FREE eBook

LEARNING Scala Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#scala

Table of Contents

About

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

It is an unofficial and free Scala 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 Scala 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

Chapter 1: Getting started with Scala Language

Remarks

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

Most given examples require a working Scala installation. This is the Scala installation page, and this is the 'How to setup Scala' example. scalafiddle.net is a good resource for executing small code examples over the web.

Versions

Version	Release Date
2.10.1	2013-03-13
2.10.2	2013-06-06
2.10.3	2013-10-01
2.10.4	2014-03-24
2.10.5	2015-03-05
2.10.6	2015-09-18
2.11.0	2014-04-21
2.11.1	2014-05-21
2.11.2	2014-07-24
2.11.4	2014-10-30
2.11.5	2014-01-14
2.11.6	2015-03-05
2.11.7	2015-06-23
2.11.8	2016-03-08
2.11.11	2017-04-19

Version	Release Date
2.12.0	2016-11-03
2.12.1	2016-12-06
2.12.2	2017-04-19

Examples

Hello World by Defining a 'main' Method

Place this code in a file named `HelloWorld.scala`:

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

Live demo

To compile it to bytecode that is executable by the JVM:

```
$ scalac HelloWorld.scala
```

To run it:

```
$ scala Hello
```

When the Scala runtime loads the program, it looks for an object named `Hello` with a `main` method. The `main` method is the program entry point and is executed.

Note that, unlike Java, Scala has no requirement of naming objects or classes after the file they're in. Instead, the parameter `Hello` passed in the command `scala Hello` refers to the object to look for that contains the `main` method to be executed. It is perfectly possible to have multiple objects with main methods in the same `.scala` file.

The `args` array will contain the command-line arguments given to the program, if any. For instance, we can modify the program like this:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
  }
}
```

Compile it:

```
$ scalac HelloWorld.scala
```

And then execute it:

```
$ scala HelloWorld 1 2 3
Hello World!
Arg=1
Arg=2
Arg=3
```

Hello World by extending App

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

Live demo

By extending the `App` trait, you can avoid defining an explicit `main` method. The entire body of the `HelloWorld` object is treated as "the main method".

2.11.0

> ## Delayed Initialization
>
> Per the official documentation, `App` makes use of a feature called *Delayed Initialization*. This means that the object fields are initialized *after* the main method is called.

2.11.0

> ## Delayed Initialization
>
> Per the official documentation, `App` makes use of a feature called *Delayed Initialization*. This means that the object fields are initialized *after* the main method is called.
>
> `DelayedInit` is now **deprecated** for general use, but is *still supported* for `App` as a special case. Support will continue until a replacement feature is decided upon and implemented.

To access command-line arguments when extending `App`, use `this.args`:

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
}
```

When using `App`, the body of the object will be executed as the `main` method, there is no need to override `main`.

Hello World as a script

Scala can be used as a scripting language. To demonstrate, create `HelloWorld.scala` with the following content:

```
println("Hello")
```

Execute it with the command-line interpreter (the `$` is the command line prompt):

```
$ scala HelloWorld.scala
Hello
```

If you omit `.scala` (such as if you simply typed `scala HelloWorld`) the runner will look for a compiled `.class` file with bytecode instead of compiling and then executing the script.

> **Note:** If scala is used as a scripting language no package can be defined.

In operating systems utilizing `bash` or similar shell terminals, Scala scripts can be executed using a 'shell preamble'. Create a file named `HelloWorld.sh` and place the following as its content:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

The parts between `#!` and `!#` is the 'shell preamble', and is interpreted as a bash script. The rest is Scala.

Once you have saved the above file, you must grant it 'executable' permissions. In the shell you can do this:

```
$ chmod a+x HelloWorld.sh
```

(Note that this gives permission to everyone: read about chmod to learn how to set it for more specific sets of users.)

Now you can execute the script like this:

```
$ ./HelloWorld.sh
```

Using the Scala REPL

When you execute `scala` in a terminal without additional parameters it opens up a REPL (Read-Eval-Print Loop) interpreter:

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

The REPL allows you to execute Scala in a worksheet fashion: the execution context is preserved and you can manually try out commands without having to build a whole program. For instance, by typing `val poem = "As halcyons we shall be"` would look like this:

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

Now we can print our `val`:

```
scala> print(poem)
As halcyons we shall be
```

Note that `val` is immutable and cannot be overwritten:

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

But in the REPL you *can* redefine a `val` (which would cause an error in a normal Scala program, if it was done in the same scope):

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

For the remainder of your REPL session this newly defined variable will shadow the previously defined variable. REPLs are useful for quickly seeing how objects or other code works. All of Scala's features are available: you can define functions, classes, methods, etc.

Scala Quicksheet

Description	Code
Assign immutable int value	`val x = 3`
Assign mutable int value	`var x = 3`
Assign immutable value with explicit type	`val x: Int = 27`
Assign lazily evaluated value	`lazy val y = print("Sleeping in.")`
Bind a function to a name	`val f = (x: Int) => x * x`
Bind a function to a name with explicit type	`val f: Int => Int = (x: Int) => x * x`

Description	Code
Define a method	`def f(x: Int) = x * x`
Define a method with explicit typing	`def f(x: Int): Int = x * x`
Define a class	`class Hopper(someParam: Int) { ... }`
Define an object	`object Hopper(someParam: Int) { ... }`
Define a trait	`trait Grace { ... }`
Get first element of sequence	`Seq(1,2,3).head`
If switch	`val result = if(x > 0) "Positive!"`
Get all elements of sequence except first	`Seq(1,2,3).tail`
Loop through a list	`for { x <- Seq(1,2,3) } print(x)`

Nested Looping	`for {`
	`  x <- Seq(1,2,3)`
	`  y <- Seq(4,5,6)`
	`} print(x + ":" + y)`

For each list element execute function	`List(1,2,3).foreach { println }`
Print to standard out	`print("Ada Lovelace")`
Sort a list alphanumerically	`List('b','c','a').sorted`

Read Getting started with Scala Language online: https://riptutorial.com/scala/topic/216/getting-started-with-scala-language

Chapter 2: Annotations

Syntax

- @AnAnnotation def someMethod = {...}
- @AnAnnotation class someClass {...}
- @AnnotatioWithArgs(annotation_args) def someMethod = {...}

Parameters

Parameter	Details
@	Indicates that the token following is an annotation.
SomeAnnotation	The name of the annotation
constructor_args	(optional) The arguments passed to the annotation. If none, the parentheses are unneeded.

Remarks

Scala-lang provides a list of standard annotations and their Java equivalents.

Examples

Using an Annotation

This sample annotation indicates that the following method is `deprecated`.

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

This can also be equivalently written as:

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Annotating the main constructor

```
/**
 * @param num Numerator
```

```
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom

    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

The visibility modifier (in this case `protected`) should come after the annotations in the same line. In case the annotation accepts optional parameters (as in this case `@throws` accepts an optional cause), you have to specify an empty parameter list for the annotation: `()` before the constructor parameters.

Note: Multiple annotations can be specified, even from the same type (repeating annotations).

Similarly with a case class without auxiliary factory method (and cause specified for the annotation):

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

Creating Your Own Annotations

You can create you own Scala annotations by creating classes derived from scala.annotation.StaticAnnotation or scala.annotation.ClassfileAnnotation

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

Annotations can then be interrogated using the reflection API.

```
scala>import scala.reflect.runtime.{universe ⇒ u}

scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus

scala>val platypusSymbol = platypusType.typeSymbol.asClass
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus
```

```
scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

Read Annotations online: https://riptutorial.com/scala/topic/3783/annotations

Chapter 3: Best Practices

Remarks

> Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

-- *Programming in Scala*, by Odersky, Spoon, and Venners

There are more example and guideline in this presentation by Odersky.

Examples

Keep it simple

Do not overcomplicate simple tasks. Most of the time you will need only:

- algebraic datatypes
- structural recursion
- monad-like api (`map`, `flatMap`, `fold`)

There is plenty of complicated stuff in Scala, such as:

- `Cake pattern` or `Reader Monad` for Dependency Injection.
- Passing arbitrary values as `implicit` arguments.

These things are not clear for newcomers: avoid using them before you understand them. Using advanced concepts without a real need obfuscates the code, making it *less* maintainable.

Don't pack too much in one expression.

- Find meaningful names for computation units.
- Use `for` comprehensions or `map` to combine computations together.

Let's say you have something like this:

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
      }
    ...
  }
}
```

If all your functions return `Either` or another `Validation`-like type, you can write:

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

Prefer a Functional Style, Reasonably

By default:

- Use `val`, not `var`, wherever possible. This allows you to take seamless advantage of a number of functional utilities, including work distribution.
- Use `recursion` and `comprehensions`s, not loops.
- Use immutable collections. This is a corrolary to using `val` whenever possible.
- Focus on data transformations, CQRS-style logic, and not CRUD.

There are good reasons to choose non-functional style:

- `var` can be used for local state (for example, inside an actor).
- `mutable` gives better performance in certain situations.

Read Best Practices online: https://riptutorial.com/scala/topic/4376/best-practices

Chapter 4: Case Classes

Syntax

- case class Foo() // Case classes with no parameters must have an empty list
- case class Foo(a1: A1, ..., aN: AN) // Create a case class with fields a1 ... aN
- case object Bar // Create a singleton case class

Examples

Case Class Equality

One feature provided for free by case classes is an auto-generated `equals` method that checks the value equality of all individual member fields instead of just checking the reference equality of the objects.

With ordinary classes:

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

With case classes:

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

Generated Code Artifacts

The `case` modifier causes the Scala compiler to automatically generate common boilerplate code for the class. Implementing this code manually is tedious and a source of errors. The following case class definition:

```
case class Person(name: String, age: Int)
```

... will have the following code automatically generated:

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)

  def productArity: Int = 2
```

```
  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

The `case` modifier also generates a companion object:

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

When applied to an `object`, the `case` modifier has similar (albeit less dramatic) effects. Here the primary gains are a `toString` implementation and a `hashCode` value that is consistent across processes. Note that case objects (correctly) use reference equality:

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
}
```

It is still possible to manually implement methods that would otherwise be provided by the `case`

modifier in both the class itself and its companion object.

Case Class Basics

In comparison to regular classes – case classes notation provides several benefits:

- All constructor arguments are `public` and can be accessed on initialized objects (normally this is not the case, as demonstrated here):

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- It provides an implementation for the following methods: `toString`, `equals`, `hashCode` (based on properties), `copy`, `apply` and `unapply`:

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- It provides a convenient mechanism for pattern matching:

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

Case Classes and Immutabilty

The Scala compiler prefixes every argument in the parameter list by default with `val`. This means that, by default, case classes are immutable. Each parameter is given an accessor method, but there are no mutator methods. For example:

```
case class Foo(i: Int)

val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

Declaring a parameter in a case class as `var` overrides the default behavior and makes the case class mutable:

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
barInstance.i = 2           // set
```

Another instance when a case class is 'mutable' is when the value in the case class is mutable:

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

Note that the 'mutation' that is occurring here is in the map that `m` points to, not to `m` itself. Thus, if some other object had `m` as a member, it would see the change as well. Note how in the following example changing `instanceA` also changes `instanceB`:

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

Create a Copy of an Object with Certain Changes

Case classes provide a `copy` method that creates a new object that shares the same fields as the old one, with certain changes.

We can use this feature to create a new object from a previous one that has some of the same characteristics. This simple case class to demonstrates this feature:

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

In this example we can see that the two objects share similar characteristics (`grade = A1`, `subject = Math`), except where they have been specified in the copy (`firstName` and `lastName`).

Single Element Case Classes for Type Safety

In order to achieve type safety sometimes we want to avoid the use of primitive types on our domain. For instance, imagine a `Person` with a `name`. Typically, we would encode the `name` as a

`String`. However, it would not be hard to mix a `String` representing a `Person`'s `name` with a `String` representing an error message:

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

To avoid such pitfalls you can encode the data like this:

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

and now our code will not compile if we mix `PersonName` with `ErrorMessage`, or even an ordinary `String`.

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

But this incurs a small runtime overhead as we now have to box/unbox `String`s to/from their `PersonName` containers. In order to avoid this, one can make `PersonName` and `ErrorMessage` value classes:

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

Read Case Classes online: https://riptutorial.com/scala/topic/1022/case-classes

Chapter 5: Classes and Objects

Syntax

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)` or with a different type: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

Instantiate Class Instances

A class in Scala is a 'blueprint' of a class instance. An instance contains the state and behavior as defined by that class. To declare a class:

```
class MyClass{}  // curly braces are optional here as class body is empty
```

An instance can be instantiated using `new` keyword:

```
var instance = new MyClass()
```

or:

```
var instance = new MyClass
```

Parentheses are optional in Scala for creating objects from a class that has a no-argument constructor. If a class constructor takes arguments:

```
class MyClass(arg : Int)       // Class definition
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   // not allowed
```

Here `MyClass` requires one `Int` argument, which can only be used internally to the class. `arg` cannot be accessed outside `MyClass` unless it is declared as a field:

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
obj.prop     // legal statement
```

Alternatively it can be declared public in the constructor:

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

Instantiating class with no parameter: {} vs ()

Let's say we have a class MyClass with no constructor argument:

```
class MyClass
```

In Scala we can instantiate it using below syntax:

```
val obj = new MyClass()
```

Or we can simply write:

```
val obj = new MyClass
```

But, if not paid attention, in some cases optional parenthesis may produce some unexpected behavior. Suppose we want to create a task that should run in a separate thread. Below is the sample code:

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

We may think that this sample code if executed will print `Performing task.`, but to our surprise, it won't print anything. Let's see what's happening here. If you pay a closer look, we have used curly braces `{}`, right after `new Thread`. It created an annonymous class which extends `Thread`:

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

And then in the body of this annonymous class, we defined our task (again creating an annonymous class implementing `Runnable` interface). So we might have thought that we used `public Thread(Runnable target)` constructor but in fact (by ignoring optional `()`) we used `public Thread()` constructor with nothing defined in the body of `run()` method. To rectify the problem, we need to use parenthesis instead of curly braces.

```
val newThread = new Thread ( new Runnable {
```

```
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    )
```

In other words, here `{}` and `()` are not *interchangeable*.

Singleton & Companion Objects

Singleton Objects

Scala supports static members, but not in the same manner as Java. Scala provides an alternative to this called *Singleton Objects*. Singleton objects are similar to a normal class, except they can not be instantiated using the `new` keyword. Below is a sample singleton class:

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

Note that we have used `object` keyword to define singleton object (instead of 'class' or 'trait'). Since singleton objects can not be instantiated they can not have parameters. Accessing a singleton object looks like this:

```
Factorial.getCache() //returns the cache
```

Note that this looks exactly like accessing a static method in a Java class.

Companion Objects

In Scala singleton objects may share the name of a corresponding class. In such a scenario the singleton object is referred to as a *Companion Object*. For instance, below the class `Factorial` is defined, and a companion object (also named `Factorial`) is defined below it. By convention companion objects are defined in the same file as their companion class.

```
class Factorial(num : Int) {

  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }

    Factorial.cache(num)
  }
}
```

```
object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

In this example we are using a private `cache` to store factorial of a number to save calculation time for repeated numbers.

Here `object Factorial` is a companion object and `class Factorial` is its corresponding companion class. Companion objects and classes can access each other's `private` members. In the example above `Factorial` class is accessing the private `cache` member of it's companion object.

Note that a new instantiation of the class will still utilize the same companion object, so any modification to member variables of that object will carry over.

Objects

Whereas Classes are more like blueprints, Objects are static (i.e. already instantiated):

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

They are often used as a companion to a class, they allow you to write:

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}

val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

Instance type checking

Type check: `variable.isInstanceOf[Type]`

With pattern matching (not so useful in this form):

```
variable match {
  case _: Type => true
  case _ => false
}
```

Both `isInstanceOf` and pattern matching are checking only the object's type, not its generic parameter (no type reification), except for arrays:

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

But

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                               //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

Type casting: `variable.asInstanceOf[Type]`

With pattern matching:

```
variable match {
  case _: Type => true
}
```

Examples:

```
  val x = 3                                       //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
    case _ => false
  }                                               //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
```

```
  }                                               //> res1: Boolean = true

  x.isInstanceOf[Int]                             //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                          //> y  : Valuable = V(3)
  y.isInstanceOf[V]                               //> res3: Boolean = true
  y.asInstanceOf[V]                               //> res4: V = V(3)
```

Remark: This is only about the behaviour on the JVM, on other platforms (JS, native) type casting/checking might behave differently.

