Kostenloses eBook

LERNEN

Scala Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#scala

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit Scala Language

Bemerkungen

Scala ist eine moderne Multi-Paradigma-Programmiersprache, die gängige Programmiermuster auf prägnante, elegante und typsichere Weise ausdrückt. Es integriert nahtlos Merkmale objektorientierter und funktionaler Sprachen.

Die meisten Beispiele erfordern eine funktionierende Scala-Installation. Dies ist die Scala-Installationsseite. Dies ist das Beispiel für das Einrichten von Scala . scalafiddle.net ist eine gute Ressource, um kleine Codebeispiele über das Web auszuführen.

Versionen

Ausführung	Veröffentlichungsdatum
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
2.12.0	2016-11-03

Ausführung	Veröffentlichungsdatum
2.12.1	2016-12-06
2.12.2	2017-04-19

Examples

Hallo Welt durch Definieren einer "Hauptmethode"

Fügen Sie diesen Code in eine Datei mit dem Namen `HelloWorld.scala` :

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

Live-Demo

So kompilieren Sie es in einen Bytecode, der von der JVM ausgeführt werden kann:

```
$ scalac HelloWorld.scala
```

Um es auszuführen:

```
$ scala Hello
```

Wenn die Scala-Laufzeit das Programm lädt, sucht es mit einer `main` nach einem Objekt namens `Hello` . Die `main` ist der Programmeintrittspunkt und wird ausgeführt.

Beachten Sie, dass Scala im Gegensatz zu Java keine Benennung von Objekten oder Klassen nach der Datei verlangt, in der sie enthalten sind. Stattdessen verweist der im Befehl `scala Hello` Parameter `Hello` auf das zu `scala Hello` Objekt, das die `main` Hauptmethode enthält. Es ist durchaus möglich, mehrere Objekte mit Hauptmethoden in derselben `.scala` Datei zu haben.

Das `args` Array enthält ggf. die Befehlszeilenargumente, die dem Programm übergeben werden. Zum Beispiel können wir das Programm folgendermaßen ändern:

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

Kompiliere es:

```
$ scalac HelloWorld.scala
```

Und dann führe es aus:

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

Hallo Welt durch Erweiterung der App

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

Live-Demo

Durch die Erweiterung der `App` Eigenschaft können Sie die Definition einer expliziten `main` vermeiden. Der gesamte Körper des `HelloWorld` Objekts wird als "Hauptmethode" behandelt.

2.11.0

> ## Verzögerte Initialisierung
>
> Laut der offiziellen Dokumentation verwendet `App` eine Funktion namens " *Verzögerte Initialisierung"* . Dies bedeutet, dass die Objektfelder *nach* dem Aufruf der Hauptmethode initialisiert werden.

2.11.0

> ## Verzögerte Initialisierung
>
> Laut der offiziellen Dokumentation verwendet `App` eine Funktion namens " *Verzögerte Initialisierung"* . Dies bedeutet, dass die Objektfelder *nach* dem Aufruf der Hauptmethode initialisiert werden.
>
> `DelayedInit` ist jetzt für die allgemeine Verwendung **veraltet** , wird jedoch als Sonderfall für `App` *weiterhin unterstützt* . Der Support wird fortgesetzt, bis eine Ersatzfunktion festgelegt und implementiert ist.

Um auf Befehlszeilenargumente zuzugreifen, wenn Sie `App` , verwenden Sie `this.args`

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

Bei der Verwendung von `App` , wird der Körper des Objekts als die ausgeführt wird `main` gibt es keine Notwendigkeit , außer Kraft zu setzen `main` .

Hallo Welt als Drehbuch

Scala kann als Skriptsprache verwendet werden. Um dies zu demonstrieren, erstellen Sie `HelloWorld.scala` mit dem folgenden Inhalt:

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

Führen Sie es mit dem Befehlszeileninterpreter aus (das `$` ist die Befehlszeilenaufforderung):

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

Wenn Sie `.scala` auslassen (z. B. wenn Sie einfach `scala HelloWorld .scala` eingegeben haben), sucht der Läufer nach einer kompilierten `.class` Datei mit Bytecode, anstatt das Skript zu kompilieren und dann auszuführen.

> **Hinweis:** Wenn Scala als Skriptsprache verwendet wird, kann kein Paket definiert werden.

In Betriebssystemen, die `bash` oder ähnliche Shell-Terminals verwenden, können Scala-Skripts unter Verwendung einer 'Shell-Präambel' ausgeführt werden. Erstellen Sie eine Datei mit dem Namen `HelloWorld.sh` und geben Sie Folgendes als Inhalt ein:

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

Die Teile zwischen `#!` und `!#` ist die 'Shell-Präambel' und wird als Bash-Skript interpretiert. Der Rest ist Scala.

Nachdem Sie die obige Datei gespeichert haben, müssen Sie ihr die Berechtigung "ausführbar" erteilen. In der Shell können Sie Folgendes tun:

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

(Beachten Sie, dass dies allen Benutzern die Berechtigung gibt: Lesen Sie mehr über chmod , um zu erfahren, wie Sie es für spezifischere Benutzergruppen festlegen.)

Jetzt können Sie das Skript folgendermaßen ausführen:

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

Verwenden der Scala REPL

Wenn Sie `scala` in einem Terminal ohne zusätzliche Parameter ausführen, wird ein REPL - Interpreter (Read-Eval-Print Loop) geöffnet:

```
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>
```

Mit REPL können Sie Scala auf einem Arbeitsblatt ausführen: Der Ausführungskontext bleibt erhalten und Sie können Befehle manuell ausprobieren, ohne ein gesamtes Programm erstellen zu müssen. Wenn Sie beispielsweise `val poem = "As halcyons we shall be"` würde dies folgendermaßen aussehen:

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

Jetzt können wir unseren `val` drucken:

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

Beachten Sie, dass `val` unveränderlich ist und nicht überschrieben werden kann:

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

In der REPL können Sie *jedoch* einen `val` neu definieren (der zu einem Fehler in einem normalen Scala-Programm führen würde, wenn er im gleichen Umfang ausgeführt würde):

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

Für den Rest Ihrer REPL-Sitzung spiegelt diese neu definierte Variable die zuvor definierte Variable. REPLs sind hilfreich, um schnell zu sehen, wie Objekte oder anderer Code funktionieren. Alle Funktionen von Scala sind verfügbar: Sie können Funktionen, Klassen, Methoden usw. definieren.

Scala Quicksheet

Beschreibung	Code
Weisen Sie einen unveränderlichen int-Wert zu	`val x = 3`
Veränderlichen int-Wert zuweisen	`var x = 3`
Weisen Sie einen unveränderlichen Wert mit explizitem Typ zu	`val x: Int = 27`

Beschreibung	Code
Faul bewerteten Wert zuweisen	`lazy val y = print("Sleeping in.")`
Binden Sie eine Funktion an einen Namen	`val f = (x: Int) => x * x`
Binden Sie eine Funktion an einen Namen mit explizitem Typ	`val f: Int => Int = (x: Int) => x * x`
Methode definieren	`def f(x: Int) = x * x`
Definieren Sie eine Methode mit expliziter Typisierung	`def f(x: Int): Int = x * x`
Definieren Sie eine Klasse	`class Hopper(someParam: Int) { ... }`
Definieren Sie ein Objekt	`object Hopper(someParam: Int) { ... }`
Definiere ein Merkmal	`trait Grace { ... }`
Holen Sie sich das erste Element der Sequenz	`Seq(1,2,3).head`
Wenn wechseln	`val result = if(x > 0) "Positive!"`
Holen Sie sich alle Elemente der Sequenz außer zuerst	`Seq(1,2,3).tail`
Eine Liste durchgehen	`for { x <- Seq(1,2,3) } print(x)`

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

Für jedes Listenelement die Funktion ausführen	`List(1,2,3).foreach { println }`
Druck auf Standard aus	`print("Ada Lovelace")`
Sortieren Sie eine Liste alphanumerisch	`List('b','c','a').sorted`

Erste Schritte mit Scala Language online lesen: https://riptutorial.com/de/scala/topic/216/erste-schritte-mit-scala-language

Kapitel 2: Abhängigkeitsspritze

Examples

Kuchenmuster mit innerer Umsetzungsklasse.

```
//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()
}
```

Im obigen Beispiel habe ich gezeigt, wie TimeUtil in den MainController injiziert wird.

Die wichtigste Syntax ist die Selbstanmerkung (`_: TimeUtil =>`), die `TimeUtil` in den `MainController` injizieren `MainController` . Mit anderen Worten, `MainController` hängt von `TimeUtil` .

Ich verwende in jeder Komponente eine innere Klasse (z. B. `TimeUtilImpl`), da dies meiner Meinung nach einfacher ist, da wir die innere Klasse nachahmen können. Und es ist auch einfacher für die Nachverfolgung, wo die Methode aufgerufen wird, wenn das Projekt komplexer wird.

Zum Schluss verdrahle ich alle Komponenten miteinander. Wenn Sie mit Guice vertraut sind, entspricht dies der `Binding`

Abhängigkeitsspritze online lesen: https://riptutorial.com/de/scala/topic/5909/abhangigkeitsspritze

Kapitel 3: Anmerkungen

Syntax

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

Parameter

Parameter	Einzelheiten
@	Gibt an, dass das folgende Token eine Anmerkung ist.
SomeAnnotation	Der Name der Anmerkung
constructor_args	(optional) Die an die Annotation übergebenen Argumente. Wenn keine, werden die Klammern nicht benötigt.

Bemerkungen

Scala-lang bietet eine Liste der Standardanmerkungen und ihrer Java-Entsprechungen .

Examples

Verwenden einer Anmerkung

Diese Beispielanmerkung weist darauf hin, dass die folgende Methode nicht mehr `deprecated` .

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

Dies kann auch äquivalent geschrieben werden als:

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

Anmerkungen zum Hauptkonstruktor

```
/**
 * @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)
}
```

Der Sichtbarkeitsmodifizierer (in diesem Fall `protected`) sollte nach den Anmerkungen in derselben Zeile stehen. `@throws` die Annotation optionale Parameter akzeptiert (da in diesem Fall `@throws` eine optionale Ursache akzeptiert), müssen Sie vor den Konstruktorparametern eine leere Parameterliste für Annotation: `()` angeben.

Hinweis: Es können mehrere Anmerkungen angegeben werden, auch vom selben Typ (wiederholte Anmerkungen).

Ähnlich bei einer Fallklasse ohne zusätzliche Factory-Methode (und der für die Annotation angegebenen Ursache):

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

Eigene Anmerkungen erstellen

Sie können Ihre eigenen Scala-Anmerkungen erstellen, indem Sie Klassen erstellen, die von scala.annotation.StaticAnnotation oder scala.annotation.ClassfileAnnotation abgeleitet sind

```
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{}
```

Anmerkungen können dann mit der Reflection-API abgefragt werden.

```
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"))
```

Anmerkungen online lesen: https://riptutorial.com/de/scala/topic/3783/anmerkungen

Kapitel 4: Aufzählungen

Bemerkungen

Ein Ansatz mit `sealed trait` und `case objects` wird bevorzugt, da die Aufzählung von Scala einige Probleme aufweist:

1. Aufzählungen haben nach dem Löschen denselben Typ.
2. Der Compiler beschwert sich nicht über "Match ist nicht erschöpfend". Wenn die Groß- / Kleinschreibung übersehen wird, `scala.MatchError` in Laufzeit `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)
```

Vergleichen mit:

```
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 {
                                              ^
```

Eine ausführlichere Erklärung zu Scala Enumeration finden Sie in diesem Artikel .

Examples

Wochentage mit Scala Enumeration

Java-ähnliche Aufzählungen können durch Erweitern der Aufzählung erstellt werden .

```
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
```

Es ist auch möglich, einen von Menschen lesbaren Namen für Werte in einer Aufzählung

hinzuzufügen:

```
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
```

Beachten Sie das nicht so typsichere Verhalten, bei dem verschiedene Enumerationen als derselbe Instanztyp ausgewertet werden können:

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

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

Verwendung versiegelter Eigenschaften und Fallobjekte

Eine Alternative zur Erweiterung der `Enumeration` besteht in der Verwendung `sealed` Fallobjekte:

```
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
}
```

Das `sealed` Schlüsselwort garantiert, dass das Merkmal `WeekDay` nicht in einer anderen Datei erweitert werden kann. Dadurch kann der Compiler bestimmte Annahmen treffen, einschließlich, dass alle möglichen Werte von `WeekDay` bereits aufgelistet sind.

Ein Nachteil ist, dass Sie mit dieser Methode keine Liste aller möglichen Werte abrufen können. Um eine solche Liste zu erhalten, muss diese explizit angegeben werden:

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

Fallklassen können auch eine `sealed` Eigenschaft erweitern. Daher können Objekte und

Fallklassen gemischt werden, um komplexe Hierarchien zu erstellen:

```
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
}
```

Ein weiterer Nachteil besteht darin, dass es nicht möglich ist, auf den Variablennamen der Aufzählung eines `sealed` Objekts zuzugreifen oder danach zu suchen. Wenn Sie einen Namen für jeden Wert benötigen, muss dieser manuell definiert werden:

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

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

Oder nur:

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

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

Verwendung von versiegelten Merkmals- und Fallobjekten und allValues-Makro

Dies ist nur eine Erweiterung der versiegelten Merkmalsvariante, bei der ein Makro zur Kompilierzeit einen Satz mit allen Instanzen generiert. Dadurch wird der Nachteil vermieden, dass ein Entwickler der Aufzählung einen Wert hinzufügen kann, aber vergessen, ihn dem Satz allElements hinzuzufügen.

Diese Variante eignet sich besonders für große Aufzählungen.

```
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]
}
```

Damit dies funktioniert, benötigen Sie dieses Makro:

```
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(_))
        )
      }
    }
  }
}
```

Aufzählungen online lesen: https://riptutorial.com/de/scala/topic/1499/aufzahlungen

Kapitel 5: Benutzerdefinierte Funktionen für Hive

Examples

Eine einfache Hive-UDF in 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)))
```

Benutzerdefinierte Funktionen für Hive online lesen:
https://riptutorial.com/de/scala/topic/8241/benutzerdefinierte-funktionen-fur-hive

Kapitel 6: Best Practices

Bemerkungen

> Bevorzugen Sie Werte, unveränderliche Objekte und Methoden ohne Nebenwirkungen. Erstmal nach ihnen greifen. Verwenden Sie Variablen, veränderliche Objekte und Methoden mit Nebenwirkungen, wenn Sie einen bestimmten Bedarf und eine entsprechende Begründung haben.

- *Programmierung in Scala* von Odersky, Spoon und Venners

Es gibt weitere Beispiele und Richtlinien in dieser Präsentation von Odersky.

Examples

Halte es einfach

Machen Sie einfache Aufgaben nicht zu kompliziert. Meistens benötigen Sie nur:

- algebraische Datentypen
- strukturelle Rekursion
- `flatMap` api (`map` , `flatMap` , `fold`)

Es gibt viele komplizierte Dinge in Scala, wie zum Beispiel:

- `Cake pattern` oder `Reader Monad` für Abhängigkeitsinjektion.
- Beliebige Werte als `implicit` Argumente übergeben.

Diese Dinge sind für Neuankömmlinge nicht klar: Vermeiden Sie die Verwendung, bevor Sie sie verstehen. Die Verwendung fortschrittlicher Konzepte ohne wirkliche Notwendigkeit verschleiert den Code und macht ihn *weniger* wartbar.

Packen Sie nicht zu viel in einen Ausdruck.

- Finden Sie aussagekräftige Namen für Berechnungseinheiten.
- Verwenden Sie `for` Comprehensions oder `map` Berechnungen miteinander zu kombinieren.

Nehmen wir an, Sie haben so etwas:

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

```
}
```

Wenn alle Funktionen zurückkehren `Either` oder eine andere `Validation` -ähnlichen Art, können Sie schreiben:

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

Ziehe einen funktionalen Stil vor

Standardmäßig:

- Verwenden Sie möglichst nicht `val` , sondern `var` . Auf diese Weise können Sie eine Reihe funktionaler Hilfsprogramme, einschließlich der Arbeitsverteilung, nahtlos nutzen.
- Verwenden Sie `recursion` und `comprehensions` , nicht Schleifen.
- Verwenden Sie unveränderliche Sammlungen. Dies ist eine Korrelation zur Verwendung von `val` wann immer dies möglich ist.
- Konzentrieren Sie sich auf Datentransformationen, CQRS-Logik und nicht auf CRUD.

Es gibt gute Gründe, sich für einen nicht funktionalen Stil zu entscheiden:

- `var` kann für den lokalen Zustand verwendet werden (z. B. innerhalb eines Schauspielers).
- `mutable` führt in bestimmten Situationen zu einer besseren Leistung.

