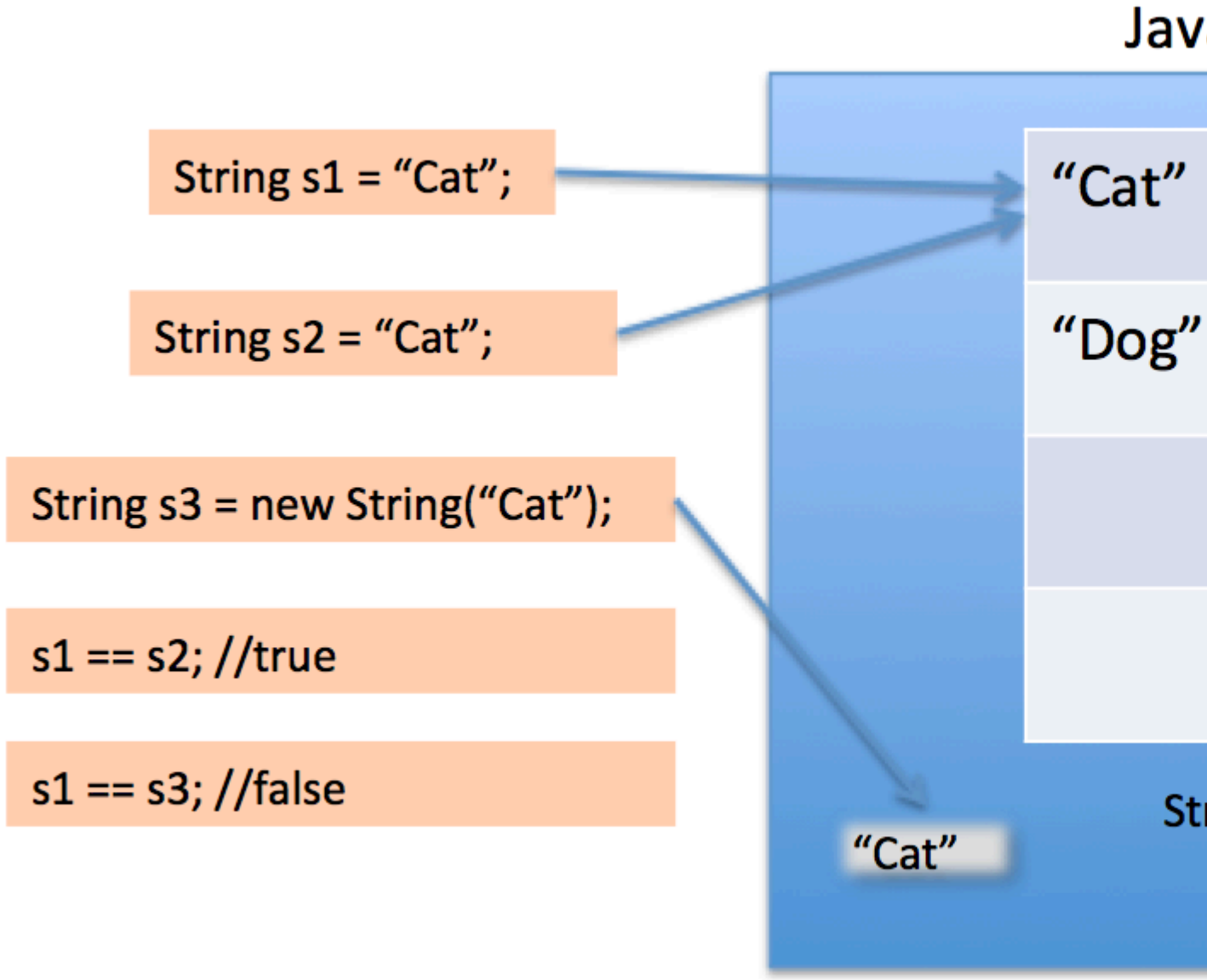
Как и многие объекты Java, **все** экземпляры String создаются в куче, даже в литералах. Когда JVM находит String буквальные , что не имеют аналогов ссылки в куче, виртуальная машина создает соответствующий String экземпляр в куче , **и** он также сохраняет ссылку на вновь созданном String например в строке пуле. Любые другие ссылки на один и тот же String литерал заменяются ранее созданным экземпляром String в куче.

Давайте посмотрим на следующий пример:

```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //All three strings are equivalent
        System.out.println(a.equals(b) && b.equals(c));

        //Although only a and b reference the same heap object
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

Вышеуказанный результат:

```
true
true
true
true
```



Когда мы используем двойные кавычки для создания String, сначала он ищет String с одинаковым значением в пуле String, если он просто возвращает ссылку else, он создает новую строку в пуле и затем возвращает ссылку.

Однако, используя новый оператор, мы вынуждаем класс String создавать новый объект String в кучном пространстве. Мы можем использовать метод intern (), чтобы поместить его в пул или обратиться к другому объекту String из пула строк, имеющего такое же значение.

Сам пул строк также создается в куче.

Java SE 7

До Java 7 String **литералы** сохранялись в пуле постоянной среды выполнения в области методов PermGen , у которых был фиксированный размер.

Пул строк также находился в PermGen .

Java SE 7

RFC: 6962931

> В JDK 7 интернированные строки больше не выделяются в постоянном поколении кучи Java, а вместо этого выделяются в основной части кучи Java (так называемые молодые и старые поколения) вместе с другими объектами, созданными приложением , Это изменение приведет к большему количеству данных, находящихся в основной куче Java, и меньше данных в постоянном поколении, и, следовательно, может потребоваться корректировка размеров кучи. Из-за этого большинства приложений будут наблюдаться лишь относительно небольшие различия в использовании кучи, но более крупные приложения, загружающие многие классы или интенсивно использующие метод String.intern() будут видеть более значительные различия.

Нечувствительный к регистру выключатель

Java SE 7

сам switch не может быть параметризован как нечувствительный к регистру, но если он абсолютно необходим, он может вести себя нечувствительно к входной строке, используя toLowerCase() или toUpperCase :

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

берегись

- Locale может повлиять на изменение ситуации.
- Необходимо соблюдать осторожность, чтобы в ярлыках не было символов верхнего регистра - они никогда не будут выполнены!

Прочитайте Струны онлайн: https://riptutorial.com/ru/java/topic/109/струны

глава 171: супер ключевое слово

Examples

Использование ключевого слова Super с примерами

Супер ключевое слово выполняет важную роль в трех местах

1. Уровень конструктора
2. Уровень метода
3. Переменный уровень

Уровень конструктора

super используется для вызова конструктора родительского класса. Этот конструктор может быть конструктором по умолчанию или параметризованным конструктором.

- Конструктор по умолчанию: super();
- Параметрированный конструктор: super(int no, double amount, String name);

```
class Parentclass
{
   Parentclass(){
      System.out.println("Constructor of Superclass");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor of Subclass");
   }
   Subclass(int n1){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor with arg");
   }
   void display(){
    System.out.println("Hello");
   }
   public static void main(String args[]){
    // Creating object using default constructor
    Subclass obj= new Subclass();
    //Calling sub class method
       obj.display();
       //Creating object 2 using arg constructor
       Subclass obj2= new Subclass(10);
       obj2.display();
  }
}
```

Примечание : super() должен быть первым оператором в конструкторе, иначе мы получим сообщение об ошибке компиляции.

Уровень метода

super ключевое слово также может быть использовано в случае переопределения метода. super можно использовать для вызова или вызова метода родительского класса.

```
class Parentclass
{
   //Overridden method
   void display(){
    System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
    System.out.println("Child class method");
   }
   void printMsg(){
    //This would call Overriding method
    display();
    //This would call Overridden method
    super.display();
   }
   public static void main(String args[]){
    Subclass obj= new Subclass();
    obj.printMsg();
   }
}
```

Примечание . Если переопределение метода отсутствует, нам не нужно использовать super ключевое слово, чтобы вызвать метод родительского класса.

Переменный уровень

super используется для ссылки на экземпляр экземпляра родительского класса. В случае наследования может быть возможность базового класса, а производный класс может иметь одинаковые элементы данных. Чтобы различать элемент данных базового / родительского класса и производного / дочернего класса, в контексте производного класса данные базового класса членам должно предшествовать ключевое слово super .

```
//Parent class or Superclass
class Parentclass
{
    int num=100;
}
//Child class or subclass
class Subclass extends Parentclass
{
    /* I am declaring the same variable
     * num in child class too.
     */
    int num=110;
    void printNumber(){
     System.out.println(num); //It will print value 110
     System.out.println(super.num); //It will print value 100
    }
    public static void main(String args[]){
       Subclass obj= new Subclass();
       obj.printNumber();
```

```
    }
}
```

Примечание . Если мы не будем писать super ключевое слово до имени элемента данных базового класса, оно будет называться текущим членом данных класса и элементом данных базового класса, которые скрыты в контексте производного класса.

Прочитайте супер ключевое слово онлайн: https://riptutorial.com/ru/java/topic/5764/супер-ключевое-слово

глава 172: Тестирование устройства

Вступление

Модульные испытания являются неотъемлемой частью разработки, основанной на тестах, и важной особенностью для создания любого надежного приложения. В Java модульное тестирование почти исключительно выполняется с использованием внешних библиотек и фреймворков, большинство из которых имеют свой собственный тег документации. Этот заглушка служит средством для ознакомления читателя с доступными инструментами и их соответствующей документацией.

замечания

Структуры тестовых модулей

Для модульного тестирования в Java существует множество возможностей. Самым популярным вариантом является JUnit. Он документируется следующим образом:

JUnit

JUnit4 - *Предлагаемый тег для функций JUnit4; еще не реализованы* .

Существуют другие платформы тестирования модулей и доступны документация:

TestNG

Инструменты для тестирования единиц

Для модульного тестирования используется несколько других инструментов:

Мокито - Издевательская структура; позволяет копировать объекты. Полезно для имитации **ожидаемого** поведения внешнего устройства в рамках теста данного устройства, чтобы не связывать поведение внешнего устройства с тестами данного устройства.

JBehave - BDD Framework. Позволяет связать тесты с пользовательским поведением (разрешая проверку требований / сценариев). *Отсутствие ярлыков документов на момент написания; вот внешняя ссылка* .

Examples

Что такое Unit Testing?

Это немного грунт. В основном это связано с тем, что документация вынуждена иметь пример, даже если он предназначен как статья-заглушка. Если вы уже знаете основы модульного тестирования, не стесняйтесь переходить к замечаниям, где упоминаются конкретные рамки.

Модульное тестирование - это обеспечение того, что данный модуль ведет себя так, как ожидалось. В крупномасштабных приложениях обеспечение надлежащего выполнения модулей в вакууме является неотъемлемой частью обеспечения верности приложения.

Рассмотрим следующий (тривиальный) псевдо-пример:

```
public class Example {
  public static void main (String args[]) {
    new Example();
  }

  // Application-level test.
  public Example() {
    Consumer c = new Consumer();
    System.out.println("VALUE = " + c.getVal());
```

```
  }

  // Your Module.
  class Consumer {
    private Capitalizer c;

    public Consumer() {
      c = new Capitalizer();
    }

    public String getVal() {
      return c.getVal();
    }
  }

  // Another team's module.
  class Capitalizer {
    private DataReader dr;

    public Capitalizer() {
      dr = new DataReader();
    }

    public String getVal() {
      return dr.readVal().toUpperCase();
    }
  }

  // Another team's module.
  class DataReader {
    public String readVal() {
      // Refers to a file somewhere in your application deployment, or
      // perhaps retrieved over a deployment-specific network.
      File f;
      String s = "data";
      // ... Read data from f into s ...
      return s;
    }
  }
}
```

Итак, этот пример тривиален; DataReader получает данные из файла, передает его в « Capitalizer , который преобразует все символы в верхний регистр, который затем передается Consumer . Но DataReader сильно связан с нашей прикладной средой, поэтому мы откладываем тестирование этой цепочки, пока не готовы к развертыванию тестовой версии.

Теперь предположим, что где-то по пути в релизе по неизвестным причинам метод getVal() в Capitalizer изменился с возврата строки toUpperCase() строку toLowerCase() :

```
  // Another team's module.
  class Capitalizer {
    ...

    public String getVal() {
      return dr.readVal().toLowerCase();
    }
  }
```

Ясно, что это нарушает ожидаемое поведение. Но из-за трудных процессов, связанных с выполнением DataReader , мы не заметим этого до нашего следующего тестового развертывания. Таким образом, дни / недели / месяцы проходят с этой ошибкой, сидящей в нашей системе, а затем менеджер

продукта видит это и мгновенно обращается к вам, руководителю группы, связанному с Consumer . «Почему это происходит? Что вы, ребята, изменили?» Очевидно, ты не знаешь. Вы понятия не имеете, что происходит. Вы не изменили код, который должен касаться этого; почему это внезапно нарушено?

В конце концов, после обсуждения между командами и совместной работой проблема прослеживается, и проблема решена. Но он задает вопрос; как это могло быть предотвращено?

Есть две очевидные вещи:

Тесты должны быть автоматизированы

Наша уверенность в ручном тестировании позволяет этой ошибке проходить незаметно слишком долго. Нам нужен способ автоматизировать процесс, при котором ошибки вводятся **мгновенно** . Не через 5 недель. Не через 5 дней. Не через 5 минут. Прямо сейчас.

Вы должны оценить, что в этом примере я изложил одну **очень тривиальную** ошибку, которая была введена и незаметна. В промышленном применении с постоянно обновляющимися десятками модулей они могут ползти повсюду. Вы исправляете что-то с помощью одного модуля, только чтобы понять, что само поведение, которое вы «зафиксировали», каким-то образом опиралось на другое (внутреннее или внешнее).

Без строгой проверки все будет проникать в систему. Возможно, что, если пренебречь достаточно далеко, это приведет к такой дополнительной работе, которая поможет исправить изменения (а затем исправить эти исправления и т. Д.), Что продукт действительно **увеличится** в оставшейся работе, так как в нее будут включены усилия. Вы не хотите быть в такой ситуации.

Тесты должны быть мелкозернистыми

Вторая проблема, отмеченная в нашем примере выше, - это время, затраченное на прослеживание ошибки. Менеджер продукта пинговал вас, когда тестеры заметили это, вы исследовали и обнаружили, что « Capitalizer возвращал, казалось бы, плохие данные, вы пинали команду « Capitalizer » вашими результатами, они исследовали и т. Д. И т. Д. И т. Д.

То же самое, что я сделал выше о количестве и сложности этого тривиального примера, здесь. Очевидно, что любой разумно хорошо разбирающийся в Java может быстро найти введенную проблему. Но часто и гораздо труднее отслеживать и передавать проблемы. Возможно, команда Capitalizer предоставила вам JAR без источника. Возможно, они расположены на другой стороне мира, а часы общения очень ограничены (возможно, по электронной почте, которые отправляются один раз в день). Это может привести к ошибкам, требующим недель или больше для отслеживания (и, опять же, для данной версии может быть несколько).

Для того , чтобы смягчать против этого, мы хотим , тщательное тестирование на максимально **тонкий** уровне , насколько это возможно (вы также хотите крупнозернистое тестирование , чтобы обеспечить модули взаимодействуют должным образом, но это не наш фокус здесь). Мы хотим строго указать, как работают все внешние функции (как минимум), и тесты для этой функциональности.

Введите модульное тестирование

Представьте себе, если бы у нас был тест, в частности, чтобы метод getVal() Capitalizer возвращал getVal() строку для данной входной строки. Кроме того, представьте, что тест был выполнен до того, как мы даже совершили какой-либо код. Ошибка, введенная в систему (то есть toUpperCase() заменяется на toLowerCase()), не вызовет проблем, потому что ошибка никогда не **будет** введена в систему. Мы поймаем его в тесте, разработчик (надеюсь) осознает свою ошибку, и будет найдено альтернативное решение о том, как внедрить их предполагаемый эффект.

Здесь приводятся некоторые упущения в отношении **того**, **как** выполнять эти тесты, но они описаны в документации по структуре (см. Примечания). Надеюсь, это служит примером того, **почему** модульное тестирование важно.

Прочитайте Тестирование устройства онлайн: https://riptutorial.com/ru/java/topic/8155/тестирование-устройства

глава 173: Типы ссылок

Examples

Различные типы ссылок

Пакет java.lang.ref предоставляет классы ссылочных объектов, которые поддерживают ограниченную степень взаимодействия с сборщиком мусора.

Java имеет четыре основных типа ссылок. Они есть:

- Сильная ссылка
- Слабая ссылка
- Мягкая ссылка
- Ссылка на Phantom

1. Сильная ссылка

Это обычная форма создания объектов.

```
MyObject myObject = new MyObject();
```

Держатель переменной содержит значительную ссылку на созданный объект. Пока эта переменная активна и сохраняет это значение, экземпляр MyObject не будет собираться сборщиком мусора.

2. Слабая ссылка

Если вы не хотите, чтобы объект был длиннее, и вам нужно как можно скорее очистить / освободить память, выделенную для объекта, это способ сделать это.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Просто слабая ссылка - это ссылка, которая недостаточно сильна, чтобы заставить объект оставаться в памяти. Слабые ссылки позволяют вам использовать способность сборщика мусора определять доступность для вас, поэтому вам не нужно делать это самостоятельно.

Когда вам нужен созданный объект, просто используйте .get() :

```
myObjectRef.get();
```

Следующий код иллюстрирует это:

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the object reference address
System.gc();
System.out.println(myObjectRef.get()); // This will print 'null' if the GC cleaned up the
object
```

3. Мягкая ссылка

Мягкие ссылки немного сильнее слабых ссылок. Вы можете создать мягкий ссылочный объект следующим образом:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

Они могут удерживать память сильнее, чем слабая ссылка. Если у вас достаточно ресурсов / ресурсов памяти, сборщик мусора не будет чистить мягкие ссылки с энтузиазмом, как слабые ссылки.

Мягкие ссылки удобны для использования в кешировании. Вы можете создавать объекты с мягкими

ссылками в виде кеша, где они сохраняются до тех пор, пока ваша память не закончится. Когда ваша память не может предоставить достаточное количество ресурсов, сборщик мусора удалит мягкие ссылки.