Constructors

Primary Constructor

In Scala the primary constructor is the body of the class. The class name is followed by a parameter list, which are the constructor arguments. (As with any function, an empty parameter list may be omitted.)

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

The construction parameters of an instance are not accessible outside its constructor body unless marked as an instance member by the `val` keyword:

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
val bar1 = new Bar() // Constructor parentheses are optional here
```

Any operations that should be performed when an instance of an object is instantiated are written directly in the body of the class:

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
```

```
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
    ...
}
```

Note that it is considered good practice to put as few side effects into the constructor as possible; instead of the above code, one should consider having `connect` and `disconnect` methods so that consumer code is responsible for scheduling IO.

Auxiliary Constructors

A class may have additional constructors called 'auxiliary constructers'. These are defined by constructor definitions in the form `def this(...) = e`, where `e` must invoke another constructor:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

This implies each constructor can have a different modifier: only some may be available publicly:

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

In this way you can control how consumer code may instantiate the class.

Read Classes and Objects online: https://riptutorial.com/scala/topic/2047/classes-and-objects

Chapter 6: Collections

Examples

Sort A List

Supposing the following list we can sort a variety of ways.

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

The default behavior of `sorted()` is to use `math.Ordering`, which for strings results in a lexographic sort:

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith` allows you to provide your own ordering utilizing a comparison function:

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy` allows you to provide a transformation function:

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

You can always reverse a list, or a sorted list, using `reverse:

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

Lists can also be sorted using Java method `java.util.Arrays.sort` and its Scala wrapper `scala.util.Sorting.quickSort`

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

These methods can improve performance when sorting larger collections if the collection conversions and unboxing/boxing can be avoided. For a more detailed discussion on the performance differences, read about Scala Collection sorted, sortWith and sortBy Performance.

Create a List containing n copies of x

To create a collection of `n` copies of some object `x`, use the fill method. This example creates a `List`, but this can work with other collections for which `fill` makes sense:

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

List and Vector Cheatsheet

> It is now a best-practice to use `Vector` instead of `List` because the implementations have better performance Performance characteristics can be found here. `Vector` can be used wherever `List` is used.

List creation

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

Take element

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

Prepend Elements

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

Append Elements

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

Join (Concatenate) Lists

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

Common operations

```
List(1, 2, 3).find(_ == 3)                    // Some(3)
List(1, 2, 3).map(_ * 2)                      // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)              // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

Map Collection Cheatsheet

> Note that this deals with the creation of a collection of type `Map`, which is distinct from the `map` method.

Map Creation

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

A map can be considered a collection of `tuples` for most operations, where the first element is the key and the second is the value.

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

Get element

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

Add element(s)

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)              // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)              // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

Common operations

In operations where an iteration over a map occurs (`map`, `find`, `forEach`, etc), the elements of the collection are `tuples`. The function parameter can either use the tuple accessors (`_1`, `_2`), or a partial function with a case block:

```
m.find(_._1 == "a")  // Some((a,1))
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
```

```
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

Map and Filter Over A Collection

Map

'Mapping' across a collection uses the `map` function to transform each element of that collection in a similar way. The general syntax is:

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

You can provide an anonymous function:

```
collection.map((x: T) => /*Do something with x*/)
```

Multiplying integer numbers by two

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

Filter

`filter` is used when you want to exclude or 'filter out' certain elements of a collection. As with `map`, the general syntax takes a function, but that function must return a `Boolean`:

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

You can provide an anonymous function directly:

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

Checking pair numbers

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

More Map and Filter examples

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

Introduction to Scala Collections

The Scala Collections framework, according to its authors, is designed to be easy to use, concise, safe, fast, and universal.

The framework is made up of Scala traits that are designed to be building blocks for creating collections. For more information on these building blocks, read the official Scala collections overview.

These built-in collections are separated into the immutable and mutable packages. By default, the immutable versions are used. Constructing a `List()` (without importing anything) will construct an *immutable* list.

One of the most powerful features of the framework is the consistent and easy-to-use interface across like-minded collections. For example, summing all elements in a collection is the same for Lists, Sets, Vectors, Seqs and Arrays:

```
val numList = List[Int](1, 2, 3, 4, 5)
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
```

```
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

These like-minded types inherit from the `Traversable` trait.

> It is now a best-practice to use `Vector` instead of `List` because the implementations have better performance Performance characteristics can be found here. `Vector` can be used wherever `List` is used.

Traversable types

Collection classes that have the `Traversable` trait implement `foreach` and inherit many methods for performing common operations to collections, which all function identically. The most common operations are listed here:

- Map - `map`, `flatMap`, and `collect` produce new collections by applying a function to each element in the original collection.

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- Conversions - `toList`, `toArray`, and many other conversion operations change the current collection into a more specific kind of collection. These are usually methods prepended with 'to' and the more specific type (i.e. 'toList' converts to a `List`).

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- Size info - `isEmpty`, `nonEmpty`, `size`, and `hasDefiniteSize` are all metadata about the set. This allows conditional operations on the collection, or for code to determine the size of the collection, including whether it's infinite or discrete.

```
List().isEmpty // true
List(1).nonEmpty // true
```

- Element retrieval - `head`, `last`, `find`, and their `Option` variants are used to retrieve the first or last element, or find a specific element in the collection.

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- Sub-collection retrieval operations - `filter`, `tail`, `slice`, `drop`, and other operations allow for choosing parts of the collection to operate on further.

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- Subdivision operations - `partition`, `splitAt`, `span`, and `groupBy` split the current collection into different parts.

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- Element tests - `exists`, `forall`, and `count` are operations used to check this collection to see if it satisfies a predicate.

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Folds - `foldLeft` (`/:`), `foldRight` (`:\`), `reduceLeft`, and `reduceRight` are used to apply binary functions to successive elements in the collection. Go here for fold examples and go here for reduce examples.

Fold

The `fold` method iterates over a collection, using an initial accumulator value and applying a function that uses each element to update the accumulator successfully:

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

In the above example, an anonymous function was supplied to `fold()`. You can also use a named function that takes two arguments. Bearing this in my, the above example can be re-written thus:

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

Changing the initial value will affect the result:

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

The `fold` method has two variants - `foldLeft` and `foldRight`.

`foldLeft()` iterates from left to right (from the first element of the collection to the last in that order).
`foldRight()`

iterates from right to left (from the last element to the first element). `fold()` iterates from left to right like `foldLeft()`. In fact, `fold()` actually calls `foldLeft()` internally.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

`fold()`, `foldLeft()` and `foldRight()` will return a value that has the same type with the initial value it takes. However, unlike `foldLeft()` and `foldRight()`, the initial value given to `fold()` can only be of the same type or a supertype of the type of the collection.

In this example the order is not relevant, so you can change `fold()` to `foldLeft()` or `foldRight()` and the result will remain the same. Using a function that is sensitive to order will alter results.

If in doubt, prefer `foldLeft()` over `foldRight()`. `foldRight()` is less performant.

Foreach

`foreach` is unusual among the collections iterators in that it does not return a result. Instead it applies a function to each element that has only side effects. For example:

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
1
2
3
```

The function supplied to `foreach` can have any return type, but the result will be discarded. Typically `foreach` is used when side effects are desirable. If you want to transform data consider using `map`, `filter`, a `for comprehension`, or another option.

Example of discarding results

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

Reduce

The `reduce()`, `reduceLeft()` and `reduceRight` methods are similar to folds. The function passed to reduce takes two values and yields a third. When operating on a list, the first two values are the first two values in the list. The result of the function and the next value in the list are then re-applied to the function, yielding a new result. This new result is applied with the next value of the list and so on until there are no more elements. The final result is returned.

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")
```

```
def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

There are some differences in how the reduce functions work as compared to the fold functions. They are:

1. The reduce functions have no initial accumulator value.
2. Reduce functions cannot be called on empty lists.
3. Reduce functions can only return the type or supertype of the list.

Read Collections online: https://riptutorial.com/scala/topic/686/collections

Chapter 7: Continuations Library

Introduction

Continuation passing style is a form of control flow that involves passing to functions the rest of the computation as a "continuation" argument. The function in question later invokes that continuation to continue program execution. One way to think of a continuation is as a closure. The Scala continuations library brings delimited continuations in the form of the primitives `shift`/`reset` to the language.

continuations library: https://github.com/scala/scala-continuations

Syntax

- reset { ... } // Continuations extend up to the end of the enclosing reset block
- shift { ... } // Create a continuation stating from after the call, passing it to the closure
- A @cpsParam[B, C] // A computation that requires a function A => B to create a value of C
- @cps[A] // Alias for @cpsParam[A, A]
- @suspendable // Alias for @cpsParam[Unit, Unit]

Remarks

`shift` and `reset` are primitive control flow structures, like `Int.+` is a primitive operation and `Long` is a primitive type. They are more primitive than either in that delimited continuations can actually be used to construct almost all control flow structures. They are not very useful "out-of-the-box", but they truly shine when they are used in libraries to create rich APIs.

Continuations and monads are also closely linked. Continuations can be made into the continuation monad, and monads are continuations because their `flatMap` operation takes a continuation as parameter.

Examples

Callbacks are Continutations

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???

readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

The function argument to `readFile` is a continuation, in that `readFile` invokes it to continue program

execution after it has done its job.

In order to rein in what can easily become callback hell, we use the continuations library.

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

Creating Functions That Take Continuations

If `shift` is called outside of a delimiting `reset` block, it can be used to create functions that themselves create continuations inside a `reset` block. It is important to note that `shift`'s type is not just `(((A => B) => C) => A)`, it is actually `(((A => B) => C) => (A @cpsParam[B, C]))`. That annotation marks where CPS transformations are needed. Functions that call `shift` without `reset` have their return type "infected" with that annotation.

Inside a `reset` block, a value of `A @cpsParam[B, C]` seems to have a value of `A`, though really it's just pretending. The continuation that is needed to complete the computation has type `A => B`, so the code following a method that returns this type must return `B`. `C` is the "real" return type, and after CPS transformation the function call has the type `C`.

Now, the example, taken from the Scaladoc of the library

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

Here, `ask` will store the continuation into a map, and later some other code can retrieve that "session" and pass in the result of the query to the user. In this way, `go` can actually be using an asynchronous library while its code looks like normal imperative code.

Read Continuations Library online: https://riptutorial.com/scala/topic/8312/continuations-library

Chapter 8: Currying

Syntax

- aFunction(10)_ //Using '_' Tells the compiler that all the parameters in the rest of the parameter groups will be curried.
- nArityFunction.curried //Converts an n-arity Function to an equivalent curried version
- anotherFunction(x)(_: String)(z) // Currying an arbitrary parameter. It needs its type explicitly stated.

Examples

A configurable multiplier as a curried function

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

Multiple parameter groups of different types, currying parameters of arbitrary positions

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

Currying a function with a single parameter group

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

Currying

Let's define a function of 2 arguments:

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

Currying `add` transforms it into a function that takes **one** `Int` and returns a **function** (from **one** `Int` to an `Int`)

```
val addCurried: (Int) => (Int => Int) = add2.curried
//                ^~~ take *one* Int
//                          ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

You can apply this concept to any function that takes multiple arguments. Currying a function that takes multiple arguments, transforms it into a series of applications of functions that take **one** argument:

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

Currying

Currying, according to Wikipedia,

> is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions.

Concretely, in terms of scala types, in the context of a function that take two arguments, (has arity 2) it is the conversion of

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

to

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

So for arity-2 functions we can write the curry function as:

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
```

```
}
```

Usage:

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scala gives us a few language features that help with this:

1. You can write curried functions as methods. so `curriedF` can be written as:

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. You can un-curry (i.e. go from `A => B => C` to `(A, B) => C`) using a standard library method: `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

When to use Currying

Currying *is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument*.

This is normally useful when for example:

1. different arguments of a function are calculated **at different times**. *(Example 1)*
2. different arguments of a function are calculated **by different tiers of the application**. *(Example 2)*

Example 1

Let's assume that the total yearly income is a function composed by the income and a bonus:

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

The curried version of the above 2-arity function is:

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

Note in the above definition that the type can be also viewed/written as:

```
Int => (Int => Int)
```

Let's assume that the yearly income portion is known in advance:

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

And at some point down the line the bonus is known:

```
partialTotalYearlyIncome(100)
```

Example 2

Let's assume that the car manufacturing involves the application of car wheels and car body:

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

These parts are applied by different factories:

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

Notice that the `CarWheelsFactory` above curries the car manufacturing function and only applies the wheels.

The car manufacturing process then will take the below form:

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

A real world use of Currying.

What we have is a list of credit cards and we'd like to calculate the premiums for all those cards that the credit card company has to pay out. The premiums themselves depend on the total number of credit cards, so that the company adjust them accordingly.

We already have a function that calculates the premium for a single credit card and takes into account the total cards the company has issued:

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
```

```
}
```

Now a reasonable approach to this problem would be to map each credit card to a premium and reduce it to a sum. Something like this:

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

However the compiler isn't going to like this, because `CreditCard.getPremium` requires two parameters. Partial application to the rescue! We can partially apply the total number of credit cards and use that function to map the credit cards to their premiums. All we need to do is curry the `getPremium` function by changing it to use multiple parameter lists and we're good to go.

The result should look something like this:

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Read Currying online: https://riptutorial.com/scala/topic/1636/currying

Chapter 9: Dependency Injection

Examples

Cake Pattern with inner implementation class.

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

In the above example, I demonstrated how to inject TimeUtil into MainController.

The most important syntax is the self-annotation (`_: TimeUtil =>`) which is to inject `TimeUtil` into `MainController`. In another word, `MainController` depends on `TimeUtil`.

I use inner class (e.g. `TimeUtilImpl`) in each component because, in my opinion, that it is easier for testing as we can mock the inner class. And it is also easier for tracing where the method is called from when project grows more complex.

Lastly, I wire all component together. If you are familiar with Guice, this is equivalent to `Binding`

Read Dependency Injection online: https://riptutorial.com/scala/topic/5909/dependency-injection

Chapter 10: Dynamic Invocation

Introduction

Scala allows you to use dynamic invocation when calling methods or accessing fields on an object. Instead of having this built deep into the language, this is accomplished through rewriting rules similar to those of implicit conversions, enabled by the marker trait [`scala.Dynamic`][Dynamic scaladoc]. This allows you to emulate the ability to dynamically add properties to objects present in dynamic languages, and more. [Dynamic scaladoc]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

Syntax

- class Foo extends Dynamic
- foo.field
- foo.field = value
- foo.method(args)
- foo.method(namedArg = x, y)

Remarks

In order to declare subtypes of `Dynamic`, the language feature `dynamics` must be enabled, either by importing `scala.language.dynamics` or by the `-language:dynamics` compiler option. Users of this `Dynamic` who do not define their own subtypes do not need to enable this.

Examples

Field Accesses

This:

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

allows for simple access to fields:

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
foo.unrealField // Becomes foo.selectDynamic(unrealField)
```

```
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

Method Calls

This:

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

allows for calls to methods, with and without named parameters:

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

Interaction Between Field Access and Update Method

Slightly counterintuitively (but also the only sane way to make it work), this:

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

is equivalent to:

```
dyn.selectDynamic("x").update(y, z)
```

while

```
dyn.x(y)
```

is still

```
dyn.applyDynamic("x")(y)
```

It is important to be aware of this, or else it may sneak by unnoticed and cause strange errors.

Read Dynamic Invocation online: https://riptutorial.com/scala/topic/8296/dynamic-invocation

Chapter 11: Enumerations

Remarks

Approach with `sealed trait` and `case objects` is preferred because Scala enumeration has a few problems:

1. Enumerations have the same type after erasure.
2. Compiler doesn't complain about “Match is not exhaustive", if case is missed it will fail in runtime `scala.MatchError`:

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

Compare with:

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

More detailed explanation is presented in this article about Scala Enumeration.

Examples

Days of the week using Scala Enumeration

Java-like enumerations can be created by extending Enumeration.

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

It is also possible to add a human-readable name for values in an enumeration:

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

Beware of the not-so-typesafe behavior, wherein different enumerations can evaluate as the same instance type:

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

Using sealed trait and case objects

An alternative to extending `Enumeration` is using `sealed` case objects:

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

The `sealed` keyword guarantees that the trait `WeekDay` cannot be extended in another file. This allows the compiler to make certain assumptions, including that all possible values of `WeekDay` are already enumerated.

One drawback is that this method does not allow you to obtain a list of all possible values. To get such a list it must be provided explicitly:

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

Case classes can also extend a `sealed` trait. Thus, objects and case classes can be mixed to create complex hierarchies:

```
sealed trait CelestialBody

object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

Another drawback is that there is no way to access a the variable name of a `sealed` object's enumeration, or search by it. If you need some kind of name associated to each value, it must be manually defined:

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

Or just:

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

Using sealed trait and case objects and allValues-macro

This is just an extension on the sealed trait variant where a macro generates a set with all instances at compile time. This nicely omits the drawback that a developer can add a value to the enumeration but forget to add it to the allElements set.