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

Kapitel 7: Betreiber in Scala

Examples

Eingebaute Operatoren

Scala verfügt über die folgenden integrierten Operatoren (Methoden / Sprachelemente mit vordefinierten Vorrangregeln):

Art	Symbol	Beispiel
Rechenzeichen	`+ - * / %`	`a + b`
Beziehungsoperatoren	`== != > < >= <=`	`a > b`
Logische Operatoren	`&& & || | !`	`a && b`
Bitweise Operatoren	`& | ^ ~ << >> >>>`	`a & b` , `~a` , `a >>> b`
Zuweisungsoperatoren	`= += -= *= /= %= <<= >>= &= ^= |=`	`a += b`

Scala-Operatoren haben dieselbe Bedeutung wie in Java

Hinweis : Methoden mit der Endung `:` bindet nach rechts (und nach rechts assoziativ), sodass der Aufruf mit `list.::(value)` als `value :: list` mit Operatorsyntax geschrieben werden kann. (`1 :: 2 :: 3 :: Nil` ist das Gleiche wie `1 :: (2 :: (3 :: Nil))`)

Überladung des Bedieners

In Scala können Sie Ihre eigenen Operatoren definieren:

```
class Team {
   def +(member: Person) = ...
}
```

Mit den obigen Definitionen können Sie es wie folgt verwenden:

```
ITTeam + Jack
```

oder

```
ITTeam.+(Jack)
```

Um unäre Operatoren zu definieren, können Sie `unary_` . ZB `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

Vorrang des Bedieners

Kategorie	Operator	Assoziativität
Postfix	`() []`	Links nach rechts
Unary	`! ~`	Rechts nach links
Multiplikativ	`* / %`	Links nach rechts
Zusatzstoff	`+ -`	Links nach rechts
Verschiebung	`>> >>> <<`	Links nach rechts
Relational	`> >= < <=`	Links nach rechts
Gleichberechtigung	`== !=`	Links nach rechts
Bitweise und	`&`	Links nach rechts
Bitweises xor	`^`	Links nach rechts
Bitweise oder	`|`	Links nach rechts
Logisch und	`&&`	Links nach rechts
Logisch oder	`||`	Links nach rechts
Zuordnung	`= += -= *= /= %= >>= <<= &= ^= | =`	Rechts nach links
Komma	`,`	Links nach rechts

Die Programmierung in Scala gibt die folgende Gliederung basierend auf dem 1. Zeichen des Operators an. ZB `>` ist das 1. Zeichen im Operator `>>>` :

Operator
(alle anderen Sonderzeichen)
`* / %`
`+ -`
`:`

Operator
= !
< >
&
^
|
(alle Buchstaben)
(alle Zuweisungsoperatoren)

Die einzige Ausnahme von dieser Regel betrifft *Zuweisungsoperatoren* , z. B. `+=` , `*=` usw. Wenn ein Operator mit einem Gleichheitszeichen (=) endet und keiner der Vergleichsoperatoren ist `<=` , `>=` , `==` bzw. `!=` der Vorrang des Operators ist dann dasselbe wie eine einfache Zuweisung. Mit anderen Worten, niedriger als bei jedem anderen Bediener.

Betreiber in Scala online lesen: https://riptutorial.com/de/scala/topic/6604/betreiber-in-scala

Kapitel 8: Currying

Syntax

- aFunction (10) _ // Using '_' Sagt dem Compiler, dass alle Parameter in den restlichen Parametergruppen aktuell sind.
- nArityFunction.curried // Konvertiert eine N-Arity-Funktion in eine entsprechende aktuelle Version
- anotherFunction (x) (_: String) (z) // Ein beliebiger Parameter wird geändert. Es muss der Typ ausdrücklich angegeben werden.

Examples

Ein konfigurierbarer Multiplikator als Curryfunktion

```
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
```

Mehrere Parametergruppen verschiedener Typen, aktuelle Parameter für beliebige Positionen

```
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"
```

Eine Funktion mit einer einzelnen Parametergruppe erstellen

```
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

Definieren wir eine Funktion von 2 Argumenten:

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

Currying `add` verwandelt es in eine Funktion, die **ein** `Int` und eine **Funktion** zurückgibt (von **einem** `Int` zu einem `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)
```

Sie können dieses Konzept auf jede Funktion anwenden, die mehrere Argumente akzeptiert. Das Currying einer Funktion, die mehrere Argumente übernimmt, wandelt sie in eine Reihe von Anwendungen von Funktionen um, die **ein** Argument annehmen:

```
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 nach Wikipedia ,

> ist die Technik zum Übersetzen der Auswertung einer Funktion, die mehrere Argumente zur Auswertung einer Funktionsfolge benötigt.

Konkret bedeutet dies im Zusammenhang mit Scala-Typen im Zusammenhang mit einer Funktion, die zwei Argumente enthält (die Arität 2 hat) die Konvertierung von

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

zu

```
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`
```

Für arity-2-Funktionen können wir die Curry-Funktion folgendermaßen schreiben:

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

```
}
```

Verwendungszweck:

```
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 gibt uns einige Sprachfunktionen, die dabei helfen:

1. Sie können Curried-Funktionen als Methoden schreiben. so kann `curriedF` geschrieben werden als:

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

2. Sie können entladen (dh von `A => B => C` zu `(A, B) => C`) gehen, indem Sie eine Standardbibliotheksmethode verwenden: `Function.uncurried`

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

Wann sollten Sie Currying verwenden?

Currying *ist die Technik zum Übersetzen der Auswertung einer Funktion, die mehrere Argumente zur Auswertung einer Funktionsfolge mit jeweils einem einzigen Argument verwendet* .

Dies ist normalerweise nützlich, wenn zum Beispiel:

1. Verschiedene Argumente einer Funktion werden **zu unterschiedlichen Zeiten** berechnet. *(Beispiel 1)*
2. Verschiedene Argumente einer Funktion werden **nach verschiedenen Ebenen der Anwendung** berechnet. *(Beispiel 2)*

Beispiel 1

Nehmen wir an, dass das jährliche Jahreseinkommen eine Funktion ist, die sich aus dem Einkommen und einem Bonus zusammensetzt:

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

Die aktuelle Version der obigen 2-Arity-Funktion lautet:

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

Beachten Sie in der obigen Definition, dass der Typ auch als angesehen / geschrieben werden kann:

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

Nehmen wir an, dass der jährliche Einkommensanteil im Voraus bekannt ist:

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

Und irgendwann ist der Bonus bekannt:

```
partialTotalYearlyIncome(100)
```

Beispiel 2

Nehmen wir an, dass die Automobilherstellung die Anwendung von Felgen und Karosserien umfasst:

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

Diese Teile werden von verschiedenen Fabriken angewendet:

```
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..")
}
```

Beachten Sie, dass die `CarWheelsFactory` oben die Fahrzeugherstellungsfunktion `CarWheelsFactory` und nur die Räder verwendet.

Der Autoherstellungsprozess wird dann die folgende Form annehmen:

```
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)
```

Ein echter Einsatz von Currying.

Was wir haben, ist eine Liste mit Kreditkarten, und wir möchten die Prämien für alle Karten berechnen, die das Kreditkartenunternehmen auszahlen muss. Die Prämien selbst hängen von der Gesamtzahl der Kreditkarten ab, so dass das Unternehmen sie entsprechend anpasst.

Wir haben bereits eine Funktion, die die Prämie für eine einzelne Kreditkarte berechnet und die

Gesamtzahl der von der Firma ausgegebenen Karten berücksichtigt:

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

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

Ein vernünftiger Ansatz für dieses Problem wäre nun, jede Kreditkarte einer Prämie zuzuordnen und auf eine Summe zu reduzieren. Etwas wie das:

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

Dem Compiler wird dies jedoch nicht gefallen, da `CreditCard.getPremium` zwei Parameter erfordert. Teilanwendung zur Rettung! Wir können die Gesamtanzahl der Kreditkarten teilweise anwenden und diese Funktion verwenden, um die Kreditkarten ihren Prämien zuzuordnen. Wir müssen nur die `getPremium` Funktion curry `getPremium` , indem wir sie so ändern, dass mehrere Parameterlisten verwendet werden, und wir können `getPremium` .

Das Ergebnis sollte ungefähr so aussehen:

```
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
```

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

Kapitel 9: Dynamischer Aufruf

Einführung

Mit Scala können Sie dynamisches Aufrufen verwenden, wenn Sie Methoden aufrufen oder auf Felder eines Objekts zugreifen. Anstatt diese tief in die Sprache zu integrieren, wird dies durch Umschreiben von Regeln erreicht, die denen impliziter Konvertierungen ähnlich sind, die durch das Merkmalsmerkmal [`scala.Dynamic`] [Dynamic scaladoc] ermöglicht werden. Auf diese Weise können Sie die Fähigkeit zum dynamischen Hinzufügen von Eigenschaften zu Objekten in dynamischen Sprachen usw. emulieren. [Dynamic scaladoc]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

Syntax

- Klasse Foo erweitert Dynamic
- foo.field
- foo.field = Wert
- foo.method (args)
- foo.method (namedArg = x, y)

Bemerkungen

Um Subtypen zu erklären `Dynamic` , die Sprache - Funktion `dynamics` muss aktiviert werden, entweder durch den Import `scala.language.dynamics` oder durch die `-language:dynamics` - Compiler - Option. Benutzer dieses `Dynamic` , die keine eigenen Subtypen definieren, müssen dies nicht aktivieren.

Examples

Feldzugriffe

Diese:

```
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) = ???
}
```

ermöglicht den einfachen Zugriff auf Felder:

```
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.
```

Methodenaufrufe

Diese:

```
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)
  }
}
```

erlaubt Aufrufe von Methoden mit und ohne benannte Parameter:

```
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)
```

Interaktion zwischen Feldzugriff und Aktualisierungsmethode

Etwas uninteressant (aber auch der einzige vernünftige Weg, damit es funktioniert):

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

ist äquivalent zu:

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

während

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

ist immer noch

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

Es ist wichtig, sich dessen bewusst zu sein, andernfalls könnte es unbemerkt schleichen und seltsame Fehler verursachen.

Dynamischer Aufruf online lesen: https://riptutorial.com/de/scala/topic/8296/dynamischer-aufruf

Kapitel 10: Einzelne abstrakte Methodentypen (SAM-Typen)

Bemerkungen

Einzelne abstrakte Methoden sind in Java 8 eingeführte Typen, die genau ein abstraktes Mitglied haben.

Examples

Lambda-Syntax

HINWEIS: Dies ist nur in Scala 2.12+ verfügbar (und in aktuellen 2.11.x-Versionen mit den `-Xexperimental -Xfuture` Compilerflags)

Ein SAM-Typ kann mit einem Lambda implementiert werden:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

Der Typ kann optional weitere nicht abstrakte Elemente enthalten:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Einzelne abstrakte Methodentypen (SAM-Typen) online lesen:
https://riptutorial.com/de/scala/topic/3664/einzelne-abstrakte-methodentypen--sam-typen-

Kapitel 11: Extraktoren

Syntax

- val-extractor (extrahierterWert1, _ / * ignorierter zweiter extrahierter Wert * /) = valueToBeExtracted
- valueToBeExtracted match {case extractor (extrahierterWert1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- object Foo {def unapply (foo: Foo): Option [String] = Some (foo.x); }

Examples

Tupel-Extraktoren

`x` und `y` werden aus dem Tupel extrahiert:

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

Um einen Wert zu ignorieren, verwenden Sie _ :

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

So entpacken Sie einen Extraktor:

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

Beachten Sie, dass Tupel eine maximale Länge von 22 haben und somit `._1` bis `._22` funktionieren (vorausgesetzt, das Tupel hat mindestens diese Größe).

Tupel-Extraktoren können verwendet werden, um symbolische Argumente für wörtliche Funktionen bereitzustellen:

```
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 Extraktoren

Eine Fallklasse ist eine Klasse mit einer Menge Standardcode, der automatisch enthalten ist. Ein Vorteil davon ist, dass Scala die Verwendung von Extraktoren mit Fallklassen vereinfacht.

```
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
```

Zu diesem Zeitpunkt sind sowohl `n` als auch `a` `val` im Programm, auf die als solche zugegriffen werden kann: Sie sollen aus p "extrahiert" worden sein. Auch weiterhin:

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

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

Hier sehen wir zwei wichtige Dinge:

- Die Extraktion kann auf tiefen Ebenen erfolgen: Eigenschaften von verschachtelten Objekten können extrahiert werden.
- Nicht alle Elemente *müssen* extrahiert werden. Das Platzhalterzeichen _ gibt an, dass dieser bestimmte Wert alles sein kann und ignoriert wird. Es wird kein `val` erstellt.

Dies kann insbesondere das Abgleichen von Sammlungen erleichtern:

```
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))
})
```

Hier haben wir Code, der den Extraktor verwendet, um explizit zu prüfen, ob `person` ein `Person` ist, und sofort die Variablen herauszuholen, die uns wichtig sind: `n` und `a` .

Unangenehm - benutzerdefinierte Extraktoren

Eine benutzerdefinierte Extraktion kann geschrieben werden, indem die `unapply` Methode implementiert und ein Wert vom Typ `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"
```

Der Rückgabetyp von `unapply` kann etwas anderes als `Option` , sofern der zurückgegebene Typ die Methoden `get` und `isEmpty` . In diesem Beispiel wird `Bar` mit diesen Methoden definiert und gibt eine Instanz von `Bar` `unapply` zurück:

```
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"
```

Der Rückgabetyp von `unapply` kann auch ein `Boolean` `unapply` ist ein Sonderfall, der `get` `isEmpty` Anforderungen von `get` und `isEmpty` nicht enthält. Beachten Sie jedoch in diesem Beispiel, dass `DivisibleByTwo` ein Objekt und keine Klasse ist und keinen Parameter verwendet (daher kann dieser Parameter nicht gebunden werden):

```
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
```

Denken `unapply` daran, dass das Begleitobjekt einer Klasse `unapply` nicht in die Klasse geht. Das obige Beispiel ist klar, wenn Sie diesen Unterschied verstehen.

Extraktor-Infix-Notation

Wenn eine Fallklasse genau zwei Werte hat, kann der Extraktor in Infix-Notation verwendet

werden.

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

Jeder Extraktor, der ein 2-Tupel zurückgibt, kann auf diese Weise arbeiten.

```
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-Extraktoren

Ein regulärer Ausdruck mit gruppierten Teilen kann als Extraktor verwendet werden:

```
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
```

Beachten Sie, dass ein `MatchError` zur Laufzeit `MatchError` wird, wenn er nicht übereinstimmt:

```
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 Extraktoren

Das Verhalten des Extraktors kann verwendet werden, um aus ihrer Eingabe beliebige Werte abzuleiten. Dies kann in Szenarien hilfreich sein, in denen Sie die Ergebnisse einer Umwandlung für den Fall berücksichtigen können, dass die Umwandlung erfolgreich ist.

Betrachten Sie als Beispiel die verschiedenen Benutzernamenformate, die in einer Windows-Umgebung verwendet werden können :

```
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
}
```

Tatsächlich ist es möglich, einen Extraktor mit beiden Verhaltensweisen zu erstellen, indem die Typen erweitert werden, auf die er passen kann:

```
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
  }
}
```

Im Allgemeinen sind Extraktoren einfach eine bequeme Neuformulierung des `Option` , die auf Methoden mit Namen wie `tryParse` :

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

Extraktoren online lesen: https://riptutorial.com/de/scala/topic/930/extraktoren

Kapitel 12: Fallklassen

Syntax

- Fallklasse Foo () // Fallklassen ohne Parameter müssen eine leere Liste haben
- Fallklasse Foo (a1: A1, ..., aN: AN) // Eine Fallklasse mit den Feldern a1 ... aN erstellen
- case object Bar // Eine Singleton-Case-Klasse erstellen

Examples

Fallklassengleichheit

Ein Feature, das von Fallklassen frei bereitgestellt wird, ist eine automatisch generierte `equals` Methode, die die Wertegleichheit aller einzelnen `equals` prüft, anstatt nur die Referenzgleichheit der Objekte zu prüfen.

Bei gewöhnlichen Klassen:

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

Mit Fallklassen:

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

Generierte Code-Artefakte

Der `case` Modifikator bewirkt , dass die Scala - Compiler automatisch gemeinsamen Standardcode für die Klasse generieren. Die manuelle Implementierung dieses Codes ist langwierig und eine Fehlerquelle. Die folgende Fallklassendefinition:

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

... wird der folgende Code automatisch generiert:

```
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)
}
```

Der `case` Modifikator erzeugt auch ein Begleitobjekt:

```
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))
}
```

Wenn auf ein angelegtes `object` , der `case` hat Modifikator ähnlich (wenn auch weniger dramatisch) Effekte. Hier sind die wichtigsten Vorteile eine `toString` Implementierung und ein über alle Prozesse hinweg konsistenter `hashCode` . Beachten Sie, dass Fallobjekte (korrekt) eine Referenzgleichheit verwenden:

```
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"
}
```

Es ist immer noch möglich, manuell Methoden zu implementieren , die sonst durch den zur Verfügung gestellt werden würde `case` Modifikator sowohl in der Klasse selbst und seine Begleiter Objekt.