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the reference address of the Object
System.gc();
System.out.println(myObjectRef.get()); // This may or may not print the reference address of
the Object
```

4. Фантомная ссылка

Это самый слабый ссылочный тип. Если вы создали ссылку на объект с помощью Phantom Reference, метод get() всегда будет возвращать значение null!

Использование этой ссылки состоит в том, что «Объекты ссылки Phantom, которые выставляются после коллектора, определяют, что их референты могут быть повторно возвращены. Фантомные ссылки чаще всего используются для планирования действий по предотвращению вскрытия более гибким способом, чем это возможно с помощью Механизм завершения Java. " - От Phantom Reference Javadoc от Oracle.

Вы можете создать объект Phantom Reference следующим образом:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Прочитайте Типы ссылок онлайн: https://riptutorial.com/ru/java/topic/4017/типы-ссылок

глава 174: Типы ссылочных данных

Examples

Создание экземпляра ссылочного типа

```
Object obj = new Object(); // Note the 'new' keyword
```

Куда:

- Object является ссылочным типом.
- obj - это переменная, в которой сохраняется новая ссылка.
- Object() - это вызов конструктора Object .

Что просходит:

- Пространство в памяти выделяется для объекта.
- Конструктор Object() вызывается для инициализации этого пространства памяти.
- Адрес памяти сохраняется в obj , так что он *ссылается* на вновь созданный объект.

Это отличается от примитивов:

```
int i = 10;
```

Если фактическое значение 10 хранится в i .

Разыменование

Выделение происходит с помощью . оператор:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Выделение *следует* за адресом памяти, хранящимся в ссылке, до места в памяти, где находится фактический объект. Когда объект найден, запрашиваемый метод вызывается (toString в этом случае).

Когда ссылка имеет значение null , разыменование приводит к исключению NullPointerException :

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null указывает на отсутствие значения, то есть *после* адреса памяти нигде не происходит. Таким образом, нет объекта, на который может быть вызван запрошенный метод.

Прочитайте Типы ссылочных данных онлайн: https://riptutorial.com/ru/java/topic/1046/типы-ссылочных-данных

глава 175: Удаленный вызов метода (RMI)

замечания

RMI требует 3 компонента: клиент, сервер и общий удаленный интерфейс. Общий удаленный интерфейс определяет контракт клиент-сервер, указывая методы, которые должен выполнить сервер. Интерфейс должен быть видимым для сервера, чтобы он мог реализовать методы; интерфейс должен быть виден клиенту, чтобы он знал, какие методы («службы») предоставляют сервер.
Любой объект, реализующий удаленный интерфейс, должен взять на себя роль сервера. Таким образом, отношения клиент-сервер, в которых сервер также может вызывать методы в клиенте, на самом деле являются отношениями сервер-сервер. Это называется *обратным вызовом*, так как сервер может перезванивать «клиент». Имея это в виду, допустимо использовать *клиент* назначения для серверов, которые функционируют как таковые.

Общий удаленный интерфейс - это любой интерфейс, расширяющий Remote . Объект, который функционирует как сервер, претерпевает следующие действия:

1. Реализует общий удаленный интерфейс, явно или неявно, путем расширения UnicastRemoteObject который реализует Remote .
2. Экспортируется либо неявно, если он расширяет UnicastRemoteObject , либо явно передается в UnicastRemoteObject#exportObject .
3. Связан в реестре, либо напрямую через Registry либо косвенно через Naming . Это необходимо только для установления первоначальной связи, поскольку дальнейшие заглушки могут быть переданы непосредственно через RMI.

В настройке проекта проекты клиента и сервера полностью не связаны друг с другом, но каждый указывает общий проект в своем пути сборки. Общий проект содержит удаленные интерфейсы.

Examples

Client-Server: вызов методов в одном JVM из другого

Общий удаленный интерфейс:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

Сервер, реализующий общий удаленный интерфейс:

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class Server implements RemoteServer {

    @Override
    public int stringToInt(String string) throws RemoteException {
```

```
        System.out.println("Server received: \"" + string + "\"");
        return Integer.parseInt(string);
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Server server = new Server();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("ServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Клиент, вызывающий метод на сервере (удаленно):

```
package client;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.RemoteServer;

public class Client {

    static RemoteServer server;

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.getRegistry();
            server = (RemoteServer) reg.lookup("ServerName");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        Client client = new Client();
        client.callServer();
    }

    void callServer() {

        try {
            int i = server.stringToInt("120");
            System.out.println("Client received: " + i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Выход:

```
Полученный сервер: «120»
Полученный клиент: 120
```

Обратный вызов: вызов методов на «клиент»

обзор

В этом примере 2 клиента отправляют информацию друг другу через сервер. Один клиент отправляет серверу номер, который передается второму клиенту. Второй клиент уменьшает число и отправляет его первому клиенту через сервер. Первый клиент делает то же самое. Сервер останавливает связь, когда число, возвращаемое им любым из клиентов, меньше 10. Возвращаемое значение от сервера к клиентам (номер, который он преобразовал в строковое представление), затем возвращает процесс.

1. Сервер входа в систему привязывается к реестру.
2. Клиент просматривает сервер входа в систему и вызывает метод login в login со своей информацией. Затем:
    - Клиентский сервер хранит информацию о клиенте. Он включает в себя заглушку клиента с методами обратного вызова.
    - Сервер входа в систему создает и возвращает серверную заглушку («соединение» или «сеанс») для хранения клиента. Он включает в себя заглушку сервера с его методами, включая метод logout из logout (в этом примере не используется).
3. Клиент вызывает passInt сервера с именем клиента-получателя и int .
4. Сервер вызывает half на клиенте получателя с этим int . Это инициирует обратную связь (обратные вызовы и обратные вызовы) до остановки сервера.

Общие удаленные интерфейсы

Сервер входа в систему:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

Сервер:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {

    void logout() throws RemoteException;

    String passInt(String name, int i) throws RemoteException;
}
```

Клиент:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
```

```
}
```

Реализации

Сервер входа в систему:

```
package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {

    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {

        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Сервер:

```
package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;

public class Connection implements RemoteConnection, Unreferenced {
```

```
    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}
```

Клиент:

```
package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import callbackRemote.RemoteClient;
```

```
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        if ("Client1".equals(client.name)) {
            try {
                client.connection.passInt(client.target, 120);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void half(int i) throws RemoteException {

        String result = connection.passInt(target, i / 2);
        System.out.println(name + " received: \"" + result + "\"");
    }
}
```

Запуск примера:

1. Запустите сервер входа.
2. Запустите клиент с аргументами Client2 Client1 1097 .
3. Запустите клиент с аргументами Client1 Client2 1098 .

Выходы появятся на 3 консолях, так как есть 3 JVM. здесь они собраны вместе:

```
Клиент2 вошел в систему
Клиент1 вошел в систему
Сервер, полученный от Client1: 120
Сервер, полученный от Client2: 60
Сервер, полученный от Client1: 30
Сервер, полученный от Client2: 15
Сервер, полученный от Client1: 7
Клиент1 получил: «7»
Клиент2 получил: «15»
Клиент1 получил: «30»
Клиент2 получил: «60»
```

Простой пример RMI с внедрением Client и Server

Это простой пример RMI с пятью классами Java и двумя пакетами, *сервером* и *клиентом* .

Серверный пакет

PersonListInterface.java

```
public interface PersonListInterface extends Remote
{
    /**
     * This interface is used by both client and server
     * @return List of Persons
     * @throws RemoteException
     */
    ArrayList<String> getPersonList() throws RemoteException;
}
```

PersonListImplementation.java

```
public class PersonListImplementation
extends UnicastRemoteObject
implements PersonListInterface
{

    private static final long serialVersionUID = 1L;

    // standard constructor needs to be available
    public PersonListImplementation() throws RemoteException
    {}

    /**
     * Implementation of "PersonListInterface"
     * @throws RemoteException
     */
    @Override
    public ArrayList<String> getPersonList() throws RemoteException
    {
        ArrayList<String> personList = new ArrayList<String>();

        personList.add("Peter Pan");
        personList.add("Pippi Langstrumpf");
        // add your name here :)

        return personList;
    }
}
```

Server.java

```
public class Server {

    /**
     * Register servicer to the known public methods
     */
    private static void createServer() {
        try {
            // Register registry with standard port 1099
            LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
```

```
            System.out.println("Server : Registry created.");

            // Register PersonList to registry
            Naming.rebind("PersonList", new PersonListImplementation());
            System.out.println("Server : PersonList registered");

        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(final String[] args) {
        createServer();
    }
}
```

Клиентский пакет

PersonListLocal.java

```
public class PersonListLocal {
    private static PersonListLocal instance;
    private PersonListInterface personList;

    /**
     * Create a singleton instance
     */
    private PersonListLocal() {
        try {
            // Lookup to the local running server with port 1099
            final Registry registry = LocateRegistry.getRegistry("localhost",
                    Registry.REGISTRY_PORT);

            // Lookup to the registered "PersonList"
            personList = (PersonListInterface) registry.lookup("PersonList");
        } catch (final RemoteException e) {
            e.printStackTrace();
        } catch (final NotBoundException e) {
            e.printStackTrace();
        }
    }

    public static PersonListLocal getInstance() {
        if (instance == null) {
            instance = new PersonListLocal();
        }

        return instance;
    }

    /**
     * Returns the servers PersonList
     */
    public ArrayList<String> getPersonList() {
        if (instance != null) {
            try {
                return personList.getPersonList();
            } catch (final RemoteException e) {
                e.printStackTrace();
            }
        }
```

```
        return new ArrayList<>();
    }
       }
```

PersonTest.java

```
public class PersonTest
{
    public static void main(String[] args)
    {
        // get (local) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // print all persons
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}
```

Проверьте свое приложение

- Начать основной метод Server.java. Выход:

```
Server : Registry created.
Server : PersonList registered
```

- Начните основной метод PersonTest.java. Выход:

```
Peter Pan
Pippi Langstrumpf
```

Прочитайте Удаленный вызов метода (RMI) онлайн: https://riptutorial.com/ru/java/topic/171/удаленный-вызов-метода--rmi-

глава 176: Управление памятью Java

замечания

В Java объекты выделяются в куче, а память кучи утилизируется с помощью автоматической сборки мусора. Приложение не может явно удалить объект Java.

Основные принципы сборки мусора Java описаны в примере коллекции Мусора . В других примерах описывается окончательная доработка, как вручную запускать сборщик мусора и проблема утечек хранилища.

Examples

завершение

Объект Java может объявить метод finalize . Этот метод вызывается непосредственно перед тем, как Java освобождает память для объекта. Он будет выглядеть следующим образом:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

Однако есть некоторые важные предостережения о поведении завершения Java.

- Java не дает никаких гарантий относительно того, когда вызывается метод finalize() .
- Java даже не гарантирует, что метод finalize() будет вызываться некоторое время в течение всего срока действия исполняемого приложения.
- Единственное, что гарантировано, это то, что метод будет вызываться до того, как объект будет удален ... если объект будет удален.

Предостережения выше означают, что плохой идеей полагаться на метод finalize для выполнения действий по очистке (или других), которые должны выполняться своевременно. Опора на завершение может привести к утечкам памяти, утечкам памяти и другим проблемам.

Короче говоря, очень мало ситуаций, когда финализация на самом деле является хорошим решением.

Финализаторы запускаются только один раз

Обычно объект удаляется после его завершения. Однако этого не происходит постоянно. Рассмотрим следующий пример [1] :

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

Когда экземпляр CaptainJack становится недоступным и сборщик мусора пытается его восстановить, метод finalize() назначит ссылку на экземпляр переменной notDeadYet . Это сделает экземпляр доступным еще раз, и сборщик мусора не удалит его.

Вопрос: Капитан Джек бессмертен?

Ответ: Нет.

Ловушка JVM будет запускать только финализатор на объект один раз в жизни. Если вы назначаете null для notDeadYet чтобы notDeadYet недопустимый экземпляр, сборщик мусора не будет вызывать finalize() для объекта.

1 - См. Https://en.wikipedia.org/wiki/Jack_Harkness .

Ручное включение GC

Вы можете вручную запустить сборщик мусора, позвонив

```
System.gc();
```

Однако Java не гарантирует, что сборщик мусора запускается при возврате вызова. Этот метод просто «предлагает» JVM (виртуальная машина Java), что вы хотите, чтобы он запускал сборщик мусора, но не заставлял его это делать.

Обычно считается неправильной практикой попытки вручную запускать сборку мусора. JVM можно запустить с -XX:+DisableExplicitGC чтобы отключить вызовы System.gc() . Запуск сбора мусора, вызвав System.gc() может привести к нарушению нормальной работы по управлению мусором / объектам для конкретной реализации сборщика мусора, используемого JVM.

Вывоз мусора

Подход C ++ - новый и удаленный

На языке, подобном C ++, прикладная программа отвечает за управление памятью, используемой динамически распределенной памятью. Когда объект создается в куче C ++ с использованием new оператора, должно быть соответствующее использование оператора delete для утилизации объекта:

- Если программа забывает delete объект и просто «забывает» об этом, связанная с ним память будет потеряна для приложения. Термин для этой ситуации - *утечка памяти* , и слишком много утечек памяти приложение может использовать все больше и больше памяти и в конечном итоге сбой.
- С другой стороны, если приложение пытается дважды delete тот же объект или использовать объект после его удаления, приложение может быть повреждено из-за проблем с повреждением памяти

В сложной программе на C ++ реализация управления памятью с использованием new и delete может занять много времени. Действительно, управление памятью является общим источником ошибок.

Подход Java - сбор мусора

Java использует другой подход. Вместо явного оператора delete Java предоставляет автоматический механизм, известный как сбор мусора, для восстановления памяти, используемой объектами, которые больше не нужны. Система времени выполнения Java берет на себя ответственность за поиск объектов, которые должны быть удалены. Эта задача выполняется компонентом, называемым *сборщиком мусора* , или GC для краткости.

В любое время во время выполнения Java-программы мы можем разделить набор всех существующих объектов на два разных подмножества [1] :

- Достижимые объекты определяются JLS следующим образом:

  > Досягаемым объектом является любой объект, к которому можно получить доступ в любом потенциальном продолжающемся вычислении из любой живой нити.

  На практике это означает, что существует цепочка ссылок, начиная с локальной переменной

области или static переменной, с помощью которой какой-то код может быть доступен для объекта.

- Недостижимыми объектами являются объекты, которые *не могут* быть достигнуты, как указано выше.

Любые объекты, недостижимые, имеют *право* на сбор мусора. Это не означает, что они *будут* собирать мусор. По факту:

- Недоступный объект *не* получает сразу, когда становится недоступным. [1] .
- Недостижимый объект *может не* быть собранным мусором.

Спецификация языка Java дает большую широту реализации JVM, чтобы решить, когда собирать недостижимые объекты. Он также (на практике) дает разрешение для реализации JVM быть консервативным в том, как он обнаруживает недоступные объекты.

Единственное , что гарантирует JLS , что никакие *достижимые* объекты никогда не будет мусора.

Что происходит, когда объект становится недоступным

Прежде всего, ничего не происходит, когда объект *становится* недоступным. Все происходит только тогда, когда работает сборщик мусора, *и* он обнаруживает, что объект недоступен. Кроме того, для GC-GC обычно не обнаруживать все недостижимые объекты.

Когда GC обнаруживает недостижимый объект, могут произойти следующие события.

1. Если есть какие-либо объекты Reference , относящиеся к объекту, эти ссылки будут удалены до того, как объект будет удален.
2. Если объект *финализируемый,* то он будет завершен. Это происходит до удаления объекта.
3. Объект можно удалить, и память, которую он занимает, может быть восстановлена.

Обратите внимание, что существует четкая последовательность, в которой *могут* произойти указанные события, но ничто не требует, чтобы сборщик мусора выполнял окончательное удаление какого-либо конкретного объекта в любом конкретном временном кадре.

Примеры достижимых и недоступных объектов

Рассмотрим следующие примеры классов:

```
// A node in simple "open" linked-list.
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test();                    // M1
        System.out.prinln("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node();      // T1
        Node n2 = new Node();      // T2
        Node n3 = new Node();      // T3
        n1.next = n2;              // T4
```

```
        n2 = null;                  // T5
        n3 = null;                  // T6
    }
}
```

Давайте посмотрим, что происходит, когда вызывается test() . Заявления T1, T2 и T3 создают объекты Node , и все объекты достижимы через переменные n1 , n2 и n3 соответственно. Заявление T4 назначает ссылку на объект второго Node на next поле первого. Когда это будет сделано, 2-й Node доступен по двум путям:

```
n2 -> Node2
n1 -> Node1, Node1.next -> Node2
```

В заявлении T5 мы присваиваем значение null для n2 . Это разрушает первую из цепей достижимости для Node2 , но вторая остается неизменной, поэтому Node2 все еще доступен.

В заявлении T6 мы присваиваем значение null для n3 . Это нарушает единственную цепочку достижимости для Node3 , что делает Node3 недоступным. Однако Node1 и Node2 все еще доступны через переменную n1 .

Наконец, когда метод test() возвращается, его локальные переменные n1 , n2 и n3 выходят за пределы области видимости и поэтому не могут быть доступны никем. Это разрушает оставшиеся цепи достижимости для Node1 и Node2 , а все объекты Node также недоступны и не *могут быть использованы* для сбора мусора.

---

1 - Это упрощение, которое игнорирует завершение и Reference классы. 2 - Гипотетически, реализация Java может сделать это, но стоимость выполнения этого делает ее нецелесообразной.

Установка размеров кучи, пермгана и стека

Когда запускается виртуальная машина Java, она должна знать, как увеличить размер кучи и размер по умолчанию для стеков потоков. Они могут быть указаны с помощью параметров командной строки в команде java . Для версий Java до Java 8 вы также можете указать размер области PermGen кучи.

Обратите внимание, что PermGen был удален в Java 8, и если вы попытаетесь установить размер PermGen, этот параметр будет проигнорирован (с предупреждающим сообщением).

Если вы явно не укажете размеры кучи и стека, JVM будет использовать значения по умолчанию, которые рассчитываются в версии и на платформе. Это может привести к тому, что ваше приложение будет использовать слишком мало или слишком много памяти. Обычно это нормально для стеков потоков, но это может быть проблематично для программы, которая использует много памяти.

Настройка размеров кучи, PermGen и стандартных стеков:

Следующие параметры JVM задают размер кучи:

- -Xms<size> - устанавливает начальный размер кучи
- -Xmx<size> - устанавливает максимальный размер кучи
- -XX:PermSize<size> - устанавливает начальный размер PermGen
- -XX:MaxPermSize<size> - устанавливает максимальный размер PermGen
- -Xss<size> - устанавливает размер стека по умолчанию

Параметр <size> может быть числом байтов или может иметь суффикс k , m или g . Последние определяют размер в килобайтах, мегабайтах и гигабайтах соответственно.

Примеры:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Поиск размеров по умолчанию:

Опция -XX:+printFlagsFinal может использоваться для печати значений всех флагов перед запуском JVM. Это можно использовать для печати значений по умолчанию для настроек размера кучи и размера стека следующим образом:

- Для Linux, Unix, Solaris и Mac OSX

  ```
  $ java -XX: + PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'
  ```

- Для Windows:

  ```
  java -XX: + PrintFlagsFinal -version | findstr / i "HeapSize PermSize
  ThreadStackSize"
  ```

Вывод приведенных выше команд будет выглядеть следующим образом:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
 intx ThreadStackSize                           = 1024            {pd product}
```

Размеры указаны в байтах.