This variant especially becomes handy for large enums.

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
}
```

For this to work you need this macro:

```
import scala.collection.immutable.TreeSet
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

Read Enumerations online: https://riptutorial.com/scala/topic/1499/enumerations

Chapter 12: Error Handling

Examples

Try

Using Try with `map`, `getOrElse` and `flatMap`:

```
import scala.util.Try

val i = Try("123".toInt)      // Success(123)
i.map(_ + 1).getOrElse(321)   // 124

val j = Try("abc".toInt)      // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)   // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                             // Success(357)
```

Using `Try` with pattern matching:

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

Either

Different data types for error/success

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

Pattern matching on Either value

```
getPersonFromWebService("http://some-webservice.com/person") match {
    case Left(errorMessage) => println(errorMessage)
```

```
    case Right(person) => println(person.surname)
}
```

Convert Either value to Option

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

Option

The use of `null` values is strongly discouraged, unless interacting with legacy Java code that expects `null`. Instead, `Option` should be used when the result of a function might either be something (`Some`) or nothing (`None`).

A try-catch block is more appropriate for error-handling, but if the function might legitimately return nothing, `Option` is appropriate to use, and simple.

An `Option[T]` can either be `Some(value)` (contains a value of type `T`) or `None`:

```
def findPerson(name: String): Option[Person]
```

If no person is found, `None` can be returned. Otherwise, an object of type `Some` containing a `Person` object is returned. What follows are ways to handle an object of type `Option`.

Pattern Matching

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

Using map and getOrElse

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

Using fold

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

Converting to Java

If you need to convert an `Option` type to a null-able Java type for interoperability:

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

Handling Errors Originating in Futures

When an `exception` is thrown from within a `Future`, you can (should) use `recover` to handle it.

For instance,

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

...will throw an `Exception` from within the `Future`. But seeing as we can predict that an `Exception` of type `FairlyStupidException` with a high probability, we can specifically handle this case in an elegant way:

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

As you can see the method given to `recover` is a `PartialFunction` over the domain of all `Throwable`, so you can handle just a certain few types and then let the rest go into the ether of exception handling at higher levels in the `Future` stack.

Note that this is similar to running the following code in a non-`Future` context:

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

It is really important to handle exceptions generated within `Future`s because much of the time they are more insidious. They don't get all in your face usually, because they run in a different execution context and thread, and thus do not prompt you to fix them when they happen, especially if you don't notice anything in logs or the behavior of the application.

Using try-catch clauses

In addition to functional constructs such as `Try`, `Option` and `Either` for error handling, Scala also supports a syntax similar to Java's, using a try-catch clause (with a potential finally block as well). The catch clause is a pattern match:

```
try {
  // ... might throw exception
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

Convert Exceptions into Either or Option Types

To convert exceptions into `Either` or `Option` types, you can use methods that provided in `scala.util.control.Exception`

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

Read Error Handling online: https://riptutorial.com/scala/topic/910/error-handling

Chapter 13: Extractors

Syntax

- val extractor(extractedValue1, _ /* ignored second extracted value */) = valueToBeExtracted
- valueToBeExtracted match { case extractor(extractedValue1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- object Foo { def unapply(foo: Foo): Option[String] = Some(foo.x); }

Examples

Tuple Extractors

`x` and `y` are extracted from the tuple:

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

To ignore a value use _:

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

To unpack an extractor:

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

Note that tuples have a maximum length of 22, and thus `._1` through `._22` will work (assuming the tuple is at least that size).

Tuple extractors may be used to provide symbolic arguments for literal functions:

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}

assert {
  names ==
    (persons map { case (given, surname) =>
      s"$surname, $given"
```

```
    })
}
```

Case Class Extractors

A case class is a class with a lot of standard boilerplate code automatically included. One benefit of this is that Scala makes it easy to use extractors with case classes.

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

At this juncture, both `n` and `a` are `val`s in the program and can be accessed as such: they are said to have been 'extracted' from p. Continuing:

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

Here we see two important things:

- Extraction can happen at 'deep' levels: properties of nested objects can be extracted.
- Not all elements *need* to be extracted. The wildcard _ character indicates that that particular value can be anything, and is ignored. No `val` is created.

In particular, this can make matching over collections easy:

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

Here, we have code that uses the extractor to explicitly check that `person` is a `Person` object and immediately pull out the variables that we care about: `n` and `a`.

Unapply - Custom Extractors

A custom extraction can be written by implementing the `unapply` method and returning a value of type `Option`:

```
class Foo(val x: String)

object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}
```

```
new Foo("42") match {
  case Foo(x) => x
}
// "42"
```

The return type of `unapply` may be something other than `Option`, provided the type returned provides `get` and `isEmpty` methods. In this example, `Bar` is defined with those methods, and `unapply` returns an instance of `Bar`:

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

The return type of `unapply` can also be a `Boolean`, which is a special case that does not carry the `get` and `isEmpty` requirements above. However, note in this example that `DivisibleByTwo` is an object, not a class, and does not take a parameter (and therefore that parameter cannot be bound):

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

Remember that `unapply` goes in the companion object of a class, not in the class. The example above will be clear if you understand this distinction.

Extractor Infix notation

If a case class has exactly two values, its extractor can be used in infix notation.

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
```

```
//x: String = hello
//y: String = world
```

Any extractor that returns a 2-tuple can work this way.

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

Regex Extractors

A regular expression with grouped parts can be used as an extractor:

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

Note that when it is not matched, a `MatchError` will be thrown at runtime:

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

Transformative extractors

Extractor behavior can be used to derive arbitrary values from their input. This can be useful in scenarios where you want to be able to act on the results of a transformation in the event that the transformation is successful.

Consider as an example the various user name formats usable in a Windows environment:

```
object UserPrincipalName {
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
    case _ => None
  }
}

def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
```

```
  case _ => None
}
```

In fact it is possible to create an extractor exhibiting both behaviors by broadening the types it can match:

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

In general, extractors are simply a convenient reformulation of the `Option` pattern, as applied to methods with names like `tryParse`:

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

Read Extractors online: https://riptutorial.com/scala/topic/930/extractors

Chapter 14: For Expressions

Syntax

- for {clauses} body
- for {clauses} yield body
- for (clauses) body
- for (clauses) yield body

Parameters

Parameter	Details
for	Required keyword to use a for loop/comprehension
clauses	The iteration and filters over which the for works.
yield	Use this if you want to create or 'yield' a collection. Using `yield` will cause the return type of the `for` to be a collection instead of `Unit`.
body	The body of the for expression, executed on each iteration.

Examples

Basic For Loop

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

This demonstrates iterating a variable, `x`, from `1` to `10` and doing something with that value. The return type of this `for` comprehension is `Unit`.

Basic For Comprehension

This demonstrates a filter on a for-loop, and the use of `yield` to create a 'sequence comprehension':

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

The output for this is:

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

A for comprehension is useful when you need to create a new collection based on the iteration and it's filters.

Nested For Loop

This shows how you can iterate over multiple variables:

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

(Note that `to` here is an infix operator method that returns an inclusive range. See the definition here.)

This creates the output:

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

Note that this is an equivalent expression, using parentheses instead of brackets:

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

In order to get all of the combinations into a single vector, we can `yield` the result and set it to a `val`:

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

Monadic for comprehensions

If you have several objects of monadic types, we can achieve combinations of the values using a 'for comprehension':

```
for {
   x <- Option(1)
   y <- Option("b")
   z <- List(3, 4)
```

```
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}

// This prints
// (1, "b", 3)
// (1, "b", 4)
```

The return type of this block is `Unit`.

If the objects are of the *same* monadic type `M` (e.g. `Option`) then using `yield` will return an object of type `M` instead of `Unit`.

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

Note that the `yield` keyword *cannot* be used in the original example, where there is a mix of monadic types (`Option` and `List`). Trying to do so will yield a compile-time type mismatch error.

Iterate Through Collections Using a For Loop

This demonstrates how to print each element of a Map

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

This demonstrates how to print each element of a list

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

Desugaring For Comprehensions

`for` comprehensions in Scala are just syntactic sugar. These comprehensions are implemented using the `withFilter`, `foreach`, `flatMap` and `map` methods of their subject types. For this reason, only types that have these methods defined can be utilized in a `for` comprehension.

A `for` comprehension of the following form, with patterns `pN`, generators `gN` and conditions `cN`:

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... will de-sugar to nested calls using `withFilter` and `foreach`:

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

Whereas a `for`/`yield` expression of the following form:

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... will de-sugar to nested calls using `withFilter` and either `flatMap` or `map`:

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

(Note that `map` is used in the innermost comprehension, and `flatMap` is used in every outer comprehension.)

A `for` comprehension can be applied to any type implementing the methods required by the de-sugared representation. There are no restrictions on the return types of these methods, so long as they are composable.

Read For Expressions online: https://riptutorial.com/scala/topic/669/for-expressions

Chapter 15: Functions

Remarks

Scala has first-class functions.

Difference between functions and methods:

A function is not a method in Scala: functions are a value, and may be assigned as such. Methods (created using `def`), on the other hand, must belong to a class, trait or object.

- Functions are compiled to a class extending a trait (such as `Function1`) at compile-time, and are instantiated to a value at runtime. Methods, on the other hand, are members of their class, trait or object, and do not exist outside of that.
- A method may be converted to a function, but a function cannot be converted to a method.
- Methods can have type parameterization, whereas functions do not.
- Methods can have parameter default values, whereas functions can not.

Examples

Anonymous Functions

Anonymous functions are functions that are defined but not assigned a name.

The following is an anonymous function that takes in two integers and returns the sum.

```
(x: Int, y: Int) => x + y
```

The resultant expression can be assigned to a `val`:

```
val sum = (x: Int, y: Int) => x + y
```

Anonymous functions are primarily used as arguments to other functions. For instance, the `map` function on a collection expects another function as its argument:

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

The types of the arguments of the anonymous function can be omitted: the types are inferred automatically:

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

If there is just one argument, the parentheses around that argument can be omitted:

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

Underscores shorthand

There is an even shorter syntax that doesn't require names for the arguments. The above snippet can be written:

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_ represents the anonymous function arguments positionally. With an anonymous function that has multiple parameters, each occurrence of _ will refer to a different argument. For instance, the two following expressions are equivalent:

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

When using this shorthand, any argument represented by the positional _ can only be referenced a single time and in the same order.

Anonymous Functions with No Parameters

To create a value for an anonymous function that does not take parameters, leave the parameter list blank:

```
val sayHello = () => println("hello")
```

Composition

Function composition allows for two functions to operate and be viewed as a single function. Expressed in mathematical terms, given a function `f(x)` and a function `g(x)`, the function `h(x) = f(g(x))`.

When a function is compiled, it is compiled to a type related to `Function1`. Scala provides two methods in the `Function1` implementation related to composition: `andThen` and `compose`. The `compose` method fits with the above mathematical definition like so:

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

The `andThen` (think `h(x) = g(f(x))`) has a more 'DSL-like' feeling:

```
val f: A => B = ...
```

```
val g: B => C = ...

val h: A => C = f andThen g
```

A new anonymous function is allocated with that is closed over `f` and `g`. This function is bound to the new function `h` in both cases.

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

If either `f` or `g` works via a side-effect, then calling `h` will cause all side-effects of `f` and `g` to happen in the order. The same is true of any mutable state changes.

Relationship to PartialFunctions

```
trait PartialFunction[-A, +B] extends (A => B)
```

Every single-argument `PartialFunction` is also a `Function1`. This is counter-intuitive in a formal mathematical sense, but better fits object oriented design. For this reason `Function1` does not have to provide a constant `true isDefinedAt` method.

To define a partial function (which is also a function), use the following syntax:

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

For further details, take a look at PartialFunctions.

Read Functions online: https://riptutorial.com/scala/topic/477/functions

Chapter 16: Futures

Examples

Creating a Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

Quite simply, the `divide` method creates a Future that will resolve with the quotient of `a` over `b`.

Consuming a Successful Future

The easiest way to consume a *successful* Future-- or rather, get the value inside the Future-- is to use the `map` method. Suppose some code calls the `divide` method of the `FutureDivider` object from the "Creating a Future" example. What would the code need to look like to get the quotient of `a` over `b`?

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

Consuming a Failed Future

Sometimes the computation in a Future can create an exception, which will cause the Future to fail. In the "Creating a Future" example, what if the calling code passed `55` and `0` to the `divide` method? It'd throw an `ArithmeticException` after trying to divide by zero, of course. How would that be handled in consuming code? There are actually a handful of ways to deal with failures.

Handle the exception with `recover` and pattern matching.

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
```

```
        }
    }
}
```

Handle the exception with the `failed` projection, where the exception becomes the value of the Future:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Putting the Future Together

The previous examples demonstrated the individual features of a Future, handling success and failure cases. Usually, however, both features are handled much more tersely. Here's the example, written in a neater and more realistic way:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Sequencing and traversing Futures

In some cases it is necessary to calculate a variable amount of values on separate Futures. Assume to have a `List[Future[Int]]`, but instead a `List[Int]` needs to be processed. Then the question is how to turn this instance of `List[Future[Int]]` into a `Future[List[Int]]`. For this purpose there is the `sequence` method on the `Future` companion object.

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

In general `sequence` is a commonly known operator within the world of functional programming that transforms `F[G[T]]` into `G[F[T]]` with restrictions to `F` and `G`.

There is an alternate operator called `traverse`, which works similar but takes a function as an extra argument. With the identity function `x => x` as a parameter it behaves like the `sequence` operator.

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

However, the extra argument allows to modify each future instance inside the given `listOfFuture`. Furthermore, the first argument doesn't need to be a list of `Future`. Therefore it is possible to transform the example as follows:

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

In this case the `List(1,2,3)` is directly passed as first argument and the identity function `x => x` is replaced with the function `Future(_)` to similarly wrap each `Int` value into a `Future`. An advantage of this is that the intermediary `List[Future[Int]]` can be omitted to improve performance.

Combine Multiple Futures – For Comprehension

The *for comprehension* is a compact way to run a block of code that depends on the successful result of multiple futures.

With `f1, f2, f3` three `Future[String]`'s that will contain the strings `one, two, three` respectively,

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

`fCombined` will be a `Future[String]` containing the string `one - two - three` once all the futures have completed successfully.

Note that an implicit ExectionContext is assumed here.