Fallklassen-Grundlagen

Im Vergleich zu regulären Klassen bietet die Notation von Fallklassen mehrere Vorteile:

- Alle Konstruktorargumente sind `public` und können auf initialisierte Objekte zugegriffen werden (normalerweise ist dies nicht der Fall, wie hier gezeigt):

```
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"
```

- Es stellt eine Implementierung für die folgenden Methoden: `toString` , `equals` , `hashCode` (basierend auf Eigenschaften), `copy` , `apply` und `unapply` :

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

- Es bietet einen praktischen Mechanismus für die Mustererkennung:

```
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.")
}
```

Fallklassen und Unveränderlichkeit

Der Scala-Compiler stellt jedem Argument in der Parameterliste standardmäßig den `val` . Dies bedeutet, dass Fallklassen standardmäßig unveränderlich sind. Jeder Parameter erhält eine Zugriffsmethode, es gibt jedoch keine Mutator-Methoden. Zum Beispiel:

```
case class Foo(i: Int)

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

Das Deklarieren eines Parameters in einer Fallklasse als `var` überschreibt das Standardverhalten

und macht die Fallklasse veränderbar:

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

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

Eine andere Instanz, in der eine Fallklasse "veränderbar" ist, ist der veränderliche Wert in der Fallklasse:

```
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)
```

Beachten Sie, dass die hier vorkommende 'Mutation' sich in der Map befindet, auf die `m` zeigt, nicht auf `m` selbst. Wenn also ein anderes Objekt `m` als Member hatte, würde es auch die Änderung sehen. Beachten Sie, wie im folgenden Beispiel die `instanceA` geändert wird, `instanceB` auch die `instanceB` B geändert wird:

```
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)
```

Erstellen Sie eine Kopie eines Objekts mit bestimmten Änderungen

Fallklassen bieten eine `copy` , mit der ein neues Objekt erstellt wird, das bei bestimmten Änderungen dieselben Felder wie das alte Objekt verwendet.

Wir können diese Funktion verwenden, um ein neues Objekt aus einem vorherigen Objekt zu erstellen, das einige der gleichen Merkmale aufweist. Diese einfache Fallklasse veranschaulicht diese Funktion:

```
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 diesem Beispiel sehen wir, dass die beiden Objekte ähnliche Merkmale aufweisen (`grade = A1` , `subject = Math`), es sei denn, sie wurden in der Kopie (`firstName` und `lastName`) angegeben.

Einzelelement-Gehäuseklassen für die Typsicherheit

Um die Typsicherheit zu erreichen, möchten wir manchmal die Verwendung primitiver Typen in unserer Domäne vermeiden. Stellen Sie sich zum Beispiel eine `Person` mit einem `name` . Normalerweise würden wir den `name` als `String` kodieren. Allerdings wäre es nicht schwer sein , ein mischen `String` einer darstellt `Person` ,s `name` mit einem `String` repräsentiert eine Fehlermeldung:

```
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!
```

Um solche Fehler zu vermeiden, können Sie die Daten folgendermaßen codieren:

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

und jetzt wird unser Code nicht kompiliert, wenn wir `PersonName` mit `ErrorMessage` oder sogar einen normalen `String` mischen.

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

Dies verursacht jedoch einen geringen Laufzeit-Overhead, da wir nun `String` s in / aus seinen `PersonName` Containern `PersonName` . Um dies zu vermeiden, können Klassen für `PersonName` und `ErrorMessage` Werte erstellt werden:

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

Fallklassen online lesen: https://riptutorial.com/de/scala/topic/1022/fallklassen

Kapitel 13: Fehlerbehandlung

Examples

Versuchen

Verwenden Sie Try with `map` , `getOrElse` und `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)
```

Verwenden `Try` with pattern match:

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

Entweder

Unterschiedliche Datentypen für Fehler / Erfolg

```
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")
    }
}
```

Musterabgleich bei jedem Wert

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

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

Wandeln Sie einen der beiden Werte in eine Option um

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

Möglichkeit

Von der Verwendung von `null` wird dringend abgeraten, es sei denn, es wird mit älterem Java-Code interagiert, der `null` erwartet. Stattdessen sollte `Option` verwendet werden, wenn das Ergebnis einer Funktion entweder etwas (`Some`) oder nichts (`None`) ist.

Ein try-catch-Block ist eher für die Fehlerbehandlung geeignet, aber wenn die Funktion legitimerweise nichts `Option` ist `Option` angemessen und einfach zu verwenden.

Eine `Option[T]` kann entweder `Some(value)` (enthält einen Wert vom Typ `T`) oder `None` :

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

Wenn keine Person gefunden wird, kann `None` zurückgegeben werden. Andernfalls wird ein Objekt vom Typ `Some` das ein `Person` Objekt enthält. Im Folgenden werden Möglichkeiten beschrieben, mit einem Objekt vom Typ `Option` .

Musterabgleich

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

Map und getOrElse verwenden

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

Falte verwenden

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

Konvertieren nach Java

Wenn Sie einen `Option` für die Interoperabilität in einen nullfähigen Java-Typ konvertieren müssen:

```
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
```

Fehler in der Zukunft behandeln

Wenn eine `exception` innerhalb einer `Future` ausgelöst wird, können Sie (sollten) es verwenden, `recover` damit umzugehen.

Zum Beispiel,

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

val itWillBeAwesome: Future = runFuture
```

... wird eine `Exception` aus der `Future` werfen. Da wir jedoch mit einer hohen Wahrscheinlichkeit vorhersagen können, dass eine `Exception` vom Typ `FairlyStupidException` , können wir diesen Fall auf elegante Weise gezielt behandeln:

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

Wie Sie sehen, ist die `recover` Methode eine `PartialFunction` über die Domäne von All `Throwable` . Sie können also nur einige wenige Typen behandeln und dann den Rest in den Äther der Ausnahmebehandlung auf höheren Ebenen des `Future` Stacks gehen lassen.

Beachten Sie, dass dies dem Ausführen des folgenden Codes in einem nicht- `Future` Kontext ähnelt:

```
def runNotFuture: Unit = throw new FairlyStupidException

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

Es ist sehr wichtig, Ausnahmen zu behandeln, die in `Future` s generiert wurden, da sie meistens hinterlistiger sind. Sie bekommen in der Regel nicht alles ins Gesicht, weil sie in einem anderen Ausführungskontext und Thread ausgeführt werden und Sie daher nicht dazu aufgefordert werden, sie zu beheben, wenn sie auftreten, insbesondere wenn Sie nichts in den Protokollen oder das Verhalten des Befehls feststellen Anwendung.

Try-Catch-Klauseln verwenden

Neben funktionalen Konstrukten wie `Try` , `Option` und `Either` für die Fehlerbehandlung unterstützt Scala auch eine ähnliche Syntax wie die von Java, wobei eine Try-Catch-Klausel verwendet wird (die möglicherweise auch zum Schluss blockiert). Die catch-Klausel ist eine Musterübereinstimmung:

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

Konvertieren Sie Ausnahmen in entweder einen oder einen Optionstyp

Um Ausnahmen in die Typen " `Either` oder " `Option` zu konvertieren, können Sie Methoden verwenden, die in `scala.util.control.Exception` bereitgestellt `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 }
```

Fehlerbehandlung online lesen: https://riptutorial.com/de/scala/topic/910/fehlerbehandlung

Kapitel 14: Fortsetzungen Bibliothek

Einführung

Continuation Passing Style ist eine Form des Kontrollflusses, bei der der Rest der Berechnung als "Fortsetzung" -Argument an Funktionen übergeben wird. Die fragliche Funktion ruft diese Fortsetzung später auf, um die Programmausführung fortzusetzen. Eine Möglichkeit, sich eine Fortsetzung vorzustellen, ist die Schließung. Die Scala Fortsetzungen Bibliothek begrenzt Fortsetzungen in Form der Primitiven bringt `shift` / `reset` auf die Sprache.

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

Syntax

- reset {...} // Fortsetzungen erstrecken sich bis zum Ende des umschließenden Rücksetzblocks
- shift {...} // Erstellen Sie eine Fortsetzung, die aus dem Aufruf hervorgeht, und leitet sie an die Schließung weiter
- A @cpsParam [B, C] // Eine Berechnung, die eine Funktion A => B erfordert, um einen Wert von C zu erstellen
- @cps [A] // Alias für @cpsParam [A, A]
- @suspendable // Alias für @cpsParam [Einheit, Einheit]

Bemerkungen

`shift` und `reset` sind einfache Kontrollflussstrukturen, wie `Int.+` eine primitive Operation und `Long` ein primitiver Typ. Sie sind primitiver als beides, indem mit begrenzten Fortsetzungen tatsächlich fast alle Kontrollflussstrukturen erstellt werden können. Sie sind nicht sehr nützlich "out-of-the-box", aber sie leuchten wirklich, wenn sie in Bibliotheken zum Erstellen von umfangreichen APIs verwendet werden.

Fortsetzungen und Monaden sind ebenfalls eng miteinander verbunden. Es können Fortsetzungen in die Fortsetzungsmonade aufgenommen werden , und Monaden sind Fortsetzungen, da ihre `flatMap` Operation eine Fortsetzung als Parameter `flatMap` .

Examples

Rückrufe sind Kontinuationen

```
// 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)
```

```
  }}
}}
```

Das Funktionsargument für `readFile` ist eine Fortsetzung, indem `readFile` aufruft, um die Programmausführung fortzusetzen, nachdem es seine Aufgabe `readFile` hat.

Um zu kontrollieren, was leicht zur Callback-Hölle werden kann, verwenden wir die Fortsetzungsbibliothek.

```
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!
```

Erstellen von Funktionen, die Fortsetzung benötigen

Wird die `shift` außerhalb eines begrenzenden `reset` aufgerufen, können mit ihr Funktionen erstellt werden, die selbst Fortsetzungen innerhalb eines `reset` erzeugen. Es ist wichtig zu wissen, dass der `shift` -Typ nicht nur `(((A => B) => C) => A)` , sondern `(((A => B) => C) => (A @cpsParam[B, C]))` . Diese Anmerkung markiert, wo CPS-Transformationen benötigt werden. Funktionen, die die `shift` Funktion ohne `reset` aufrufen `shift` haben ihren Rückgabetyp mit dieser Anmerkung "infiziert".

Innerhalb eines `reset` scheint ein Wert von `A @cpsParam[B, C]` einen Wert von `A` , obwohl es wirklich nur so ist. Die Fortsetzung, die zum Abschließen der Berechnung erforderlich ist, hat den Typ `A => B` Der Code, der einer Methode folgt, die diesen Typ zurückgibt, muss `B . C` ist der "echte" Rückgabetyp, und nach der CPS-Umwandlung hat der Funktionsaufruf den Typ `C`

Nun das Beispiel aus dem Scaladoc der Bibliothek

```
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)
}
```

Hier speichert `ask` die Fortsetzung in einer Map, und später kann ein anderer Code diese "Sitzung" abrufen und das Ergebnis der Abfrage an den Benutzer weiterleiten. Auf diese Weise kann `go` tatsächlich eine asynchrone Bibliothek verwenden, während der Code wie normaler imperativer Code aussieht.

Fortsetzungen Bibliothek online lesen: https://riptutorial.com/de/scala/topic/8312/fortsetzungen-bibliothek

Kapitel 15: Funktion höherer Ordnung

Bemerkungen

Scala legt großen Wert darauf, Methoden und Funktionen als syntaktisch identisch zu behandeln. Aber unter der Haube sind sie unterschiedliche Konzepte.

Eine Methode ist ausführbarer Code und hat keine Wertdarstellung.

Eine Funktion ist eine tatsächliche Objektinstanz des Typs `Function1` (oder eines ähnlichen Typs einer anderen Art). Der Code ist in der `apply` Methode enthalten. Im Grunde fungiert es einfach als Wert, der herumgereicht werden kann.

Im Übrigen *ist* genau die Fähigkeit , Funktionen als Werte zu behandeln , was durch eine Sprache gemeint ist die Unterstützung für Funktionen höherer Ordnung aufweist. Funktionsinstanzen sind Scalas Ansatz zur Implementierung dieser Funktion.

Eine tatsächliche Funktion höherer Ordnung ist eine Funktion, die entweder einen Funktionswert als Argument verwendet oder einen Funktionswert zurückgibt. Da in Scala alle Operationen Methoden sind, ist es allgemeiner, über Methoden nachzudenken, die Funktionsparameter empfangen oder zurückgeben. `map` , wie in `Seq` definiert, kann daher als "Funktion höherer Ordnung" betrachtet werden, da es sich bei seinem Parameter um eine Funktion handelt, aber es handelt sich nicht um eine Funktion. Es ist eine Methode.

Examples

Methoden als Funktionswerte verwenden

Der Scala-Compiler konvertiert Methoden automatisch in Funktionswerte, um sie an Funktionen höherer Ordnung zu übergeben.

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

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

Im obigen Beispiel ist `MyObject.mapMethod` kein Funktionsaufruf, sondern wird stattdessen als Wert an `map` . In der Tat *erfordert* `map` einen Funktionswert, der an sie übergeben wird, wie aus der Signatur ersichtlich ist. Die Signatur für die `map` einer `List[A]` (eine Liste von Objekten des Typs `A`) lautet:

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

Der Teil `f: (A) => B` gibt an, dass der Parameter für diesen Methodenaufruf eine Funktion ist, die

ein Objekt vom Typ `A` übernimmt und ein Objekt vom Typ `B` zurückgibt. `A` und `B` sind beliebig definiert. Wenn wir zum ersten Beispiel zurückkehren, können wir sehen, dass `mapMethod` ein `Int` (das `A`) übernimmt und einen `String` (der `B` entspricht) zurückgibt. Daher ist `mapMethod` ein gültiger Funktionswert, der an die `map` . Wir könnten den gleichen Code wie folgt umschreiben:

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

Dadurch wird der Funktionswert angezeigt, was für einfache Funktionen Klarheit schaffen kann.

Funktionen höherer Ordnung (Funktion als Parameter)

Eine Funktion höherer Ordnung kann im Gegensatz zu einer Funktion erster Ordnung eine von drei Formen haben:

- Einer oder mehrere seiner Parameter ist eine Funktion und gibt einen Wert zurück.
- Es gibt eine Funktion zurück, aber keiner seiner Parameter ist eine Funktion.
- Beide der oben genannten: Einer oder mehrere seiner Parameter ist eine Funktion und gibt eine Funktion zurück.

```
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
    }
```

Hier verwendet die Map-Funktion eine `getFullName(n._1,n._2)` Funktion `getFullName(n._1,n._2)` als Parameter. Dies wird als **HOF (Funktion höherer Ordnung) bezeichnet.**

Argumente faul Auswertung

Scala unterstützt die verzögerte Auswertung von Funktionsargumenten mithilfe der Notation: `def func(arg: => String)` . Ein solches Funktionsargument kann ein reguläres `String` Objekt oder eine Funktion höherer Ordnung mit dem Rückgabetyp `String` annehmen. Im zweiten Fall würde das Funktionsargument beim Zugriff auf Werte ausgewertet.