Утечка памяти в Java

В примере коллекции Garbage мы подразумеваем, что Java решает проблему утечки памяти. На самом деле это не так. Программа Java может просачивать память, хотя причины утечек довольно разные.

Доступные объекты могут протекать

Рассмотрим следующую реализацию наивного стека.

```
public class NaiveStack {
    private Object[] stack = new Object[100];
    private int top = 0;

    public void push(Object obj) {
        if (top >= stack.length) {
            throw new StackException("stack overflow");
        }
        stack[top++] = obj;
    }

    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        return stack[--top];
    }

    public boolean isEmpty() {
        return top == 0;
    }
}
```

Когда вы push объект, а затем сразу же pop , все равно будет ссылка на объект в массиве stack .

Логика реализации стека означает, что эта ссылка не может быть возвращена клиенту API. Если объект был вытолкнут, мы можем доказать, что его нельзя *«получить доступ к любому потенциальному продолжающемуся вычислению из любой живой нити»* . Проблема в том, что JVM текущего поколения не

может этого доказать. Существующие JVM поколения не рассматривают логику программы при определении доступности ссылок. (Для начала это не практично.)

Но, отвлекаясь от вопроса о том, что означает действительно *достижимость* , у нас явно есть ситуация, когда реализация NaiveStack «висит на» объектах, которые должны быть исправлены. Это утечка памяти.

В этом случае решение прост:

```
    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        Object popped = stack[--top];
        stack[top] = null;              // Overwrite popped reference with null.
        return popped;
    }
```

Кэши могут быть утечками памяти

Общей стратегией повышения эффективности обслуживания является кэширование результатов. Идея состоит в том, что вы сохраняете запись общих запросов и их результатов в структуре данных в памяти, известной как кеш. Затем каждый раз, когда запрос выполняется, вы просматриваете запрос в кеше. Если поиск выполняется успешно, вы возвращаете соответствующие сохраненные результаты.

Эта стратегия может быть очень эффективной, если ее выполнить правильно. Однако, если они реализованы неправильно, кеш может быть утечкой памяти. Рассмотрим следующий пример:

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result == doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

Проблема с этим кодом заключается в том, что, хотя любой вызов doRequest может добавить новую запись в кеш, их не удалять. Если служба постоянно выполняет разные задачи, то кэш в конечном итоге будет потреблять всю доступную память. Это форма утечки памяти.

Один из подходов к решению этого - использовать кеш с максимальным размером и выкидывать старые записи, когда кеш превышает максимальный. (Выдача наименее недавно использованной записи - хорошая стратегия.) Другой подход заключается в создании кеша с использованием WeakHashMap чтобы JVM могла выселить записи кэша, если куча начинает слишком WeakHashMap .

Прочитайте Управление памятью Java онлайн: https://riptutorial.com/ru/java/topic/2804/управление-памятью-java

глава 177: Установка Java (стандартная версия)

Вступление

На этой странице документации вы найдете инструкции по установке java standard edition на Windows , Linux и macOS .

Examples

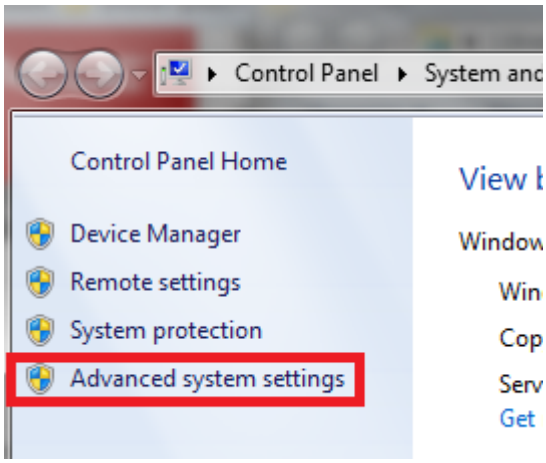
Установка% PATH% и% JAVA_HOME% после установки в Windows

Предположения:

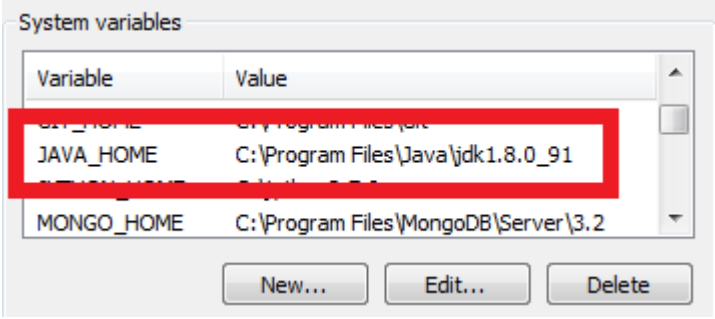
- Был установлен Oracle JDK.
- JDK был установлен в каталог по умолчанию.

Шаги настройки

1. Откройте проводник Windows.
2. В навигационной панели слева щелкните правой кнопкой мыши *этот компьютер* (или *компьютер* для более старых версий Windows). Существует более короткий путь, не используя проводник в реальных версиях Windows: просто нажмите Win + Pause
3. В открывшемся окне панели управления щелкните левой кнопкой мыши « *Дополнительные системные настройки*», который должен находиться в верхнем левом углу. Откроется окно « *Свойства системы*» .

   В качестве альтернативы, введите SystemPropertiesAdvanced (без SystemPropertiesAdvanced регистра) в *Run* (Win + R) и нажмите Enter .
4. На вкладке « *Дополнительно* » в разделе « *Свойства системы*» выберите кнопку « Переменные среды ...» в правом нижнем углу окна.
5. Добавьте **новую системную переменную** , нажав кнопку « Создать ...» в *системных переменных* с именем JAVA_HOME и значением которой является путь к каталогу, в котором установлен JDK. После ввода этих значений нажмите OK .



6. Прокрутите список *системных переменных* и выберите переменную Path .

7. **ВНИМАНИЕ:** Windows использует Path для поиска важных программ. Если какой-либо или все из них удалено, Windows может не работать должным образом. Он должен быть изменен, чтобы позволить Windows запускать JDK. Имея это в виду, нажмите кнопку «Изменить ...» с выбранной переменной Path . Добавить %JAVA_HOME%\bin; к началу переменной Path .

Лучше добавить в начале строки, потому что программное обеспечение Oracle, используемое для регистрации своей собственной версии Java в Path - это приведет к игнорированию вашей версии, если это произойдет после объявления Oracle.

Проверьте свою работу

1. Откройте командную строку, нажав «Пуск», затем Enter cmd и нажмите « Enter .
2. Введите javac -version в приглашение. Если это было успешно, версия JDK будет напечатана на экране.

Примечание. Если вам нужно попробовать еще раз, закройте приглашение, прежде чем проверять свою работу. Это заставит окна получить новую версию Path .

Выбор подходящей версии Java SE

Было много выпусков Java с момента выпуска оригинальной версии Java 1.0 в 1995 году. (См . Историю версий Java для сводки.) Однако большинство выпусков прошли официальные даты окончания жизни. Это означает, что поставщик (как правило, Oracle сейчас) прекратил новую разработку для выпуска и больше не предоставляет публичные / бесплатные исправления для любых ошибок или проблем безопасности. (Частные выпуски патчей обычно доступны для людей / организаций с контрактом на поддержку, обратитесь в офис продаж вашего продавца.)

В общем, рекомендуемая версия Java SE для использования будет последним обновлением для последней публичной версии. В настоящее время это означает самую последнюю доступную версию Java 8. Java 9 должен выйти на публичный выпуск в 2017 году. (Java 7 прошла свой End Of Life, а последний публичный релиз был в апреле 2015 года. Java 7 и более ранние версии не рекомендуются.)

Эта рекомендация применяется ко всем новым разработкам Java, и любому, кто изучает Java. Это также относится к людям, которые просто хотят запускать программное обеспечение Java, предоставляемое сторонним разработчиком. Вообще говоря, хорошо написанный Java-код будет работать на более новой версии Java. (Но проверьте примечания к выпуску программного обеспечения и обратитесь к автору / поставщику / поставщику, если у вас есть сомнения.)

Если вы работаете с более старой Java-кодовой базой, вам следует убедиться, что ваш код работает в последней версии Java. Решение о том, когда начать использовать функции более новых версий Java, сложнее, так как это повлияет на вашу способность поддерживать клиентов, которые не могут или не хотят их установки на Java.

Выпуск Java и именование версий

Именование Java-релиза немного запутанно. На самом деле существуют две системы именования и нумерации, как показано в этой таблице:

Версия JDK	Название маркетинга
JDK-1,0	JDK 1.0
JDK-1,1	JDK 1.1
JDK-1,2	J2SE 1.2
...	...
JDK-1,5	J2SE 1.5 переименовал Java SE 5
JDK-1,6	Java SE 6
JDK-1,7	Java SE 7
JDK-1,8	Java SE 8
jdk-9 [1]	Java SE 9 (еще не выпущен)

1 - Похоже, что Oracle намерена отказаться от своей предыдущей практики использования схемы «семантической версии» в строках Java-версии. Остается увидеть, будут ли они следовать этому.

«SE» в маркетинговых именах относится к Standard Edition. Это базовая версия для запуска Java на большинстве ноутбуков, ПК и серверов (помимо Android).

Существуют два других официальных выпуска Java: «Java ME» - это Micro Edition, а «Java EE» - это Enterprise Edition. Android-стиль Java также значительно отличается от Java SE. Java ME, Java EE и Android Java не входят в сферу применения этой темы.

Полный номер версии для Java-версии выглядит следующим образом:

```
1.8.0_101-b13
```

Это говорит JDK 1.8.0, Update 101, Build # 13. Oracle ссылается на это в примечаниях к выпуску как:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

Номер обновления важен - Oracle регулярно выпускает обновления для основной версии с исправлениями безопасности, исправлениями ошибок и (в некоторых случаях) новыми функциями. Номер сборки обычно не имеет значения. Обратите внимание, что Java 8 и Java 1.8 *относятся к одной и той же вещи* ; Java 8 - это просто «маркетинговое» название для Java 1.8.

Что мне нужно для разработки Java

Установка JDK и текстовый редактор являются минимальным для разработки Java. (Приятно иметь текстовый редактор, который может выделять синтаксис Java, но вы можете обойтись без него.)

Однако для серьезных разработок рекомендуется также использовать следующее:

- Java IDE, такой как Eclipse, Intellij IDEA или NetBeans
- Инструмент построения Java, такой как Ant, Gradle или Maven
- Система управления версиями для управления вашей базой кода (с соответствующими резервными копиями и репликацией вне сайта)
- Инструменты тестирования и инструменты CI (непрерывная интеграция)

Использование диспетчера пакетов

JDK и / или выпуски JRE для OpenJDK или Oracle могут быть установлены с использованием диспетчера пакетов в большинстве распространенных дистрибутивов Linux. (Доступные вам варианты зависят от дистрибутива.)

Как правило, процедура заключается в открытии окна терминала и выполнении команд, показанных ниже. (Предполагается, что у вас есть достаточный доступ к командам запуска в качестве «корневого» пользователя ... это то, что делает команда sudo . Если вы этого не сделаете, обратитесь к администраторам вашей системы.)

Рекомендуется использовать диспетчер пакетов, поскольку он (как правило) упрощает обновление вашей Java-установки.

apt-get , дистрибутивы Linux на базе Debian (Ubuntu и т. д.)

Следующие инструкции установят Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Примечание. Чтобы автоматически настроить переменные среды Java 8, вы можете установить следующий пакет:

```
$ sudo apt-get install oracle-java8-set-default
```

Создание файла .deb

Если вы предпочитаете самостоятельно создавать файл .deb из файла .tar.gz загруженного из Oracle, выполните следующие действия (при условии, что вы загрузили файл .tar.gz в ./<jdk>.tar.gz):

```
$ sudo apt-get install java-package # might not be available in default repos
$ make-jpkg ./<jdk>.tar.gz          # should not be run as root
$ sudo dpkg -i *j2sdk*.deb
```

Примечание : ожидается, что вход будет представлен как файл «.tar.gz».

slackpkg , дистрибутивы Linux на основе Slackware

```
sudo slapt-get install default-jdk
```

yum , RedHat, CentOS и т. д.

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

dnf , Fedora

В последних версиях Fedora yum был заменен dnf .

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

В последних выпусках Fedora пакетов для установки Java 7 и более ранних версий нет.

pacman , дистрибутивы Linux на базе Arch

```
sudo pacman -S jdk8-openjdk
```

Использование sudo не требуется, если вы работаете как пользователь root.

Gentoo Linux

Руководство Gentoo Java поддерживается командой Gentoo Java и хранит обновленную страницу вики, включая нужные пакеты портов и флаги USE.

Установка Oracle JDK на Redhat, CentOS, Fedora

Установка JDK из файла Oracle JDK или JRE tar.gz

1. Загрузите соответствующий файл архива Oracle («tar.gz») для нужной версии с сайта загрузки Oracle Java .
2. Измените каталог на место, где вы хотите установить установку;
3. Декомпрессировать файл архива; например

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Установка из файла RPM Oracle Java.

1. Получите требуемый файл RPM для нужной версии с сайта загрузки Oracle Java .
2. Установите с помощью команды rpm . Например:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Установка Java JDK или JRE в Windows

Только Oracle JDK и JRE доступны для платформ Windows. Процедура установки проста:

1. Перейдите на страницу загрузки Oracle Java:
2. Нажмите кнопку JDK, кнопку JRE или кнопку сервера JRE. Обратите внимание, что для разработки с использованием Java вам нужен JDK. Чтобы узнать разницу между JDK и JRE, см. Здесь
3. Прокрутите вниз до версии, которую хотите загрузить. (В общем, рекомендуется использовать самую последнюю).
4. Выберите переключатель «Принять лицензионное соглашение».
5. Загрузите установщик Windows x86 (32 бит) или Windows 64 (64-разрядный).
6. Запустите программу установки ... обычным способом для вашей версии Windows.

Альтернативный способ установки Java в Windows с помощью командной строки - использовать Chocolately:

1. Установите Chocolately из https://chocolatey.org/
2. Откройте экземпляр cmd, например, нажмите Win + R, а затем введите «cmd» в окне «Run», а затем введите.
3. В вашем экземпляре cmd выполните следующую команду для загрузки и установки Java 8 JDK:

```
C:\> choco install jdk8
```

Начало и работа с портативными версиями

Существуют случаи, когда вы захотите установить JDK / JRE в систему с ограниченными правами, например VM, или вы можете установить и использовать несколько версий или архитектур (x64 / x86)

JDK / JRE. Шаги остаются такими же до момента загрузки установщика (.EXE). После этого следующие шаги (шаги применимы для JDK / JRE 7 и выше, для более старых версий они немного отличаются в именах папок и файлов):

1. Переместите файл в соответствующее место, где вы хотите, чтобы ваши двоичные файлы Java постоянно проживали.
2. Установите 7-Zip или его портативную версию, если у вас есть ограниченные привилегии.
3. С помощью 7-Zip извлеките файлы из установщика Java EXE в указанное место.
4. Откройте командную строку там, удерживая Shift и Right-Click ing в папке в проводнике или перейдите в это место из любого места.
5. Перейдите к вновь созданной папке. Скажем, имя папки - jdk-7u25-windows-x64 . Итак, введите cd jdk-7u25-windows-x64 . Затем введите следующие команды:

   ```
   cd .rsrc\JAVA_CAB10

   extrac32 111
   ```

6. Это создаст файл tools.zip в этом месте. Извлеките tools.zip с 7-Zip, чтобы файлы внутри него теперь создавались под tools в том же каталоге.
7. Теперь выполните эти команды в уже открытой командной строке:

   ```
   cd tools

   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Подождите, пока команда завершится. Скопируйте содержимое tools в место, где вы хотите, чтобы ваши двоичные файлы были.

Таким образом, вы можете установить любые версии JDK / JRE, которые необходимо установить одновременно.

Оригинальная публикация: http://stackoverflow.com/a/6571736/1448252

Установка Java JDK на macOS

Oracle Java 7 и Java 8

Java 7 и Java 8 для macOS доступны из Oracle. Эта страница Oracle отвечает на многие вопросы о Java для Mac. Обратите внимание, что Java 7 до 7u25 отключен Apple по соображениям безопасности.

В общем, Oracle Java (версия 7 и более поздняя) требует Mac на базе Mac, работающего под управлением macOS 10.7.3 или новее.

Установка Oracle Java

Установки Java 7 и 8 JDK и JRE для macOS можно загрузить с веб-сайта Oracle:

- Java 8 - Загрузки Java SE
- Java 7 - Oracle Java Archive.

После загрузки соответствующего пакета дважды щелкните по пакету и выполните обычный процесс установки. Здесь должен быть установлен JDK:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

где соответствует установленной версии.

Переключение командной строки

Когда Java установлена, установленная версия автоматически устанавливается как значение по умолчанию. Чтобы переключаться между разными, используйте:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

Следующие функции могут быть добавлены в ~/.bash_profile (если вы используете оболочку Bash по умолчанию) для удобства использования:

```
function java_version {
   echo 'java -version';
}

function java_set {
   if [[ $1 == "6" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.6';
       echo "Setting Java to version 6..."
       echo "$JAVA_HOME"
   elif [[ $1 == "7" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.7';
       echo "Setting Java to version 7..."
       echo "$JAVA_HOME"
   elif [[ $1 == "8" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.8';
       echo "Setting Java to version 8..."
       echo "$JAVA_HOME"
   fi
}
```

Apple Java 6 на macOS

В более старых версиях macOS (10.11 El Capitan и более ранних версиях) выпущена версия Java от Java 6. Если он установлен, его можно найти в этом месте:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Обратите внимание, что Java 6 давно закончила свою работу, поэтому рекомендуется перейти на более новую версию. Существует более подробная информация о переустановке Apple Java 6 на веб-сайте Oracle.

Настройка и переключение версий Java в Linux с использованием альтернатив

Использование альтернатив

Во многих дистрибутивах Linux используется команда alternatives для переключения между различными версиями команды. Вы можете использовать это для переключения между различными версиями Java, установленными на машине.

1. В командной оболочке установите $ JDK в путь к недавно установленному JDK; например

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Используйте alternatives --install чтобы добавить команды в Java SDK к альтернативам:

```
$ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2
$ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2
$ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
```

И так далее.

Теперь вы можете переключаться между различными версиями Java-команды следующим образом:

```
$ sudo alternatives --config javac

There is 1 program that provides 'javac'.