Also, keep in mind that for comprehension is just a syntactic sugar for a flatMap method, so Future objects construction inside for body would eliminate concurrent execution of code-blocks enclosed by futures and lead to sequential code. You see it on example:

```
val result1 = for {
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
  }
} yield first - second

val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
```

```
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

Value enclosed by `result1` object would be always negative while `result2` would be positive.

For more details about the *for comprehension* and `yield` in general, see http://docs.scala-lang.org/tutorials/FAQ/yield.html

Read Futures online: https://riptutorial.com/scala/topic/3245/futures

Chapter 17: Handling units (measures)

Syntax

- class Meter(val meters: Double) extends AnyVal
- type Meter = Double

Remarks

It is recommended to use value classes for units or a dedicated library for them.

Examples

Type aliases

```
type Meter = Double
```

This simple approach has serious drawbacks for unit handling as every other type that is a `Double` will be compatible with it:

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

All of the above compiles, so in this case units can only be used for marking input/output types for the readers of the code (only the intent).

Value classes

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

Value classes provide a type-safe way to encode units, even if they require a bit more characters to use them:

```
var length = Meter(3)
var weight = Gram(4)
//length = weight //type mismatch; found : Gram required: Meter
```

By extending `AnyVal`s, there is no runtime penalty for using them, on the JVM level, those are regular primitive types (`Double`s in this case).

In case you want to automatically generate other units (like `Velocity` aka `MeterPerSecond`), this

approach is not the best, though there are libraries that can be used in those cases too:

- Squants
- units
- ScalaQuantity

Read Handling units (measures) online: https://riptutorial.com/scala/topic/5966/handling-units--measures-

Chapter 18: Higher Order Function

Remarks

Scala goes to great lengths to treat methods and functions as syntactically identical. But under the hood, they are distinct concepts.

A method is executable code, and has no value representation.

A function is an actual object instance of type `Function1` (or a similar type of another arity). Its code is contained in its `apply` method. Effectively, it simply acts as a value that can be passed around.

Incidentally, the ability to treat functions as values *is* exactly what is meant by a language having support for higher-order functions. Function instances are Scala's approach to implementing this feature.

An actual higher-order function is a function that either takes a function value as an argument or returns a function value. But in Scala, as all operations are methods, it's more general to think of methods that receive or return function parameters. So `map`, as defined on `Seq` might be thought of as a "higher-order function" due to its parameter being a function, but it is not literally a function; it is a method.

Examples

Using methods as function values

The Scala compiler will automatically convert methods into function values for the purpose of passing them into higher-order functions.

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

In the example above, `MyObject.mapMethod` is not a function call, but instead is passed to `map` as a value. Indeed, `map` *requires* a function value passed to it, as can be seen in it's signature. The signature for the `map` of a `List[A]` (a list of objects of type `A`) is:

```
def map[B](f: (A) ⇒ B): List[B]
```

The `f: (A) => B` part indicates that the parameter to this method call is some function that takes an object of type `A` and returns an object of type `B`. `A` and `B` are arbitrarily defined. Returning to the first example, we can see that `mapMethod` takes an `Int` (which corresponds to `A`) and returns a `String` (which corresponds to `B`). Thus `mapMethod` is a valid function value to pass to `map`. We could rewrite

the same code like this:

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

This inlines the function value, which may add clarity for simple functions.

High Order Functions(Function as Parameter)

A higher-order function, as opposed to a first-order function, can have one of three forms:

- One or more of its parameters is a function, and it returns some value.
- It returns a function, but none of its parameters is a function.
- Both of the above: One or more of its parameters is a function, and it returns a function.

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

Here the map function takes a `getFullName(n._1,n._2)` function as a parameter. This is called **HOF (Higher order function).**

Arguments lazy evaluation

Scala supports lazy evaluation for function arguments using notation: `def func(arg: => String)`. Such function argument might take regular `String` object or a higher order function with `String` return type. In second case, function argument would be evaluated on value access.

Please see the example:

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}

def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}
```

```
def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)

dumbMediator(preconditions = false, calculateData)
```

`smartMediator` call would return None value and print message `"Applying mediator"`.

`dumbMediator` call would return None value and print message `"Calculating expensive data! Applying mediator"`.

Lazy evaluation might be extremely useful when you want to optimize an overhead of expensive arguments calculation.

Read Higher Order Function online: https://riptutorial.com/scala/topic/1642/higher-order-function

Chapter 19: If Expressions

Examples

Basic If Expressions

In Scala (in contrast to Java and most other languages), `if` is an **expression** instead of a *statement*. Regardless, the syntax is identical:

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

The implication of `if` being an expression is that you can assign the result of the evalation of the expression to a variable:

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

Above we see that the `if` expression is evaluated and `result` is set to that resulting value.

The return type of an `if` expression is the **supertype** of all logic branches. This means that for this example the return type is a `String`. Since not all `if` expressions return a value (such as an `if` statement that has no `else` branch logic), it is possible that the return type is `Any`:

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

If no value can be returned (such as if only side effects like `println` are used inside the logical branches), then the return type will be `Unit`:

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` expressions in Scala are similar to how the ternary operator in Java functions. Because of this similarity, there is no such operator in Scala: it would be redundant.

> Curly braces can be omitted in an `if` expression if the content is a single line.

Read If Expressions online: https://riptutorial.com/scala/topic/4171/if-expressions

Chapter 20: Implicits

Syntax

- implicit val x: T = ???

Remarks

Implicit classes allow custom methods to be added to existing types, without having to modify their code, thereby enriching types without needing control of the code.

Using implicit types to enrich an existing class is often referred to as an 'enrich my library' pattern.

Restrictions on Implicit Classes

1. Implicit classes may only exist within another class, object, or trait.
2. Implicit classes may only have one non-implicit primary constructor parameter.
3. There may not be another object, class, trait, or class member definition within the same scope that has the same name as the implicit class.

Examples

Implicit Conversion

An implicit conversion allows the compiler to automatically convert an object of one type to another type. This allows the code to treat an object as an object of another type.

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

The conversion is one-way: in this case you cannot convert `42` back to `Foo(42)`. To do so, a second implicit conversion must be defined:

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

Note that this is the mechanism by which a float value can be added to an integer value, for instance.

> Implicit conversions should be used sparingly because they obfuscate what is

> happening. It is a best practice to use an explicit conversion via a method call unless there's a tangible readability gain from using an implicit conversion.

There is no significant performance impact of implicit conversions.

Scala automatically imports a variety of implicit conversions in `scala.Predef`, including all conversions from Java to Scala and back. These are included by default in any file compilation.

Implicit Parameters

Implicit parameters can be useful if a parameter of a type should be defined once in the scope and then applied to all functions that use a value of that type.

A normal function call looks something like this:

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

Now lets say we have some methods that all have a timeout duration, and we want to call all those methods using the same timeout. We can define timeout as an implicit variable.

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

The way this works is that the scalac compiler looks for a value in the scope which is **marked as implicit** and **whose type matches** the one of the implicit parameter. If it finds one, it will apply it as the implicit parameter.

> Note that this won't work if you define two or even more implicits of the same type in the scope.

To customize the error message, use the `implicitNotFound` annotation on the type:

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

A timeout is a usual use case for this, or for example in Akka the ActorSystem is (most of the times) always the same, so it's usually passed implicitly. Another use case would be library design, most commonly with FP libraries that rely on typeclasses (like scalaz, cats or rapture).

> It's generally considered bad practice to use implicit parameters with basic types like *Int*, *Long*, *String* etc. since it will create confusion and make the code less readable.

Implicit Classes

Implicit classes make it possible to add new methods to previously defined classes.

The `String` class has no method `withoutVowels`. This can be added like so:

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

The implicit class has a single constructor parameter (`str`) with the type that you would like to extend (`String`) and contains the method you would like to "add" to the type (`withoutVowels`). The newly defined methods can now be used directly on the enhanced type (when the enhanced type is in implicit scope):

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

Under the hood, implicit classes define an implicit conversion from the enhanced type to the implicit class, like this:

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

Implicit classes are often defined as Value classes to avoid creating runtime objects and thus removing the runtime overhead:

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
```

```
}
```

With the above improved definition, a new instance of `StringEnhancer` doesn't need to be created every time the `withoutVowels` method gets invoked.

Resolving Implicit Parameters Using 'implicitly'

Assuming an implicit parameter list with more than one implicit parameter:

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

Now, assuming that one of the implicit instances is not available (`SomeCtx1`) while all other implicit instances needed are in-scope, to create an instance of the class an instance of `SomeCtx1` must be provided.

This can be done while preserving each other in-scope implicit instance using the `implicitly` keyword:

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

Implicits in the REPL

To view all the `implicits` in-scope during a REPL session:

```
scala> :implicits
```

To also include implicit conversions defined in `Predef.scala`:

```
scala> :implicits -v
```

If one has an expression and wishes to view the effect of all rewrite rules that apply to it (including implicits):

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

(Example:

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
"Eve")(Predef.implicitly)).mkString(", "))
```

)

Read Implicits online: https://riptutorial.com/scala/topic/1732/implicits

Chapter 21: Java Interoperability

Examples

Converting Scala Collections to Java Collections and vice versa

When you need to pass a collection into a Java method:

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

If the Java code returns a Java collection, you can turn it into a Scala collection in a similar manner:

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

Note that these are decorators, so they merely wrap the underlying collections in a Scala or Java collection interface. Therefore, the calls `.asJava` and `.asScala` do not copy the collections.

Arrays

Arrays are regular JVM arrays with a twist that they are treated as invariant and have special constructors and implicit conversions. Construct them without the `new` keyword.

```
val a = Array("element")
```

Now `a` has type `Array[String]`.

```
val acs: Array[CharSequence] = a
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

Although `String` is convertible to `CharSequence`, `Array[String]` is not convertible to `Array[CharSequence]`.

You can use an `Array` like other collections, thanks to an implicit conversion to `TraversableLike` `ArrayOps`:

```
val b: Array[Int] = a.map(_.length)
```

Most of the Scala collections (`TraversableOnce`) have a `toArray` method taking an implicit `ClassTag` to construct the result array:

```
List(0).toArray
//> res1: Array[Int] = Array(0)
```

This makes it easy to use any `TraversableOnce` in your Scala code and then pass it to Java code which expects an array.

Scala and Java type conversions

Scala offers implicit conversions between all the major collection types in the JavaConverters object.

The following type conversions are bidirectional.

Scala Type	Java Type
Iterator	java.util.Iterator
Iterator	java.util.Enumeration
Iterator	java.util.Iterable
Iterator	java.util.Collection
mutable.Buffer	java.util.List
mutable.Set	java.util.Set
mutable.Map	java.util.Map
mutable.ConcurrentMap	java.util.concurrent.ConcurrentMap

Certain other Scala collections can also be converted to Java, but do not have a conversion back to the original Scala type:

Scala Type	Java Type
Seq	java.util.List
mutable.Seq	java.util.List
Set	java.util.Set
Map	java.util.Map

Reference:

Conversions Between Java and Scala Collections

Functional Interfaces for Scala functions - scala-java8-compat

A Java 8 compatibility kit for Scala.

Most examples are copied from Readme

Converters between scala.FunctionN and java.util.function

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

Converters between scala.Option and java.util classes Optional, OptionalDouble, OptionalInt, and OptionalLong.

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

Converters from Scala collections to Java 8 Streams

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
    a.parStream.count // 8 + 2 = 10

  val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
  // LongAccumulator
  val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

Read Java Interoperability online: https://riptutorial.com/scala/topic/2441/java-interoperability

Chapter 22: JSON

Examples

JSON with spray-json

spray-json provides an easy way to work with JSON. Using implicit formats, everything happens "behind the scenes":

Make the Library Available with SBT

To manage `spray-json` with SBT managed library dependencies:

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Note that the last parameter, the version number (`1.3.2`), may be different in different projects.

The `spray-json` library is hosted at repo.spray.io.

Import the Library

```
import spray.json._
import DefaultJsonProtocol._
```

The default JSON protocol `DefaultJsonProtocol` contains formats for all basic types. To provide JSON functionality for custom types, either use convenience builders for formats or write formats explicitly.

Read JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

Write JSON

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSL is not supported.

Read-Write to Case Classes

The following example shows how to serialize a case class object into the JSON format.

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

This results in the following JSON:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

That JSON can, in turn, be deserialized back into an object:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

Custom Format

Write a custom `JsonFormat` if a special serialization of a type is required. For example, if the field names are different in Scala than in JSON. Or, if different concrete types are instantiated based on the input.

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
            address = fields("home").convertTo[Address]
        )
    }
```

```
    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

JSON with Circe

Circe provides compile-time derived codecs for en/decode json into case classes. A simple example looks like this:

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

JSON with play-json

play-json uses implicit formats as other json frameworks

SBT dependency: `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat` contains defaul formats to read/write all basic types. To provide JSON functionality for your own types, you can either use convenience builders for formats or write formats explicitly.

Read json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

Write json

```
val values = List("a", "b", "c")
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

As always prefer pattern matching against `JsSuccess`/`JsError` and try to avoid `.get`, `array(i)` calls.

Read and write to case class

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

Own Format

You can write your own JsonFormat if you require a special serialization of your type (e.g. name the fields differently in scala and Json or instantiate different concrete types based on the input)

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
//Address(Awesome Street 9,SuperCity)
```

Alternative

If the json doesn't exactly match your case class fields (`isAlive` in case class vs `is_alive` in json):

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

Json with optional fields

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

Reading timestamps from json

Imagine you have a Json object, with a Unix timestamp field:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

solution:

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

Reading custom case classes

Now, if you do wrap your object identifiers for type safety, you will enjoy this. See the following json object:

```
{
  "id": 91,
  "data": "Some data"
}
```

and the corresponding case classes:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Now you just need to read the primitive type (Long), and map to your idenfier:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

code at https://github.com/pedrorijo91/scala-play-json-examples

JSON with json4s

json4s uses implicit formats as other json frameworks.

SBT dependency:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

Imports

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats` contains default formats to read/write all basic types.

Read json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

Write json

```
val values = List("a", "b", "c")
```

```
compact(render(values))         // ["a", "b", "c"]
```

DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

Read and write to case class

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

Read and Write heterogenous lists

To serialize and deserialize an heterogenous (or polymorphic) list, specific type-hints need to be provided.

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

Own Format

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
```

```
// Address(Awesome Stree,Super City)
```

Read JSON online: https://riptutorial.com/scala/topic/2348/json

Chapter 23: Macros

Introduction

Macros are a form of compile time metaprogramming. Certain elements of Scala code, such as annotations and methods, can be made to transform other code when they are compiled. Macros are ordinary Scala code that operate on data types that represent other code. The [Macro Paradise][] plugin extends the abilities of macros beyond the base language. [Macro Paradise]: http://docs.scala-lang.org/overviews/macros/paradise.html

Syntax

- def x() = macro x_impl // x is a macro, where x_impl is used to transform code
- def macroTransform(annottees: Any*): Any = macro impl // Use in annotations to make them macros

Remarks

Macros are a language feature that need to be enabled, either by importing `scala.language.macros` or with the compiler option `-language:macros`. Only macro definitions require this; code that uses macros need not do it.

Examples

Macro Annotation

This simple macro annotation outputs the annotated item as-is.

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

The `@compileTimeOnly` annotation generates an error with a message indicating that the `paradise` compiler plugin must be included to use this macro. Instructions to include this via SBT are here.

You can use the above-defined macro like this:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

Method Macros

When a method is defined to be a macro, the compiler takes the code that is passed as its argument and turns it into an AST. It then invokes the macro implementation with that AST, and it returns a new AST that is then spliced back to its call site.

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                      => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

It is also possible to have macros that take `Tree`s as arguments. Like how `reify` is used to create `Expr`s, the `q` (for quasiquote) string interpolator lets us create and deconstruct `Tree`s. Note that we could have used `q` above (`expr.tree` is, surprise, a `Tree` itself) too, but didn't for demonstrative purposes.

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

Errors in Macros

Macros can trigger compiler warnings and errors through the use of their `Context`.

Say we're a particularly overzealous when it comes to bad code, and we want to mark every instance of technical debt with a compiler info message (let's not think about how bad this idea is). We can use a macro that does nothing except emit such a message.

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

Additionally, instead of using `???` to mark unimplemented code, we can create two macros, `!!!` and `?!?`, that serve the same purpose, but emit compiler warnings. `?!?` will cause a warning to be issued, and `!!!` will cause an outright error.

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}
```

```
def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

Read Macros online: https://riptutorial.com/scala/topic/3808/macros

Chapter 24: Monads

Examples

Monad Definition

Informally, a monad is a container of elements, notated as `F[_]`, packed with 2 functions: `flatMap` (to transform this container) and `unit` (to create this container).

Common library examples include `List[T]`, `Set[T]` and `Option[T]`.

Formal definition

Monad `M` is a parametric type `M[T]` with two operations `flatMap` and `unit`, such as:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

These functions must satisfy three laws:

1. *Associativity*: `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   That is, if the sequence is unchanged you may apply the terms in any order. Thus, applying `m` to `f`, and then applying the result to `g` will yield the same result as applying `f` to `g`, and then applying `m` to that result.
2. *Left unit*: `unit(x) flatMap f == f(x)`
   That is, the unit monad of `x` flat-mapped across `f` is equivalent to applying `f` to `x`.
3. *Right unit*: `m flatMap unit == m`
   This is an 'identity': any monad flat-mapped against unit will return a monad equivalent to itself.

Example:

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Associativity*:

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Left unit*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Right unit*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

Standard Collections are Monads

Most of the standard collections are monads (`List[T]`, `Option[T]`), or monad-like (`Either[T]`, `Future[T]`). These collections can be easily combined together within `for` comprehensions (which are an equivalent way of writing `flatMap` transformations):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

The above is equivalent to:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Because a monad preserves the *data structure* and only acts on the elements within that structure, we can endless chain monadic datastructures, as shown here in a for-comprehension.

Read Monads online: https://riptutorial.com/scala/topic/4112/monads

Chapter 25: Operator Overloading

Examples

Defining Custom Infix Operators

In Scala operators (such as `+`, `-`, `*`, `++`, etc.) are just methods. For instance, `1 + 2` can be written as `1.+(2)`. These sorts of methods are called *'infix operators'*.