Bitte sehen Sie sich das Beispiel an:

```
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` Aufruf von `smartMediator` würde None value zurückgeben und die Nachricht `"Applying mediator"` `smartMediator` .

`dumbMediator` Aufruf von `dumbMediator` würde None value zurückgeben und die Nachricht `"Calculating expensive data! Applying mediator"` `dumbMediator` .

Lazy Evaluation kann äußerst nützlich sein, wenn Sie den Aufwand für die Berechnung teurer Argumente optimieren möchten.

Funktion höherer Ordnung online lesen: https://riptutorial.com/de/scala/topic/1642/funktion-hoherer-ordnung

Kapitel 16: Funktionen

Bemerkungen

Scala hat erstklassige Funktionen.

Unterschied zwischen Funktionen und Methoden:

Eine Funktion ist keine Methode in Scala: Funktionen sind ein Wert und können als solcher zugewiesen werden. Methoden, die mit `def` erstellt wurden, müssen hingegen zu einer Klasse, einem Merkmal oder einem Objekt gehören.

- Funktionen werden zu einer Klasse kompiliert, die eine Eigenschaft (wie `Function1`) zur Kompilierzeit erweitert, und zur Laufzeit in einen Wert instanziiert. Methoden dagegen sind Mitglieder ihrer Klasse, ihres Merkmals oder Objekts und existieren nicht außerhalb davon.
- Eine Methode kann in eine Funktion konvertiert werden, eine Funktion kann jedoch nicht in eine Methode konvertiert werden.
- Methoden können eine Typparametrierung haben, Funktionen dagegen nicht.
- Methoden können Parameter-Standardwerte haben, Funktionen dagegen nicht.

Examples

Anonyme Funktionen

Anonyme Funktionen sind Funktionen, die definiert sind, aber keinen Namen erhalten.

Das Folgende ist eine anonyme Funktion, die zwei Ganzzahlen aufnimmt und die Summe zurückgibt.

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

Der resultierende Ausdruck kann einem `val` zugewiesen werden:

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

Anonyme Funktionen werden hauptsächlich als Argumente für andere Funktionen verwendet. Beispielsweise erwartet die `map` Funktion in einer Sammlung eine andere Funktion als Argument:

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

Die Typen der Argumente der anonymen Funktion können weggelassen werden: Die Typen

werden automatisch abgeleitet :

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

Wenn es nur ein Argument gibt, können die Klammern um dieses Argument weggelassen werden:

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

Unterstreicht die Kurzschrift

Es gibt eine noch kürzere Syntax, für die keine Namen für die Argumente erforderlich sind. Das obige Snippet kann geschrieben werden:

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

_ stellvertretend für die anonymen Funktionsargumente. Bei einer anonymen Funktion mit mehreren Parametern verweist jedes Vorkommen von _ auf ein anderes Argument. Zum Beispiel sind die beiden folgenden Ausdrücke gleichwertig:

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

Bei Verwendung dieser Abkürzung kann jedes von positional _ dargestellte Argument nur einmal und in derselben Reihenfolge referenziert werden.

Anonyme Funktionen ohne Parameter

Lassen Sie die Parameterliste leer, um einen Wert für eine anonyme Funktion zu erstellen, für die keine Parameter verwendet werden.

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

Zusammensetzung

Die Funktionszusammenstellung ermöglicht es zwei Funktionen zu bedienen und als eine einzige Funktion betrachtet zu werden. Mathematisch ausgedrückt wird bei einer Funktion `f(x)` und einer Funktion `g(x)` die Funktion `h(x) = f(g(x))` .

Wenn eine Funktion kompiliert wird, wird sie zu einem mit `Function1` Typ kompiliert. Scala stellt in der `Function1` Implementierung zwei Methoden für die Komposition `andThen` : `andThen` und `compose` . Die `compose` passt zu der obigen mathematischen Definition wie folgt:

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

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

Das `andThen` (denke, `h(x) = g(f(x))`) hat ein eher 'DSL-ähnliches' Gefühl:

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

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

Eine neue anonyme Funktion wird zugewiesen, die über `f` und `g` . Diese Funktion ist in beiden Fällen an die neue Funktion `h` gebunden.

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

Wenn entweder `f` oder `g` über einen Nebeneffekt wirkt, führt der Aufruf von `h` dazu, dass alle Nebeneffekte von `f` und `g` in der Reihenfolge auftreten. Gleiches gilt für alle veränderlichen Zustandsänderungen.

Beziehung zu PartialFunctions

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

Jede `PartialFunction` mit einem Argument ist auch eine `Function1` . Dies ist im formalen mathematischen Sinne kontraintuitiv, passt jedoch besser zu objektorientiertem Design. Aus diesem Grund muss `Function1` keine konstante `true isDefinedAt` Methode `isDefinedAt` .

Um eine Teilfunktion (die auch eine Funktion ist) zu definieren, verwenden Sie folgende Syntax:

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

Weitere Informationen finden Sie unter PartialFunctions .

Funktionen online lesen: https://riptutorial.com/de/scala/topic/477/funktionen

Kapitel 17: Für Ausdrücke

Syntax

- für {clauses} body
- für {Klauseln} ergeben Körper
- für (Klauseln) Körper
- für (Klauseln) Ertragskörper

Parameter

Parameter	Einzelheiten
zum	Erforderliches Schlüsselwort zur Verwendung einer for-Schleife / eines Verständnisses
Klauseln	Die Iteration und Filter, über die das funktioniert.
Ausbeute	Verwenden Sie dies, wenn Sie eine Sammlung erstellen oder ausgeben möchten. Die Verwendung von `yield` bewirkt, dass der Rückgabetyp von `for` eine Sammlung statt `Unit` .
Karosserie	Der Rumpf des for-Ausdrucks, der bei jeder Iteration ausgeführt wird.

Examples

Basic für Schleife

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

Dies zeigt, wie Sie eine Variable `x` von `1` bis `10` iterieren und mit diesem Wert etwas tun. Die Art der Rückgabe `for` Verständnis ist `Unit` .

Grundlegendes zum Verständnis

Dies demonstriert einen Filter für eine for-Schleife und die Verwendung von `yield` , um ein 'Sequenzverständnis' zu erzeugen:

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

Die Ausgabe hierfür ist:

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

Ein Verständnis ist hilfreich, wenn Sie eine neue Sammlung basierend auf der Iteration und ihren Filtern erstellen müssen.

Für Schleife geschachtelt

Dies zeigt, wie Sie mehrere Variablen durchlaufen können:

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

(Beachten Sie, dass `to` eine Infix-Operator-Methode ist, die einen inklusiven Bereich zurückgibt. Siehe die Definition hier .)

Dadurch wird die Ausgabe erstellt:

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

Beachten Sie, dass dies ein äquivalenter Ausdruck ist, der Klammern anstelle von Klammern verwendet:

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

Um alle Kombinationen in einem einzigen Vektor zusammenzufassen, können wir das Ergebnis `yield` und es auf einen `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))
```

Monadisch für Verständnis

Wenn Sie mehrere Objekte mit monadischen Typen haben, können wir die Werte mit einem 'zum Verständnis' kombinieren:

```
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)
```

Der Rückgabetyp dieses Blocks ist `Unit` .

Wenn die Objekte vom *gleichen* monadischen Typ `M` (z. B. `Option`), wird bei Verwendung von `yield` ein Objekt vom Typ `M` anstelle von `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))
```

Beachten Sie, dass das `yield` *nicht* im ursprünglichen Beispiel verwendet werden kann, wenn eine Mischung aus monadischen Typen (`Option` und `List`) vorhanden ist. Wenn Sie dies versuchen, wird ein Fehler beim Kompilieren des Typs auftreten.

Durchlaufen Sie Sammlungen mit einer for-Schleife

Dies zeigt, wie jedes Element einer Map gedruckt wird

```
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
```

Dies veranschaulicht, wie jedes Element einer Liste gedruckt wird

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

Demugaring für das Verständnis

`for` Comprehensions in Scala sind nur syntaktischer Zucker . Diese Comprehensions werden implementiert , um die Verwendung von `withFilter` , `foreach` , `flatMap` und `map` Methoden ihres Faches Typen. Aus diesem Grund können nur Typen, für die diese Methoden definiert sind, `for` Verständnis verwendet werden.

A `for` Verständnis der folgenden Form mit Mustern `pN` , Generatoren `gN` und Bedingungen `cN` :

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

... `withFilter` verschachtelte Anrufe mit `withFilter` und `foreach` :

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

Ein Ausdruck `for` / `yield` der folgenden Form:

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

... `withFilter` verschachtelte Anrufe mit `withFilter` und entweder `flatMap` oder `map` :

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

(Beachten Sie, dass `map` im innersten Verständnis verwendet wird und `flatMap` in jedem äußeren Verständnis.)

Ein `for` das Verständnis kann auf jede Art angewandt werden , um die Methoden , die von der de-gezuckerten Darstellung erforderlich implementieren. Es gibt keine Einschränkungen für die Rückgabetypen dieser Methoden, sofern sie zusammengestellt werden können.

Für Ausdrücke online lesen: https://riptutorial.com/de/scala/topic/669/fur-ausdrucke

Kapitel 18: Futures

Examples

Zukunft gestalten

```
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
    }
}
```

Die `divide` Methode erzeugt ganz einfach eine Zukunft, die mit dem Quotienten von `a` über `b` .

Eine erfolgreiche Zukunft konsumieren

Der einfachste Weg , eine *erfolgreiche* Future-- oder eher zu konsumieren, um den Wert innerhalb des Future-- zu erhalten ist , die verwenden `map` Methode. Angenommen, ein Code ruft die `divide` Methode des `FutureDivider` Objekts aus dem Beispiel "Zukunft `FutureDivider` " auf. Wie müsste der Code aussehen, um den Quotienten von `a` über `b` ?

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

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

Eine misslungene Zukunft konsumieren

Manchmal kann die Berechnung in einer Zukunft zu einer Ausnahme führen, wodurch die Zukunft scheitern kann. Was passiert, wenn der aufrufende Code in dem Beispiel "Zukunft erstellen" `55` und `0` an die `divide` Methode übergeben hat? Es würde natürlich eine `ArithmeticException` nachdem versucht wurde, durch Null zu teilen. Wie würde das mit verbrauchendem Code behandelt werden? Es gibt tatsächlich eine Reihe von Möglichkeiten, mit Fehlern umzugehen.

Behandeln Sie die Ausnahme mit `recover` und 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}")
        }
    }
}
```

Behandeln Sie die Ausnahme mit der `failed` Projektion, bei der die Ausnahme zum Wert der Zukunft wird:

```
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}")
        }
    }
}
```

Zukunft zusammenstellen

In den vorherigen Beispielen wurden die einzelnen Merkmale einer Zukunft sowie der Umgang mit Erfolg und Misserfolgen demonstriert. In der Regel werden jedoch beide Funktionen viel strenger behandelt. Hier ist das Beispiel, das auf eine schönere und realistischere Weise geschrieben wurde:

```
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}")
        }
    }
}
```

Sequenzierung und Durchquerung von Futures

In einigen Fällen ist es erforderlich, eine variable Anzahl von Werten auf separaten Futures zu berechnen. Angenommen, Sie haben eine `List[Future[Int]]` , aber stattdessen muss eine `List[Int]` verarbeitet werden. Dann stellt sich die Frage, wie diese Instanz von `List[Future[Int]]` in eine `Future[List[Int]]` . Zu diesem Zweck gibt es die `sequence` im Begleitobjekt `Future` .

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

Im Allgemeinen ist `sequence` ein allgemein bekannter Operator in der Welt der funktionalen Programmierung, der `F[G[T]]` in `G[F[T]]` mit Einschränkungen auf `F` und `G` transformiert.

Es gibt einen alternativen Operator namens `traverse` , der ähnlich arbeitet, aber eine Funktion als zusätzliches Argument verwendet. Mit der Identitätsfunktion `x => x` als Parameter verhält es sich wie der `sequence` .

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

Das zusätzliche Argument erlaubt es jedoch, jede zukünftige Instanz innerhalb der angegebenen `listOfFuture` zu ändern. Darüber hinaus muss das erste Argument keine Liste der `Future` . Daher ist es möglich, das Beispiel wie folgt umzuwandeln:

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

In diesem Fall wird die `List(1,2,3)` direkt als erstes Argument übergeben und die Identitätsfunktion `x => x` wird durch die Funktion `Future(_)` , um jeden `Int` Wert auf ähnliche Weise in einen `Future` . Ein Vorteil davon ist, dass die Zwischenliste `List[Future[Int]]` weggelassen werden kann, um die Leistung zu verbessern.

Kombinieren Sie mehrere Futures - zum Verständnis

Das *Verständnis* ist eine kompakte Art, einen Codeblock auszuführen, der vom erfolgreichen Ergebnis mehrerer Futures abhängt.

Mit `f1, f2, f3` drei `Future[String]` , die jeweils die Strings `one, two, three` enthalten.

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

`fCombined` wird eine `Future[String]` die die Zeichenfolge `one - two - three` sobald alle Futures erfolgreich abgeschlossen wurden.

Beachten Sie, dass hier ein impliziter ExectionContext angenommen wird.

Denken Sie auch daran, dass das Verständnis nur ein syntaktischer Zucker für eine flatMap-Methode ist, sodass die Konstruktion zukünftiger Objekte für body die gleichzeitige Ausführung von von Futures umschlossenen Codeblöcken eliminiert und zu sequentiellem Code führt. Sie sehen es am Beispiel:

```
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
```

`result1` Objekt `result1` eingeschlossener `result1` wäre immer negativ, während `result2` positiv wäre.

Weitere Informationen *zum Verständnis* und zum `yield` im Allgemeinen finden Sie unter http://docs.scala-lang.org/tutorials/FAQ/yield.html

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

Kapitel 19: Handling Units (Maßnahmen)

Syntax

- Klasse Meter (Wert: doppelt) erweitert AnyVal
- Typ Meter = Doppelt

Bemerkungen

Es wird empfohlen, Werteinheiten für Einheiten oder eine dedizierte Bibliothek für sie zu verwenden.

Examples

Geben Sie Aliase ein

```
type Meter = Double
```

Dieser einfache Ansatz hat gravierende Nachteile für die Handhabung von Einheiten, da jeder andere Typ, der ein `Double` ist, damit kompatibel ist:

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

Alle obigen Kompilierungen, so dass Einheiten in diesem Fall nur zum Markieren von Eingabe- / Ausgabearten für die Codeleser (nur die Absicht) verwendet werden können.

Wertklassen

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

Werteklassen bieten eine typsichere Methode zum Codieren von Einheiten, auch wenn sie etwas mehr Zeichen benötigen, um sie zu verwenden:

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

Durch die Erweiterung von `AnyVal` s besteht keine Laufzeitstrafe für deren Verwendung. Auf JVM-Ebene handelt es sich dabei um reguläre primitive Typen (in diesem Fall `Double` s).

Falls Sie automatisch andere Einheiten generieren möchten (z. B. `Velocity` aka `MeterPerSecond`), ist dieser Ansatz nicht der beste, obwohl es auch Bibliotheken gibt, die in diesen Fällen verwendet werden können:

- Squants
- Einheiten
- ScalaQuantity

Handling Units (Maßnahmen) online lesen: https://riptutorial.com/de/scala/topic/5966/handling-units--ma-nahmen-

Kapitel 20: Impliziert

Syntax

- impliziter Wert x: T = ???

Bemerkungen

Implizite Klassen ermöglichen das Hinzufügen benutzerdefinierter Methoden zu vorhandenen Typen, ohne dass der Code geändert werden muss. Dadurch werden Typen angereichert, ohne dass der Code gesteuert werden muss.

Die Verwendung impliziter Typen zum Anreichern einer vorhandenen Klasse wird häufig als Muster zum Anreichern meiner Bibliothek bezeichnet.

Einschränkungen für implizite Klassen

1. Implizite Klassen dürfen nur innerhalb einer anderen Klasse, eines anderen Objekts oder einer anderen Eigenschaft existieren.
2. Implizite Klassen dürfen nur einen nicht-impliziten primären Konstruktorparameter haben.
3. Es darf keine andere Objekt-, Klassen-, Merkmals- oder Klassenmitgliedsdefinition innerhalb desselben Bereichs geben, die denselben Namen wie die implizite Klasse hat.

Examples

Implizite Konvertierung

Bei einer impliziten Konvertierung kann der Compiler ein Objekt eines Typs automatisch in einen anderen Typ konvertieren. Dadurch kann der Code ein Objekt als Objekt eines anderen Typs behandeln.

```
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
```

Die Konvertierung ist einseitig: In diesem Fall können Sie `42` nicht in `Foo(42)` zurück konvertieren. Dazu muss eine zweite implizite Konvertierung definiert werden:

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

Beachten Sie, dass dies der Mechanismus ist, mit dem beispielsweise ein Gleitkommawert zu einem ganzzahligen Wert hinzugefügt werden kann.

> Implizite Konvertierungen sollten sparsam verwendet werden, da sie das, was geschieht, verschleiern. Es empfiehlt sich, eine explizite Konvertierung über einen Methodenaufruf zu verwenden, es sei denn, durch die Verwendung einer impliziten Konvertierung wird eine spürbare Lesbarkeit erzielt.

Es gibt keine signifikanten Auswirkungen auf die Leistung von impliziten Konvertierungen.

Scala importiert automatisch verschiedene implizite Konvertierungen in `scala.Predef` , einschließlich aller Konvertierungen von Java in Scala und zurück. Diese sind standardmäßig in jeder Dateikompilierung enthalten.

Implizite Parameter

Implizite Parameter können nützlich sein, wenn ein Parameter eines Typs einmal im Gültigkeitsbereich definiert und dann auf alle Funktionen angewendet werden soll, die einen Wert dieses Typs verwenden.

Ein normaler Funktionsaufruf sieht ungefähr so aus:

```
// 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
```

Nehmen wir an, wir haben einige Methoden, die alle eine Timeout-Dauer haben, und wir möchten alle diese Methoden mit demselben Timeout aufrufen. Wir können das Timeout als implizite Variable definieren.

```
// 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
```

Das funktioniert so, dass der scalac-Compiler im Geltungsbereich nach einem Wert sucht, **der als**

implizit markiert ist und **dessen Typ mit** dem des impliziten Parameters **übereinstimmt** . Wenn er einen findet, wendet er ihn als impliziten Parameter an.

> Beachten Sie, dass dies nicht funktioniert, wenn Sie zwei oder sogar mehrere Implikationen desselben Typs im Gültigkeitsbereich definieren.

Verwenden Sie zum Anpassen der Fehlermeldung die `implicitNotFound` Annotation für den Typ:

```
@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]!
```

Ein Timeout ist ein üblicher Anwendungsfall, oder in Akka ist das ActorSystem (meistens) immer das gleiche, daher wird es normalerweise implizit übergeben. Ein weiterer Anwendungsfall wäre Bibliotheksdesign, am häufigsten bei FP-Bibliotheken, die auf Typenklassen (wie Skalaz , Katzen oder Verzückung) angewiesen sind.

> Die Verwendung impliziter Parameter mit Grundtypen wie *Int* , *Long* , *String* usw. wird im Allgemeinen als schlechte Praxis betrachtet, da dies zu Verwirrung führt und den Code weniger lesbar macht.

Implizite Klassen

Implizite Klassen ermöglichen das Hinzufügen neuer Methoden zu zuvor definierten Klassen.