  Selection    Command
-----------------------------------------------
*+ 1           /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
   2           /Data/jdk1.8.0_67/bin/javac

Enter to keep the current selection[+], or type selection number: 2
$
```

Для получения дополнительной информации об использовании alternatives обратитесь к руководству пользователя (8) .

Установка на основе Arch

archlinux-java основе Arch Linux поставляются с командой archlinux-java для переключения версий Java.

Список установленных сред

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Переключение текущей среды

```
# archlinux-java set <JAVA_ENV_NAME>
```

Например:

```
# archlinux-java set java-8-openjdk/jre
```

Более подробную информацию можно найти на Arch Linux Wiki

Проверка и настройка после установки на Linux

После установки Java SDK рекомендуется проверить, что он готов к использованию. Вы можете сделать это, выполнив эти две команды, используя обычную учетную запись пользователя:

```
$ java -version
$ javac -version
```

Эти команды распечатывают информацию о версии для JRE и JDK (соответственно), которые находятся на пути поиска команд вашей оболочки. Найдите строку версии JDK / JRE.

- Если какая-либо из приведенных выше команд не работает, говоря «команда не найдена», JRE

или JDK вообще не находятся в пути поиска; перейдите к **настройке PATH непосредственно** ниже.
- Если какая-либо из приведенных выше команд отображает другую строку версии, которую вы ожидаете, то ваш путь поиска или система «альтернативы» должны быть настроены; перейти к **проверке альтернатив**
- Если отображаются правильные строки строк, вы почти закончили; перейдите к **разделу Проверка JAVA_HOME**

Конфигурирование PATH напрямую

Если на пути поиска нет java или javac , тогда простое решение состоит в том, чтобы добавить его к вашему пути поиска.

Сначала найдите, где вы установили Java; см. **Где была установлена Java?** ниже, если у вас есть сомнения.

Далее, если предположить, что bash - это ваша командная оболочка, используйте текстовый редактор, чтобы добавить следующие строки в конец ~/.bash_profile или ~/.bashrc (если вы используете Bash в качестве оболочки).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... заменяя <installation directory> именем пути для вашего каталога установки Java. Обратите внимание, что приведенное выше предполагает, что каталог установки содержит каталог bin , а каталог bin содержит команды java и javac которые вы пытаетесь использовать.

Затем отправьте исходный файл, который вы только что отредактировали, чтобы обновить переменные среды для текущей оболочки.

```
$ source ~/.bash_profile
```

Затем повторите проверки версии java и javac . Если все еще есть проблемы, используйте which java и which javac для проверки правильности обновления переменных среды.

Наконец, выйдите из системы и войдите в систему, чтобы обновленные переменные окружения ptopagate со всеми вашими оболочками. Теперь вы должны это сделать.

Проверка альтернатив

Если java -version или javac -version работали, но дали неожиданный номер версии, вам нужно проверить, откуда идут команды. Используйте, which и ls -l чтобы узнать это следующим образом:

```
$ ls -l `which java`
```

Если вывод выглядит следующим образом:

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

то используется переключение версий alternatives . Вам нужно решить, продолжать ли его использовать или просто переопределить, установив PATH напрямую.

- Настройка и переключение версий Java в Linux с использованием альтернатив
- См. «Настройка PATH напрямую» выше.

Где была установлена Java?

Java может быть установлен в различных местах, в зависимости от метода установки.

- RPM для Oracle помещают установку Java в «/ usr / java».
- В Fedora по умолчанию используется «/ usr / lib / jvm».
- Если Java был установлен вручную из ZIP или JAR-файлов, установка может быть в любом месте.

Если вам трудно найти каталог установки, мы предлагаем вам использовать find (или slocate) для поиска команды. Например:

```
$ find / -name java -type f 2> /dev/null
```

Это дает вам имена путей для всех файлов, называемых java в вашей системе. (Перенаправление стандартной ошибки на «/ dev / null» подавляет сообщения о файлах и каталогах, к которым у вас нет доступа.)

Установка oracle java на Linux с последним файлом tar

Следуйте приведенным ниже инструкциям, чтобы установить Oracle JDK из последнего файла tar:

1. Загрузите последний tar-файл отсюда - актуальным является Java SE Development Kit 8u112.
2. Вам нужны sudo privilages:

   ```
   sudo su
   ```

3. Создайте каталог для установки jdk:

   ```
   mkdir /opt/jdk
   ```

4. Извлеките загруженный tar в него:

   ```
   tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
   ```

5. Проверьте, извлечены ли файлы:

   ```
   ls /opt/jdk
   ```

6. Установка Oracle JDK в качестве JVM по умолчанию:

   ```
   update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
   ```

   а также

   ```
   update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
   ```

7. Проверьте версию Java:

   ```
   java -version
   ```

Ожидаемый результат:

```
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

Прочитайте Установка Java (стандартная версия) онлайн:
https://riptutorial.com/ru/java/topic/4754/установка-java--стандартная-версия-

глава 178: Утверждая

Синтаксис

- assert *expression1* ;
- assert *expression1* : *expression2* ;

параметры

параметр	подробности
выражение1	Оператор утверждения вызывает `AssertionError` если это выражение принимает значение `false` .
выражения2	Необязательный. Когда используется, `AssertionError` сбрасываемый оператором assert, имеет это сообщение.

замечания

По умолчанию утверждения отключены во время выполнения.

Чтобы включить утверждения, вы должны запустить java с флагом -ea .

```
java -ea com.example.AssertionExample
```

Утверждения - это утверждения, которые выдают ошибку, если их выражение оценивается как false . Утверждения должны использоваться только для *проверки* кода; они никогда не должны использоваться в производстве.

Examples

Проверка арифметики с утверждением

```
a = 1 - Math.abs(1 - a % 2);

// This will throw an error if my arithmetic above is wrong.
assert a >= 0 && a <= 1 : "Calculated value of " + a + " is outside of expected bounds";

return a;
```

Прочитайте Утверждая онлайн: https://riptutorial.com/ru/java/topic/407/утверждая

глава 179: Файловый ввод-вывод

Вступление

Java I / O (вход и выход) используется для обработки ввода и вывода вывода. Java использует концепцию потока для быстрой работы ввода-вывода. Пакет java.io содержит все классы, необходимые для операций ввода и вывода. Обработка файлов также выполняется в Java с помощью API ввода-вывода Java.

Examples

Чтение всех байтов в байт []

Java 7 представила очень полезный класс файлов

Java SE 7

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("path/to/file");

try {
    byte[] data = Files.readAllBytes(path);
} catch(IOException e) {
    e.printStackTrace();
}
```

Чтение изображения из файла

```
import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Запись байта [] в файл

Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

try(FileOutputStream stream = new FileOutputStream("Hello world.txt")) {
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
}
```

Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

FileOutputStream stream = null;
try {
    stream = new FileOutputStream("Hello world.txt");
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}
```

Большинство API-интерфейсов java.io поддерживают как String и File качестве аргументов, поэтому вы также можете использовать

```
File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);
```

Stream vs Writer / Reader API

Потоки обеспечивают самый прямой доступ к двоичному содержимому, поэтому любые реализации InputStream / OutputStream всегда работают с int s и byte s.

```
// Read a single byte from the stream
int b = inputStream.read();
if (b >= 0) { // A negative value represents the end of the stream, normal values are in the
range 0 - 255
    // Write the byte to another stream
    outputStream.write(b);
}

// Read a chunk
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // A negative value represents end of stream
    // Write the chunk to another stream
    outputStream.write(data, 0, nBytesRead);
}
```

Есть некоторые исключения, возможно, в первую очередь PrintStream который добавляет «способность печатать представления различных значений данных удобно». Это позволяет использовать System.out как двоичный InputStream и как текстовый вывод с использованием таких методов, как System.out.println() .

Кроме того, некоторые потоковые реализации работают как интерфейс для содержимого более высокого уровня, таких как объекты Java (см. Сериализация) или собственные типы, например DataOutputStream / DataInputStream .

С классами Writer и Reader Java также предоставляет API для явных потоков символов. Хотя большинство приложений будут основывать эти реализации на потоках, API потока символов не предоставляет никаких методов для двоичного содержимого.

```
// This example uses the platform's default charset, see below
// for a better implementation.
```

```
Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");

Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();
```

Всякий раз, когда необходимо кодировать символы в двоичные данные (например, при использовании классов InputStreamWriter / OutputStreamWriter), вы должны указать кодировку, если вы не хотите зависеть от кодировки платформы по умолчанию. Если есть сомнения, используйте кодировку, совместимую с Unicode, например UTF-8, которая поддерживается на всех платформах Java. Поэтому вам следует избегать таких классов, как FileWriter и FileReader поскольку они всегда используют кодировку платформы по умолчанию. Лучший способ доступа к файлам с использованием потоков символов - это:

```
Charset myCharset = StandardCharsets.UTF_8;

Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();

Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

Одним из наиболее часто используемых Reader является BufferedReader который предоставляет метод для считывания целых строк текста из другого считывателя и, по-видимому, является самым простым способом прочтения потока символов по строкам:

```
// Read from baseReader, one line at a time
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
  // Remember: System.out is a stream, not a writer!
  System.out.println(line);
}
```

Чтение всего файла сразу

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

\ Z - символ EOF (конец файла). При установке в качестве разделителя сканер будет считывать заполнение до тех пор, пока не будет достигнут флаг EOF.

Чтение файла со сканером

Чтение файла по строкам

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNextLine())
            {
                String line = scanner.nextLine();
```

```
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

слово за слово

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNext())
            {
                String line = scanner.next();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

и вы также можете изменить делимметр, используя метод scanner.useDelimeter ()

Итерация по каталогу и фильтрация с помощью расширения файла

```
    public void iterateAndFilter() throws IOException {
        Path dir = Paths.get("C:/foo/bar");
        PathMatcher imageFileMatcher =
            FileSystems.getDefault().getPathMatcher(
                "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

        try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
                entry -> imageFileMatcher.matches(entry.getFileName()))) {

            for (Path path : stream) {
                System.out.println(path.getFileName());
            }
        }
    }
```

Миграция из java.io.File в Java 7 NIO (java.nio.file.Path)

В этих примерах предполагается, что вы уже знаете, что такое NIO для Java 7 в целом, и вы привыкли писать код с помощью java.io.File . Используйте эти примеры в качестве средства для быстрого поиска дополнительной документации по миграции для NIO.

Существует гораздо больше для NIO Java 7, таких как файлы с отображением памяти или открытие файла ZIP или JAR с помощью FileSystem . Эти примеры будут охватывать только ограниченное число основных случаев использования.

В качестве основного правила, если вы привыкли выполнять операцию чтения / записи файловой системы с помощью метода экземпляра java.io.File , вы найдете его как статический метод в java.nio.file.Files .

Укажите путь

```
// -> IO
File file = new File("io.txt");

// -> NIO
Path path = Paths.get("nio.txt");
```

Пути по отношению к другому пути

```
// Forward slashes can be used in place of backslashes even on a Windows operating system
// -> IO
File folder = new File("C:/");
File fileInFolder = new File(folder, "io.txt");

// -> NIO
Path directory = Paths.get("C:/");
Path pathInDirectory = directory.resolve("nio.txt");
```

Преобразование файла из / в путь для использования с библиотеками

```
// -> IO to NIO
Path pathFromFile = new File("io.txt").toPath();

// -> NIO to IO
File fileFromPath = Paths.get("nio.txt").toFile();
```

Проверьте, существует ли файл и удаляет его, если он

```
// -> IO
if (file.exists()) {
    boolean deleted = file.delete();
    if (!deleted) {
        throw new IOException("Unable to delete file");
    }
}

// -> NIO
Files.deleteIfExists(path);
```

Запись в файл через OutputStream

Существует несколько способов записи и чтения из файла с помощью NIO для различных ограничений производительности, памяти, чтения и использования, таких как FileChannel , Files.write(Path path, byte\[\] bytes, OpenOption... options) . В этом примере OutputStream только OutputStream , но вам настоятельно рекомендуется узнать о файлах с отображением памяти и различных статических методах, доступных в java.nio.file.Files .

```
List<String> lines = Arrays.asList(
        String.valueOf(Calendar.getInstance().getTimeInMillis()),
        "line one",
        "line two");
```

```
// -> IO
if (file.exists()) {
    // Note: Not atomic
    throw new IOException("File already exists");
}
try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

Итерация по каждому файлу в папке

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // Note: Depending on the number of files in the directory folder.listFiles() may take a
long time to return
    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
}

// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs)
throws IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

Рекурсивная итерация папок

```
// -> IO
recurseFolder(folder);

// -> NIO
// Note: Symbolic links are NOT followed unless explicitly passed as an argument to
Files.walkFileTree
Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
```

```
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});


private static void recurseFolder(File folder) {
    for (File selectedFile : folder.listFiles()) {
        System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
        if (selectedFile.isDirectory()) {
            // Note: Symbolic links are followed
            recurseFolder(selectedFile);
        }
    }
}
```

Чтение / запись файла с использованием FileInputStream / FileOutputStream

Напишите в файл test.txt:

```
String filepath ="C:\\test.txt";
FileOutputStream fos = null;
try {
      fos = new FileOutputStream(filepath);
      byte[] buffer = "This will be written in test.txt".getBytes();
      fos.write(buffer, 0, buffer.length);
      fos.close();
} catch (FileNotFoundException e) {
      e.printStackTrace();
} catch (IOException e) {
      e.printStackTrace();
} finally{
      if(fos != null)
        fos.close();
}
```

Читайте из файла test.txt:

```
String filepath ="C:\\test.txt";
FileInputStream fis = null;
try {
   fis = new FileInputStream(filepath);
   int length = (int) new File(filepath).length();
   byte[] buffer = new byte[length];
   fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
     e.printStackTrace();
} catch (IOException e) {
     e.printStackTrace();
} finally{
   if(fis != null)
```

```
     fis.close();
}
```

Обратите внимание, что начиная с Java 1.7 был введен оператор try-with-resources, который значительно упростил реализацию операции чтения / записи:

Напишите в файл test.txt:

```
String filepath ="C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Читайте из файла test.txt:

```
String filepath ="C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Чтение из двоичного файла

Вы можете прочитать двоичный файл, используя этот фрагмент кода во всех последних версиях Java:

Java SE 1.4

```
File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();
```

Если вы используете Java 7 или более позднюю версию, существует более простой способ использования nio API :

Java SE 7

```
Path path = Paths.get("path_to_the_file");
byte [] data = Files.readAllBytes(path);
```

Блокировка

Файл можно заблокировать с FileChannel API FileChannel который можно получить из входных streams и readers

Пример с streams

```
// Открыть файловый поток FileInputStream ios = new FileInputStream (имя файла);
```

```
    // get underlying channel
    FileChannel channel = ios.getChannel();

    /*
     * try to lock the file. true means whether the lock is shared or not i.e. multiple
processes can acquire a
     * shared lock (for reading only) Using false with readable channel only will generate an
exception. You should
     * use a writable channel (taken from FileOutputStream) when using false. tryLock will
always return immediately
     */
    FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

    if (lock == null) {
        System.out.println("Unable to acquire lock");
    } else {
        System.out.println("Lock acquired successfully");
    }

    // you can also use blocking call which will block until a lock is acquired.
    channel.lock();

    // Once you have completed desired operations of file. release the lock
    if (lock != null) {
        lock.release();
    }

    // close the file stream afterwards
    // Example with reader
    RandomAccessFile randomAccessFile = new RandomAccessFile(filename,  "rw");
    FileChannel channel = randomAccessFile.getChannel();
    //repeat the same steps as above but now you can use shared as true or false as the
channel is in read write mode
```

Копирование файла с помощью InputStream и OutputStream

Мы можем напрямую копировать данные из источника в приемник данных с использованием цикла. В этом примере мы считываем данные из InputStream и в то же время записываем в OutputStream. Когда мы закончим чтение и письмо, мы должны закрыть ресурс.

```
public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } finally {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}
```

Чтение файла с использованием канала и буфера

Channel использует Buffer для чтения / записи данных. Буфер представляет собой контейнер с фиксированным размером, в котором мы можем сразу написать блок данных. Channel - это гораздо

быстрее, чем потоковый ввод-вывод.