This means custom methods can be defined on your own types, reusing these operators:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

These operators defined-as-methods can be used like so:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

Note that infix operators can only have a single argument; the object before the operator will call it's own operator on the object after the operator. Any Scala method with a single argument can be used as an infix operator.

> This should be used with parcimony. It is generally considered good practice only if your own method does exactly what one would expect from that operator. In case of doubt, use a more conservative naming, like `add` instead of `+`.

Defining Custom Unary Operators

Unary operators can be defined by prepending the operator with `unary_`. Unary operators are limited to `unary_+`, `unary_-`, `unary_!` and `unary_~`:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }

  def unary_- = {
```

```
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

The unary operator can be used as follows:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> This should be used with parcimony. Overloading a unary operator with a definition that is not what one would expect can lead to code confusion.

Read Operator Overloading online: https://riptutorial.com/scala/topic/2271/operator-overloading

Chapter 26: Operators in Scala

Examples

Built-in Operators

Scala has the following built-in operators (methods/language elements with predefined precedence rules):

Type	Symbol	Example
Arithmetic operators	`+ - * / %`	`a + b`
Relational operators	`== != > < >= <=`	`a > b`
Logical operators	`&& & || | !`	`a && b`
Bit-wise operators	`& | ^ ~ << >> >>>`	`a & b`, `~a`, `a >>> b`
Assignment operators	`= += -= *= /= %= <<= >>= &= ^= |=`	`a += b`

Scala operators have the same meaning as in Java

Note: methods ending with `:` bind to the right (and right associative), so the call with `list.::(value)` can be written as `value :: list` with operator syntax. (`1 :: 2 :: 3 :: Nil` is the same as `1 :: (2 :: (3 :: Nil))`)

Operator Overloading

In Scala you can define your own operators:

```
class Team {
   def +(member: Person) = ...
}
```

With the above defines you can use it like:

```
ITTeam + Jack
```

or

```
ITTeam.+(Jack)
```

To define unary operators you can prefix it with `unary_`. E.g. `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

Operator Precedence

Category	Operator	Associativity
Postfix	`() []`	Left to right
Unary	`! ~`	Right to left
Multiplicative	`* / %`	Left to right
Additive	`+ -`	Left to right
Shift	`>> >>> <<`	Left to right
Relational	`> >= < <=`	Left to right
Equality	`== !=`	Left to right
Bitwise and	`&`	Left to right
Bitwise xor	`^`	Left to right
Bitwise or	`|`	Left to right
Logical and	`&&`	Left to right
Logical or	`||`	Left to right
Assignment	`= += -= *= /= %= >>= <<= &= ^= |=`	Right to left
Comma	`,`	Left to right

Programming in Scala gives the following outline based on the 1st character in the operator. E.g. `>` is the 1st character in the operator `>>>`:

Operator
(all other special characters)
`* / %`
`+ -`
`:`

Operator
= !
< >
&
^
|
(all letters)
(all assignment operators)

The one exception to this rule concerns *assignment operators*, e.g. `+=`, `*=`, etc. If an operator ends with an equal character (=) and is not one of the comparison operators `<=`, `>=`, `==` or `!=`, then the precedence of the operator is the same as simple assignment. In other words, lower than that of any other operator.

Read Operators in Scala online: https://riptutorial.com/scala/topic/6604/operators-in-scala

Chapter 27: Option Class

Syntax

- class Some[+T](value: T) extends Option[T]
- object None extends Option[Nothing]
- Option[T](value: T)

  Constructor to create either a `Some(value)` or `None` as appropriate for the value provided.

Examples

Options as Collections

`Option`s have some useful higher-order functions that can be easily understood by viewing options as *collections with zero or one items* - where `None` behaves like the empty collection, and `Some(x)` behaves like a collection with a single item, `x`.

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

Using Option Instead of Null

In Java (and other languages), using `null` is a common way of indicating that there is no value attached to a reference variable. In Scala, using `Option` is preferred over using `null`. `Option` wraps values that *might* be `null`.

`None` is a subclass of `Option` wrapping a null reference. `Some` is a subclass of `Option` wrapping a non-null reference.

Wrapping a reference is easy:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

This is typical code when calling a Java library that might return a null reference:

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
// else resource = Some(resource)
```

If `getResource()` returns a `null` value, `resource` will be a `None` object. Otherwise it will be a `Some(resource)` object. The preferred way to handle an `Option` is using higher order functions available within the `Option` type. For example if you want to check if your value is not `None` (similar to checking if `value == null`), you would use the `isDefined` function:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

Similarly, to check for a `null` reference you can do this:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

It is preferred that you treat conditional execution on the wrapped value of an `Option` (without using the 'exceptional' `Option.get` method) by treating the `Option` as a monad and using `foreach`:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

If a `Resource` instance is required (versus an `Option[Resource]` instance), you can still use `Option` to protect against null values. Here the `getOrElse` method provides a default value:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

Java code won't readily handle Scala's `Option`, so when passing values to Java code it is good form to unwrap an `Option`, passing `null` or a sensible default where appropriate:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

Basics

An `Option` is a data structure that contains either a single value, or no value at all. An `Option` can be thought of as collections of zero or one elements.

Option is an abstract class with two children: `Some` and `None`.

`Some` contains a single value, and `None` contains no value.

`Option` is useful in expressions that would otherwise use `null` to represent the lack of a concrete

value. This protects against a `NullPointerException`, and allows the composition of many expressions that might not return a value using combinators such as `Map`, `FlatMap`, etc.

Example with Map

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]` is **sealed** and thus cannot be extended. Therefore it's semantics are stable and can be relied on.

Options in for comprehensions

`Option`s have a `flatMap` method. This means they can be used in a for comprehension. In this way we can lift regular functions to work on `Option`s without having to redefine them.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

When one of the values is a `None` the ending result of the calculation will be `None` as well.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = None
```

Note: this pattern extends more generally for concepts called `Monad`s. (More information should be available on pages relating to for comprehensions and `Monad`s)

In general it is not possible to mix different monads in a for comprehension. But since `Option` can be easily converted to an `Iterable`, we can easily mix `Option`s and `Iterable`s by calling the

`.toIterable` method.

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

A small note: if we had defined our for comprehension the other way around the for comprehension would compile since our option would be converted implicitly. For that reason it is useful to always add this `.toIterable` (or corresponding function depending on which collection you are using) for consistency.

Read Option Class online: https://riptutorial.com/scala/topic/2293/option-class

Chapter 28: Packages

Introduction

Packages in Scala manage namespaces in large programs. For example, the name `connection` can occur in the packages `com.sql` and `org.http`. You can use the fully qualified `com.sql.connection` and `org.http.connection`, respectively, in order to access each of these packages.

Examples

Package structure

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

Packages and files

The package clause is not directly binded with the file where it is found. It is possible to find common elements of the package clause in diferent files. For example, the package clauses bellow can be found in the file math1.scala and in the file math2.scala.

File math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

File math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

File study.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

Package naming convension

Scala packages should follow the Java package naming conventions.
Package names are written in all lower case to avoid conflict with the names of classes or interfaces. Companies use their reversed Internet domain name to begin their package names—for example,

```
io.super.math
```

Read Packages online: https://riptutorial.com/scala/topic/8231/packages

Chapter 29: Parallel Collections

Remarks

Parallel collections facilitate parallel programming by hiding low-level parallelization details. This makes taking advantage of multi-core architectures easy. Examples of parallel collections include `ParArray`, `ParVector`, `mutable.ParHashMap`, `immutable.ParHashMap`, and `ParRange`. A full list can be found in the documentation.

Examples

Creating and Using Parallel Collections

To create a parallel collection from a sequential collection, call the `par` method. To create a sequential collection from a parallel collection, call the `seq` method. This example shows how you turn a regular `Vector` into a `ParVector`, and then back again:

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

The `par` method can be chained, allowing you to convert a sequential collection to a parallel collection and immediately perform an action on it:

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

In these examples, the work is actually parceled out to multiple processing units, and then re-joined after the work is complete - without requiring developer intervention.

Pitfalls

Do not use parallel collections when the collection elements must be received in a specific order.

Parallel collections perform operations concurrently. That means that all of the work is divided into parts and distributed to different processors. Each processor is unaware of the work being done by others. If the *order of the collection* matters then work processed in parallel is nondeterministic. (Running the same code twice can yield different results.)

Non-associative Operations

If an operation is non-associative (if the order of execution matters), then the result on a parallelized collection will be nondeterministic.

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

Side Effects

Operations that have side effects, such as `foreach`, may not execute as desired on parallelized collections due to race conditions. Avoid this by using functions that have no side effects, such as `reduce` or `map`.

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity

scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

Read Parallel Collections online: https://riptutorial.com/scala/topic/3882/parallel-collections

Chapter 30: Parser Combinators

Remarks

ParseResult Cases

A `ParseResult` comes in three flavors:

- Success, with a marker as to the start of the match and the next character to be matched.
- Failure, with a marker as to the start of where the match was attempted. In this case the parser backtracks to that position, where it will be when parsing continues.
- Error, which stops the parsing. No backtracking or further parsing occurs.

Examples

Basic Example

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

The output will be as follows:

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`[1.6]` in the `Alice` example indicates that the start of the match is at position `1`, and the fist character remaining to match starts at position `6`.

Read Parser Combinators online: https://riptutorial.com/scala/topic/3730/parser-combinators

Chapter 31: Partial Functions

Examples

Composition

Partial functions are often used to define a total function in parts:

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

In this usage, the partial functions are attempted in order of concatenation with the `orElse` method. Typically, a final partial function is provided that matches all remaining cases. Collectively, the combination of these functions acts as a total function.

This pattern is typically used to separate concerns where a function may effectively act a dispatcher for disparate code paths. This is common, for example, in the receive method of an Akka Actor.

Usage with `collect`

While partial function are often used as convenient syntax for total functions, by including a final wildcard match (`case _`), in some methods, their partiality is key. One very common example in idiomatic Scala is the `collect` method, defined in the Scala collections library. Here, partial functions allow the common functions of examining the elements of a collection to map and/or filter them to occur in one compact syntax.

Example 1

Assuming that we have a square root function defined as partial function:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

We can invoke it with the `collect` combinator:

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

effectively performing the same operation as:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

Example 2

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

There are several things to note in the example above:

- The left-hand side of each pattern match effectively selects elements to process and include in the output. Any value that doesn't have a matching `case` is simply omitted.
- The right-hand side defines the case-specific processing to apply.
- Pattern matching binds variable for use in guard statements (the `if` clauses) and the right-hand side.

Basic syntax

Scala has a special type of function called a partial function, which extends normal functions -- meaning that a `PartialFunction` instance can be used wherever `Function1` is expected. Partial functions can be defined anonymously using `case` syntax also used in pattern matching:

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7

pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

As seen in the example, a partial function need not be defined over the whole domain of its first parameter. A standard `Function1` instance is assumed to be *total*, meaning that it is defined for every possible argument.

Usage as a total function

Partial functions are very common in idiomatic Scala. They are often used for their convenient `case` -based syntax to define total functions over traits:

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

This saves the additional syntax of a `match` statement in a regular anonymous function. Compare:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

It is also frequently used to perform a parameter decomposition using pattern matching, when a tuple or a case class is passed to a function:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

Usage to extract tuples in a map function

These three map functions are equivalent, so use the variation that your team finds most readable.

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

The partial function **must match all input**: any case which doesn't match will throw an exception at runtime.

Read Partial Functions online: https://riptutorial.com/scala/topic/1638/partial-functions

Chapter 32: Pattern Matching

Syntax

- selector match partialFunction
- selector match {list of case alternatives) // This is most common form of the above

Parameters

Parameter	Details
selector	The expression whose value is being pattern-matched.
alternatives	a list of `case`-delimited alternatives.

Examples

Simple Pattern Match

This example shows how to match an input against several values:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Live demo

Note: _ is the *fall through* or *default* case, but it is not required.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

To avoid throwing an exception, it is a best functional-programming practice here to handle the default case (`case _ => <do something>`). Note that matching over *a case class* can help the compiler produce a warning if a case is missing. The same goes for user-defined types which extend a sealed trait. If the match is total then a default case may not be needed

It is also possible to match against values that are not defined inline. These must be *stable identifiers*, which are obtained by either using a capitalized name or enclosing backticks.

With `One`and `two` defined somewhere else, or passed as function parameters:

```
val One: Int = 1
val two: Int = 2
```

They can be matched against in the following way:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

Unlike other programming languages as Java for example there is no fall through. If a case block matches an input, it gets executed and the matching is finished. Therefore the least specific case should be the last case block.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

Pattern Matching With Stable Identifier

In standard pattern matching, the identifier used will shadow any identifier in the enclosing scope. Sometimes it is necessary to match on the enclosing scope's variable.

The following example function takes a character and a list of tuples and returns a new list of tuples. If the character existed as the first element in one of the tuples, the second element is incremented. If it does not yet exist in the list, a new tuple is created.

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

The above demonstrates pattern matching where the method's input, `char`, is kept 'stable' in the pattern match: that is, if you call `tabulate('x', ...)`, the first case statement would be interpreted as:

```
case('x', count) => ...
```

Scala will interpret any variable demarcated with a tick mark as a stable identifier: it will also interpret any variable that starts with a capital letter in the same way.

Pattern Matching on a Seq

To check for a precise number of elements in the collection

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

Live demo

To extract the first(s) element(s) and keeping the rest as a collection:

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

In general, any form that can be used to construct a sequence can be used to pattern match against an existing sequence.

Note that while using `Nil` and `::` will work when pattern matching a Sequence, it does convert it to a `List`, and can have unexpected results. Constrain yourself to `Seq( ...)` and `+:` to avoid this.

Note that while using `::` will not work for `WrappedArray`, `Vector` etc, see:

```
scala> def f(ints:Seq[Int]) = ints match {
     | case h :: t => h
     | case _ => "No match"
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

And with `+:`:

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

Guards (if expressions)

Case statements can be combined with if expressions to provide extra logic when pattern matching.

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

It is important to ensure your guards do not create a non-exhaustive match (the compiler often will not catch this):

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

This throws a `MatchError` on odd numbers. You must either account for all cases, or use a wildcard match case:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

Pattern Matching with Case Classes

Every case class defines an extractor that can be used to capture the members of the case class when pattern matching:

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

All the normal rules of pattern-matching apply - you can use guards and constant expressions to control matching:

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

Matching on an Option

If you are matching on an Option type:

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

This is functionally equivalent to using `fold`, or `map`/`getOrElse`:

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

Pattern Matching Sealed Traits

When pattern matching an object whose type is a sealed trait, Scala will check at compile-time that all cases are 'exhaustively matched':

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
}
```

If a new `case class` for `Shape` is later added, all `match` statements on `Shape` will start to throw a compiler warning. This makes thorough refactoring easier: the compiler will alert the developer to all code that needs to be updated.

Pattern Matching with Regex

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
  case _ => println(s"This is not a valid email.")
}
```

In this example, the regex attempts to match the email address provided. If it does, the `userName` and `domain` is extracted and printed. `topDomain` is also extracted, but nothing is done with it in this example. Calling `.r` on a String `str` is equivalent to `new Regex(str)`. The `r` function is available via an implicit conversion.

Pattern binder (@)

The `@` sign binds a variable to a name during a pattern match. The bound variable can either be the entire matched object or part of the matched object:

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

The bound identifier can be used in conditional filters. Thus:

```
case Circle(r) if r > 9 => s"large circle"
```

can be written as:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

The name can be bound to only a part of the matched pattern:

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

Pattern Matching Types

Pattern matching can also be used to check the type of an instance, rather than using `isInstanceOf[B]`:

```
val anyRef: AnyRef = ""

anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

The order of the cases is important:

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

In this manner it is similar to a classical 'switch' statement, without the fall-through functionality. However, you can also pattern match and 'extract' values from the type in question. For instance:

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

Note that in the `Foo` and `Woo` case we use the underscore (_) to 'match an unbound variable'. That is to say that the value (in this case `Hadas` and `27`, respectively) is not bound to a name and thus is not available in the handler for that case. This is useful shorthand in order to match 'any' value without worrying about what that value is.

Pattern Matching compiled as tableswitch or lookupswitch

The `@switch` annotation tells the compiler that the `match` statement can be replaced with a single `tableswitch` instruction at the bytecode level. This is a minor optimization that can remove

unnecessary comparisons and variable loads during runtime.