Die `String` Klasse hat keine Methode ohne `withoutVowels` . Dies kann wie folgt hinzugefügt werden:

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

Die implizite Klasse hat einen einzigen Konstruktorparameter (`str`) mit dem Typ, den Sie erweitern möchten (`String`), und enthält die Methode, die Sie dem Typ (`withoutVowels`) "hinzufügen" `withoutVowels` . Die neu definierten Methoden können jetzt direkt für den erweiterten Typ verwendet werden (wenn der erweiterte Typ im impliziten Gültigkeitsbereich ist):

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

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

Unter der Haube definieren implizite Klassen eine implizite Konvertierung vom erweiterten Typ in

die implizite Klasse wie folgt:

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

Implizite Klassen werden häufig als Value-Klassen definiert, um das Erstellen von Laufzeitobjekten und damit das Entfernen des Laufzeit-Overhead zu vermeiden:

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

Mit der obigen verbesserten Definition muss nicht jedes Mal eine neue Instanz von `StringEnhancer` erstellt werden, wenn die `withoutVowels` Methode aufgerufen wird.

Implizite Parameter mit 'implizit' auflösen

Eine implizite Parameterliste mit mehr als einem impliziten Parameter annehmen:

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

`SomeCtx1` nun davon ausgegangen wird, dass eine der impliziten Instanzen nicht verfügbar ist (`SomeCtx1`), während alle anderen erforderlichen impliziten Instanzen im Gültigkeitsbereich liegen, muss zum Erstellen einer Instanz der Klasse eine Instanz von `SomeCtx1` bereitgestellt werden.

Dies kann durchgeführt werden, während die jeweils andere implizite Instanz im Gültigkeitsbereich mithilfe des `implicitly` Schlüsselworts beibehalten wird:

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

Implikationen in der REPL

Um alle `implicits` in-scope während einer REPL Sitzung:

```
scala> :implicits
```

Um implizite Konvertierungen `Predef.scala` die in `Predef.scala` definiert `Predef.scala` :

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

Wenn einer einen Ausdruck hat und die Auswirkung aller für ihn geltenden Umschreibungsregeln (einschließlich Implikationen) sehen möchte

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

(Beispiel:

```
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(", "))
```

)

Impliziert online lesen: https://riptutorial.com/de/scala/topic/1732/impliziert

Kapitel 21: Java-Interoperabilität

Examples

Konvertieren von Scala-Sammlungen in Java-Sammlungen und umgekehrt

Wenn Sie eine Collection an eine Java-Methode übergeben müssen:

```
import scala.collection.JavaConverters._

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

Wenn der Java-Code eine Java-Sammlung zurückgibt, können Sie diese auf ähnliche Weise in eine Scala-Sammlung umwandeln:

```
import scala.collection.JavaConverters._

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

Beachten Sie, dass es sich hierbei um Dekorateure handelt, die die zugrunde liegenden Sammlungen lediglich in eine Scala- oder Java-Sammlungsschnittstelle einschließen. Deshalb kopieren die Aufrufe `.asJava` und `.asScala` die Sammlungen nicht.

Arrays

Arrays sind reguläre JVM-Arrays mit dem gewissen Unterschied, dass sie als unveränderlich behandelt werden und spezielle Konstruktoren und implizite Konvertierungen aufweisen. Konstruieren Sie sie ohne das `new` Schlüsselwort.

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

Jetzt `a` hat Typ `Array[String]` .

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

Obwohl `String` in `CharSequence` konvertierbar `CharSequence` , kann `Array[String]` nicht in `Array[CharSequence]` .

Sie können ein `Array` wie andere Sammlungen verwenden, dank der impliziten Konvertierung in `TraversableLike` `ArrayOps` :

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

Die meisten Scala-Sammlungen (`TraversableOnce`) verfügen über eine `toArray` Methode, die ein implizites `ClassTag` zum `ClassTag` des Ergebnisarrays verwendet:

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

Dies macht es einfach, `TraversableOnce` in Ihrem Scala-Code zu verwenden und dann an Java-Code zu übergeben, der ein Array erwartet.

Konvertierungen von Scala und Java

Scala bietet implizite Konvertierungen zwischen allen wichtigen Auflistungstypen im JavaConverters-Objekt an.

Die folgenden Typkonvertierungen sind bidirektional.

Scala-Typ	Java-Typ
Iterator	java.util.Iterator
Iterator	java.util.Enumeration
Iterator	java.util
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

Bestimmte andere Scala-Sammlungen können auch nach Java konvertiert werden, haben jedoch keine Konvertierung in den ursprünglichen Scala-Typ:

Scala-Typ	Java-Typ
Seq	java.util.List
veränderbar	java.util.List
einstellen	java.util.Set
Karte	java.util.Map

Referenz :

Konvertierungen zwischen Java- und Scala-Sammlungen

Funktionsschnittstellen für Scala-Funktionen - Scala-Java 8-kompatibel

Ein Java 8-Kompatibilitätskit für Scala.

Die meisten Beispiele werden aus der Readme-Datei kopiert

Konverter zwischen scala.FunctionN und 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
```

Konverter zwischen den Klassen scala.Option und java.util Optional, OptionalDouble, OptionalInt und 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
    }
```

Konverter von Scala-Sammlungen zu 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)
```

Java-Interoperabilität online lesen: https://riptutorial.com/de/scala/topic/2441/java-interoperabilitat

Kapitel 22: JSON

Examples

JSON mit Spray-Json

spray-json bietet eine einfache Möglichkeit, mit JSON zu arbeiten. Bei Verwendung impliziter Formate geschieht alles "hinter den Kulissen":

Machen Sie die Bibliothek mit SBT verfügbar

So verwalten Sie `spray-json` mit verwalteten SBT-Bibliotheksabhängigkeiten :

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

Beachten Sie, dass der letzte Parameter, die Versionsnummer (`1.3.2`), in verschiedenen Projekten unterschiedlich sein kann.

Die `spray-json` Bibliothek wird auf repo.spray.io gehostet.

Importieren Sie die Bibliothek

```
import spray.json._
import DefaultJsonProtocol._
```

Das Standard-JSON-Protokoll `DefaultJsonProtocol` enthält Formate für alle `DefaultJsonProtocol` . Verwenden Sie zur Bereitstellung der JSON-Funktionalität für benutzerdefinierte Typen entweder Convenience Builder für Formate oder explizit Schreibformate.

Lesen Sie 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)
```

Schreibe JSON

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSL wird nicht unterstützt.

Schreib-Lese-Fallklassen

Das folgende Beispiel zeigt, wie ein Fallklassenobjekt in das JSON-Format serialisiert wird.

```
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
```

Dies führt zu folgendem JSON:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

Diese JSON kann wiederum in ein Objekt deserialisiert werden:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

Benutzerdefiniertes Format

Schreiben Sie ein benutzerdefiniertes `JsonFormat` wenn eine spezielle Serialisierung eines Typs erforderlich ist. Zum Beispiel, wenn die Feldnamen in Scala anders sind als in JSON. Oder wenn unterschiedliche konkrete Typen basierend auf der Eingabe instanziiert werden.

```
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 mit Circe

Circe stellt abgeleitete Codecs zur Kompilierzeit für en / decode json in Fallklassen bereit. Ein einfaches Beispiel sieht so aus:

```
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 mit Play-Json

play-json verwendet implizite Formate als andere Json-Frameworks

SBT-Abhängigkeit: `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` enthält Standardformate zum Lesen / Schreiben aller Basistypen. Um JSON-Funktionen für Ihre eigenen Typen bereitzustellen, können Sie Convenience Builder für Formate verwenden oder explizit Schreibformate verwenden.

Lesen Sie 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),)
```

Schreibe 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"}
```

Wie immer bevorzugen Sie den Musterabgleich mit `JsSuccess` / `JsError` und versuchen Sie, `.get` und `array(i)` -Aufrufe zu vermeiden.

Lesen und schreiben Sie in die Fallklasse

```
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))
```

Eigenes Format

Sie können Ihr eigenes JsonFormat schreiben, wenn Sie eine spezielle Serialisierung Ihres Typs benötigen (z. B. die Felder in scala und Json anders benennen oder verschiedene konkrete Typen basierend auf der Eingabe instanziieren).

```
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

Wenn der Json nicht genau mit Ihren `isAlive` (`isAlive` in `is_alive` vs `is_alive` in `is_alive`):

```
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 mit optionalen Feldern

```
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 _)
}
```

Zeitstempel von Json lesen

Stellen Sie sich vor, Sie haben ein Json-Objekt mit einem Unix-Zeitstempelfeld:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

Lösung:

```
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 _)
}
```

Kundenspezifische Fallklassen lesen

Wenn Sie nun Ihre Objektkennungen für die Typsicherheit einpacken, werden Sie dies genießen. Siehe das folgende Json-Objekt:

```
{
  "id": 91,
  "data": "Some data"
}
```

und die entsprechenden Fallklassen:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Jetzt müssen Sie nur noch den primitiven Typ (Long) lesen und Ihrer ID zuordnen:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

Code unter https://github.com/pedrorijo91/scala-play-json-examples

JSON mit Json4s

Json4s verwendet implizite Formate als andere Json-Frameworks.

SBT-Abhängigkeit:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

Importe

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats` enthält Standardformate zum Lesen / Schreiben aller `DefaultFormats` .

Lesen Sie 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)
```

Schreibe 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"}
```

Lesen und schreiben Sie in die Fallklasse

```
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]
```

Lesen und Schreiben von heterogenen Listen

Um eine heterogene (oder polymorphe) Liste zu serialisieren und zu deserialisieren, müssen bestimmte Typhinweise angegeben werden.

```
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)
```

Eigenes 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)
```

JSON online lesen: https://riptutorial.com/de/scala/topic/2348/json

Kapitel 23: Klassen eingeben

Bemerkungen

Um Serialisierungsprobleme zu vermeiden, insbesondere in verteilten Umgebungen (z. B. Apache Spark), empfiehlt es sich, die `Serializable` Eigenschaft für Typklasseninstanzen zu implementieren.

Examples

Einfache Typenklasse

Eine Typenklasse ist einfach eine `trait` mit einem oder mehreren Typparametern:

```
trait Show[A] {
  def show(a: A): String
}
```

Anstatt eine Typenklasse zu erweitern, wird für jeden unterstützten Typ eine implizite Instanz der Typklasse bereitgestellt. Durch das Platzieren dieser Implementierungen im Companion-Objekt der Typklasse kann die implizite Auflösung ohne spezielle Importe funktionieren:

```
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
}
```

Wenn Sie sicherstellen möchten, dass ein generischer Parameter, der an eine Funktion übergeben wird, über eine Instanz einer Typklasse verfügt, verwenden Sie implizite Parameter:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

Sie können auch eine Kontextbindung verwenden :

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Rufen Sie die obige `log` wie jede andere Methode auf. Es kann nicht kompiliert werden, wenn für

das `A` Sie an das `log` keine implizite `Show[A]` -Implementierung gefunden werden kann

```
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]]
```

In diesem Beispiel wird die Klasse " `Show` implementiert. Dies ist eine gebräuchliche Typenklasse, mit der beliebige Instanzen beliebiger Typen in `String` s konvertiert werden. Obwohl jedes Objekt über eine `toString` Methode verfügt, ist nicht immer klar, ob `toString` sinnvoll definiert ist oder nicht. Bei der Verwendung der `Show` Typklasse, können Sie garantieren , dass alles übergeben `log` eine gut definierte Umwandlung hat `String` .

Typenklasse erweitern

In diesem Beispiel wird die Erweiterung der folgenden Typklasse erläutert.

```
trait Show[A] {
  def show: String
}
```

Um eine Klasse, die *Sie* steuern (und in Scala geschrieben) haben, um die Typklasse zu erweitern, fügen Sie ihrem Begleitobjekt eine implizite Klasse hinzu. Lassen Sie uns zeigen , wie wir die bekommen können `Person` Klasse von diesem Beispiel erweitern `Show` :

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

Wir können diese Klasse machen verlängern `Show` durch eine implizite zum Hinzufügen `Person` ,s Begleitobjekt:

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

Ein Begleitobjekt *muss sich in derselben Datei* wie die Klasse befinden. Sie benötigen also sowohl die `class Person` als auch das `object Person` in derselben Datei.

Um eine Klasse zu erstellen, die Sie nicht kontrollieren oder in Scala nicht geschrieben sind, erweitern Sie die Typklasse, fügen Sie dem Begleitobjekt der Typklasse eine implizite Funktion hinzu, wie im Beispiel der Simple Type Class gezeigt .

Wenn Sie weder die Klasse noch die Typenklasse steuern, erstellen Sie ein implizites Objekt wie oben und `import` es. Verwenden der `log` im Beispiel für die einfache Typklasse :

```
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) }
}
```

Fügen Sie Typklassenfunktionen zu Typen hinzu

Scalas Implementierung von Typklassen ist ziemlich ausführlich. Eine Möglichkeit, die Ausführlichkeit zu reduzieren, ist die Einführung sogenannter "Operation Classes". Diese Klassen wickeln automatisch eine Variable / einen Wert ein, wenn sie zur Erweiterung der Funktionalität importiert werden.

Um dies zu veranschaulichen, erstellen wir zunächst eine einfache Typenklasse:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

Als Nächstes implementieren wir die Eigenschaft (Instanziierung der Typenklasse):

```
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
  }

}
```

Im Moment wäre es sehr umständlich, unsere Addable-Instanzen zu verwenden:

```
import Instances._
val three = addableInt.add(1,2)
```

Wir schreiben `1.add(2)` . Deshalb erstellen wir eine "Operationsklasse" (auch als "Ops-Klasse" bezeichnet), die immer einen Typ `Addable` , der `Addable` implementiert.

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
```

```
  }
}
```

Jetzt können wir unsere neue Funktion `add` als ob sie Teil von `Int` und `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" -Klassen können automatisch von Makros in der Simulacrum- Bibliothek erstellt werden:

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Klassen eingeben online lesen: https://riptutorial.com/de/scala/topic/3835/klassen-eingeben

Kapitel 24: Klassen und Objekte

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)` oder mit einem anderen Typ: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

Instanziieren von Klasseninstanzen

Eine Klasse in Scala ist ein "Entwurf" einer Klasseninstanz. Eine Instanz enthält den von dieser Klasse definierten Zustand und Verhalten. So deklarieren Sie eine Klasse:

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

Eine Instanz kann mit einem `new` Schlüsselwort instanziiert werden:

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

oder:

```
var instance = new MyClass
```

Klammern sind in Scala optional zum Erstellen von Objekten aus einer Klasse mit einem Konstruktor ohne Argumente. Wenn ein Klassenkonstruktor Argumente akzeptiert:

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

Hier benötigt `MyClass` ein `Int` Argument, das nur intern für die Klasse verwendet werden kann. `arg` kann nicht außerhalb von `MyClass` zugegriffen werden, es sei denn, es ist als Feld deklariert:

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

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

Alternativ kann es im Konstruktor als public deklariert werden:

```
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
```

Klasse ohne Parameter instanziieren: {} vs ()

Nehmen wir an, wir haben eine Klasse MyClass ohne Konstruktorargument:

```
class MyClass
```

In Scala können wir es mit folgender Syntax instanziieren:

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

Oder wir schreiben einfach:

```
val obj = new MyClass
```

Bei Nichtbeachtung kann die optionale Klammer in einigen Fällen zu unerwartetem Verhalten führen. Angenommen, wir möchten eine Aufgabe erstellen, die in einem separaten Thread ausgeführt werden soll. Unten ist der Beispielcode:

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

newThread.start   // prints no output
```

Wir denken vielleicht, dass dieser Beispielcode, wenn er ausgeführt wird, die ausführende `Performing task.` druckt `Performing task.` , aber zu unserer Überraschung wird nichts gedruckt. Mal sehen, was hier passiert. Wenn du genauer hinschaust, haben wir geschweifte Klammern `{}` , gleich nach dem `new Thread` . Es wurde eine anonyme Klasse erstellt, die den `Thread` :

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

Im Rumpf dieser anonymen Klasse haben wir dann unsere Aufgabe definiert (wieder eine anonyme Klasse, die die `Runnable` Schnittstelle implementiert). Wir haben vielleicht gedacht, dass wir einen `public Thread(Runnable target)` verwendet haben, aber in der Tat (durch Ignorieren von optionalem `()`) haben wir einen `public Thread() public Thread(Runnable target)` verwendet, bei dem nichts im Rumpf der `run()` Methode definiert ist. Um das Problem zu beheben, müssen wir Klammern anstelle von geschweiften Klammern verwenden.

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

Mit anderen Worten, hier `{}` und `()` sind nicht *austauschbar* .

Singleton & Begleitobjekte

Singleton-Objekte

Scala unterstützt statische Member, jedoch nicht auf dieselbe Weise wie Java. Als Alternative dazu bietet Scala *Singleton Objects an* . Singleton-Objekte ähneln einer normalen Klasse, können jedoch nicht mit dem `new` Schlüsselwort instanziiert werden. Hier ist ein Beispiel für eine Singleton-Klasse:

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

Beachten Sie, dass wir das `object` zur Definition des Singleton-Objekts (anstelle von 'class' oder 'trait') verwendet haben. Da Einzelobjekte nicht instanziiert werden können, können sie keine Parameter haben. Der Zugriff auf ein Singleton-Objekt sieht folgendermaßen aus:

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

Beachten Sie, dass dies genau wie der Zugriff auf eine statische Methode in einer Java-Klasse aussieht.

Begleitobjekte

In Scala können Einzelobjekte den Namen einer entsprechenden Klasse gemeinsam nutzen. In einem solchen Szenario wird das Einzelobjekt als *Begleitobjekt bezeichnet* . Unterhalb der Klasse wird beispielsweise `Factorial` definiert und darunter ein Begleitobjekt (auch `Factorial`). Standardmäßig werden Begleitobjekte in derselben Datei wie ihre Begleitklasse definiert.