Чтобы читать данные из файла с помощью Channel нам необходимо выполнить следующие шаги:

1. Нам нужен экземпляр FileInputStream . FileInputStream имеет метод с именем getChannel() который возвращает канал.
2. Вызовите метод getChannel() FileInputStream и приобретите канал.
3. Создайте ByteBuffer. ByteBuffer - контейнер с фиксированным размером байтов.
4. Канал имеет метод чтения, и мы должны предоставить ByteBuffer в качестве аргумента для этого метода чтения. ByteBuffer имеет два режима: настроение только для чтения и настроение только для записи. Мы можем изменить режим с помощью вызова метода flip() . Буфер имеет положение, лимит и емкость. Как только буфер создается с фиксированным размером, его предел и емкость совпадают с размером, и позиция начинается с нуля. Пока буфер записывается с данными, его положение постепенно увеличивается. Изменение режима означает изменение положения. Чтобы читать данные с начала буфера, мы должны установить позицию в ноль. flip () изменить позицию
5. Когда мы вызываем метод чтения Channel , он заполняет буфер, используя данные.
6. Если нам нужно прочитать данные из ByteBuffer , нам нужно перевернуть буфер, чтобы изменить его режим на запись только в режим только для чтения, а затем продолжить чтение данных из буфера.
7. Когда данных больше нет, метод read() канала возвращает 0 или -1.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

public static void main(String[] args) {

   File inputFile = new File("hello.txt");

   if (!inputFile.exists()) {
    System.out.println("The input file doesn't exit.");
    return;
   }

  try {
   FileInputStream fis = new FileInputStream(inputFile);
   FileChannel fileChannel = fis.getChannel();
   ByteBuffer buffer = ByteBuffer.allocate(1024);

   while (fileChannel.read(buffer) > 0) {
    buffer.flip();
    while (buffer.hasRemaining()) {
     byte b = buffer.get();
     System.out.print((char) b);
    }
    buffer.clear();
   }

   fileChannel.close();
  } catch (IOException e) {
   e.printStackTrace();
  }
 }
}
```

Копирование файла с использованием канала

Мы можем использовать Channel для ускорения копирования содержимого файла. Для этого мы можем использовать метод transferTo() для FileChannel .

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

public class FileCopier {

    public static void main(String[] args) {
        File sourceFile = new File("hello.txt");
        File sinkFile = new File("hello2.txt");
        copy(sourceFile, sinkFile);
    }

    public static void copy(File sourceFile, File destFile) {
        if (!sourceFile.exists() || !destFile.exists()) {
            System.out.println("Source or destination file doesn't exist");
            return;
        }


        try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();
             FileChannel sinkChanel = new FileOutputStream(destFile).getChannel()) {

            srcChannel.transferTo(0, srcChannel.size(), sinkChanel);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Чтение файла с использованием BufferedInputStream

Чтение файла с использованием BufferedInputStream обычно происходит быстрее, чем FileInputStream потому что он поддерживает внутренний буфер для хранения байтов, считанных из основного входного потока.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;


public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
```

```
}
```

Запись файла с использованием Channel и Buffer

Для записи данных в файл с использованием Channel нам необходимо выполнить следующие действия:

1. Во-первых, нам нужно получить объект FileOutputStream
2. Приобретите FileChannel вызывающий метод getChannel() из FileOutputStream
3. Создайте ByteBuffer а затем заполните его данными
4. Затем мы должны вызвать метод flip() ByteBuffer и передать его как аргумент метода write() для FileChannel
5. Когда мы закончим писать, мы должны закрыть ресурс

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

 public static void main(String[] args) {

  File outputFile = new File("hello.txt");
  String text = "I love Bangladesh.";

  try {
   FileOutputStream fos = new FileOutputStream(outputFile);
   FileChannel fileChannel = fos.getChannel();
   byte[] bytes = text.getBytes();
   ByteBuffer buffer = ByteBuffer.wrap(bytes);
   fileChannel.write(buffer);
   fileChannel.close();
  } catch (java.io.IOException e) {
   e.printStackTrace();
  }
 }
}
```

Написание файла с помощью PrintStream

Мы можем использовать класс PrintStream для записи файла. Он имеет несколько методов, которые позволяют печатать любые значения типа данных. Метод println() добавляет новую строку. Когда мы закончим печать, мы должны очистить PrintStream .

```
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;

public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow documentation seems fun.");
            ps.println();
            ps.println("I love Java!");
            ps.printf("Today is: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
```

```
        }

    }
}
```

Итерация по каталогу, в котором находятся подкаталоги

```
  public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
      if (Files.isDirectory(path)) {
        System.out.println(path.getFileName());
      }
    }
  }
```

Добавление каталогов

Чтобы создать новый каталог из экземпляра File вам нужно использовать один из двух методов: mkdirs() или mkdir() .

- mkdir() - создает каталог с именем этого абстрактного пути. (источник)
- mkdirs() - создает каталог с именем этого абстрактного пути, включая любые необходимые, но несуществующие родительские каталоги. Обратите внимание: если эта операция не удалась, возможно, ей удалось создать некоторые из необходимых родительских каталогов. (источник)

Примечание: createNewFile() не создаст новый каталог только для файла.

```
File singleDir = new File("C:/Users/SomeUser/Desktop/A New Folder/");

File multiDir = new File("C:/Users/SomeUser/Desktop/A New Folder 2/Another Folder/");

// assume that neither "A New Folder" or "A New Folder 2" exist

singleDir.createNewFile(); // will make a new file called "A New Folder.file"
singleDir.mkdir(); // will make the directory
singleDir.mkdirs(); // will make the directory

multiDir.createNewFile(); // will throw a IOException
multiDir.mkdir(); // will not work
multiDir.mkdirs(); // will make the directory
```

Блокирование или перенаправление стандартного вывода / ошибки

Иногда плохо разработанная сторонняя библиотека будет писать нежелательную диагностику в потоки System.out или System.err . Рекомендуемые решения для этого - либо найти лучшую библиотеку, либо (в случае с открытым исходным кодом) исправить эту проблему и внести исправления разработчикам.

Если вышеупомянутые решения не осуществимы, вам следует рассмотреть возможность перенаправления потоков.

Перенаправление в командной строке

В UNIX, Linux или MacOSX система может быть выполнена из оболочки с помощью > перенаправления. Например:

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

Первый перенаправляет стандартный вывод и стандартную ошибку на «/ dev / null», которая отбрасывает все, что написано в эти потоки. Второй из перенаправляет стандартный вывод на « out.log» и стандартную ошибку на «error.log».

(Для получения дополнительной информации о перенаправлении см. Документацию используемой командной оболочки. Аналогичные рекомендации относятся к Windows.)

Кроме того, вы можете реализовать перенаправление в сценарии оболочки или пакетном файле, который запускает приложение Java.

Перенаправление в приложении Java

Также возможно перенастроить потоки *в* Java-приложении, используя System.setOut() и System.setErr() . Например, следующий фрагмент перенаправляет стандартный вывод и стандартную ошибку на 2 файла журнала:

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

Если вы хотите полностью отбросить вывод, вы можете создать выходной поток, который «пишет» на недопустимый дескриптор файла. Это функционально эквивалентно записи в «/ dev / null» в UNIX.

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

Внимание: будьте осторожны, как вы используете setOut и setErr :

1. Перенаправление повлияет на всю JVM.
2. Делая это, вы убираете способность пользователя перенаправлять потоки из командной строки.

Доступ к содержимому ZIP-файла

API-интерфейс FileSystem Java 7 позволяет читать и добавлять записи из или в Zip-файл с использованием API-интерфейса Java NIO так же, как и в любой другой файловой системе.

FileSystem - это ресурс, который должен быть правильно закрыт после использования, поэтому следует использовать блок try-with-resources.

Чтение из существующего файла

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
  Path root = zipFs.getPath("/");
  ... //access the content of the zip file same as ordinary files
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Создание нового файла

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //required for creating a new zip file
env.put("encoding", "UTF-8"); //optional: default is UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
  Path newFile = zipFs.getPath("/newFile.txt");
  //writing to file
  Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
  ex.printStackTrace();
```

```
}
```

Прочитайте Файловый ввод-вывод онлайн: https://riptutorial.com/ru/java/topic/93/файловый-ввод-вывод

глава 180: Файлы с несколькими релизами JAR

Вступление

Одной из функций, представленной в Java 9, является многорежимный Jar (MRJAR), который позволяет связывать код с таргетингом на несколько выпусков Java в одном файле Jar. Функция указана в JEP 238 .

Examples

Пример содержимого файла Jar для нескольких выпусков

Установив Multi-Release: true в файле MANIFEST.MF, Jar-файл становится многорежимным Jar и временем выполнения Java (при условии, что он поддерживает формат MRJAR) выбирает соответствующие версии классов в зависимости от текущей основной версии ,

Структура такой банки состоит в следующем:

```
jar root
  - A.class
  - B.class
  - C.class
  - D.class
  - META-INF
     - versions
        - 9
           - A.class
           - B.class
        - 10
           - A.class
```

- В JDK <9 в среде выполнения Java видны только классы в корневой записи.
- На JDK 9 классы A и B будут загружены из root/META-INF/versions/9 каталога root/META-INF/versions/9 , а C и D будут загружены из базовой записи.
- На JDK 10 класс A будет загружен из каталога root/META-INF/versions/10 .

Создание многодисковой банки с использованием инструмента банки

Команда jar может использоваться для создания многоэкранного Jar, содержащего две версии одного и того же класса, скомпилированные как для Java 8, так и для Java 9, хотя и с предупреждением о том, что классы идентичны:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
```

Параметр --release 9 указывает jar включить все, что следует (demo пакет внутри sampleproject-9) внутри root/META-INF/versions/9 с версией в MRJAR, а именно под root/META-INF/versions/9 . Результатом является следующее содержание:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
           - demo
```

```
              - SampleClass.class
```

Давайте теперь создадим класс Main, который печатает URL-адрес SampleClass и добавляет его для версии Java 9:

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

Если мы скомпилируем этот класс и запустим команду jar, мы получим сообщение об ошибке:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, contains a new public class
not found in base entries
Warning: entry META-INF/versions/9/demo/Main.java, multiple resources with same name
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
invalid multi-release jar file MR.jar deleted
```

Причина в том, что инструмент jar предотвращает добавление публичных классов к версиям, если они не добавлены в базовые записи. Это делается для того, чтобы MRJAR предоставлял один и тот же публичный API для разных версий Java. Обратите внимание, что во время выполнения это правило не требуется. Он может применяться только такими инструментами, как jar . В этом конкретном случае целью Main является запуск образца кода, поэтому мы можем просто добавить копию в базовую запись. Если класс был частью более новой реализации, которая нам нужна только для Java 9, она может быть сделана непубличной.

Чтобы добавить Main в корневую запись, нам сначала нужно скомпилировать ее для таргетинга на версию до 9 Java. Это можно сделать с помощью новой опции --release javac :

```
C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../..
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
```

Запуск основного класса показывает, что SampleClass загружается из каталога версий:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

URL загруженного класса внутри многоэкранного Jar

Учитывая следующий многорежимный Jar:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
```

```
            - demo
               - SampleClass.class
```

Следующий класс печатает URL-адрес SampleClass :

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

Если класс скомпилирован и добавлен в версию для версии Java 9 в MRJAR, ее запуск приведет к:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

Прочитайте Файлы с несколькими релизами JAR онлайн: https://riptutorial.com/ru/java/topic/9866/файлы-с-несколькими-релизами-jar

глава 181: Флаги JVM

замечания

Настоятельно рекомендуется использовать только эти параметры:

- Если у вас есть полное понимание вашей системы.
- Знают, что, если они используются ненадлежащим образом, эти параметры могут негативно повлиять на стабильность или производительность вашей системы.

Информация, собранная из официальной документации Java .

Examples

-XXaggressive

-XXaggressive - это набор конфигураций, которые позволяют JVM работать на высокой скорости и достигать стабильного состояния как можно скорее. Для достижения этой цели JVM использует больше внутренних ресурсов при запуске; однако, когда цель достигнута, она требует менее адаптивной оптимизации. Мы рекомендуем использовать этот параметр для длительных приложений с интенсивной памятью, которые работают в одиночку.

Использование:

```
-XXaggressive:<param>
```

<PARAM>	Описание
opt	Распределяет адаптивную оптимизацию ранее и включает новые оптимизации, которые, как ожидается, будут использоваться по умолчанию в будущих выпусках.
memory	Настраивает систему памяти для интенсивных рабочих нагрузок и задает ожидаемое количество ресурсов памяти, чтобы обеспечить высокую пропускную способность. JRockit JVM также будет использовать большие страницы, если они доступны.

-XXallocClearChunks

Эта опция позволяет вам очистить TLA для ссылок и значений во время распределения TLA и предварительно выбрать следующий фрагмент. Когда объявляется целое число, ссылка или что-то еще, оно имеет значение по умолчанию 0 или null (в зависимости от типа). В подходящее время вам нужно будет очистить эти ссылки и значения, чтобы освободить память на куче, чтобы Java мог использовать или повторно использовать ее. Вы можете сделать это, когда объект выделен, или, используя эту опцию, при запросе нового TLA.

Использование:

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

Вышеупомянутый вариант является логическим и обычно рекомендуется для систем IA64; в конечном счете, его использование зависит от приложения. Если вы хотите установить размер очищенных кусков, объедините эту опцию с -XXallocClearChunkSize . Если вы используете этот флаг, но не указываете логическое значение, значение по умолчанию равно true .

-XXallocClearChunkSize

При использовании с -XXallocClearChunkSize эта опция устанавливает размер -XXallocClearChunkSize фрагментов. Если этот флаг используется, но значение не указано, значение по умолчанию составляет 512 байт.

Использование:

```
-XXallocClearChunks -XXallocClearChunkSize=<size>[k|K][m|M][g|G]
```

-XXcallProfiling

Эта опция позволяет использовать профилирование вызовов для оптимизации кода. Profiling записывает полезную статистику времени выполнения, специфичную для приложения, и может во многих случаях увеличивать производительность, поскольку JVM может затем воздействовать на эти статистические данные.

> Примечание. Эта опция поддерживается JRockit JVM R27.3.0 и более поздней версией. Он может стать дефолтом в будущих версиях.

Использование:

```
java -XXcallProfiling myApp
```

По умолчанию этот параметр отключен. Вы должны включить его для его использования.

-XXdisableFatSpin

Этот параметр отключает код блокировки жира в Java, позволяя потокам, которые блокируют попытку получить блокировку жира, отправляется спать напрямую.

Объекты в Java становятся блокировкой, как только любой поток входит в синхронизированный блок на этом объекте. Все замки удерживаются (т. Е. Остаются заблокированными) до тех пор, пока не отпустят стопорная нить. Если блокировка не будет выпущена очень быстро, ее можно раздуть до «жирной блокировки». «Спиннинг» происходит, когда поток, который хочет определенный замок, непрерывно проверяет эту блокировку, чтобы убедиться, что она еще снята, вращается в как он делает проверку. Вращение против жирового замка обычно полезно, хотя в некоторых случаях оно может быть дорогостоящим и может повлиять на производительность. -XXdisableFatSpin позволяет отключить вращение против блокировки жира и устранить потенциальный удар производительности.

Использование:

```
-XXdisableFatSpin
```

-XXdisableGCHeuristics

Этот параметр отключает изменения стратегии сборщика мусора. Этот вариант не влияет на эвристику уплотнения и эвристику размера ясеня. По умолчанию эвристика сбора мусора включена.

Использование:

```
-XXdisableFatSpin
```

-XXdumpSize

Этот параметр вызывает создание файла дампа и позволяет указать относительный размер этого файла (то есть маленький, средний или большой).

Использование:

```
-XXdumpsize:<size>
```

<Размер>	Описание
none	Не создает файл дампа.
small	В Windows создается небольшой файл дампа (в Linux создается полный дамп ядра). Небольшой дамп включает только стеки потоков, включая их следы и совсем немного. Это было по умолчанию в JRockit JVM 8.1 с пакетами 1 и 2, а также 7.0 с пакетом обновления 3 и выше).
normal	Вызывает создание нормального дампа на всех платформах. Этот файл дампа содержит всю память, кроме кучи java. Это значение по умолчанию для JRockit JVM 1.4.2 и более поздних версий.
large	Включает все, что есть в памяти, включая кучу Java. Этот параметр делает -XXdumpSize эквивалентным -XXdumpFullState .

-XXexitOnOutOfMemory

Эта опция делает JRockit JVM выходом при первом возникновении ошибки из памяти. Его можно использовать, если вы предпочитаете перезапускать экземпляр JVKit JVM, а не обрабатывать ошибки в памяти. Введите эту команду при запуске, чтобы заставить JRockit JVM выйти из первого вхождения ошибки из памяти.

Использование:

```
-XXexitOnOutOfMemory
```

Прочитайте Флаги JVM онлайн: https://riptutorial.com/ru/java/topic/2500/флаги-jvm

глава 182: Функциональные интерфейсы

Вступление

В Java 8+ *функциональный интерфейс* представляет собой интерфейс, который имеет только один абстрактный метод (помимо методов Object). См. JLS §9.8. Функциональные интерфейсы .

Examples

Список стандартных функциональных интерфейсов Java Runtime Library по сигнатуре

Типы параметров	Тип возврата	Интерфейс
()	недействительным	Runnable
()	Т	поставщик
()	логический	BooleanSupplier
()	ИНТ	IntSupplier
()	долго	LongSupplier
()	двойной	DoubleSupplier
(Т)	недействительным	Потребитель <T>
(Т)	Т	UnaryOperator <T>
(Т)	р	Функция <T, R>
(Т)	логический	Предиката <T>
(Т)	ИНТ	ToIntFunction <T>
(Т)	долго	ToLongFunction <T>
(Т)	двойной	ToDoubleFunction <T>
(Т, Т)	Т	BinaryOperator <T>
(Т, U)	недействительным	BiConsumer <T, U>
(Т, U)	р	BiFunction <T, U, R>
(Т, U)	логический	BiPredicate <T, U>
(Т, U)	ИНТ	ToIntBiFunction <T, U>
(Т, U)	долго	ToLongBiFunction <T, U>

Типы параметров	Тип возврата	Интерфейс
(T, U)	двойной	ToDoubleBiFunction <T, U>
(T, int)	недействительным	ObjIntConsumer <T>
(T, длинный)	недействительным	ObjLongConsumer <T>
(T, двойной)	недействительным	ObjDoubleConsumer <T>
(Целое)	недействительным	IntConsumer
(Целое)	р	IntFunction <R>
(Целое)	логический	IntPredicate
(Целое)	ИНТ	IntUnaryOperator
(Целое)	долго	IntToLongFunction
(Целое)	двойной	IntToDoubleFunction
(int, int)	ИНТ	IntBinaryOperator
(долго)	недействительным	LongConsumer
(долго)	р	LongFunction <R>
(долго)	логический	LongPredicate
(долго)	ИНТ	LongToIntFunction
(долго)	долго	LongUnaryOperator
(долго)	двойной	LongToDoubleFunction
(долго долго)	долго	LongBinaryOperator
(Двойной)	недействительным	DoubleConsumer
(Двойной)	р	DoubleFunction <R>
(Двойной)	логический	DoublePredicate
(Двойной)	ИНТ	DoubleToIntFunction
(Двойной)	долго	DoubleToLongFunction
(Двойной)	двойной	DoubleUnaryOperator

Типы параметров	Тип возврата	Интерфейс
(двойной, двойной)	двойной	DoubleBinaryOperator

Прочитайте Функциональные интерфейсы онлайн: https://riptutorial.com/ru/java/topic/10001/функциональные-интерфейсы

глава 183: Хеш-таблица

Вступление

Hashtable - это класс в **наборах** Java, который реализует интерфейс Map и расширяет класс словаря

Содержит только уникальные элементы и их синхронизированные

Examples

Хеш-таблица

```
import java.util.*;
public class HashtableDemo {
   public static void main(String args[]) {
   // create and populate hash table
   Hashtable<Integer, String> map = new Hashtable<Integer, String>();
   map.put(101,"C Language");
   map.put(102, "Domain");
   map.put(104, "Databases");
   System.out.println("Values before remove: "+ map);
   // Remove value for key 102
   map.remove(102);
   System.out.println("Values after remove: "+ map);
   }
}
```

Прочитайте Хеш-таблица онлайн: https://riptutorial.com/ru/java/topic/10709/хеш-таблица

глава 184: Читатели и писатели

Вступление

Читатели и писатели и их соответствующие подклассы обеспечивают простой ввод-вывод для текстовых / символьных данных.