The `@switch` annotation works only for matches against literal constants and `final val` identifiers. If the pattern match cannot be compiled as a `tableswitch`/`lookupswitch`, the compiler will raise a warning.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

The results are the same as a normal pattern match:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

From the Scala Documentation (2.8+) – `@switch`:

> An annotation to be applied to a match expression. If present, the compiler will verify that the match has been compiled to a tableswitch or lookupswitch, and issue an error if it instead compiles into a series of conditional expressions.

From the Java Specification:

- tableswitch: "Access jump table by index and jump"
- lookupswitch: "Access jump table by key match and jump"

Matching Multiple Patterns At Once

The `|` can be used to have a single case statement match against multiple inputs to yield the same result:

```
def f(str: String): String = str match {
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
f("fubar")  // res2: String = No match.
```

Note that while matching **values** this way works well, the following matching of **types** will cause problems:

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

If in the latter case (with _) you don't need the value of the unbound variable and just want to do something else, you're fine:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

Otherwise, you are left with splitting your cases:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

Pattern Matching on tuples

Given the following `List` of tuples:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

Pattern matching can be used to handle each element differently:

```
pastries foreach { pastry =>
  pastry match {
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

The first case shows how to match against a specific string and get the corresponding price. The second case shows a use of if and tuple extraction to match against elements of the tuple.

Read Pattern Matching online: https://riptutorial.com/scala/topic/661/pattern-matching

Chapter 33: Quasiquotes

Examples

Create a syntax tree with quasiquotes

Use quasiquotes to create a `Tree` in a macro.

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

It can be arbitrarily complex but it will be validated for correct scala syntax.

Read Quasiquotes online: https://riptutorial.com/scala/topic/4032/quasiquotes

Chapter 34: Recursion

Examples

Tail Recursion

Using regular recursion, each recursive call pushes another entry onto the call stack. When the recursion is completed, the application has to pop each entry off all the way back down. If there are much recursive function calls it can end up with a huge stack.

Scala automatically removes the recursion in case it finds the recursive call in tail position. The annotation (`@tailrec`) can be added to recursive functions to ensure that tail call optimization is performed. The compiler then shows an error message if it can't optimize your recursion.

Regular Recursion

This example is not tail recursive because when the recursive call is made, the function needs to keep track of the multiplication it needs to do with the result after the call returns.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

The function call with the parameter will result in a stack that looks like this:

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

If we try to annotate this example with `@tailrec` we will get the following error message: `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

Tail Recursion

In tail recursion, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step.

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

In contrast, the stack trace for the tail recursive factorial looks like the following:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

There is only the need to keep track of the same amount of data for every call to `fact_inside` because the function is simply returning the value it got right through to the top. This means that even if `fact_with_tail 1000000` is called, it needs only the same amount of space as `fact_with_tail 3`. This is not the case with the non-tail-recursive call, and as such large values may cause a stack overflow.

Stackless recursion with trampoline(scala.util.control.TailCalls)

It is very common to get a `StackOverflowError` error while calling recursive function. Scala standard library offers TailCall to avoid stack overflow by using heap objects and continuations to store the local state of the recursion.

Two examples from the scaladoc of TailCalls

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)

// What is the 40th entry of the Fibonacci series?
fib(40).result
```

Read Recursion online: https://riptutorial.com/scala/topic/3889/recursion

Chapter 35: Reflection

Examples

Loading a class using reflection

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Read Reflection online: https://riptutorial.com/scala/topic/5824/reflection

Chapter 36: Regular Expressions

Syntax

- re.findAllIn(s: CharSequence): MatchIterator
- re.findAllMatchIn(s: CharSequence): Iterator[Match]
- re.findFirstIn(s: CharSequence): Option[String]
- re.findFirstMatchIn(s: CharSequence): Option[Match]
- re.findPrefixMatchIn(s: CharSequence): Option[Match]
- re.findPrefixOf(s: CharSequence): Option[String]
- re.replaceAllIn(s: CharSequence, replacer: Match => String): String
- re.replaceAllIn(s: CharSequence, replacement: String): String
- re.replaceFirstIn(s: CharSequence, replacement: String): String
- re.replaceSomeIn(s: CharSequence, replacer: Match => Option[String]): String
- re.split(s: CharSequence): Array[String]

Examples

Declaring regular expressions

The `r` method implicitly provided via scala.collection.immutable.StringOps produces an instance of scala.util.matching.Regex from the subject string. Scala's triple-quoted string syntax is useful here, as you do not have to escape backslashes as you would in Java:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex implements an idiomatic regular expression API for Scala as a wrapper over java.util.regex.Pattern, and the supported syntax is the same. That being said, Scala's support for multi-line string literals makes the `x` flag substantially more useful, enabling comments and ignoring pattern whitespace:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

There is an overloaded version of `r`, `def r(names: String*): Regex` which allows you to assign group names to your pattern captures. This is somewhat brittle as the names are disassociated from the captures, and should only be used if the regular expression will be used in multiple locations:

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
  case Some(matched) =>
    val y = matched.group("y").toInt
```

```
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
}
```

Repeating matching of a pattern in a string

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

Read Regular Expressions online: https://riptutorial.com/scala/topic/2891/regular-expressions

Chapter 37: Scala.js

Introduction

`Scala.js` is a port from `Scala` that compiles to `JavaScript`, which at the end will be running outside the `JVM`. It has benefits as strong typing, code optimization at compile time, full interoperability with JavaScript libraries.

Examples

console.log in Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

Fat arrow functions

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

Simple Class

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

Collections

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

Manipulating DOM

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
```

```
  target.appendChild(pNode)
}
```

Using with SBT

Sbt dependency

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

Running

```
sbt run
```

Running with continous compilation:

```
sbt ~run
```

Compile to a single JavaScript file:

```
sbt fastOptJS
```

Read Scala.js online: https://riptutorial.com/scala/topic/9426/scala-js

Chapter 38: Scaladoc

Syntax

- Goes above methods, fields, classes or packages.
- Starts with `/**`
- Each line has an starting `*` proceding with the comments
- Ends with `*/`

Parameters

Parameter	Details
Class specific	_
`@constructor detail`	Explains the main constructor of the class
Method specific	_
`@return detail`	Details about what is returned on the method.
Method, Constructor and/or Class tags	_
`@param x detail`	Details about the value parameter `x` on a method or constructor.
`@tparam x detail`	Details about the type parameter `x` on a method or constructor.
`@throws detail`	What exceptions may be thrown.
Usage	_
`@see detail`	References other sources of information.
`@note detail`	Adds a note for pre or post conditions, or any other notable restrictions or expectations.
`@example detail`	Provides example code or related example documentation.
`@usecase detail`	Provides a simplified method definition for when the full method definition is too complex or noisy.
Other	_
`@author detail`	Provides information about the author of the following.

Parameter	Details
`@version detail`	Provides the version that this portion belongs to.
`@deprecated detail`	Marks the following entity as deprecated.

Examples

Simple Scaladoc to method

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

Read Scaladoc online: https://riptutorial.com/scala/topic/4518/scaladoc

Chapter 39: scalaz

Introduction

Scalaz is a Scala library for functional programming.

It provides purely functional data structures to complement those from the Scala standard library. It defines a set of foundational type classes (e.g. `Functor`,`Monad`) and corresponding instances for a large number of data structures.

Examples

ApplyUsage

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/96982033@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

ArrowUsage

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/11131196496a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/9243295486bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

Read scalaz online: https://riptutorial.com/scala/topic/9893/scalaz

Chapter 40: Scope

Introduction

Scope on Scala defines where a value (`def`, `val`, `var` or `class`) can be accessed from.

Syntax

- declaration
- private declaration
- private[this] declaration
- private[fromWhere] declaration
- protected declaration
- protected[fromWhere] declaration

Examples

Public (default) scope

By default, the scope is `public`, the value can be accessed from anywhere.

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

A private scope

When the scope is private, it can only be accessed from the current class or other instances of the current class.

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
    }
  }
  class BarClass {
    val f = new FooClass
```

```
    f.x // <- This will not compile
  }
}
```

A private package-specific scope

You can specify a package where the private value can be accessed.

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

Object private scope

The most restrictive scope is *"object-private"* scope, which only allows that value to be accessed from the same instance of the object.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

Protected scope

The protected scope allows the value to be accessed from any subclasses of the current class.

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

Package protected scope

The package protected scope allows the value to be accessed only from any subclass in a specific package.

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

Read Scope online: https://riptutorial.com/scala/topic/9705/scope

Chapter 41: Self types

Syntax

- trait Type { selfId => /*other members can refer to `selfId` in case `this` means something*/ }
- trait Type { selfId: OtherType => /* other members can use `selfId` and it will be of type `OtherType` */
- trait Type { selfId: OtherType1 with OtherType2 => /* `selfId` is of type `OtherType1` and `OtherType2` */

Remarks

Often used with the cake pattern.

Examples

Simple self type example

Self types can be used in traits and classes to define constraints on the concrete classes it is mixed to. It is also possible to use a different identifier for the `this` using this syntax (useful when outer object has to be referenced from an inner object).

Assume you want to store some objects. For that, you create interfaces for the storage and to add values to a container:

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

This way those are not in the same object hierarchy, but `PermanentStorage` cannot be implemented without also implementing `Container`.

Read Self types online: https://riptutorial.com/scala/topic/4639/self-types

Chapter 42: Setting up Scala

Examples

On Linux via dpkg

On Debian-based distributions, including Ubuntu, the most straightforward way is to use the `.deb` installation file. Go to the Scala website. Choose the version you want to install then scroll down and look for `scala-x.x.x.deb`.

You can install the scala deb from command line:

```
sudo dpkg -i scala-x.x.x.deb
```

To verify that it is installed correctly, in the terminal command prompt:

```
which scala
```

The response returned should be the equivalent to what you placed in your PATH variable. To verify that scala is working:

```
scala
```

This should start the Scala REPL, and report the version (which, in turn, should match the version you downloaded).

Ubuntu Installation via Manual Download and Configuration

Download your preferred version from Lightbend with `curl`:

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Unzip the `tar` file to `/usr/local/share` or `/opt/bin`:

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Add the `PATH` to `~/.profile` or `~/.bash_profile` or `~/.bashrc` by including this text to one of those files:

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

To verify that it is installed correctly, in the terminal command prompt:

```
which scala
```

The response returned should be the equivalent to what you placed in your `PATH` variable. To verify that `scala` is working:

```
scala
```

This should start the Scala REPL, and report the version (which, in turn, should match the version you downloaded).

Mac OSX via Macports

On Mac OSX computers with MacPorts installed, open a terminal window and type:

```
port list | grep scala
```

This will list all the Scala-related packages available. To install one (in this example the 2.11 version of Scala):

```
sudo port install scala2.11
```

(The `2.11` may change if you want to install a different version.)

All dependencies will automatically be installed and your `$PATH` parameter updated. To verify everything worked:

```
which scala
```

This will show you the path to the Scala installation.

```
scala
```

This will open up the Scala REPL, and report the version number installed.

Read Setting up Scala online: https://riptutorial.com/scala/topic/2921/setting-up-scala

Chapter 43: Single Abstract Method Types (SAM Types)

Remarks

Single Abstract Methods are types, introduced in Java 8, that have exactly one abstract member.

Examples

Lambda Syntax

NOTE: This is only available in Scala 2.12+ (and in recent 2.11.x versions with the `-Xexperimental -Xfuture` compiler flags)

A SAM type can be implemented using a lambda:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

The type can optionally have other non-abstract members:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Read Single Abstract Method Types (SAM Types) online:
https://riptutorial.com/scala/topic/3664/single-abstract-method-types--sam-types-

Chapter 44: Streams

Remarks

Streams are lazily-evaluated, meaning they can be used to implement generators, which will provide or 'generate' a new item of the specified type on-demand, rather than before the fact. This ensures only the computations necessary are done.

Examples

Using a Stream to Generate a Random Sequence

`genRandom` creates a stream of random numbers that has a one in four chance of terminating each time it's called.

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

Note the `#::` construct, which *lazily recurses*: because it is prepending the current random number to a stream, it does not evaluate the remainder of the stream until it is iterated through.

Infinite Streams via Recursion

Streams can be built that reference themselves and thus become infinitely recursive.

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 36726740705505779255899443

// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

In this context the difference between Var, Val, and Def is interesting. As a `def` each element is recalculated every time it is referenced. As a `val` each element is retained and reused after it's been calculated. This can be demonstrated by creating a side-effect with each calculation.

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

This also explains why the random number `Stream` doesn't work as a `val`.

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

Infinite self-referent stream

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

Read Streams online: https://riptutorial.com/scala/topic/3702/streams

Chapter 45: String Interpolation

Remarks

This feature exists in Scala 2.10.0 and above.

Examples

Hello String Interpolation

The **s** interpolator allows the usage of variables within a string.

```
val name = "Brian"
println(s"Hello $name")
```

prints "Hello Brian" to the console when ran.

Formatted String Interpolation Using the f Interpolator

```
val num = 42d
```

Print two decimal places for `num` using *f*

```
println(f"$num%2.2f")
42.00
```

Print `num` using scientific notation using *e*

```
println(f"$num%e")
4.200000e+01
```

Print `num` in hexadecimal with *a*

```
println(f"$num%a")
0x1.5p5
```

Other format strings can be found at
https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

Using expression in string literals

You can use curly braces to interpolate expressions into string literals:

```
def f(x: String) = x + x
val a = "A"
```

```
s"${a}"    // "A"
s"${f(a)}" // "AA"
```

Without the braces, scala would only interpolate the *identifier* after the `$` (in this case `f`). Since there is no implicit conversion from `f` to a `String` this is an exception in this example:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

Custom string interpolators

It is possible to define custom string interpolators in addition to the built-in ones.

```
my"foo${bar}baz"
```

Is expanded by the compiler to:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` has no `my` method, therefore it can be provided by implicit conversion. A custom interpolator with the same behavior as the builtin `s` interpolator would then be implemented as follows:

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

And the interpolation `my"foo${bar}baz"` would desugar to:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Note that there is no restriction on the arguments or return type of the interpolation function. This leads us down a dark path where interpolation syntax can be used creatively to construct arbitrary objects, as illustrated in the following example:

```
case class Let(name: Char, value: Int)

implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
}
```

```
let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

String interpolators as extractors

It is also possible to use Scala's string interpolation feature to create elaborate extractors (pattern matchers), as perhaps most famously employed in the quasiquotes API of Scala macros.

Given that `n"p0${i0}p1"` desugars to `new StringContext("p0", "p1").n(i0)`, it is perhaps unsurprising that extractor functionality is enabled by providing an implicit conversion from `StringContext` to a class with property `n` of a type defining an `unapply` or `unapplySeq` method.

As an example, consider the following extractor which extracts path segments by constructing a regular expression from the `StringContext` parts. We can then delegate most of the heavy lifting to the `unapplySeq` method provided by the resulting scala.util.matching.Regex:

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

Note that the `path` object could also define an `apply` method in order to behave as a regular interpolator as well.

Raw String Interpolation

You can use the **raw** interpolator if you want a String to be printed as is and without any escaping of literals.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

With the use of the **raw** interpolator, you should see the following printed in the console:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Without the **raw** interpolator, `\n` and `\t` would have been escaped.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
```

```
Monde")
```

Prints:

```
Hello World In English And French
English:       Hello World
French:        Bonjour Le Monde
```

Read String Interpolation online: https://riptutorial.com/scala/topic/1629/string-interpolation

Chapter 46: Symbol Literals

Remarks

Scala comes with a concept of **symbols** - strings that are *interned*, that is: two symbols with the same name (the same character sequence), in contrary to strings, will refer to the same object during execution.

Symbols are a feature of many languages: Lisp, Ruby and Erlang and more, however in Scala they are of relatively small use. Good feature to have nevertheless.

Use:

Any literal beginning with a single quote ', followed by one or more digits, letters, or under-scores _ is a symbol literal. The first character is an exception as it can't be a digit.

Good definitions:

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

Bad definitions:

```
'8'th_division
'94_pattern
'bad-format
```

Examples

Replacing strings in case clauses

Let's say we have multiple data sources which include *database, file, prompt* and *argumentList*. Depending on chosen source we change our approach:

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
}
```

We could have very well used `String` in place of `Symbol`. We didn't, because none of strings's features are useful in this context.

This makes the code simpler and less error prone.

Read Symbol Literals online: https://riptutorial.com/scala/topic/6419/symbol-literals

Chapter 47: synchronized

Syntax

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

synchronize on an object

`synchronized` is a low-level concurrency construct that can help preventing multiple threads access the same resources. Introduction for the JVM using the Java language.