```
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 diesem Beispiel verwenden wir einen privaten `cache` , um die Fakultät einer Zahl zu speichern, um Rechenzeit für wiederholte Zahlen zu sparen.

Das `object Factorial` ist hier ein Begleitobjekt und die `class Factorial` ist die entsprechende Begleitklasse. Begleitobjekte und Klassen können auf ihre `private` Mitglieder zugreifen. Im obigen Beispiel `Factorial` die `Factorial` Klasse auf das private `cache` Member des zugehörigen Objekts zu.

Beachten Sie, dass eine neue Instanziierung der Klasse immer noch dasselbe Begleitobjekt verwendet, sodass Änderungen an Member-Variablen dieses Objekts übernommen werden.

Objekte

Während Klassen eher wie Blaupausen sind, sind Objekte statisch (dh bereits instanziiert):

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

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

Sie werden oft als Begleiter einer Klasse verwendet, und Sie können schreiben:

```
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
```

Instanztypüberprüfung

Typüberprüfung : `variable.isInstanceOf[Type]`

Mit Mustervergleich (in dieser Form nicht so nützlich):

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

Sowohl `isInstanceOf` als auch Pattern Matching prüfen nur den Typ des Objekts, nicht dessen generischen Parameter (keine Typänderung), außer bei 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
```

Aber

```
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
```

Typcasting : `variable.asInstanceOf[Type]`

Mit Musterabgleich :

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

Beispiele:

```
  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)
```

Anmerkung: Hierbei handelt es sich nur um das Verhalten in der JVM. Auf anderen Plattformen (JS, native) kann das Casting / die Überprüfung von Typen sich möglicherweise unterscheiden.

Konstrukteure

Primärer Konstruktor

In Scala ist der Hauptkonstruktor der Körper der Klasse. Auf den Klassennamen folgt eine Parameterliste, die die Konstruktorargumente darstellt. (Wie bei jeder Funktion kann eine leere Parameterliste weggelassen werden.)

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

class Bar {
    ...
}
```

Die Konstruktionsparameter einer Instanz sind außerhalb ihres Konstruktorkörpers nicht verfügbar, sofern sie nicht mit dem Schlüsselwort `val` als Instanzmitglied gekennzeichnet sind:

```
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
```

Alle Operationen, die ausgeführt werden sollen, wenn eine Instanz eines Objekts instanziiert wird, werden direkt in den Hauptteil der Klasse geschrieben:

```
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
    ...
}
```

Beachten Sie, dass es als gute Praxis angesehen wird, so wenig Nebenwirkungen wie möglich in den Konstruktor einzufügen. Anstelle des obigen Codes sollten Sie über `connect` und `disconnect` nachdenken, sodass der Verbrauchercode für die Planung der E / A verantwortlich ist.

Hilfskonstruktoren

Eine Klasse kann zusätzliche Konstruktoren enthalten, die als "Hilfskonstruktoren" bezeichnet werden. Diese werden durch Konstruktordefinitionen in der Form `def this(...) = e` , wobei `e` einen anderen Konstruktor aufrufen muss:

```
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
```

Dies bedeutet, dass jeder Konstruktor einen anderen Modifikator haben kann: nur einige sind möglicherweise öffentlich verfügbar:

```
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
```

Auf diese Weise können Sie steuern, wie der Verbrauchercode die Klasse instanziiert.

Klassen und Objekte online lesen: https://riptutorial.com/de/scala/topic/2047/klassen-und-objekte

Kapitel 25: Makros

Einführung

Makros sind eine Form der Kompilierzeit-Metaprogrammierung. Bestimmte Elemente des Scala-Codes, z. B. Anmerkungen und Methoden, können so erstellt werden, dass sie anderen Code konvertieren, wenn sie kompiliert werden. Makros sind gewöhnlicher Scala-Code, der mit Datentypen arbeitet, die anderen Code darstellen. Das Plugin [Macro Paradise] [] erweitert die Fähigkeiten von Makros über die Basissprache hinaus. [Macro Paradise]: http://docs.scala-lang.org/overviews/macros/paradise.html

Syntax

- def x () = Makro x_impl // x ist ein Makro, wobei x_impl zur Transformation von Code verwendet wird
- def macroTransform (annottees: Any *): Any = macro impl // In Annotationen verwenden, um Makros zu erstellen

Bemerkungen

Makros sind eine `scala.language.macros` , die aktiviert werden muss, indem entweder `scala.language.macros` oder mit der Compiler-Option `-language:macros` importiert wird. Nur Makrodefinitionen erfordern dies. Code, der Makros verwendet, muss dies nicht tun.

Examples

Makro-Anmerkung

Diese einfache Makroannotation gibt das kommentierte Element unverändert aus.

```
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}")
  }
}
```

Die Annotation `@compileTimeOnly` generiert einen Fehler mit einer Meldung, die darauf hinweist,

dass das `paradise` Compiler-Plug-In für die Verwendung dieses Makros enthalten sein muss. Anweisungen zum Einbinden über SBT finden Sie hier .

Sie können das oben definierte Makro folgendermaßen verwenden:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

Methodenmakros

Wenn eine Methode als Makro definiert ist, übernimmt der Compiler den übergebenen Code als Argument und wandelt ihn in ein AST um. Sie ruft dann die Makroimplementierung mit dieser AST auf und gibt eine neue AST zurück, die dann an ihre Aufrufstelle zurückgespleißt wird.

```
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
    }
  }
}
```

Es ist auch möglich, Makros zu verwenden, die `Tree` s als Argumente verwenden. Wie , wie `reify` dient zum Erstellen `Expr` s, die `q` (für quasiquote) string Interpolator lässt uns erstellen und

dekonstruieren `Tree` s. Beachten Sie, dass wir `q` oben verwenden konnten (`expr.tree` ist überraschend ein `Tree` selbst), dies jedoch nicht zu Demonstrationszwecken.

```
// 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"
  }
}
```

Fehler in Makros

Makros können Warnungen und Fehler des Compilers durch die Verwendung ihres `Context` auslösen.

Nehmen wir an, wir sind besonders übereifrig, wenn es um schlechten Code geht, und wir möchten jede technische Schuld mit einer Compiler-Infomeldung kennzeichnen. Wir können ein Makro verwenden, das nur eine solche Nachricht ausgibt.

```
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 ()
}
```

Zusätzlich anstelle von `???` um nicht implementierten Code zu kennzeichnen, können wir zwei Makros erstellen, `!!!` und `?!?` , die dem gleichen Zweck dienen, aber Compiler-Warnungen ausgeben. `?!?` wird eine Warnung auslösen und `!!!` wird einen eindeutigen Fehler verursachen.

```
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.???"
}
```

Makros online lesen: https://riptutorial.com/de/scala/topic/3808/makros

Kapitel 26: Mit Daten unveränderlich arbeiten

Bemerkungen

Wert- und Variablennamen sollten im unteren Kamelfall stehen

> Konstante Namen sollten im oberen Kamel sein. Wenn das Member also final ist, unveränderlich ist und zu einem Paketobjekt oder einem Objekt gehört, kann es als Konstante betrachtet werden
>
> Methoden-, Wert- und Variablennamen sollten in Kleinbuchstaben stehen

Quelle: http://docs.scala-lang.org/style/naming-conventions.html

Dieses kompilieren:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

aber das tut nicht:

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

Es ist nicht nur val vs. var

`val` und `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` Verweise sind nicht veränderbar: Wie eine `final` Variable in `Java` kann sie nach der

Initialisierung nicht mehr geändert werden
- `var` Referenzen können als einfache Variablendeklaration in Java erneut zugewiesen werden

Unveränderliche und veränderliche Sammlungen

```
  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)
```

Die Standardbibliothek von Scala bietet sowohl unveränderliche als auch veränderliche Datenstrukturen, nicht den Verweis darauf. Jedes Mal, wenn eine unveränderliche Datenstruktur "geändert" wird, wird eine neue Instanz erstellt, anstatt die ursprüngliche Sammlung direkt zu ändern. Jede Instanz der Sammlung kann eine signifikante Struktur mit einer anderen Instanz gemeinsam nutzen.

Veränderliche und unveränderliche Sammlung (offizielle Scala-Dokumentation)

Aber ich kann in diesem Fall keine Unveränderlichkeit verwenden!

Nehmen wir als Beispiel eine Funktion, die 2 `Map` und gibt eine `Map` die jedes Element in `ma` und `mb` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

Ein erster Versuch könnte darin bestehen, die Elemente einer der Karten mit `for ((k, v) <- map)` zu durchlaufen und die zusammengeführte Karte irgendwie zurückzugeben.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

Dieser allererste Schritt fügt sofort eine Einschränkung hinzu: **Jetzt wird eine Mutation außerhalb von `for` *benötigt*** . Dies ist deutlicher beim Entzuckern `for` :

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

"Warum müssen wir mutieren?"

`foreach` stützt sich auf Nebenwirkungen. Jedes Mal, wenn wir möchten, dass etwas innerhalb eines `foreach` geschieht, müssen wir etwas "Nebeneffekt" haben. In diesem Fall könnten wir ein variables `var result` ändern oder eine veränderliche Datenstruktur verwenden.

Erstellen und Füllen der `result`

Nehmen wir an, `ma` und `mb` sind `scala.collection.immutable.Map` , wir könnten das `result` Map aus `ma` erstellen:

```
val result = mutable.Map() ++ ma
```

Iterieren dann durch `mb` seine Elemente hinzuzufügen und wenn der `key` des aktuellen Elements auf `ma` bereits vorhanden ist , machen sie es mit dem außer Kraft setzen `mb` ein.

```
mb.foreach { case (k, v) => result += (k -> v) }
```

Veränderliche Implementierung

So weit so gut, wir mussten "veränderliche Sammlungen verwenden" und eine korrekte Implementierung könnte sein:

```
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
}
```

Wie erwartet:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

Zur Rettung falten

Wie können wir in diesem Szenario `foreach` loswerden? Wenn wir im Grunde nur die Collection-Elemente `.foldLeft` und eine Funktion anwenden, während das Ergebnis mit der Option akkumuliert wird, könnte `.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 diesem Fall ist unser "Ergebnis" der akkumulierte Wert ausgehend von `ma` , der `zero` von `.foldLeft` .

Zwischenergebnis

Offensichtlich produziert und zerstört diese unveränderliche Lösung viele `Map` Instanzen beim Falten, aber es ist erwähnenswert, dass diese Instanzen kein vollständiger Klon der `Map` , sondern eine signifikante Struktur (Daten) mit der vorhandenen Instanz gemeinsam nutzen.

Einfachere Angemessenheit

Es ist einfacher, über die Semantik zu `.foldLeft` wenn sie deklarativer ist als der `.foldLeft` Ansatz. Die Verwendung unveränderlicher Datenstrukturen könnte dazu beitragen, die Implementierung der Implementierung zu erleichtern.

Mit Daten unveränderlich arbeiten online lesen: https://riptutorial.com/de/scala/topic/6298/mit-daten-unveranderlich-arbeiten

Kapitel 27: Mit Gradle arbeiten

Examples

Grundeinstellung

1. Erstellen Sie eine Datei mit dem Namen `SCALA_PROJECT/build.gradle` mit folgendem Inhalt:

```
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. Führen Sie `gradle tasks` , um die verfügbaren Tasks `gradle tasks` .

3. Führen Sie `gradle create-dirs` , um ein Verzeichnis `src/scala, src/resources` zu erstellen.

4. Führen Sie `gradle build` , um das Projekt zu erstellen und Abhängigkeiten herunterzuladen.

Erstellen Sie Ihr eigenes Gradle Scala-Plugin

Nachdem Sie das **grundlegende Setup-** Beispiel **durchgearbeitet haben** , wiederholen Sie möglicherweise den größten Teil davon in jedem einzelnen Scala Gradle-Projekt. Riecht nach Boilerplate-Code ...

Was wäre, wenn Sie statt des von Gradle angebotenen Scala-Plugins Ihr eigenes Scala-Plugin anwenden könnten, das für die Handhabung Ihrer gesamten Build-Logik verantwortlich ist und gleichzeitig das bereits vorhandene Plugin erweitert.

In diesem Beispiel wird die vorherige Build-Logik in ein wiederverwendbares Gradle-Plugin umgewandelt.

Glücklicherweise können Sie in Gradle problemlos benutzerdefinierte Plugins mithilfe der Gradle-API schreiben, wie in der Dokumentation beschrieben . Als Implementierungssprache können Sie Scala selbst oder sogar Java verwenden. Die meisten Beispiele, die Sie in allen Dokumenten finden, sind jedoch in Groovy geschrieben. Wenn Sie mehr Code-Beispiele benötigen oder wissen möchten, was sich hinter dem Scala-Plugin verbirgt, können Sie das Gradle Github-Repo überprüfen.

Das Plugin schreiben

Bedarf

Das benutzerdefinierte Plugin fügt die folgenden Funktionen hinzu, wenn es auf ein Projekt angewendet wird:

- ein `scalaVersion` Eigenschaftsobjekt, das zwei überschreibbare Standardeigenschaften haben wird
  - major = "2.12"
  - minor = "0"
- Eine `withScalaVersion` Funktion, die auf einen Abhängigkeitsnamen angewendet wird, fügt die Scala-Hauptversion hinzu, um die Binärkompatibilität zu gewährleisten (sbt `%%` Operator `%%` möglicherweise, sonst gehen Sie hier vor dem `%%`)
- Eine `createDirs` Task zum Erstellen der erforderlichen Verzeichnisstruktur, genau wie im vorherigen Beispiel

Implementierungsrichtlinie

1. Erstellen Sie ein neues Gradle-Projekt und fügen Sie Folgendes zu `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

Anmerkungen :

- Die Implementierung des Plugins ist in Scala geschrieben, daher benötigen wir das Gradle Scala Plugin
- Um das Plugin aus anderen Projekten zu verwenden, wird das Gradle Maven Plugin verwendet. Dadurch wird die `install` zum Speichern des Projektbehälters zum lokalen Repository von Maven hinzugefügt
- `compile gradleApi()` fügt dem `gradle-api-<gradle_version>.jar` die `gradle-api-`

```
    <gradle_version>.jar
```

2. Erstellen Sie eine neue Scala-Klasse für die Plugin-Implementierung

```
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")
    }
}
```

Anmerkungen :

- Um ein Plugin zu implementieren, erweitern Sie einfach die `Plugin` Eigenschaft des Typs `Project` und überschreiben die `apply` Methode
- Innerhalb der Apply-Methode haben Sie Zugriff auf die `Project` Instanz, auf die das Plugin angewendet wird, und Sie können es zum Hinzufügen von Build-Logik verwenden
- Dieses Plugin übernimmt nichts anderes als das bereits vorhandene Gradle Scala Plugin

3. `scalaVersion` Sie die `scalaVersion` Objekteigenschaft hinzu

Zunächst erstellen wir eine `ScalaVersion` Klasse, die die beiden Versionseigenschaften enthält

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Eine coole Sache an Gradle-Plugins ist die Tatsache, dass Sie immer bestimmte Eigenschaften hinzufügen oder überschreiben können. Ein Plugin empfängt diese Art von Benutzereingaben über den `ExtensionContainer` , der an eine Gradle- `Project` Instanz angehängt ist. Für weitere Informationen, lesen Sie diese aus.
Indem wir der `apply` Methode Folgendes hinzufügen, tun wir im Wesentlichen Folgendes:

- Wenn im Projekt keine `scalaVersion` Eigenschaft definiert ist, fügen Sie eine mit den Standardwerten hinzu
- Andernfalls erhalten wir die vorhandene als Instanz von `ScalaVersion` , um sie weiter zu verwenden

```
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]
```

Dies entspricht dem Schreiben der folgenden Datei in die Erstellungsdatei des Projekts, in dem das Plugin angewendet wird:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. `scalaVersion` Sie die `scala-lang` Bibliothek mithilfe der `scalaVersion` zu den Projektabhängigkeiten `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

Dies entspricht dem Schreiben der folgenden Datei in die Erstellungsdatei des Projekts, in dem das Plugin angewendet wird:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. `withScalaVersion` Sie die `withScalaVersion` Funktion hinzu

```
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. Erstellen `createDirs` schließlich die `createDirs` Task und fügen Sie sie dem Projekt hinzu Implementieren Sie eine Gradle-Aufgabe, indem Sie `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())
    }
  }
}
```

Hinweis : Der `SourceSetContainer` enthält Informationen zu allen im Projekt vorhandenen Quellverzeichnissen. Was das Gradle Scala Plugin macht, ist das Hinzufügen der zusätzlichen Quellensätze zu den Java-Quelldateien, wie Sie in den Plugin-Dokumenten sehen können .

Fügen Sie die `createDir` Aufgabe für das Projekt durch das an den Anhängen `apply` Methode:

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

Am Ende sollte Ihre `ScalaCustomPlugin` Klasse so aussehen:

```
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])
  }
}
```

Installieren des Plugin-Projekts im lokalen Maven-Repository

Dies ist sehr einfach durch Ausführen der `gradle install`
Sie können die Installation überprüfen, indem Sie in das lokale Repository-Verzeichnis `~/.m2/repository` , das sich normalerweise unter `~/.m2/repository`

Wie findet Gradle unser neues Plugin?

Jedes Gradle-Plugin hat eine `id` die in der `apply` Anweisung verwendet wird. Wenn Sie zum Beispiel Folgendes in die Build-Datei schreiben, wird dies in einen Trigger für Gradle übersetzt, um das Plugin mit der ID- `scala` zu finden und anzuwenden.