Examples

BufferedReader

Вступление

Класс BufferedReader является оберткой для других классов Reader которая выполняет две основные цели:

1. BufferedReader обеспечивает буферизацию для обернутого Reader . Это позволяет приложению читать символы по одному без лишних накладных расходов ввода-вывода.
2. BufferedReader обеспечивает функциональность для чтения текста строки за раз.

Основы использования BufferedReader

Обычный шаблон для использования BufferedReader выглядит следующим образом. Во-первых, вы получаете Reader , с которого хотите читать символы. Затем вы создаете экземпляр BufferedReader который обертывает Reader . Затем вы читаете символьные данные. Наконец, вы закрываете BufferedReader который закрывает завернутый «Reader». Например:

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
     BufferedReader br = new BufferedReader(fr)) {
    // Count the number of 'a' characters.
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("There are " + aCount + " 'a' characters in " + someFile);
}
```

Вы можете применить этот шаблон к любому Reader

Заметки:

1. Мы использовали Java 7 (или более поздние) *try-with-resources,* чтобы гарантировать, что основной читатель всегда закрыт. Это позволяет избежать потенциальной утечки ресурсов. В более ранних версиях Java вы должны явно закрыть BufferedReader в блоке finally .
2. Код внутри блока try практически идентичен тому, что мы будем использовать, если мы будем читать непосредственно из FileReader . Фактически, BufferedReader функционирует точно так же, как и Reader который он будет обертывать. Разница в том, что *эта* версия намного эффективнее.

Размер буфера BufferedReader

Метод BufferedReader.readLine ()

Пример: чтение всех строк файла в список

Это делается путем ввода каждой строки в файл и добавления ее в List<String> . Затем список возвращается:

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line = null;
        while ((line = reader.readLine) != null) {
            lines.add(line);
        }
    }
    return lines;
}
```

Java 8 предоставляет более сжатый способ сделать это с помощью метода lines() :

```
public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.empty();
}
```

Пример StringWriter

Класс Java StringWriter представляет собой поток символов, который собирает выходные данные из строкового буфера, который может использоваться для построения строки.

Класс StringWriter расширяет класс Writer.

В классе StringWriter системные ресурсы, такие как сетевые сокеты и файлы, не используются, поэтому закрытие StringWriter не требуется.

```
import java.io.*;
public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
                    writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}
```

Вышеприведенный пример помогает нам узнать простой пример StringWriter с использованием BufferedReader для чтения данных файла из потока.

Прочитайте Читатели и писатели онлайн: https://riptutorial.com/ru/java/topic/10618/читатели-и-писатели

глава 185: Шифрование RSA

Examples

Пример использования гибридной криптосистемы, состоящей из OAEP и GCM

Следующий пример шифрует данные с использованием гибридной криптосистемы, состоящей из AES GCM и OAEP, используя их размеры по умолчанию и размер ключа AES 128 бит.

OAEP менее уязвим для атаки оракула, чем PKCS # 1 v1.5. GCM также защищен от атак наложения.

Дешифрование может быть выполнено путем первого получения длины инкапсулированного ключа, а затем путем извлечения инкапсулированного ключа. Затем инкапсулированный ключ может быть дешифрован с использованием закрытого ключа RSA, который формирует пару ключей с открытым ключом. После этого зашифрованный шифрованный файл AES / GCM может быть расшифрован в исходный текст.

Протокол состоит из:

1. поле длины для завернутого ключа (RSAPrivateKey пропускает метод getKeySize());
2. завернутый / инкапсулированный ключ того же размера, что и размер ключа RSA в байтах;
3. зашифрованный текст GCM и 128-битный тег аутентификации (автоматически добавленный Java).

Заметки:

- Чтобы правильно использовать этот код, вы должны предоставить ключ RSA не менее 2048 бит, лучше - лучше (но медленнее, особенно при расшифровке);
- Чтобы использовать AES-256, вы должны сначала установить неограниченные файлы политики криптографии ;
- Вместо создания собственного протокола вы можете использовать формат контейнера, такой как Синтаксис криптографического сообщения (CMS / PKCS # 7) или PGP.

Итак, вот пример:

```
/**
 * Encrypts the data using a hybrid crypto-system which uses GCM to encrypt the data and OAEP
to encrypt the AES key.
 * The key size of the AES encryption will be 128 bit.
 * All the default parameter choices are used for OAEP and GCM.
 *
 * @param publicKey the RSA public key used to wrap the AES key
 * @param plaintext the plaintext to be encrypted, not altered
 * @return the ciphertext
 * @throws InvalidKeyException if the key is not an RSA public key
 * @throws NullPointerException if the plaintext is null
 */
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)", e);
    } catch (NoSuchPaddingException e) {
```

```
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)", e);
    }
    oaep.init(Cipher.WRAP_MODE, publicKey);

    // --- wrap the plaintext in a buffer

    // will throw NullPointerException if plaintext is null
    ByteBuffer plaintextBuffer = ByteBuffer.wrap(plaintext);

    // --- generate a new AES secret key ---

    KeyGenerator aesKeyGenerator;
    try {
        aesKeyGenerator = KeyGenerator.getInstance("AES");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES key generator (mandatory algorithm for
runtimes)", e);
    }
    // for AES-192 and 256 make sure you've got the rights (install the
    // Unlimited Crypto Policy files)
    aesKeyGenerator.init(128);
    SecretKey aesKey = aesKeyGenerator.generateKey();

    // --- wrap the new AES secret key ---

    byte[] wrappedKey;
    try {
        wrappedKey = oaep.wrap(aesKey);
    } catch (IllegalBlockSizeException e) {
        throw new RuntimeException(
                "AES key should always fit OAEP with normal sized RSA key", e);
    }

    // --- setup the AES GCM cipher mode ---

    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly generated
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode encryption
        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
        aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)", e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for GCM (present in the standard Java runtime
sinze XX)", e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
                "IvParameterSpec not accepted by this implementation of GCM", e);
    }

    // --- create a buffer of the right size for our own protocol ---
```

```
    ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
            Short.BYTES
            + oaep.getOutputSize(128 / Byte.SIZE)
            + aesGCM.getOutputSize(plaintext.length));

    // - element 1: make sure that we know the size of the wrapped key
    ciphertextBuffer.putShort((short) wrappedKey.length);

    // - element 2: put in the wrapped key
    ciphertextBuffer.put(wrappedKey);

    // - element 3: GCM encrypt into buffer
    try {
        aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
        throw new RuntimeException("Cryptographic exception, AES/GCM encryption should not
fail here", e);
    }

    return ciphertextBuffer.array();
}
```

Конечно, шифрование не очень полезно без расшифровки. Обратите внимание, что при дешифровании это приведет к минимальной информации.

```
/**
 * Decrypts the data using a hybrid crypto-system which uses GCM to encrypt
 * the data and OAEP to encrypt the AES key. All the default parameter
 * choices are used for OAEP and GCM.
 *
 * @param privateKey
 *            the RSA private key used to unwrap the AES key
 * @param ciphertext
 *            the ciphertext to be encrypted, not altered
 * @return the plaintext
 * @throws InvalidKeyException
 *             if the key is not an RSA private key
 * @throws NullPointerException
 *             if the ciphertext is null
 * @throws IllegalArgumentException
 *             with the message "Invalid ciphertext" if the ciphertext is invalid (minimize
information leakage)
 */
public static byte[] decryptData(PrivateKey privateKey, byte[] ciphertext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
```

```
runtime sinze XX)",
                e);
    }
    oaep.init(Cipher.UNWRAP_MODE, privateKey);

    // --- wrap the ciphertext in a buffer

    // will throw NullPointerException if ciphertext is null
    ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

    // sanity check #1
    if (ciphertextBuffer.remaining() < 2) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }
    // - element 1: the length of the encapsulated key
    int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
    // sanity check #2
    if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }

    // --- unwrap the AES secret key ---

    byte[] wrappedKey = new byte[wrappedKeySize];
    // - element 2: the encapsulated key
    ciphertextBuffer.get(wrappedKey);
    SecretKey aesKey;
    try {
        aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
                Cipher.SECRET_KEY);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (InvalidKeyException e) {
        throw new RuntimeException(
                "Invalid ciphertext");
    }

    // --- setup the AES GCM cipher mode ---

    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly
        // generated
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode
        // encryption
        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128,
                new byte[12]);
        aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for GCM (present in the standard Java runtime
```

```
sinze XX)",
                e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
                "IvParameterSpec not accepted by this implementation of GCM",
                e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer plaintextBuffer = ByteBuffer.allocate(aesGCM
            .getOutputSize(ciphertextBuffer.remaining()));

    // - element 3: GCM ciphertext
    try {
        aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException
            | BadPaddingException e) {
        throw new RuntimeException(
                "Invalid ciphertext");
    }

    return plaintextBuffer.array();
}
```

Прочитайте Шифрование RSA онлайн: https://riptutorial.com/ru/java/topic/1889/шифрование-rsa

кредиты

S. No	Главы	Contributors
1	Начало работы с Java Language	aa_oo, Aaqib Akhtar, abhinav, Abhishek Jain, Abob, acdcjunior, Adeel Ansari, adsalpha, AER, akhilsk, Akshit Soota, Alex A, alphaloop, altomnr, Amani Kilumanga, AndroidMechanic, Ani Menon, ankit dassor, Ankur Anand, antonio, Arkadiy, Ashish Ahuja, Ben Page, Blachshma, bpoiss, Burkhard, Carlton, Charlie H, Coffeehouse Coder, c□ LDS□ EED, Community, Confiqure, CraftedCart, dabansal, Daksh Gupta, Dan Hulme, Dan Morenus, DarkV1, David G., David Grinberg, David Newcomb, DeepCoder, Do Nhu Vy, Draken, Durgpal Singh, Dushko Jovanovski, E_net4, Edvin Tenovimas, Emil Sierżęga, Emre Bolat, enrico.bacis, Eran, explv, fgb, Francesco Menzani, Functino, garg10may, Gautam Jose, GingerHead, Grzegorz Górkiewicz, iliketocode, ıɯɐƃoʇ əızuəʞ, intboolstring, ipsi, J F , James Taylor, Jason, JavaHopper, Javant, javydreamercsw, Jean Vitor, Jean-François Savard, Jeffrey Brett Coleman, Jeffrey Lin, Jens Schauder, John Fergus, John Riddick, John Slegers, Jojodmo, JonasCz, Jonathan, Jonny Henly, Jorn Vernee, kaartic, Lambda Ninja, LostAvatar, madx, Magisch, Makoto, manetsus, Marc, Mark Adelsberger, Maroun Maroun, Matt, Matt, mayojava, Mitch Talmadge , mnoronha, Mrunal Pagnis, Mukund B, Mureinik, NageN, Nathan Arthur, nevster, Nithanim, Nuri Tasdemir, nyarasha, ochi, OldMcDonald, Onur, Ortomala Lokni, OverCoder, P.J.Meisch, Pavneet_Singh, Petter Friberg, philnate, Phrancis, Pops, ppeterka, Přemysl Šťastný, Pritam Banerjee, Radek Postołowicz, Radouane ROUFID, Rafael Mello, Rakitić, Ram, RamenChef, rekire, René Link, Reut Sharabani, Richard Hamilton, Ronnie Wang, ronnyfm, Ross Drew, RotemDev, Ryan Hilbert, SachinSarawgi, Sanandrea, Sandeep Chatterjee, Sayakiss, ShivBuyya, Shoe, Siguza , solidcell, stackptr, Stephen C, Stephen Leppik, sudo, Sumurai8 , Знаđошfa□, tbodt, The Coder, ThePhantomGamer, Thisaru Guruge, Thomas Gerot, ThomasThiebaud, ThunderStruct, tonirush, Tushar Mudgal, Unihedron, user1133275, user124993, uzaif, Vaibhav Jain, Vakerrian, vasili111, Victor Stafusa, Vin, VinayVeluri, Vogel612 , vorburger, Wilson, worker_bee, Yash Jain, Yury Fedorov, Zachary David Saunders, Ze Rubeus
2	2D-графика в Java	17slim, ABDUL KHALIQ
3	Apache Commons Lang	Jonathan Barbero
4	API Reflection	Ali786, ArcticLord, Aurasphere, Blubberguy22, Bohemian, Christophe Weis, Drizzt321, fabian, hd84335, Joeri Hendrickx, Luan Nico, madx, Michael Myers, Onur, Petter Friberg, RamenChef, Ravindra babu, Squidward, Stephen C, Tony BenBrahim, Universal Electricity, ФХосę □ Переúра ツ
5	API стека	manouti
6	AppDynamics и TIBCO BusinessWorks для легкой интеграции	Alexandre Grimaud
7	Autoboxing	17slim, Anony-Mousse, Bob Rivers, Chuck Daniels, cshubhamrao, fabian, hd84335, J Atkin, janos, kaartic, Kirill Sokolov, Luan Nico, Nayuki, piyush_baderia, Ram, RamenChef, Saagar Jha, Stephen C, Unihedron, Vladimir Vagaytsev

8	BigDecimal	alain.janinm, Christian, Dth, Enigo, ggolding, Harish Gyanani, John Nash, Loris Securo, Łukasz Piaszczyk, Manish Kothari, mszymborski, RamenChef, sudo, xwoker
9	BigInteger	Alek Mieczkowski, Alex Shesterov, Amani Kilumanga, Andrii Abramov, azurefrog, Byte1518, dimo414, dorukayhan, Emil Sierżęga, fabian, GPI, Ha., hd84335, janos, Kaushal28, Maarten Bodewes, Makoto, matt freake, Md. Nasir Uddin Bhuiyan, Nufail, Pritam Banerjee, Ruslan Bes, ShivBuyya, Stendika, Vogel612
10	BufferedWriter	Andrii Abramov, fabian, Jorn Vernee, Robin, VatsalSura
11	ByteBuffer	Community, Jon Ericson, Jorn Vernee, Tarık Yılmaz, Tomasz Bawor, victorantunes, Vogel612
12	CompletableFuture	Adowrath, Kishore Tulsiani, WillShackleford
13	Enum, начиная с номера	Sugan
14	FileUpload для AWS	Amit Gujarathi
15	FTP (протокол передачи файлов)	Kelvin Kellner
16	HttpURLConnection	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
17	InputStreams и OutputStreams	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash, Kip, KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612
18	Java Pitfalls - использование исключений	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
19	Java Pitfalls - синтаксис языка	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453
20	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
21	Java-агенты	Display Name, mnoronha
22	Java-версии, версии, выпуски и дистрибутивы	Gal Dreiman, screab, Stephen C
23	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram, Stephan, Thomas Fritsch, vallismortis, Walery Strauch
24	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik
25	JMX	esin88
26	JNDI	EJP, neohope, RamenChef
27	JShell	ostrichofevil, Sudip Bhandari
28	JSON в Java	Asaph, Bogdan Korinnyi, Burkhard, Cache Staheli, hd84335, ipsi, Jared Hooper, Kurzalead, MikaelF, Mrunal Pagnis, Nicholas J

		Panella, Nikita Kurtin, ppeterka, Prem Singh Bist, RamenChef, Ray Kiddy, SirKometa, still_learning, Stoyan Dekov, systemfreund, Tim, Vikas Gupta, vsminkov, Yury Fedorov
29	LinkedHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
30	log4j / log4j2	Daniel Wild, Fildor, HCarrasko, hd84335, Mrunal Pagnis, Rens van der Heijden
31	NIO - Сеть	Matthieu, mnoronha
32	NumberFormat	arpit pandey, John Nash, RamenChef, ΦΧοςę □ Περεύρα ツ
33	ServiceLoader	fabian, Florian Genser, Gerald Mücke
34	SortedMap	Amit Gujarathi
35	Streams	4castle, Abubakkar, acdcjunior, Aimee Borda, Akshit Soota, Amitay Stern, Andrew Tobilko, Andrii Abramov, ArsenArsen, Bart Kummel, berko, Blubberguy22, bpoiss, Brendan B, Burkhard, Cerbrus, Charlie H, Claudio, Community, Conrad.Dean, Constantine, Daniel Käfer, Daniel M., Daniel Stradowski, Dariusz, David G., DonyorM, Dth, Durgpal Singh, Dushko Jovanovski, DVarga, dwursteisen, Eirik Lygre, enrico.bacis, Eran, explv, Fildor, Gal Dreiman, gontard, GreenGiant, Grzegorz Oledzki, Hank D, Hulk, iliketocode, ItachiUchiha, izikovic, J Atkin, Jamie Rees, JavaHopper, Jean-François Savard, John Slegers, Jon Erickson, Jonathan, Jorn Vernee, Jude Niroshan, JudgingNotJudging, Justin, Kapep, Kip, LisaMM, Makoto, Malt, malteo, Marc, MasterBlaster, Matt, Matt, Matt S., Matthieu, Michael Piefel, MikeW, Mitch Talmadge, Mureinik, Muto, Naresh Kumar, Nathaniel Ford, Nuri Tasdemir, OldMcDonald, Oleg L., omiel, Ortomala Lokni, Pawan, Paweł Albecki, Petter Friberg, Philipp Wendler, philnate, Pirate_Jack, ppeterka, Radnyx, Radouane ROUFID, Rajesh Kumar, Rakitić, RamenChef, Ranadip Dutta, ravthiru, reto, Reut Sharabani, RobAu, Robin, Roland Illig, Ronnie Wang, rrampage, RudolphEst, sargue, Sergii Bishyr, sevenforce, Shailesh Kumar Dayananda, shmosel, Shoe, solidcell, Spina, Squidward, SRJ, stackptr, stark, Stefan Dollase, Stephen C, Stephen Leppik, Steve K, Sugan, suj1th, thiagogcm, tpunt, Tunaki, Unihedron, user1133275, user1803551, Valentino, vincentvanjoe, vsnyc, Wilson, Ze Rubeus, zwl
36	StringBuffer	Amit Gujarathi
37	StringBuilder	Andrii Abramov, Cache Staheli, David Soroko, Enigo, fabian, fgb, JudgingNotJudging, KIRAN KUMAR MATAM, Nicktar, P.J.Meisch, Stephen C
38	sun.misc.Unsafe	4444, Daniel Nugent, Grexis, Stephen C, Suminda Sirinath S. Dharmasena
39	ThreadLocal	Dariusz, Liju Thomas, Manish Kothari, Nithanim, taer
40	TreeMap и TreeSet	Malt, Stephen C
41	Varargs (переменный аргумент)	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
42	WeakHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
43	XJC	Danilo Guimaraes, fabian