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

In case of `object`s it might synchronize on the class of the object, not on the singleton instance.

synchronize implicitly on this

```
/* within a class, def, trait or object, but not a constructor */
synchronized {
  /* code to run when an intrisctic lock on `this` is acquired */
  /* no other thread can get the this lock unless execution is suspended with
   * `wait` on `this`
   */
}
```

Read synchronized online: https://riptutorial.com/scala/topic/3371/synchronized

Chapter 48: Testing with ScalaCheck

Introduction

ScalaCheck is a library written in Scala and used for automated property-based testing of Scala or Java programs. ScalaCheck was originally inspired by the Haskell library QuickCheck, but has also ventured into its own.

ScalaCheck has no external dependencies other than the Scala runtime, and works great with sbt, the Scala build tool. It is also fully integrated in the test frameworks ScalaTest and specs2.

Examples

Scalacheck with scalatest and error messages

Example of usage scalacheck with scalatest. Below we have four tests:

- "show pass example" - it passes
- "show simple example without custom error message " - just failed message without details, `&&` boolean operator is used
- "show example with error messages on argument" - error message on argument (`"argument" |:`) Props.all method is used instead of `&&`
- "show example with error messages on command" - error message on command (`"command" |:`) Props.all method is used instead of `&&`

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }

  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }

  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
```

```
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }
```

```
  }
}
```

The output (fragments):

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

Read Testing with ScalaCheck online: https://riptutorial.com/scala/topic/9430/testing-with-scalacheck

Chapter 49: Testing with ScalaTest

Examples

Hello World Spec Test

Create a testing class in the `src/test/scala` directory, in a file named `HelloWorldSpec.scala`. Put this inside the file:

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- This example is making use of `FlatSpec` and `Matchers`, which are part of the ScalaTest library.
- `FlatSpec` allows tests to be written in the **Behavior-Driven Development (BDD)** style. In this style, a sentence is used to describe the expected behavior of a given unit of code. The test confirms that the code adheres to that behavior. See the documentation for additional information.

Spec Test Cheatsheet

Setup

The tests below uses these values for the examples.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

Type check

To verify the type for a given `val`:

```
helloWorld shouldBe a [String]
```

Note that the brackets here are used to get type `String`.

Equal check

To test equality:

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

Not Equal check

To test inequality:

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

Length check

To verify length and/or size:

```
helloWorld should have length 11
helloWorldList should have size 2
```

Exceptions check

To verify the type and message of an exception:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

Include the ScalaTest Library with SBT

Using SBT to manage the library dependency, add this to `build.sbt`:

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

More details can be found at the ScalaTest site.

Read Testing with ScalaTest online: https://riptutorial.com/scala/topic/5506/testing-with-scalatest

Chapter 50: Traits

Syntax

- trait ATrait { ... }
- class AClass (...) extends ATrait { ... }
- class AClass extends BClass with ATrait
- class AClass extends ATrait with BTrait
- class AClass extends ATrait with BTrait with CTrait
- class ATrait extends BTrait

Examples

Stackable Modification with Traits

You can use traits to modify methods of a class, using traits in stackable fashion.

The following example shows how traits can be stacked. The ordering of the traits are important. Using different order of traits, different behavior is achieved.

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

Note that `super` is used to invoke `roll()` method in both the traits. Only in this way we can achieve stackable modification. In cases of stackable modification, method invocation order is determined by *linearization rule*.

Trait Basics

This is the most basic version of a trait in Scala.

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

Since no super class is declared for trait `Identifiable`, by default it extends from `AnyRef` class. Because no definition for `getIdentifier` is provided in `Identifiable`, the `Puppy` class must implement it. However, `Puppy` inherits the implementation of `printIdentification` from `Identifiable`.

In the REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

Solving the Diamond Problem

The diamond problem, or multiple inheritance, is handled by Scala using Traits, which are similar to Java interfaces. Traits are more flexible than interfaces and can include implemented methods. This makes traits similar to mixins in other languages.

Scala does not support inheritance from multiple classes, but a user can extend multiple traits in a single class:

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}

trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}

object grandChild extends traitB with traitC

grandChild.name
```

Here `grandChild` is inheriting from both `traitB` and `traitC`, which in turn both inherit from `traitA`. The output (below) also shows the order of precedence when resolving which method implementations are called first:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Note that, when `super` is used to invoke methods in `class` or `trait`, linearization rule come into play to decide call hierarchy. Linearization order for `grandChild` will be:

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Any

Below is another example:

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

This program prints:

```
*************
-------------
Hello World!
```

Linearization for `CustomPrinter` will be:

> CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Printer -> AnyRef -> Any

Linearization

In case of stackable modification, Scala arranges classes and traits in a linear order to determine method call hierarchy, which is known as *linearization*. The linearization rule is used *only* for those methods that involve method invocation via `super()`. Let's consider this by an example:

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

Linearization happens from *back to front*. In this case,

1. First `Shape` will be linearized, which looks like:

   ```
   Shape -> AnyRef -> Any
   ```

2. Then `Dotted` is linearized:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. Next in line is `Blue`. Normally `Blue`'s linearization will be:

   ```
   Blue -> Color -> Shape -> AnyRef -> Any
   ```

   because, in `MyShape`'s linearization until now (*Step 2*), `Shape -> AnyRef -> Any` has already appeared. Hence, it is ignored. Thus, the `Blue` linearization will be:

   ```
   Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
   ```

4. Finally, `Circle` will be added and final linearization order will be:

    Circle -> Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any

This linearization order decides invocation order of methods when `super` is used in any class or trait. The first method implementation from the right is invoked, in the linearization order. If `new MyShape().paint("Circle ")` is executed, it will print:

```
Circle with Blue Color with Dotted Border
```

More information on linearization can be found here.

Read Traits online: https://riptutorial.com/scala/topic/1056/traits

Chapter 51: Tuples

Remarks

Why are tuples limited to length 23?

Tuples are rewritten as objects by the compiler. The compiler has access to `Tuple1` through `Tuple22`. This arbitrary limit was decided by language designers.

Why do tuple lengths count from 0?

A `Tuple0` is equivalent to a `Unit`.

Examples

Creating a new Tuple

A tuple is a heterogeneous collection of two to twenty-two values. A tuple can be defined using parentheses. For tuples of size `2` (also called a 'pair') there's an arrow syntax.

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x` is a tuple of size two. To access the elements of a tuple use `._1`, through `._22`. For instance, we can use `x._1` to access the first element of the `x` tuple. `x._2` accesses the second element. More elegantly, you can use tuple extractors.

The arrow syntax for creating tuples of size two is primarily used in Maps, which are collections of `(key -> value)` pairs:

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

The syntax for the pair in the map is the arrow syntax, making it clear that 1 is the key and a is the value associated with that key.

Tuples within Collections

Tuples are often used within collections but they must be handled in a specific way. For example, given the following list of tuples:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

It may seem natural to add the elements together using implicit tuple-unpacking:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

However this results in the following error:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala cannot implicitly unpack the tuples in this manner. We have two options to fix this map. The first is to use the positional accessors `_1` and `_2`:

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

The other option is to use a `case` statement to unpack the tuples using pattern matching:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

These restrictions apply for any higher-order-function applied to a collection of tuples.

Read Tuples online: https://riptutorial.com/scala/topic/4971/tuples

Chapter 52: Type Classes

Remarks

To avoid serialization problems, particularly in distributed environments (e.g. Apache Spark), it is a best practice to implement the `Serializable` trait for type class instances.

Examples

Simple Type Class

A type class is simply a `trait` with one or more type parameters:

```
trait Show[A] {
  def show(a: A): String
}
```

Instead of extending a type class, an implicit instance of the type class is provided for each supported type. Placing these implementations in the companion object of the type class allows implicit resolution to work without any special imports:

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

If you want to guarantee that a generic parameter passed to a function has an instance of a type class, use implicit parameters:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

You can also use a context bound:

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Call the above `log` method like any other method. It will fail to compile if an implicit `Show[A]` implementation can't be found for the `A` you pass to `log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

This example implements the `Show` type class. This is a common type class used to convert arbitrary instances of arbitrary types into `String`s. Even though every object has a `toString` method, it's not always clear whether or not `toString` is defined in a useful way. With use of the `Show` type class, you can guarantee that anything passed to `log` has a well-defined conversion to `String`.

Extending a Type Class

This example discusses extending the below type class.

```
trait Show[A] {
  def show: String
}
```

To make a class *you* control (and is written in Scala) extend the type class, add an implicit to its companion object. Let us show how we can get the `Person` class from this example to extend `Show`:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

We can make this class extend `Show` by adding an implicit to `Person`'s companion object:

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

A companion object *must be in the same file* as the class, so you need both `class Person` and `object Person` in the same file.

To make a class you do not control, or is not written in Scala, extend the type class, add an implicit to the companion object of the type class, as shown in the Simple Type Class example.

If you control neither the class nor the type class, create an implicit as above anywhere, and `import` it. Using the `log` method on the Simple Type Class example:

```
object MyShow {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}

def logPeople(persons: Person*): Unit = {
```

```
  import MyShow.personShow
  persons foreach { p => log(p) }
}
```

Add type class functions to types

Scala's implementation of type classes is rather verbose. One way to reduce the verbosity is to introduce so-called "Operation Classes". These classes will automatically wrap a variable/value when they are imported to extend functionality.

To illustrate this, let us first create a simple type class:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

Next we will implement the trait (instantiate the type class):

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

At the moment it would be very cumbersome to use our Addable instances:

```
import Instances._
val three = addableInt.add(1,2)
```

We would rather just write write `1.add(2)`. Therefore we'll create an "Operation Class" (also called an "Ops Class") that will always wrap over a type that implements `Addable`.

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
  }
}
```

Now we can use our new function `add` as if it was part of `Int` and `String`:

```
object Main {
```

```
  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers

  def main(args: Array[String]): Unit = {

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

"Ops" classes can be created automatically by macros in simulacrum library:

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Read Type Classes online: https://riptutorial.com/scala/topic/3835/type-classes

Chapter 53: Type Inference

Examples

Local Type Inference

Scala has a powerful type-inference mechanism built-in to the language. This mechanism is termed as 'Local Type Inference':

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

The compiler can infer the type of variables from the initialization expression. Similarly, the return type of methods can be omitted, since they are equivalent to the type returned by the method body. The above examples are equivalent to the below, explicit type declarations:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

Type Inference And Generics

The Scala compiler can also deduce type parameters when polymorphic methods are called, or when generic classes are instantiated:

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

The above form of type inference is similar to the Diamond Operator, introduced in Java 7.

Limitations to Inference

There are scenarios in which Scala type-inference does not work. For instance, the compiler cannot infer the type of method parameters:

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

The compiler cannot infer the return type of recursive methods:

```
// Does not compile
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

Preventing inferring Nothing

Based on this blog post.

Assume you have a method like this:

```
  def get[T]: Option[T] = ???
```

When you try to call it without specifying the generic parameter, `Nothing` gets inferred, which is not very useful for an actual implementation (and its result is not useful). With the following solution the `NotNothing` context bound can prevent using the method without specifying the expected type (in this example `RuntimeClass` is also excluded as for `ClassTags` not `Nothing`, but `RuntimeClass` is inferred):

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

Example usage:

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()

  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

Read Type Inference online: https://riptutorial.com/scala/topic/4918/type-inference

Chapter 54: Type Parameterization (Generics)

Examples

The Option type

A nice example of a parameterized type is the Option type. It is essentially just the following definition (with several more methods defined on the type):

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

We can also see that this has a parameterized method, `fold`, which returns something of type `B`.

Parameterized Methods

The return type of a method can depend on the *type* of the parameter. In this example, `x` is the parameter, `A` is the *type* of `x`, which is known as the *type parameter*.

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala will use type inference to determine the return type, which constrains what methods may be called on the parameter. Thus, care must be taken: the following is a compile-time error because `*` is not defined for every type `A`:

```
def g[A](x: A): A = 2 * x  // Won't compile
```

Generic collection

Defining the list of Ints

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

but what if we need to define the list of Boolean, Double etc?

Defining generic list

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

Read Type Parameterization (Generics) online: https://riptutorial.com/scala/topic/782/type-parameterization--generics-

Chapter 55: Type Variance

Examples

Covariance

The `+` symbol marks a type parameter as *covariant* - here we say that "`Producer` is covariant on `A`":

```
trait Producer[+A] {
  def produce: A
}
```

A covariant type parameter can be thought of as an "output" type. Marking `A` as covariant asserts that `Producer[X] <: Producer[Y]` provided that `X <: Y`. For example, a `Producer[Cat]` is a valid `Producer[Animal]`, as all produced cats are also valid animals.

A covariant type parameter cannot appear in contravariant (input) position. The following example will not compile as we are asserting that `Co[Cat] <: Co[Animal]`, but `Co[Cat]` has `def handle(a: Cat): Unit` which cannot handle any `Animal` as required by `Co[Animal]`!

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

One approach to dealing with this restriction is to use type parameters bounded by the covariant type parameter. In the following example, we know that `B` is a supertype of `A`. Therefore given `Option[X] <: Option[Y]` for `X <: Y`, we know that `Option[X]`'s `def getOrElse[B >: X](b: => B): B` can accept any supertype of `X` - which includes the supertypes of `Y` as required by `Option[Y]`:

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

Invariance

By default all type parameters are invariant - given `trait A[B]`, we say that "`A` is invariant on `B`". This means that given two parametrizations `A[Cat]` and `A[Animal]`, we assert no sub/superclass relationship between these two types - it does not hold that `A[Cat] <: A[Animal]` nor that `A[Cat] >: A[Animal]` regardless of the relationship between `Cat` and `Animal`.

Variance annotations provide us with a means of declaring such a relationship, and imposes rules on the usage of type parameters so that the relationship remains valid.

Contravariance

The `-` symbol marks a type parameter as *contravariant* - here we say that "`Handler` is contravariant

on `A`":

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

A contravariant type parameter can be thought of as an "input" type. Marking `A` as contravariant asserts that `Handler[X] <: Handler[Y]` provided that `X >: Y`. For example a `Handler[Animal]` is a valid `Handler[Cat]`, as a `Handler[Animal]` must also handle cats.

A contravariant type parameter cannot appear in covariant (output) position. The following example will not compile as we are asserting that a `Contra[Animal] <: Contra[Cat]`, however a `Contra[Animal]` has `def produce: Animal` which is not guaranteed to produce cats as required by `Contra[Cat]`!

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Beware however: for the purposes of overloading resolution, contravariance also counterintuitively inverts the specificity of a type on the contravariant type parameter - `Handler[Animal]` is considered to be "more specific" than `Handler[Cat]`.

As it is not possible to overload methods on type parameters, this behavior generally only becomes problematic when resolving implicit arguments. In the following example `ofCat` will never be used, as the return type of `ofAnimal` is more specific:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

This behavior is currently slated to change in dotty, and is why (as an example) `scala.math.Ordering` is invariant on its type parameter `T`. One workaround is to make your typeclass invariant, and type-parametrize the implicit definition in the event that you want it to apply to subclasses of a given type:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

Covariance of a collection

Because collections are typically covariant in their element type*, a collection of a subtype may be passed where a super type is expected:

```
trait Animal { def name: String }
```

```
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
// Moe
```

It may not seem like magic, but the fact that a `Seq[Dog]` is accepted by a method that expects a `Seq[Animal]` is the entire concept of a higher-kinded type (here: `Seq`) being covariant in its type parameter.

* A counterexample being the standard library's Set

Covariance on an invariant trait

There is also a way to have a single method accept a covariant argument, instead of having the whole trait covariant. This may be necessary because you would like to use `T` in a contravariant position, but still have it covariant.

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}

object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
  lv.handle(new B {})
}
```

Read Type Variance online: https://riptutorial.com/scala/topic/1651/type-variance

Chapter 56: Type-level Programming

Examples

Introduction to type-level programming

If we consider a heterogenous list, wherein the elements of the list have varied but known types, it might be desirable to be able to perform operations on the elements of the list collectively without discarding the elements' type information. The following example implements a mapping operation over a simple heterogenous list.

Because the element type varies, the class of operations we can perform is restricted to some form of type projection, so we define a trait `Projection` having abstract `type Apply[A]` computing the result *type* of the projection, and `def apply[A](a: A): Apply[A]` computing the result *value* of the projection.

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

In implementing `type Apply[A]` we are programming at the type level (as opposed to the value level).