```
apply plugin: 'scala'
```

Genauso möchten wir unser neues Plugin folgendermaßen einsetzen:

```
apply plugin: "com.btesila.scala.plugin"
```

`com.btesila.scala.plugin` bedeutet, dass unser Plugin die ID `com.btesila.scala.plugin` .

Um diese ID festzulegen, fügen Sie die folgende Datei hinzu:
src / main / resources / META-INF / gradle-plugin / com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

Führen Sie danach `gradle install` erneut `gradle install` .

Verwendung des Plugins

1. Erstellen Sie ein neues leeres Gradle-Projekt und fügen Sie der Build-Datei Folgendes hinzu

```
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. Führen Sie `gradle createDirs` - Sie sollten jetzt alle Quellverzeichnisse erstellt haben
3. Überschreiben Sie die Scala-Version, indem Sie dies der Build-Datei hinzufügen:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. Fügen Sie eine Abhängigkeitsbibliothek hinzu, die mit der Scala-Version binär kompatibel ist

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

Das ist es! Sie können dieses Plugin jetzt für alle Ihre Projekte verwenden, ohne dieselbe alte Boilerplate zu wiederholen.

Mit Gradle arbeiten online lesen: https://riptutorial.com/de/scala/topic/3304/mit-gradle-arbeiten

Kapitel 28: Monaden

Examples

Monade-Definition

Informell handelt es sich bei einer Monade um einen Container mit Elementen, der als `F[_]` , und enthält zwei Funktionen: `flatMap` (um diesen Container zu transformieren) und `unit` (um diesen Container zu erstellen).

Übliche Bibliotheksbeispiele sind `List[T]` , `Set[T]` und `Option[T]` .

Formale Definition

Monade `M` ist ein parametrischer Typ `M[T]` mit zwei Operationen `flatMap` und `unit` , wie:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

Diese Funktionen müssen drei Gesetze erfüllen:

1. *Assoziativität* : `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   Das heißt, wenn die Reihenfolge unverändert ist, können Sie die Bedingungen in beliebiger Reihenfolge anwenden. Wenn Sie also `m` auf `f` anwenden und dann das Ergebnis auf `g` anwenden, erhalten Sie dasselbe Ergebnis wie das Anwenden von `f` auf `g` und das anschließende Anwenden von `m` auf dieses Ergebnis.
2. *Linke Einheit* : `unit(x) flatMap f == f(x)`
   Das heißt, die Einheitsmonade von `x` flach über `f` abgebildet ist, entspricht der Anwendung von `f` auf `x` .
3. *Rechte Einheit* : `m flatMap unit == m`
   Dies ist eine 'Identität': Jede gegen Einheit gemappte Monade gibt eine Monade zurück, die ihrer selbst entspricht.

Beispiel :

```
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. *Assoziativität* :

```
(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. *Linke Einheit*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Rechte Einheit*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

Standardsammlungen sind Monaden

Die meisten Standardsammlungen sind Monaden (`List[T]` , `Option[T]`) oder monadenähnlich (`Either[T]` , `Future[T]`). Diese Sammlungen können leicht miteinander kombiniert werden , in `for` Comprehensions (welche eine äquivalente Art des Schreibens `flatMap` Transformations):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

Das obige ist äquivalent zu:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Da eine Monade die *Datenstruktur* bewahrt und nur auf die Elemente innerhalb dieser Struktur einwirkt, können wir endlose monadische Datenstrukturen verketten, wie hier zum Verständnis gezeigt.

Monaden online lesen: https://riptutorial.com/de/scala/topic/4112/monaden

Kapitel 29: Musterabgleich

Syntax

- Selektor stimmen mit PartialFunction überein
- Selektorübereinstimmung {Liste der Fallalternativen) // Dies ist die häufigste Form der obigen

Parameter

Parameter	Einzelheiten
Wähler	Der Ausdruck, dessen Wert mit einem Muster abgeglichen wird.
Alternativen	eine Liste von Alternativen, `case` denen der `case` begrenzt ist.

Examples

Einfache Musterübereinstimmung

Dieses Beispiel zeigt, wie eine Eingabe mit mehreren Werten abgeglichen wird:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Live-Demo

Hinweis: _ ist der *Fall* oder *Standardfall* , ist aber nicht erforderlich.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

Um das Auslösen einer Ausnahme zu vermeiden, ist es hier am besten, die Standardprogrammierung zu behandeln (`case _ => <do something>`). Beachten Sie, dass der Vergleich über *eine Fallklasse* helfen kann, dass der Compiler eine Warnung ausgibt, wenn ein Fall fehlt. Dasselbe gilt für benutzerdefinierte Typen, die eine versiegelte Eigenschaft erweitern.

Wenn die Übereinstimmung insgesamt ist, ist möglicherweise kein Standardfall erforderlich

Es ist auch möglich, mit Werten zu vergleichen, die nicht inline definiert sind. Dies müssen *stabile Bezeichner sein* , die entweder durch Verwendung eines Großbuchstaben oder durch Einschließen von Backticks erhalten werden.

Mit `One` und `two` die anderswo definiert oder als Funktionsparameter übergeben werden:

```
val One: Int = 1
val two: Int = 2
```

Sie können auf folgende Weise gegeneinander abgeglichen werden:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

Im Gegensatz zu anderen Programmiersprachen wie Java gibt es keinen Durchbruch. Wenn ein Fallblock mit einer Eingabe übereinstimmt, wird er ausgeführt und der Abgleich ist abgeschlossen. Daher sollte der kleinste spezifische Fall der letzte Fallblock sein.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

Musterabgleich mit stabiler Kennung

Bei der Standardmusterübereinstimmung spiegelt der verwendete Bezeichner jeden Bezeichner im umschließenden Bereich. Manchmal ist es notwendig, die Variable des umgebenden Bereichs abzugleichen.

Die folgende Beispielfunktion nimmt ein Zeichen und eine Liste von Tupeln und gibt eine neue Liste von Tupeln zurück. Wenn das Zeichen als erstes Element in einem der Tupel vorhanden war, wird das zweite Element inkrementiert. Wenn es noch nicht in der Liste vorhanden ist, wird ein neues Tupel erstellt.

```
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)
}
```

Das obige Beispiel zeigt einen Mustervergleich, bei dem die Eingabe `char` der Methode im Mustervergleich "stabil" gehalten wird. Wenn Sie also `tabulate('x', ...)` aufrufen, wird die erste case-Anweisung folgendermaßen interpretiert:

```
case('x', count) => ...
```

Scala interpretiert jede mit einem Häkchen markierte Variable als stabile Kennung: Sie interpretiert auch jede Variable, die mit einem Großbuchstaben beginnt, auf dieselbe Weise.

Pattern Matching auf einer Seq

Nach einer genauen Anzahl von Elementen in der Sammlung suchen

```
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

Das erste (s) Element (e) extrahieren und den Rest als Sammlung behalten:

```
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"
}
```

Im Allgemeinen kann jedes Formular, das zum Erstellen einer Sequenz verwendet werden kann, zum Musterabgleich mit einer vorhandenen Sequenz verwendet werden.

Beachten Sie, dass `Nil` und `::` funktionieren, wenn ein Pattern mit einer Sequenz übereinstimmt, es jedoch in eine `List` konvertiert und unerwartete Ergebnisse erzielen kann. Beschränken Sie sich auf `Seq( ...)` und `+:` um dies zu vermeiden.

Beachten Sie, dass die Verwendung von `::` für `WrappedArray` , `Vector` usw. nicht funktioniert, siehe:

```
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
```

Und mit `+:`:

```
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
```

Wachen (wenn Ausdrücke)

Case-Anweisungen können mit if-Ausdrücken kombiniert werden, um beim Pattern-Matching zusätzliche Logik bereitzustellen.

```
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"
    }
}
```

Es ist wichtig sicherzustellen, dass Ihre Wachen keine nicht erschöpfenden Übereinstimmungen erstellen (der Compiler fängt das oft nicht an)

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

Dies wirft einen `MatchError` auf ungerade Zahlen. Sie müssen entweder für alle Fälle ein Konto verwenden oder einen Platzhalter für Übereinstimmungen mit Platzhaltern verwenden:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

Musterabgleich mit Fallklassen

Jede Fallklasse definiert einen Extraktor, mit dem die Mitglieder der Fallklasse beim Mustervergleich erfasst werden können:

```
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
}
```

Es gelten alle normalen Regeln des Musterabgleichs. Sie können Wachen und konstante Ausdrücke verwenden, um den Abgleich zu steuern:

```
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
}
```

Übereinstimmung mit einer Option

Wenn Sie mit einem Optionstyp übereinstimmen:

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

Dies entspricht funktional der Verwendung von `fold` oder `map` / `getOrElse` :

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

Musterübereinstimmung mit versiegelten Eigenschaften

Wenn ein Muster mit einem Objekt übereinstimmt, dessen Typ eine versiegelte Eigenschaft ist, prüft Scala zur Kompilierzeit, ob alle Fälle "vollständig übereinstimmen":

```
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.
```

```
}
```

Wenn später eine neue `case class` für `Shape` hinzugefügt wird, werden alle `match` in `Shape` mit einer Compiler-Warnung ausgelöst. Dies erleichtert das gründliche Refactoring: Der Compiler weist den Entwickler auf den gesamten Code hin, der aktualisiert werden muss.

Musterabgleich mit 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 diesem Beispiel versucht der Regex, die angegebene E-Mail-Adresse abzugleichen. Wenn dies der Fall ist, werden `userName` und `domain` extrahiert und gedruckt. `topDomain` wird ebenfalls extrahiert, aber in diesem Beispiel wird nichts damit gemacht. Das Aufrufen von `.r` für einen String `str` ist gleichbedeutend mit dem `new Regex(str)` . Die Funktion `r` steht über eine implizite Konvertierung zur Verfügung .

Musterordner (@)

Das `@` -Zeichen bindet eine Variable während eines Mustervergleichs an einen Namen. Die gebundene Variable kann entweder das gesamte übereinstimmende Objekt oder ein Teil des übereinstimmenden Objekts sein:

```
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
```

Der gebundene Bezeichner kann in bedingten Filtern verwendet werden. Somit:

```
case Circle(r) if r > 9 => s"large circle"
```

kann geschrieben werden als:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

Der Name kann nur an einen Teil des übereinstimmenden Musters gebunden sein:

```
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)
```

Musterübereinstimmungsarten

Pattern Matching kann auch verwendet werden, um den Typ einer Instanz zu prüfen, anstatt `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
```

Die Reihenfolge der Fälle ist wichtig:

```
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
```

Auf diese Weise ähnelt es einer klassischen "switch" -Anweisung ohne die Durchfallfunktionalität. Sie können jedoch auch Werte aus dem betreffenden Typ mit Mustern anpassen und "extrahieren". Zum Beispiel:

```
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?'
```

Beachten Sie, dass wir im Fall von `Foo` und `Woo` den Unterstrich (`_`) verwenden, um eine ungebundene Variable zu finden. Das heißt, dass der Wert (in diesem Fall `Hadas` bzw. `27`) an

keinen Namen gebunden ist und daher für diesen Fall nicht im Handler verfügbar ist. Dies ist eine nützliche Abkürzung, um einen beliebigen Wert abzugleichen, ohne sich um den Wert zu kümmern.

Pattern Matching als Tableswitch oder Lookupswitch kompiliert

Die `@switch` Annotation teilt dem Compiler mit, dass die `match` durch eine einzige `tableswitch` auf Bytecode-Ebene ersetzt werden kann. Dies ist eine geringfügige Optimierung, durch die unnötige Vergleiche und variable Ladevorgänge während der Laufzeit entfernt werden können.

Die `@switch` Annotation funktioniert nur für die Spiele gegen wörtliche Konstanten und `final val` Bezeichner. Wenn die `tableswitch` nicht als `tableswitch` / `lookupswitch` kompiliert werden `lookupswitch` , gibt der Compiler eine Warnung aus.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

Die Ergebnisse sind die gleichen wie bei einer normalen Musterübereinstimmung:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

Aus der Scala-Dokumentation (2.8+) - `@switch` :

> Eine Anmerkung, die auf einen Übereinstimmungsausdruck angewendet werden soll. Wenn vorhanden, überprüft der Compiler, ob die Übereinstimmung mit einem Tabellenschalter oder einem Suchschalter erstellt wurde, und gibt einen Fehler aus, wenn er stattdessen in eine Reihe von bedingten Ausdrücken kompiliert wird.

Aus der Java-Spezifikation:

- Tabellenumschaltung : "Zugriff auf Sprungtabelle über Index und Sprung"
- Lookupswitch : "Zugriff auf Sprungtabelle durch Schlüsselübereinstimmung und Sprung"

Mehrere Muster gleichzeitig abgleichen

Die `|` kann verwendet werden, um eine Übereinstimmung der einzelnen Case-Anweisungen mit mehreren Eingaben zu erzielen, um dasselbe Ergebnis zu erzielen:

```
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.
```

Beachten Sie, dass der Abgleich von **Werten auf** diese Weise gut funktioniert, der folgende Abgleich von **Typen** führt jedoch zu Problemen:

```
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"
  }
}
```

Wenn Sie im letzteren Fall (mit _) nicht den Wert der ungebundenen Variablen benötigen und einfach etwas anderes tun möchten, können Sie Folgendes tun:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

Ansonsten müssen Sie Ihre Fälle aufteilen:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

Musterabgleich auf Tupeln

Gegeben die folgende `List` von Tupeln:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

Mit Pattern Matching kann jedes Element unterschiedlich behandelt werden:

```
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")
  }
}
```

Der erste Fall zeigt, wie Sie mit einer bestimmten Zeichenfolge abgleichen und den entsprechenden Preis erhalten. Im zweiten Fall wird die Extraktion von if und tuple verwendet , um mit Elementen des Tupels übereinzustimmen.

Musterabgleich online lesen: https://riptutorial.com/de/scala/topic/661/musterabgleich

Kapitel 30: Optionsklasse

Syntax

- Klasse Einige [+ T] (Wert: T) erweitert Option [T]
- Objekt Keine erweitert Option [Nichts]
- Option [T] (Wert: T)

  Konstruktor, um je nach angegebenem `Some(value)` entweder einen `Some(value)` oder `None` zu erstellen.

Examples

Optionen als Sammlungen

`Option` haben einige nützliche Funktionen höherer Ordnung, die leicht verstanden werden können, wenn Optionen als *Sammlungen mit null oder einem Element* `None` verhält sich `None` wie die leere Sammlung und `Some(x)` wie eine Sammlung mit einem einzelnen Element, `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
```

Option anstelle von Null verwenden

In Java (und anderen Sprachen) ist die Verwendung von `null` eine häufige Methode, um anzuzeigen, dass an eine Referenzvariable kein Wert angehängt ist. In Scala wird die Verwendung von `Option` Verwendung von `null` vorgezogen. `Option` umschließt Werte, die *möglicherweise* `null` .

`None` ist eine Unterklasse von `Option` die eine Nullreferenz einschließt. `Some` sind eine Unterklasse von `Option` die eine nicht-null-Referenz einschließt.

Eine Referenz einwickeln ist einfach:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

Dies ist ein typischer Code, wenn eine Java-Bibliothek aufgerufen wird, die möglicherweise eine Nullreferenz zurückgibt:

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
// else resource = Some(resource)
```

Wenn `getResource()` einen `null` zurückgibt, ist die `resource` ein `None` Objekt. Andernfalls handelt es sich um ein `Some(resource)` Objekt `Some(resource)` . Die bevorzugte Methode zum Umgang mit einer `Option` ist die Verwendung von Funktionen höherer Ordnung innerhalb des `Option` . Wenn Sie beispielsweise prüfen möchten, ob Ihr Wert nicht `None` (ähnlich wie beim Überprüfen, ob `value == null`), würden Sie die `isDefined` Funktion verwenden:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

Um nach einer `null` suchen, können Sie Folgendes tun:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

Es wird bevorzugt, dass Sie die bedingte Ausführung des umschlossenen Werts einer `Option` (ohne die `Option.get` Methode "außergewöhnlich" `Option.get` verwenden), indem Sie die `Option` als Monade behandeln und `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
```

Wenn eine `Resource` erforderlich ist (im Gegensatz zu einer `Option[Resource]` -Instanz), können Sie `Option` zum Schutz vor NULL-Werten verwenden. Hier liefert die `getOrElse` Methode einen Standardwert:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

Java-Code kann Scalas `Option` nicht ohne weiteres verarbeiten. Wenn Sie Werte an Java-Code übergeben, ist es eine gute Form, eine `Option` zu entpacken, wobei `null` oder ein sinnvoller Standardwert übergeben wird:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

Grundlagen

Eine `Option` ist eine Datenstruktur, die entweder einen einzelnen oder keinen Wert enthält. Eine `Option` kann als Sammlung von null oder einem Element betrachtet werden.

Option ist eine abstrakte Klasse mit zwei Kindern: `Some` und `None` .

`Some` enthält einen einzelnen Wert und " `None` enthält keinen Wert.

`Option` ist in Ausdrücken nützlich, die andernfalls `null` , um das Fehlen eines konkreten Werts darzustellen. Dies schützt vor einer `NullPointerException` und ermöglicht die Zusammensetzung vieler Ausdrücke, die möglicherweise keinen Wert mit Kombinatoren wie `Map` , `FlatMap` usw. zurückgeben.

Beispiel mit 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]` ist **versiegelt** und kann daher nicht erweitert werden. Daher ist die Semantik stabil und verlässlich.

Optionen für Verständnis

`Option` haben eine `flatMap` Methode. Dies bedeutet, dass sie für ein Verständnis verwendet werden können. Auf diese Weise können wir reguläre Funktionen zur Bearbeitung von `Option` ohne sie neu definieren zu müssen.