44	XOM - Объектная модель XML	Arthur, Makoto
45	Альтернативные коллекции	mnoronha, ppeterka, Viacheslav Vedenin
46	Анализ XML с использованием API JAXP	GPI
47	Аннотации	Ad Infinitum, Alon .G., Andrei Maieras, Andrii Abramov, bruno, Conrad.Dean, Dariusz, Demon Coldmist, Drizzt321, Dushko Jovanovski, fabian, faraa, GhostCat, hd84335, Hendrik Ebbers, J Atkin, Jorn Vernee, Kapep, Malt, MasterBlaster, matt freake, Nolequen, Ortomala Lokni, Ram, shmosel, Stephen C, Umberto Raimondi, Vogel612, ФХосę Переúра ツ
48	Апплеты	ArcticLord, Enigo, MadProgrammer, ppeterka
49	Атомные типы	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena, xTrollxDudex
50	аудио	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
51	Безопасность и криптография	John Nash, shibli049
52	Бит-манипуляция	Aimee Borda, Blubberguy22, dosdebug, esin88, Gerald Mücke, Jorn Vernee, Kineolyan, mnoronha, Nayuki, Rednivrug, Ryan Hilbert, Stephen C, thatguy
53	Валюта и деньги	Alexey Lagunov
54	Ведение журнала (java.util.logging)	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
55	Видимость (контроль доступа к членам класса)	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M., intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo , Stephen C, Vogel612
56	Виртуальная машина Java (JVM)	Dushman, RamenChef, Rory McCrossan, Stephen Leppik
57	Виртуальный доступ Java	Ezekiel Baniaga, Stephan, Stephen C
58	Вложенные и внутренние классы	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh , sargue, Slava Babin, Stephen C, tynn
59	Возможности Java SE 7	compuhosny, RamenChef
60	Возможности Java SE 8	compuhosny, RamenChef, sun-solar-arrow
61	Выбор коллекций	John DiFini
62	Выражения	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik

63	Генерация случайных чисел	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q, Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir
64	Геттеры и сеттеры	Fildor, Ironcache, Krö w, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
65	Даты и время (java.time. *)	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon, Squidward, Tarun Maganti, Vogel612, ФХосę Переúра ヅ
66	Двигатель JavaScript Nashorn	ben75, ekaerovets, Francesco Menzani, hd84335, Ilya, InitializeSahib, kasperjj, VatsalSura
67	Дженерики	1d0m3n30, 4444, Aaron Digulla, Abhishek Jain, Alex Meiburg, alex s, Andrei Maieras, Andrii Abramov, Anony-Mousse, Bart Enkelaar, bitek, Blubberguy22, Bob Brinks, Burkhard, Cache Staheli, Cannon , Ce7, Chriss, code11, Codebender, Daniel Figueroa, daphshez, DVarga, Emil Sierżęga, enrico.bacis, Eran, faraa, hd84335, hexafraction, Jan Vladimir Mostert, Jens Schauder, Jorn Vernee, Jude Niroshan, kcoppock, Kevin Montrose, Lahiru Ashan, Lii, manfcas, Mani Muthusamy, Marc, Matt, Mistalis, Mshnik, mvd, Mzzzzzz, NatNgs, nishizawa23, Oleg Sklyar, Onur, Ortomala Lokni, paisanco, Paul Bellora, Paweł Albecki, PcAF, Petter Friberg, phant0m, philnate, Radouane ROUFID, RamenChef, rap-2-h, rd22, Rogério, rolve, RutledgePaulV, S.K. Venkat, Siguza, Stephen C, Stephen Leppik, suj1th, tainy, ThePhantomGamer, Thomas, TNT, ʇolɐəz əɥʇ qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yury Fedorov
68	Документирование кода Java	Blubberguy22, Burkhard, Caleb Brinkman, Carter Brainerd, Community, Do Nhu Vy, Emil Sierżęga, George Bailey, Gerald Mücke , hd84335, ipsi, Kevin Thorne, Martijn Woudstra, Mitch Talmadge, Nagesh Lakinepally, PizzaFrog, Radouane ROUFID, RamenChef, sargue, Stephan, Stephen C, Trevor Sears, Universal Electricity
69	Загрузчики классов	FFY00, Flow, Holger, Makoto, Stephen C
70	Защищенные объекты	Ankit Katiyar
71	Изменение байтового кода	bloo, Display Name, rakwaht, Squidward
72	Инкапсуляция	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
73	Интерфейс Dequeue	Suketu Patel
74	Интерфейс Java Native	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
75	Интерфейс инструмента JVM	desilijic
76	Интерфейсы	100rabh, A Boschman, Abhishek Jain, Adowrath, Alex Shesterov, Andrew Tobilko, Andrii Abramov, Cà phê đen, Chirag Parmar, Conrad.Dean, Daniel Käfer, devguy, DVarga, Hilikus, inovaovao, intboolstring, James Oswald, Jan Vladimir Mostert, JavaHopper, Johannes, Jojodmo, Jonathan, Jorn Vernee, Kai, kstandell, Laurel , Marvin, MikeW, Paul Nelson Baker, Peter Rader, ppovoski, Prateek Agarwal, Radouane ROUFID, RamenChef, Robin, Simulant, someoneigna, Stephen C, Stephen Leppik, Sujith Niraikulathan, Thomas Gerot, user187470, Vasiliy Vlasov, Vince Emigh, xwoker, Zircon

77	Исключения и обработка исключений	Adrian Krebs, agilob, akhilsk, Andrii Abramov, Bhavik Patel, Burkhard, Cache Staheli, Codebender, Dariusz, DarkV1, dimo414, Draken, EAX, Emil Sierżęga, enrico.bacis, fabian, FMC, Gal Dreiman, GreenGiant, Hernanibus, hexafraction, Ilya, intboolstring, Jabir, James Jensen, JavaHopper, Jens Schauder, John Nash, John Slegers, JonasCz, Kai, Kevin Thorne, Malt, Manish Kothari, Md. Nasir Uddin Bhuiyan, michaelbahr, Miljen Mikic, Mitch Talmadge, Mrunal Pagnis, Myridium, mzc, Nikita Kurtin, Oleg Sklyar, P.J.Meisch, Paweł Albecki, Peter Gordon, Petter Friberg, ppeterka, Radek Postołowicz, Radouane ROUFID, Raj, RamenChef, rdonuk, Renukaradhya, RobAu, sandbo00, Saša Šijak, sharif.io, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, sv3k, tatoalo, Thomas Fritsch, Tripta Kiroula, vic-3, Vogel612, Wilson, yiwei
78	Исполнители, Исполнительные службы и пулы потоков	Andrii Abramov, Cache Staheli, Fildor, hd84335, Jens Schauder, JonasCz, noscreenname, Olivier Grégoire, philnate, Ravindra babu , Shettyh, Stephen C, Suminda Sirinath S. Dharmasena, sv3k, tones, user1121883, Vlad-HC, Vogel612
79	Использование ThreadPoolExecutor в приложениях MultiThreaded.	Brendon Dugan
80	Использование других языков сценариев в Java	Nikhil R
81	Использование ключевого слова static	17slim, Amir Rachum, Andrew Brooke, Arthur, ben75, CarManuel, Daniel Nugent, EJP, Hi I'm Frogatto, Mark Yisri, Sadiq Ali, Skepter, Squidward
82	Итератор и Итерабель	Abubakkar, Comic Sans, Dariusz, Hulk, Lukas Knuth, RamenChef, Stephen C, user1121883, WillShackleford
83	Календарь и его подклассы	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
84	Карта Enum	KIRAN KUMAR MATAM
85	Карты	17slim, agilob, alain.janinm, ata, Binary Nerd, Burkhard, coobird, Dmitriy Kotov, Durgpal Singh, Emil Sierżęga, Emily Mabrey, Enigo, fabian, GPI, hd84335, J Atkin, Jabir, Javant, Javier Diaz, Jeffrey Bosboom, johnnyaug, Jonathan, Kakarot, KartikKannapur, Kenster, michaelbahr, Mo.Ashfaq, Nathaniel Ford, phatfingers, Ram, RamenChef, ravthiru, sebkur, Stephen C, Stephen Leppik, Viacheslav Vedenin, VISHWANATH N P, Vogel612
86	Класс - отражение Java	gobes, KIRAN KUMAR MATAM
87	Класс EnumSet	KIRAN KUMAR MATAM
88	Класс java.util.Objects	mnoronha, RamenChef, Stephen C
89	Класс даты	A_Arnold, alain.janinm, arcy, Bob Rivers, Christian Wilkie, explv, Jabir, Jean-Baptiste Yunès, John Smith, Matt Clark, Miles , NamshubWriter, Nicktar, Nishant123, Ph0bi4, ppeterka, Ralf Kleberhoff, Ram, skia.heliou, Squidward, Stephen C, Vinod Kumar Kashyap

90	Класс свойств	17slim, Arthur, J Atkin, Jabir, KIRAN KUMAR MATAM, Marvin, peterh, Stephen C, VGR, vorburger
91	Классы и объекты	Community, Confiqure, Daniel LIn, Dave Ranjan, EJP, eveysky, fabian, Jens Schauder, Kevin Johnson, KIRAN KUMAR MATAM, MasterBlaster, Mureinik, Rakitić, Ram, RamenChef, Ryan Cocuzzo, Salman Kazmi, Tyler Zika
92	Клонирование объектов	Ayush Bansal, Christophe Weis, Jonathan
93	Кодировка символов	Ilya
94	Коллекции	4castle, A_Arnold, Ad Infinitum, Alek Mieczkowski, alex s, altomnr, Andy Thomas, Anony-Mousse, Ashok Felix, Aurasphere, Bob Rivers, ced-b, ChandrasekarG, Chirag Parmar, clinomaniac, Codebender, Craig Gidney, Daniel Stradowski, dcod, DimaSan, Dušan Rychnovský, Enigo, Eran, fabian, fgb, GPI, Grzegorz Górkiewicz, ionyx, Jabir, Jan Vladimir Mostert, KartikKannapur, Kenster, KIRAN KUMAR MATAM, koder23, KudzieChase, Makoto, Maroun Maroun, Martin Frank, Matsemann, Mike H, Mo.Ashfaq, Mrunal Pagnis, mystarrocks, Oleg Sklyar, Pablo, Paweł Albecki, Petter Friberg, philnate, Polostor, Poonam, Powerlord, ppeterka, Prasad Reddy, Radiodef, rajadilipkolli, rd22, rdonuk, Ruslan Bes, Samk, SjB, Squidward, Stephen C, Stephen Leppik, Unihedron, user2296600, user3105453, Vasiliy Vlasov, Vasily Kabunov, VatsalSura, vsminkov, webo80, xploreraj
95	Команда Java - «java» и «javaw»	4444, Ben, mnoronha, Stephen C, Vogel612
96	Команды выполнения	RamenChef
97	Компилятор Java - «javac»	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C, Snađoшfa□, Tom Gijselinck
98	Компилятор Just in Time (JIT)	Liju Thomas, Stephen C
99	Консольный ввод-вывод	Aaron Franke, Ani Menon, Erkan Haspulat, Francesco Menzani, jayantS, Lankymart, Loris Securo, manetsus, Olivier Grégoire, Petter Friberg, rolve, Saagar Jha, Stephen C
100	Конструкторы	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot, tynn, Vogel612
101	литералы	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
102	Локализация и интернационализация	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
103	Лямбда-выражения	Abhishek Jain, Ad Infinitum, Adam, aioobe, Amit Gupta, Andrei Maieras, Andrew Tobilko, Andrii Abramov, Ankit Katiyar, Anony-Mousse, assylias, Brian Goetz, Burkhard, Conrad.Dean, cringe, Daniel M., David Soroko, dimitrisli, Draken, DVarga, Emre Bolat, enrico.bacis, fabian, fgb, Gal Dreiman, gar, GPI, Hank D, hexafraction, Ivan Vergiliev, J Atkin, Jean-François Savard, Jeroen Vandevelde, John Slegers, JonasCz, Jorn Vernee, Jude Niroshan, JudgingNotJudging, Kevin Raoofi, Malt, Mark Green, Matt, Matthew Trout, Matthias Braun, ncmathsadist, nobeh, Ortomala Lokni, Paŭlo Ebermann, Paweł Albecki, Petter Friberg,

		philnate, Pujan Srivastava, Radouane ROUFID, RamenChef, rolve, Saclyr Barlonium, Sergii Bishyr, Skylar Sutton, solomonope, Stephen C, Stephen Leppik, timbooo, Tunaki, Unihedron, vincentvanjoe, Vlasec, Vogel612, webo80, William Ritson, Wolfgang, Xaerxess, xploreraj, Yogi, Ze Rubeus
104	Массивы	3442, 416E64726577, A Boschman, A.M.K, A_Arnold, Abhishek Jain, Abubakkar, acdcjunior, Ad Infinitum, Addis, Adrian Krebs, AER, afzalex, agilob, Alan, Alex Shesterov, Alexandru, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, AndroidMechanic, Anil , ankidaemon, ankit dassor, anotherGatsby, antonio, Ares, Arthur , Ashish Ahuja, assylias, AstroCB, baao, Beggs, Berzerk, Big Fan , BitNinja, bjb568, Blubberguy22, Bob Rivers, bpoiss, Bryan, BudsNanKis, Burkhard, bwegs, c1phr, Cache Staheli, Cerbrus, Charitha, Charlie H, Chris Midgley, Christophe Weis, Christopher Schneider, Codebender, coder-croc, Cold Fire, Colin Pickard, Community, Confiqure, CptEric, Daniel Käfer, Daniel Stradowski, Dariusz, DarkV1, David G., DeepCoder, Devid Farinelli, Dhrubajyoti Gogoi, Dmitry Ginzburg, dorukayhan, Duh-Wayne-101, Durgpal Singh, DVarga, Ed Cottrell, Edvin Tenovimas, Eilit, eisbehr, Elad, Emil Sierżęga, Emre Bolat, Eng.Fouad, enrico.bacis, Eran, Erik Minarini, Etki, explv, fabian, fedorqui , Filip Haglund, Forest White, fracz, Franck Dernoncourt, Functino, futureelite7, Gal Dreiman, gar, Gene Marin, GingerHead , granmirupa, Grexis, Grzegorz Sancewicz, Gubbel, Guilherme Torres Castro, Gustavo Coelho, hhj8i, Hiren, Idos, ihatecsv, iliketocode, Ilya, Ilyas Mimouni, intboolstring, Irfan, J Atkin, jabbathehutt1234, JakeD, James Taylor, Jamie, Jamie Rees, Janez Kuhar, Jared Rummler, Jargonius, Jason Sturges, JavaHopper, Javant, Jeeter, Jeffrey Bosboom, Jens Schauder, Jérémie Bolduc, Jeutnarg, jhnance, Jim Garrison, jitendra varshney, jmattheis, Joffrey, Johannes, johannes_preiser, John Slegers, JohnB, Jojodmo, Jonathan, Jordi Castilla, Jorn, Jorn Vernee, Josh, JStef, JudgingNotJudging, Justin, Kapep, KartikKannapur, Kayathiri, Kaz Wolfe, Kenster, Kevin Thorne, Lambda Ninja, Liju Thomas, llamositopia, Loris Securo, Luan Nico, Lucas Paolillo, maciek, Magisch, Makoto, Makyen, Malt, Marc, Markus, Marvin, MasterBlaster, Matas Vaitkevicius, matsve, Matt, Matt, Matthias Braun, Maxim Kreschishin, Maxim Plevako, Maximillian Laumeister, MC Emperor, Menasheh, Michael Piefel, michaelbahr, Miljen Mikic, Minhas Kamal, Mitch Talmadge, Mohamed Fadhl, Muhammed Refaat, Muntasir, Mureinik, Mzzzzzz, NageN, Nathaniel Ford, Nayuki, nicael, Nigel Nop, niyasc, noʏʜʌdʌzɐɔƆ, Nuri Tasdemir, Ocracoke, OldMcDonald, Onur, orccrusher99, Ortomala Lokni, Panda, Paolo Forgia, Paul Bellora, Paweł Albecki, PeerNet, Peter Gordon, phatfingers, Pimgd, Piyush, ppeterka, Přemysl Šťastný, PSN, Pujan Srivastava, QoP, Radiodef, Radouane ROUFID, Raidri, Rajesh , Rakitić, Ram, RamenChef, Ravi Chandra, René Link, Reut Sharabani, Richard Hamilton, Robert Columbia, rolfedh, rolve, Roman Cherepanov, roottraveller, Ross, Ryan Hilbert, Sam Hazleton, sandbo00, Saurabh, Sayakiss, sebkur, Sergii Bishyr, sevenforce, shmosel, Shoe, Siguza, Simulant, Slayther, Smi, solidcell, Spencer Wieczorek, Squidward, stackptr, stark, Stephen C, Stephen Leppik, Sualeh Fatehi, sudo, Sumurai8, Sunnyok, syb0rg, tbodt, tdelev, tharkay, Thomas, ThunderStruct, Tol182, ʇolɐəz əɥʇ qoq, tpunt, Travis J, Tunaki, Un3qual, Unihedron, user6653173, uzaif, vasili111, VedX, Ven, Victor G., Vikas Gupta, vincentvanjoe, Vogel612, Wilson, Winter, X.lophix, YCF_L, Yohanes Khosiawan □□□, yuku, Yury Fedorov, zamonier, Ф Хосę □ Перейра ヅ
105	Менеджер по безопасности	alphaloop, hexafraction, Uux