Our heterogenous list type defines a `map` operation parametrized by the desired projection as well as the projection's type. The result of the map operation is abstract, will vary by implementing class and projection, and must naturally still be an `HList`:

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

In the case of `HNil`, the empty heterogenous list, the result of any projection will always be itself. Here we declare `trait HNil` as a convenience so that we may write `HNil` as a type in lieu of `HNil.type`:

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons` is the non-empty heterogenous list. Here we assert that when applying a map operation, the resulting head type is that which results from the application of the projection to the head value (`P#Apply[H]`), and that the resulting tail type is that which results from mapping the projection over the tail (`T#Map[P]`), which is known to be an `HList`:

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

The most obvious such projection is to perform some form of wrapping operation - the following example yields an instance of `HCons[Option[String], HCons[Option[Int], HNil]]`:

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

Read Type-level Programming online: https://riptutorial.com/scala/topic/3738/type-level-programming

Chapter 57: User Defined Functions for Hive

Examples

A simple Hive UDF within Apache Spark

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

Read User Defined Functions for Hive online: https://riptutorial.com/scala/topic/8241/user-defined-functions-for-hive

Chapter 58: Var, Val, and Def

Remarks

As `val` are semantically static, they are initialized "in-place" wherever they appear in the code. This can produce surprising and undesirable behavior when used in abstract classes and traits.

For example, let's say we would like to make a trait called `PlusOne` that defines an increment operation on a wrapped `Int`. Since `Int`s are immutable, the value plus one is known at initialization and will never be changed afterwards, so semantically it's a `val`. However, defining it this way will produce an unexpected result.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

No matter what value `i` you construct `IntWrapper` with, calling `.incr` on the returned object will always return 1. This is because the val `incr` is initialized *in the trait*, before the extending class, and at that time `i` only has the default value of `0`. (In other conditions, it might be populated with `Nil`, `null`, or a similar default.)

The general rule, then, is to avoid using `val` on any value that depends on an abstract field. Instead, use `lazy val`, which does not evaluate until it is needed, or `def`, which evaluates every time it is called. Note however that if the `lazy val` is forced to evaluate by a `val` before initialization completes, the same error will occur.

A fiddle (written in Scala-Js, but the same behavior applies) can be found here.

Examples

Var, Val, and Def

var

A `var` is a reference variable, similar to variables in languages like Java. Different objects can be freely assigned to a `var`, so long as the given object has the same type that the `var` was declared with:

```
scala> var x = 1
x: Int = 1

scala> x = 2
```

```
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

Note in the example above the type of the `var` was inferred by the compiler given the first value assignment.

val

A `val` is a constant reference. Thus, a new object cannot be assigned to a `val` that has already been assigned.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

However, the object that a `val` points to is *not* constant. That object may be modified:

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

def

A `def` defines a method. A method cannot be re-assigned to.

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

In the above examples, `val y` and `def z` return the same value. However, a `def` is evaluated *when it is called*, whereas a `val` or `var` is evaluated *when it is assigned*. This can result in differing behavior when the definition has side effects:

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2

scala> b + 1
Hi
res3: Int = 2
```

Functions

Because functions are values, they can be assigned to `val`/`var`/`def`s. Everything else works in the same manner as above:

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

Lazy val

`lazy val` is a language feature where the initialization of a `val` is delayed until it is accessed for the first time. After that point, it acts just like a regular `val`.

To use it add the `lazy` keyword before `val`. For example, using the REPL:

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
     |   "my bar value"
     | }
```

```
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

This example demonstrates the execution order. When the `lazy val` is declared, all that is saved to the `foo` value is a lazy function call that hasn't been evaluated yet. When the regular `val` is set, we see the `println` call execute and the value is assigned to `bar`. When we evalute `foo` the first time we see `println` execute - but not when it's evaluated the second time. Similarly, when `bar` is evaluated we don't see `println` execute - only when it is declared.

When To Use 'lazy'

1. **Initialization is computationally expensive and usage of `val` is rare.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` takes a long time to calculate, and it's not always used. Making it a `lazy val` saves unnecessary computation.

2. **Resolving cyclic dependencies**

   Let's look at an example with two objects that need to be declared at the same time during instantiation:

   ```
   object comicBook {
     def main(args:Array[String]): Unit = {
       gotham.hero.talk()
       gotham.villain.talk()
     }
   }

   class Superhero(val name: String) {
     lazy val toLockUp = gotham.villain
     def talk(): Unit = {
       println(s"I won't let you win ${toLockUp.name}!")
     }
   }

   class Supervillain(val name: String) {
     lazy val toKill = gotham.hero
     def talk(): Unit = {
   ```

```
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

Without the keyword `lazy`, the respective objects can not be members of an object. Execution of such a program would result in a `java.lang.NullPointerException`. By using `lazy`, the reference can be assigned before it is initialized, without fear of having an uninitialized value.

Overloading Def

You can overload a `def` if the signature is different:

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

This functions the same whether inside classes, traits, objects or not.

Named Parameters

When invoking a `def`, parameters may be assigned explicitly by name. Doing so means they needn't be correctly ordered. For example, define `printUs()` as:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Now it can be called in these ways (amongst others):

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

This results in `one, two, three` being printed in all cases.

If not all arguments are named, the first arguments are matched by order. No positional (non-named) argument may follow a named one:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Read Var, Val, and Def online: https://riptutorial.com/scala/topic/3155/var--val--and-def

Chapter 59: While Loops

Syntax

- while (boolean_expression) { block_expression }
- do { block_expression } while (boolean_expression)

Parameters

Parameter	Details
`boolean_expression`	Any expression that will evaluate to `true` or `false`.
`block_expression`	Any expression or set of expressions that will be evaluated if the `boolean_expression` evaluates to `true`.

Remarks

The primary difference between `while` and `do-while` loops is whether they execute the `block_expression` before they check to see if they should loop.

Because `while` and `do-while` loops rely on an expression to evaluate to `false` to terminate, they often require mutable state to be declared outside the loop and then modified inside the loop.

Examples

While Loops

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Do-While Loops

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
  println("Line number: " + line)
} while (line < maximum_lines)
```

The `do`/`while` loop is infrequently used in functional programming, but can be used to work around the lack of support for the `break`/`continue` construct, as seen in other languages:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Read While Loops online: https://riptutorial.com/scala/topic/650/while-loops

Chapter 60: Working with data in immutable style

Remarks

Value and variable names should be in lower camel case

> Constant names should be in upper camel case. That is, if the member is final, immutable and it belongs to a package object or an object, it may be considered a constant
>
> Method, Value and variable names should be in lower camel case

Source: http://docs.scala-lang.org/style/naming-conventions.html

This compile:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

but this doesn't:

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

It is not just val vs. var

`val` and `var`

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123

scala> b = 321
b: Int = 321
```

- `val` references are unchangeable: like a `final` variable in `Java`, once it has been initialized

you cannot change it
- `var` references are reassignable as a simple variable declaration in Java

Immutable and Mutable collections

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

The Scala standard library offers both immutable and mutable data structures, not the reference to it. Each time an immutable data structure get "modified", a new instance is produced instead of modifying the original collection in-place. Each instance of the collection may share significant structure with another instance.

Mutable and Immutable Collection (Official Scala Documentation)

But I can't use immutability in this case!

Let's pick as an example a function that takes 2 `Map` and return a `Map` containing every element in `ma` and `mb`:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

A first attempt could be iterating through the elements of one of the maps using `for ((k, v) <- map)` and somehow return the merged map.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

This very first move immediately add a constrain: **a mutation outside that `for` is now *needed***. This is more clear when de-sugaring the `for`:

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

"Why we have to mutate?"

`foreach` relies on side-effects. Every time we want something to happen within a `foreach` we need to "side-effect something", in this case we could mutate a variable `var result` or we can use a mutable data structure.

Creating and filling the `result` map

Let's assume the `ma` and `mb` are `scala.collection.immutable.Map`, we could create the `result` Map from `ma`:

```
val result = mutable.Map() ++ ma
```

Then iterate through `mb` adding its elements and if the `key` of the current element on `ma` already exist, let's override it with the `mb` one.

```
mb.foreach { case (k, v) => result += (k -> v) }
```

Mutable implementation

So far so good, we "had to use mutable collections" and a correct implementation could be:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

As expected:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

Folding to the rescue

How can we get rid of `foreach` in this scenario? If all we what to do is basically iterate over the collection elements and apply a function while accumulating the result on option could be using `.foldLeft`:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

In this case our "result" is the accumulated value starting from `ma`, the `zero` of the `.foldLeft`.

Intermediate result

Obviously this immutable solution is producing and destroying many `Map` instances while folding, but it is worth mentioning that those instances are not a full clone of the `Map` accumulated but instead are sharing significant structure (data) with the existing instance.

Easier reasonability

It is easier to reason about the semantic if it is more declarative as the `.foldLeft` approach. Using immutable data structures could help making our implementation easier to reason on.

Read Working with data in immutable style online: https://riptutorial.com/scala/topic/6298/working-with-data-in-immutable-style

Chapter 61: Working With Gradle

Examples

Basic Setup

1. Create a file named `SCALA_PROJECT/build.gradle` with these contents:

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. Run `gradle tasks` to see available tasks.

3. Run `gradle create-dirs` to create a `src/scala, src/resources` directory.

4. Run `gradle build` to build the project and download dependencies.

Create your own Gradle Scala plugin

After going through the **Basic Setup** example, you may find yourself repeating most part of it in every single Scala Gradle project. Smells like boilerplate code...

What if, instead of applying the Scala plugin offered by Gradle, you could apply your own Scala plugin, which would be responsible for handling all your common build logic, extending, at the same time, the already existing plugin.

This example is going to transform the previous build logic into a reusable Gradle plugin.

Luckyly, in Gradle, you can easily write custom plugins with the help of the Gradle API, as outlined in the documentation. As language of implementation, you can use Scala itself or even Java.

However, most of the examples you can find throughout the docs are written in Groovy. If you need more code samples or you want to understand what lies behind the Scala plugin, for instance, you can check the gradle github repo.

Writing the plugin

Requirements

The custom plugin will add the following functionality when applied to a project:

- a `scalaVersion` property object, which will have two overridable default properties
  - major = "2.12"
  - minor = "0"
- a `withScalaVersion` function, which applied to a dependency name, will add the scala major version to ensure binary compatibility (sbt `%%` operator might ring a bell, otherwise go here before proceeding)
- a `createDirs` task to create the necessary directory tree, exactly as in the previous example

Implementation guideline

1. create a new gradle project and add the following to `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

Notes:

- the plugin implementation is written in Scala, thus we need the Gradle Scala Plugin
- in order to use the plugin from other projects, the Gradle Maven Plugin is used; this adds the `install` task used for saving the project jar to the Maven Local Repository
- `compile gradleApi()` adds the `gradle-api-<gradle_version>.jar` to the classpath

2. create a new Scala class for the plugin implementation

```
package com.btesila.gradle.plugins

import org.gradle.api.{Plugin, Project}

class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
```

```
    }
}
```

Notes:

- in order to implement a Plugin, just extend `Plugin` trait of type `Project` and override the `apply` method
- within the apply method, you have access to the `Project` instance that the plugin is applied to and you can use it for adding build logic to it
- this plugin does nothing but apply the already existing Gradle Scala Plugin

3. add the `scalaVersion` object property

Firstly, we create a `ScalaVersion` class, which will hold the two version properties

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

One cool thing about Gradle plugins is the fact that you can always add or override specific properties. A plugin receives this kind of user input via the `ExtensionContainer` attached to a gradle `Project` instance. For more details, check this out.
By adding the following to the `apply` method, we are basically doing this:

- if there is not a `scalaVersion` property defined in the project, we add one with the default values
- otherwise, we get the existing one as instance of `ScalaVersion`, to use it further

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

This is equivalent to writing the following to the build file of the project that applies the plugin:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. add the `scala-lang` library to the project dependencies, using the `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

This is equivalent to writing the following to the build file of the project that applies the plugin:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. add the `withScalaVersion` function

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. finally, create the `createDirs` task and add it to the project
   Implement a Gradle task by extending `DefaultTask`:

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

Note: the `SourceSetContainer` has information about all source directories present in the project. What the Gradle Scala Plugin does, is to add the extra source sets to the Java ones, as you can see in theplugin docs.

Add the `createDir` task to the project by appending this to the `apply` method:

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

In the end, your `ScalaCustomPlugin` class should look like this:

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]

    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

```
    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

Installing the plugin project to the local Maven repository

This is done really easy by running `gradle install`
You can check the installation by going to local repository directory, usually found at `~/.m2/repository`

How does Gradle find our new plugin?

Each Gradle plugin has an `id` which is used in the `apply` statement. For instance, by writing the following to the build file, it translates to a trigger to Gradle to find and apply the plugin with id `scala`.

```
apply plugin: 'scala'
```

In the same way, we would like to apply our new plugin in the following way,

```
apply plugin: "com.btesila.scala.plugin"
```

meaning that our plugin will have the `com.btesila.scala.plugin` id.

In order to set this id, add the following file:
src/main/resources/META-INF/gradle-plugin/com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

Afterwards, run again `gradle install`.

Using the plugin

1. create a new empty Gradle project and add the following to the build file

```
buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
```

```
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}

repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. run `gradle createDirs` - you should now have all the source directories generated
3. override the scala version by adding this to the build file:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. add a dependency library that is binary compatible with the Scala version

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

That's it! You can now use this plugin across all your projects without repeating the same old boilerplate.

Read Working With Gradle online: https://riptutorial.com/scala/topic/3304/working-with-gradle

Chapter 62: XML Handling

Examples

Beautify or Pretty-Print XML

The `PrettyPrinter` utility will 'pretty print' XML documents. The following code snippet pretty prints unformatted xml:

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

This will output the content using a page width of `150` and an indentation constant of `4` white-space characters:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

You can use `XML.loadFile("nameoffile.xml")` to load xml from a file instead of from a string.

Read XML Handling online: https://riptutorial.com/scala/topic/1453/xml-handling

Credits

S. No	Chapters	Contributors
1	Getting started with Scala Language	4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz
2	Annotations	Gábor Bakos, Nathaniel Ford, Thomas Matecki
3	Best Practices	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
4	Case Classes	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
5	Classes and Objects	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
6	Collections	Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin
7	Continuations Library	dmitry, HTNW
8	Currying	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890
9	Dependency Injection	Hoang Ong
10	Dynamic Invocation	HTNW
11	Enumerations	Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
12	Error Handling	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos, Tzach Zohar

13	Extractors	Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler, Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill
14	For Expressions	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
15	Functions	Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192, Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka , Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties
16	Futures	isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
17	Handling units (measures)	Gábor Bakos
18	Higher Order Function	acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini
19	If Expressions	corvus_192, Nathaniel Ford, ScientiaEtVeritas
20	Implicits	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
21	Java Interoperability	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
22	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
23	Macros	gregghz, HTNW, Nathaniel Ford
24	Monads	ipoteka, Nathaniel Ford
25	Operator Overloading	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
26	Operators in Scala	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
27	Option Class	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
28	Packages	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
29	Parallel Collections	Nathaniel Ford, Shuklaswag

30	Parser Combinators	Nathaniel Ford
31	Partial Functions	acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
32	Pattern Matching	Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio, Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe
33	Quasiquotes	gregghz
34	Recursion	Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
35	Reflection	Sachin Janani
36	Regular Expressions	dmitry, J Cracknell, Nathaniel Ford
37	Scala.js	Camilo Sampedro
38	Scaladoc	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
39	scalaz	chengpohi
40	Scope	Camilo Sampedro
41	Self types	Gábor Bakos, irundaia
42	Setting up Scala	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk
43	Single Abstract Method Types (SAM Types)	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
44	Streams	jwvh, Nathaniel Ford, Oleg Pyzhcov
45	String Interpolation	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
46	Symbol Literals	ZbyszekKr
47	synchronized	Gábor Bakos
48	Testing with ScalaCheck	Andrzej Jozwik
49	Testing with ScalaTest	Nadim Bahadoor, Nathaniel Ford

50	Traits	André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz
51	Tuples	corvus_192, evan.oman, Lawsy, Nathaniel Ford
52	Type Classes	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
53	Type Inference	Gábor Bakos, Nathaniel Ford, suj1th
54	Type Parameterization (Generics)	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
55	Type Variance	acjay, J Cracknell, Reactormonk
56	Type-level Programming	J Cracknell
57	User Defined Functions for Hive	Camilo Sampedro
58	Var, Val, and Def	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr
59	While Loops	J Cracknell, Nathaniel Ford
60	Working with data in immutable style	Filippo Vitale
61	Working With Gradle	Bianca Tesila, Nathaniel Ford, Rjk
62	XML Handling	Nathaniel Ford, Rockie Yang, vsnyc