106	Местное время	100rabh, A_Arnold, Alex, Andrii Abramov, Bob Rivers, Cache Staheli, DimaSan, Jasper, Kakarot, Kuroda, Manuel Vieda, Michael Piefel, phatfingers, RamenChef, Skylar Sutton, Vivek Anoop
107	Местный внутренний класс	KIRAN KUMAR MATAM
108	Методы и конструкторы классов объектов	A Boschman, Ad Infinitum, Andrii Abramov, Ani Menon, anuvab1911, Arthur Noseda, augray, Brett Kail, Burkhard, CaffeineToCode, Chris Midgley, cricket_007, Dariusz, Elazar, Emil Sierżęga, Enigo, fabian, fgb, Floern, fzzfzzfzz, hd84335, intboolstring, james large, JamesENL, Jens Schauder, John Slegers, Jorn Vernee, kstandell, Lahiru Ashan, Laurel, Miljen Mikic, mnoronha, mykey, NageN, Nayuki, Nicktar, Pace, Petter Friberg, Radouane ROUFID, Ram, Robert Columbia, Ronnie Wang, shmosel, Stephen C, TNT
109	Методы по умолчанию	ar4ers, hd84335, intboolstring, javac, Jeffrey Bosboom, Jens Schauder, Kai, matt freake, o_nix, philnate, Ravindra HV, richersoon, Ruslan Bes, Stephen C, Stephen Leppik, Vasiliy Vlasov
110	Методы сбора коллекции	Jacob G.
111	Модель памяти Java	Shree, Stephen C, Suminda Sirinath S. Dharmasena
112	Модификаторы без доступа	Ankit Katiyar, Arash, fabian, Florian Weimer, FlyingPiMonster, Grzegorz Górkiewicz, J-Alex, JavaHopper, Ken Y-N, KIRAN KUMAR MATAM, Miljen Mikic, NageN, Nuri Tasdemir, Onur, ppeterka, Prateek Agarwal
113	Модули	Jonathan, user140547
114	наборы	A_Arnold, atom, ced-b, Chirag Parmar, Daniel Stradowski, demongolem, DimaSan, fabian, Kaushal28, Kenster
115	наследование	Ad Infinitum, Adam, Adrian Krebs, agoeb, Ali Dehghani, Andrii Abramov, ar4ers, Arkadiy, Blubberguy22, Bohemian, Brad Larson, Burkhard, CodeCore, coder-croc, Dariusz, David Grinberg, devnull69, DonyorM, DVarga, Emre Bolat, explv, fabian, gattsbr, geniushkg, GhostCat, Gubbel, hirosht, HON95, J Atkin, Jason V, JavaHopper, Jeffrey Bosboom, Jens Schauder, Jonathan, Jorn Vernee, Kai, Kevin DiTraglia, kiuby_88, Lahiru Ashan, Luan Nico, maheshkumar, Mshnik, Muhammed Refaat, OldMcDonald, Oleg Sklyar, Ortomala Lokni, PM 77-1, Prateek Agarwal, QoP, Radouane ROUFID, RamenChef, Ravindra babu, Shog9, Simulant, SjB, Slava Babin, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, Theo, ToTheMaximum, uhrm, Unihedron, Vasiliy Vlasov, Vucko
116	Настройка производительности Java	Gene Marin, jatanp, Stephen C, Vogel612
117	Неизменяемые объекты	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
118	Неизменяемый класс	Mykola Yashchenko
119	Необязательный	A Boschman, Abubakkar, Andrey Rubtsov, Andrii Abramov, assylias, bowmore, Charlie H, Chris H., Christophe Weis, compuhosny, Dair, Emil Sierżęga, enrico.bacis, fikovnik, Grzegorz Górkiewicz, gwintrob, Hadson, hd84335, hzpz, J Atkin, Jean-François Savard,

		John Slegers, Jude Niroshan, Maroun Maroun, Michael Wiles, OldMcDonald, shmosel, Squidward, Stefan Dollase, Stephen C, ultimate_guy, Unihedron, user140547, Vince, vsminkov, xwoker
120	Новый ввод-вывод файлов	dorukayhan, niheno, TuringTux
121	Обработка аргументов командной строки	Burkhard, Michael von Wenckstern, Stephen C
122	Общие ошибки Java	akvyalkov, Anand Vaidya, Andy Thomas, Anton Hlinisty, anuvab1911, Conrad.Dean, Daniel Nugent, Dushko Jovanovski, Enwired, Gal Dreiman, Gerald Mücke, HTNW, james large, Jenny T-Type, John Starich, Lahiru Ashan, Makoto, Morgan Zhang, NamshubWriter, P.J.Meisch, Pirate_Jack, ppeterka, RamenChef, screab, Siva Sankar Rajendran, Squidward, Stephen C, Stephen Leppik, Steve Harris, tonirush, TuringTux, user3105453
123	Одиночки	aasu, Andrew Antipov, Daniel Käfer, Dave Ranjan, David Soroko, Emil Sierżęga, Enigo, fabian, Filip Smola, GreenGiant, Gubbel, Hulk, Jabir, Jens Schauder, JonasCz, Jonathan, JonK, Malt, Matsemann, Michael Lloyd Lee mlk, Mifeet, Miroslav Bradic, NamshubWriter, Pablo, Peter Rader, RamenChef, riyaz-ali, sanastasiadis, shmosel, Stefan Dollase, stefanobaghino, Stephen C, Stephen Leppik, still_learning, Uri Agassi, user3105453, Vasiliy Vlasov, Vlad-HC, Vogel612, xploreraj
124	операторы	17slim, 1d0m3n30, A Boschman, acdcjunior, afuc func, AJ Jwair, Amani Kilumanga, Andreas, Andrew, Andrii Abramov, Blake Yarbrough, Blubberguy22, Bobas_Pett, c.uent, Cache Staheli, Chris Midgley, Claudia, clinomaniac, Dariusz, Darth Shadow, Davis, EJP, Emil Sierżęga, Eran, fabian, FedeWar, FlyingPiMonster, futureelite7, Harsh Vakharia, hd84335, J Atkin, JavaHopper, Jérémie Bolduc, jimrm, Jojodmo, Jorn Vernee, kanhaiya agarwal, Kevin Thorne, Li357, Loris Securo, Lynx Brutal, Maarten Bodewes, Mac70, Makoto, Marvin, Michael Anderson, Mshnik, NageN, Nuri Tasdemir, Ortomala Lokni, OverCoder, ParkerHalo, Peter Gordon, ppeterka, qxz, rahul tyagi, RamenChef, Ravan, Reut Sharabani, Rubén, sargue, Sean Owen, ShivBuyya, shmosel, SnoringFrog, Stephen C, tonirush, user3105453, Vogel612, Winter
125	Операции с плавающей точкой Java	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
126	Ориентиры	esin88
127	Основные управляющие структуры	Adrian Krebs, AJNeufeld, Andrew Brooke, AshanPerera, Buddy, Caleb Brinkman, Cas Eliëns, Coffeehouse Coder, CraftedCart, dedmass, ebo, fabian, intboolstring, Inzimam Tariq IT, Jens Schauder, JonasCz, Jorn Vernee, juergen d, Makoto, Matt Champion, philnate, Ram, Santhosh Ramanan, sevenforce, Stephen C, teek, Unihedron, Uri Agassi, xwoker
128	Отправка динамического метода	Jeet
129	Оценка XML XPath	17slim, manouti
130	Очереди и Deques	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan, engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C

131	Ошибки Java - Nulls и NullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
132	Ошибки Java - потоки и параллелизм	dorukayhan, james large, Stephen C
133	Ошибки Java - проблемы с производительностью	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
134	пакеты	JamesENL, KIRAN KUMAR MATAM
135	Параллельное программирование (темы)	adino, Alex, assylias, bfd, Bhagyashree Jog, bowmore, Burkhard, Chetya, corsiKa, Dariusz, Diane Chastain, DimaSan, dimo414, Fildor, Freddie Coleman, GPI, Grzegorz Górkiewicz, hd84335, hellrocker, hexafraction, Ilya, james large, Jens Schauder, Johannes, Jorn Vernee, Kakarot, Lance Clark, Malt, Matěj Kripner, Md. Nasir Uddin Bhuiyan, Michael Piefel, michaelbahr, Mitchell Tracy, MSB, Murat K., Mureinik, mvd, NatNgs, nickguletskii, Olivier Durin, OlivierTheOlive, Panda, parakmiakos, Paweł Albecki, ppeterka, RamenChef, Ravindra babu, rd22, RudolphEst, snowe2010, Squidward, Stephen C, Sudhir Singh, Tobias Friedinger, Unihedron, Vasiliy Vlasov, Vlad-HC, Vogel612, wolfcastle, xTrollxDudex, YCF_L, Yury Fedorov, ZX9
136	Параллельное программирование с использованием структуры Fork / Join	Community, Joe C
137	Параллельные коллекции	GPI, Kenster, Powerlord, user2296600
138	Перечисления	1d0m3n30, A Boschman, aioobe, Amani Kilumanga, Andreas Fester, Andrew Sklyarevsky, Andrew Tobilko, Andrii Abramov, Anony-Mousse, bcosynot, Bob Rivers, coder-croc, Community, Constantine, Daniel Käfer, Daniel M., Danilo Guimaraes, DVarga, Emil Sierżęga, enrico.bacis, f_puras, fabian, Gal Dreiman, Gene Marin, Grexis, Grzegorz Oledzki, ipsi, J Atkin, Jared Hooper, javac, Jérémie Bolduc, Johannes, Jon Ericson, k3b, Kenster, Lahiru Ashan, Maarten Bodewes, madx, Mark, Michael Myers, Mick Mnemonic, NageN, Nef10, Nolequen, OldCurmudgeon, OliPro007, OverCoder, P.J.Meisch, Panther, Paweł Albecki, Petter Friberg, Punika, Radouane ROUFID, RamenChef, rd22, Ronon Dex, Ryan Hilbert, S.K. Venkat, Samk, shmosel, Spina, Stephen Leppik, Tarun Maganti, Tim, Torsten, VGR, Victor G., Vinay , Wolf, Yury Fedorov, Zefick, Φ Xocę □ Пereúpa ツ
139	Полиморфизм	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
140	предпочтения	RAnders00
141	Преобразование в строки и из них	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark, PSo, RamenChef, Shaan, Stephen C, still_learning
142	Преобразование типа	4castle, Filip Smola, Joshua Carmody, Nick Donnelly, RamenChef, Squidward

143	Примитивные типы данных	17slim, 1d0m3n30, Amani Kilumanga, Ani Menon, Anony-Mousse, Bilesh Ganguly, Bob Rivers, Burkhard, Conrad.Dean, Daniel, Dariusz, DimaSan, dnup1092, Do Nhu Vy, enrico.bacis, fabian, Francesco Menzani, Francisco Guimaraes, gar, Ilya, IncrediApp, ipsi, J Atkin, JakeD, javac, Jean-François Savard, Jojodmo, Kapep, KdgDev, Lahiru Ashan, Master Azazel, Matt, mayojava, MBorsch, nimrod, Pang, Panther, ParkerHalo, Petter Friberg, Radek Postołowicz, Radouane ROUFID, RAnders00, RobAu, Robert Columbia, Simulant, Squidward, Stephen C, Stephen Leppik, Sundeep, SuperStormer, ThePhantomGamer, TMN, user1803551, user2314737, Veedrac, Vogel612
144	Процесс	Andy Thomas, Bob Rivers, ppeterka, vorburger, yitzih
145	Путь Класса	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C, trashgod
146	Разборка и декомпиляция	ipsi, mnoronha
147	Развертывание Java	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit, Stephen C
148	Разделение строки на части с фиксированной длиной	Bohemian
149	Реализации Java-плагинов	Alexiy
150	Регулярные выражения	Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron, Vogel612
151	Рекурсия	Andy Thomas, atom, Bobas_Pett, Ce7, charlesreid1, Confiqure, David Soroko, fabian, hamena314, hd84335, JavaHopper, Javant, Matej Kormuth, mayojava, Nicktar, Peter Gordon, RamenChef, Raviteja, Ruslan Bes, Stephen C, sumit
152	Ресурсы (на пути к классам)	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke , Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
153	Розетки	Ordiel
154	Свободный интерфейс	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
155	Сериализация	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov
156	сетей	Arthur, Burkhard, devnull69, DonyorM, glee8e, Grayson Croom, Ilya, Malt, Matej Kormuth, Matthieu, Mine_Stone, ppeterka, RamenChef, Stephen C, Tot Zam, vsav
157	сканер	Alek Mieczkowski, Chirag Parmar, Community, Jon Ericson, JonasCz , Ram, RamenChef, Redterd, Stephen C, sun-solar-arrow, ФХосę □ П epeúpa ツ
158	Служба печати Java	Danilo Guimaraes, Leonardo Pina
159	Создание изображений	alain.janinm, Dariusz, kajacx, Kenster, mnoronha

	программно	
160	Создание кода Java	Tony
161	Сокеты Java	Nikhil R
162	Списки	17slim, A Boschman, Arthur, Avinash Kumar Yadav, Blubberguy22, ced-b, Daniel Nugent, granmirupa, Ilya, Jan Vladimir Mostert, janos, JD9999, jopasserat, Karthikeyan Vaithilingam, Kenster, Krzysztof Krasoń, Oleg Sklyar, RamenChef, Sheshnath, Stephen C, sudo, Thisaru Guruge, Vasilis Vasilatos, ΦΧοςę Переúра ツ
163	Список против SET	KIRAN KUMAR MATAM
164	Сравнение C ++	John DiFini
165	Сравнительный и компаратор	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
166	Ссылки на объекты	Andrii Abramov, arcy, Vasiliy Vlasov
167	Стандарт официального кода Oracle	Ahmed Ashour, aioobe, akhilsk, alex s, Andrii Abramov, Cassio Mazzochi Molin, Dan Whitehouse, Enigo, erickson, f_puras, fabian , giucal, hd84335, J.D. Sandifer, Lahiru Ashan, Mac70, NamshubWriter, Nicktar, Petter Friberg, Pradatta, Pritam Banerjee, RamenChef, sanjaykumar81, Santa Claus, Santhosh Ramanan, VGR
168	Строковый токенизатор	M M
169	Струны	17slim, A.J. Brown, A_Arnold, Abhishek Jain, Abubakkar, Adam Ratzman, Adrian Krebs, agilob, Aiden Deom, Alex Meiburg, Alex Shesterov, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, Andy Thomas, Anony-Mousse, Asaph, Ataeraxia, Austin, Austin Day, ben75, bfd, Bob Brinks, bpoiss, Burkhard, Cache Staheli, Caner Balım, Chris Midgley, Christian, Christophe Weis, coder-croc, Community, cyberscientist, Daniel Käfer, Daniel Stradowski, DarkV1, dedmass, DeepCoder, dnup1092, dorukayhan, drov, DVarga, ekeith, Emil Sierżęga, emotionlessbananas, enrico.bacis, Enwired, fabian, FlyingPiMonster, Gabriele Mariotti, Gal Dreiman, Gergely Toth, Gihan Chathuranga, GingerHead, giucal, Gray, GreenGiant, hamena314, Harish Gyanani, HON95, iliketocode, Ilya, Infuzed guy, intboolstring, J Atkin, Jabir, javac, JavaHopper, Jeffrey Lin, Jens Schauder, Jérémie Bolduc, John Slegers, Jojodmo, Jon Ericson, JonasCz, Jordi Castilla, Jorn Vernee, JSON C11, Jude Niroshan, Kamil Akhuseyinoglu, Kapep, Kaushal28, Kaushik NP, Kehinde Adedamola Shittu, Kenster, kstandell, Lachlan Dowding, Lahiru Ashan, Laurel, Leo Aso, Liju Thomas, LisaMM, M.Sianaki, Maarten Bodewes , Makoto, Malav, Malt, Manoj, Manuel Spigolon, Mark Stewart, Marvin, Matej Kormuth, Matt Clark, Matthias Braun, maxdev, Maxim Plevako, mayha, Michael, MikeW, Miles, Miljen Mikic, Misa Lazovic, mr5, Myridium, NikolaB, Nufail, Nuri Tasdemir, OldMcDonald, OliPro007, Onur, Optimiser, ozOli, P.J.Meisch, Paolo Forgia, Paweł Albecki, Petter Friberg, phant0m, piyush_baderia, ppeterka, Přemysl Šťastný, PSo, QoP, Radouane ROUFID, Raj, RamenChef, RAnders00, Rocherlee, Ronnie Wang, Ryan Hilbert, ryanyuyu, Sayakiss, SeeuD1, sevenforce, Shaan, ShivBuyya, Shoe, Sky, SmS, solidcell, Squidward, Stefan Isele - prefabware.com, stefanobaghino, Stephen C, Stephen Leppik,

		Steven Benitez, still_learning, Sudhir Singh, Swanand Pathak, S нađошƒаl□, TDG, TheLostMind, ThePhantomGamer, Tony BenBrahim, Unihedron, VGR, Vishal Biyani, Vogel612, vsminkov, vvtx, Wilson, winseybash, xwoker, yuku, Yury Fedorov, Zachary David Saunders, Zack Teater, Ze Rubeus, ФХосę □ Переúра ツ
170	супер ключевое слово	Abhijeet
171	Тестирование устройства	Ironcache
172	Типы ссылок	EJP, NageN, Thisaru Guruge
173	Типы ссылочных данных	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan □□□
174	Удаленный вызов метода (RMI)	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
175	Управление памятью Java	Daniel M., engineercoding, fgb, John Nash, jwd630, mnoronha, OverCoder, padippist, RamenChef, Squidward, Stephen C
176	Установка Java (стандартная версия)	4444, Adeel Ansari, ajablonski, akhilsk, Alex A, altomnr, Ani Menon, Anthony Raymond, anuvab1911, Confiqure, CraftedCart, Emil Sierżęga, Gautam Jose, hd84335, ipsi, Jeffrey Brett Coleman, Lambda Ninja, Nithanim, Radouane ROUFID, Rakitić, ronnyfm, Sanandrea, Sandeep Chatterjee, sohnryang, Stephen C, Sнаđошƒаl□, tonirush, Walery Strauch, Ze Rubeus
177	Утверждая	Jonathan, Makoto, rajah9, RamenChef, The Guy with The Hat, Uri Agassi
178	Файловый ввод-вывод	Alper Fırat Kaya, Arthur, assylias, ata, Aurasphere, Burkhard, Conrad.Dean, Daniel M., Enigo, FlyingPiMonster, Gerald Mücke, Gubbel, Hay, hd84335, Jabir, James Jensen, Jason Sturges, Jordy Baylac, leaqui, mateuscb, MikaelF, Moshiour, Myridium, Nicktar, Peter Gordon, Petter Friberg, ppeterka, RAnders00, RobAu, rokonoid, Sampada, sebkur, ShivBuyya, Squidward, Stephen C, still_learning, Tilo, Tobias Friedinger, TuringTux, Will Hardwick-Smith
179	Файлы с несколькими релизами JAR	manouti
180	Флаги JVM	Confiqure, RamenChef
181	Функциональные интерфейсы	Andreas
182	Хеш-таблица	KIRAN KUMAR MATAM
183	Читатели и писатели	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
184	Шифрование RSA	Dennis Kriechel, Drunix, iqbal_cs, Maarten Bodewes, Nicktar, Shog9