```
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)
```

Wenn einer der Werte " `None` ist, lautet das Endergebnis der Berechnung " `None` .

```
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
```

Hinweis: Dieses Muster erstreckt sich allgemeiner auf Konzepte, die `Monad` . (Weitere Informationen sollten auf Seiten verfügbar sein, die sich auf Verständnis und `Monad` 's beziehen.)

Im Allgemeinen ist es nicht möglich, verschiedene Monaden zum Verständnis zu mischen. Da `Option` jedoch leicht in eine `Iterable` konvertiert werden kann, können Sie `Option` und `Iterable` s einfach mischen, indem Sie die `.toIterable` Methode aufrufen.

```
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)
```

Eine kleine Anmerkung: Wenn wir unser Verständnis für das Verständnis definiert hätten, würde sich das Gegenteil um das Verständnis herum ausrechnen, da unsere Option implizit konvertiert würde. Aus diesem Grund ist es sinnvoll, diese `.toIterable` Funktion (oder die entsprechende Funktion, abhängig von der verwendeten Sammlung) immer aus `.toIterable` hinzuzufügen.

Optionsklasse online lesen: https://riptutorial.com/de/scala/topic/2293/optionsklasse

Kapitel 31: Pakete

Einführung

Pakete in Scala verwalten Namespaces in großen Programmen. Zum Beispiel kann der Name `connection` kann in den Paketen auftreten `com.sql` und `org.http` . Sie können die vollständig qualifizierten `com.sql.connection` und `org.http.connection` verwenden, um auf jedes dieser Pakete zuzugreifen.

Examples

Paketstruktur

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

Pakete und Dateien

Die Paketklausel ist nicht direkt an die Datei gebunden, in der sie gefunden wurde. Es ist möglich, gemeinsame Elemente der Paketklausel in verschiedenen Dateien zu finden. Die unten aufgeführten Paketklauseln finden Sie beispielsweise in der Datei math1.scala und in der Datei math2.scala.

Datei math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

Datei math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

Datei 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()
    }
}
```

Convension der Paketbenennung

Scala-Pakete sollten den Namenskonventionen für Java-Pakete folgen.
Paketnamen werden in Kleinbuchstaben geschrieben, um Konflikte mit den Namen von Klassen oder Interfaces zu vermeiden. Unternehmen verwenden ihren umgekehrten Internet-Domänennamen, um ihre Paketnamen zu beginnen, z. B.

```
io.super.math
```

Pakete online lesen: https://riptutorial.com/de/scala/topic/8231/pakete

Kapitel 32: Parallele Sammlungen

Bemerkungen

Parallele Kollektionen erleichtern die parallele Programmierung, indem Parallelisierungsdetails auf niedriger Ebene ausgeblendet werden. Dies macht es einfach, Multi-Core-Architekturen zu nutzen. Beispiele für parallele Sammlungen umfassen `ParArray` , `ParVector` , `mutable.ParHashMap` , `immutable.ParHashMap` und `ParRange` . Eine vollständige Liste finden Sie in der Dokumentation .

Examples

Parallele Sammlungen erstellen und verwenden

Um eine parallele Sammlung aus einer sequentiellen Sammlung zu erstellen, rufen Sie die `par` Methode auf. Um eine sequentielle Sammlung aus einer parallelen Sammlung zu erstellen, rufen Sie die Methode `seq` . Dieses Beispiel zeigt, wie Sie einen regulären `Vector` in einen `ParVector` und dann wieder zurückkehren:

```
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)
```

Die `par` Methode kann verkettet werden, sodass Sie eine sequentielle Auflistung in eine parallele Auflistung konvertieren und sofort eine Aktion darauf ausführen können:

```
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 diesen Beispielen wird die Arbeit in mehrere Verarbeitungseinheiten aufgeteilt und nach Abschluss der Arbeit wieder zusammengefügt - ohne dass ein Eingreifen des Entwicklers erforderlich ist.

Fallstricke

Verwenden Sie keine parallelen Sammlungen, wenn die Sammlungselemente in einer bestimmten Reihenfolge empfangen werden müssen.

Parallele Sammlungen führen gleichzeitig Operationen durch. Das bedeutet, dass die gesamte Arbeit in Teile aufgeteilt und an verschiedene Prozessoren verteilt wird. Jeder Verarbeiter weiß

nicht, welche Arbeit andere erledigen. Wenn die *Reihenfolge der Sammlung von* Belang ist, ist die parallel verarbeitete Arbeit nicht deterministisch. (Wenn Sie denselben Code zweimal ausführen, kann dies zu unterschiedlichen Ergebnissen führen.)

Nicht assoziative Operationen

Wenn eine Operation nicht assoziativ ist (wenn die Reihenfolge der Ausführung von Belang ist), ist das Ergebnis einer parallelisierten Sammlung nicht deterministisch.

```
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
```

Nebenwirkungen

Operationen, die Nebeneffekte haben, wie z. B. `foreach` , werden bei parallelisierten Sammlungen aufgrund von `foreach` möglicherweise nicht wie gewünscht ausgeführt. Vermeiden Sie dies, indem Sie Funktionen verwenden, die keine Nebenwirkungen haben, wie z. B. `reduce` oder `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...
```

Parallele Sammlungen online lesen: https://riptutorial.com/de/scala/topic/3882/parallele-sammlungen

Kapitel 33: Parser-Kombinatoren

Bemerkungen

ParseResult-Fälle

Ein `ParseResult` in drei `ParseResult` :

- Erfolg, mit einer Markierung für den Beginn des Matches und dem nächsten übereinstimmenden Zeichen.
- Fehler, mit einer Markierung für den Beginn des Versuchs In diesem Fall führt der Parser zu dieser Position zurück, wo er sich befinden wird, wenn die Analyse fortgesetzt wird.
- Fehler, der die Analyse stoppt. Es erfolgt kein Backtracking oder weiteres Parsen.

Examples

Basisbeispiel

```
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)
```

Die Ausgabe lautet wie folgt:

```
[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]` im Beispiel von `Alice` zeigt an, dass der Beginn des Matches an Position `1` ist und der erste Charakter, der noch passt, an Position `6` beginnt.

Parser-Kombinatoren online lesen: https://riptutorial.com/de/scala/topic/3730/parser-kombinatoren

Kapitel 34: Programmierung auf Typebene

Examples

Einführung in die Programmierung auf Typebene

Wenn wir eine heterogene Liste betrachten, bei der die Elemente der Liste unterschiedliche, aber bekannte Typen haben, kann es wünschenswert sein, Operationen an den Elementen der Liste gemeinsam durchführen zu können, ohne die Typinformationen der Elemente zu verwerfen. Im folgenden Beispiel wird eine Mapping-Operation über eine einfache heterogene Liste implementiert.

Da der Elementtyp variiert, ist die Klasse von Operationen , die wir durchführen kann , um eine Form des Typs Projektion beschränkt, so definieren wir eine trait `Projection` abstract mit `type Apply[A]` der Berechnung der *Ergebnistyp* des Vorsprunges und `def apply[A](a: A): Apply[A]` den Ergebniswert der Projektion zu *berechnen* .

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

Bei der Implementierung des `type Apply[A]` programmieren wir auf der Typebene (im Gegensatz zur Wertebene).

Unser heterogener Listentyp definiert eine `map` Operation durch den gewünschten Vorsprung parametrisiert sowie die Typ der Projektion. Das Ergebnis der `HList` ist abstrakt, variiert je nach Implementierung von Klasse und Projektion und muss natürlich immer noch eine `HList` :

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

Im Fall von `HNil` , der leeren heterogenen Liste, ist das Ergebnis einer Projektion immer selbst. Hier erklären wir das `trait HNil` als Bequemlichkeit, damit wir `HNil` anstelle von `HNil.type` als Typ schreiben `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` ist die nicht leere heterogene Liste. Hier behaupten wir, dass der resultierende Kopftyp bei Anwendung einer Kartenoperation derjenige ist, der sich aus der Anwendung der Projektion auf den Kopfwert (`P#Apply[H]`) ergibt, und dass der resultierende Schwanztyp derjenige ist, der sich

aus der Abbildung des ergibt Projektion über den Schwanz (`T#Map[P]`), der als `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))
}
```

Die naheliegendste Projektion ist die Ausführung einer Form der Umhüllung. Das folgende Beispiel führt zu einer Instanz von `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)
})
```

Programmierung auf Typebene online lesen:
https://riptutorial.com/de/scala/topic/3738/programmierung-auf-typebene

Kapitel 35: Quasiquoten

Examples

Erstellen Sie einen Syntaxbaum mit Quasiquoten

Verwenden Sie Quasiquoten, um einen `Tree` in einem Makro zu erstellen.

```
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)
  }
}
```

Es kann beliebig komplex sein, wird aber für die korrekte Scala-Syntax validiert.

Quasiquoten online lesen: https://riptutorial.com/de/scala/topic/4032/quasiquoten

Kapitel 36: Reflexion

Examples

Laden einer Klasse mit Reflektion

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Reflexion online lesen: https://riptutorial.com/de/scala/topic/5824/reflexion

Kapitel 37: Reguläre Ausdrücke

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, Ersetzer: Match => String): String
- re.replaceAllIn (s: CharSequence, Ersetzung: String): String
- re.replaceFirstIn (s: CharSequence, Ersetzung: String): String
- re.replaceSomeIn (s: CharSequence, Ersetzer: Match => Option [String]): String
- re.split (s: CharSequence): Array [String]

Examples

Reguläre Ausdrücke deklarieren

Die `r` Methode, die implizit über scala.collection.immutable.StringOps bereitgestellt wird, erzeugt eine Instanz von scala.util.matching.Regex aus der Betreffzeichenfolge. Die dreifach zitierte Stringsyntax von Scala ist hier hilfreich, da Sie Backslashes nicht wie in Java umgehen müssen:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex implementiert eine idiomatische API für reguläre Ausdrücke für Scala als Wrapper über java.util.regex.Pattern . Die unterstützte Syntax ist dieselbe. Die Unterstützung von Scala für mehrzeilige String-Literale macht das `x` Flag jedoch wesentlich nützlicher, indem Kommentare aktiviert und Muster-Whitespace ignoriert werden:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

Es gibt eine überladene Version von `r` , `def r(names: String*): Regex` , mit dem Sie Ihren Pattern-Captures Gruppennamen zuweisen können. Dies ist etwas spröde, da die Namen von den Captures getrennt werden und sollten nur verwendet werden, wenn der reguläre Ausdruck an mehreren Stellen verwendet wird:

```
"""(\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 => ???
}
```

Wiederholen des Abgleichs eines Musters in einer Zeichenfolge

```
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)
```

Reguläre Ausdrücke online lesen: https://riptutorial.com/de/scala/topic/2891/regulare-ausdrucke

Kapitel 38: Rekursion

Examples

Schwanzrekursion

Bei der regulären Rekursion wird bei jedem rekursiven Aufruf ein weiterer Eintrag in den Aufrufstapel verschoben. Wenn die Rekursion abgeschlossen ist, muss die Anwendung jeden Eintrag ganz nach unten ziehen. Wenn es viele rekursive Funktionsaufrufe gibt, kann dies zu einem riesigen Stack führen.

Scala entfernt automatisch die Rekursion, falls der rekursive Aufruf in Endposition gefunden wird. Die Annotation (`@tailrec`) kann zu rekursiven Funktionen hinzugefügt werden, um sicherzustellen, dass die Optimierung der `@tailrec` durchgeführt wird. Der Compiler zeigt dann eine Fehlermeldung an, wenn er Ihre Rekursion nicht optimieren kann.

Regelmäßige Rekursion

Dieses Beispiel ist nicht rekursiv, da die Funktion beim Aufruf des rekursiven Aufrufs die Multiplikation verfolgen muss, die sie mit dem Ergebnis zu tun hat, nachdem der Aufruf zurückgekehrt ist.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

Der Funktionsaufruf mit dem Parameter führt zu einem Stack, der wie folgt aussieht:

```
(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
```

Wenn Sie versuchen, dieses Beispiel mit `@tailrec` zu kommentieren, wird die folgende Fehlermeldung `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

Schwanzrekursion

Bei der Tail-Rekursion führen Sie zuerst Ihre Berechnungen aus. Anschließend führen Sie den rekursiven Aufruf aus und übergeben die Ergebnisse Ihres aktuellen Schritts an den nächsten rekursiven Schritt.

```
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))
```

Im Gegensatz dazu sieht die Stapelverfolgung für die rekursive Faktorsicht aus dem Schwanz folgendermaßen aus:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

Es ist nur erforderlich, für jeden Aufruf von `fact_inside` die gleiche Datenmenge zu `fact_inside` da die Funktion einfach den Wert `fact_inside` den sie direkt nach oben erhalten hat. Das heißt, auch wenn `fact_with_tail 1000000` aufgerufen wird, benötigt es nur so viel Platz wie `fact_with_tail 3` . Dies ist bei einem nicht rekursiven Aufruf nicht der Fall, und große Werte können einen Stapelüberlauf verursachen.

Stapellose Rekursion mit Trampolin (scala.util.control.TailCalls)

Es ist sehr üblich, beim Aufrufen der rekursiven Funktion einen `StackOverflowError` Fehler zu erhalten. Die Scala-Standardbibliothek bietet TailCall an, um einen Stapelüberlauf zu vermeiden, indem Heap-Objekte und Fortsetzungen verwendet werden, um den lokalen Zustand der Rekursion zu speichern.

Zwei Beispiele aus dem scaladoc von 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
```

Rekursion online lesen: https://riptutorial.com/de/scala/topic/3889/rekursion

Kapitel 39: Sammlungen

Examples

Liste sortieren

Angenommen, die folgende Liste kann auf verschiedene Arten sortiert werden.

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

Das Standardverhalten von `sorted()` ist die Verwendung von `math.Ordering` , was für Strings zu einer lexographischen Sortierung führt:

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

`sortWith` können Sie Ihre eigene Bestellung über eine Vergleichsfunktion `sortWith` :

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

`sortBy` können Sie eine Transformationsfunktion bereitstellen:

```
//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)
```

Sie können eine Liste oder eine sortierte Liste jederzeit umkehren, indem Sie Folgendes verwenden:

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

Listen können auch mit der Java-Methode `java.util.Arrays.sort` und ihrem Scala-Wrapper `scala.util.Sorting.quickSort` sortiert werden

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

Diese Methoden können die Leistung beim Sortieren größerer Sammlungen verbessern, wenn die Konvertierung der Sammlung und das Unboxing / Boxing vermieden werden können. Weitere

Informationen zu den Leistungsunterschieden finden Sie in der Scala Collection, sortiert, sortWith und sortBy Performance .

Erstellen Sie eine Liste mit n Kopien von x

Verwenden Sie die Füllmethode, um eine Sammlung von `n` Kopien eines Objekts `x` zu erstellen. In diesem Beispiel wird eine `List` . Dies kann jedoch mit anderen Sammlungen funktionieren, für die `fill` sinnvoll ist:

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

Liste und Vektor-Spickzettel

> Es ist jetzt eine bewährte Methode, `Vector` anstelle von `List` da die Implementierungen eine bessere Leistung bieten. Performance-Eigenschaften finden Sie hier . `Vector` kann überall verwendet werden, wo `List` verwendet wird.

Listenerstellung

```
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
```

Nimm das 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
```

Elemente voranstellen

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

Elemente anhängen

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

Listen verbinden (verketten)

```
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)
```

Gemeinsame Operationen

```
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"
```

Kartensammlung Cheatsheet

> Beachten Sie, dass dies die Erstellung einer Sammlung vom Typ `Map` betrifft, die sich von der `map` Methode unterscheidet.

Kartenerstellung

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

Eine Map kann für die meisten Operationen als eine Sammlung von `tuples` werden, wobei das erste Element der Schlüssel und das zweite Element der Wert ist.

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

Element abrufen

```
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)
```

Element (e) hinzufügen

```
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)
```

Gemeinsame Operationen

Bei Operationen, bei denen eine Iteration über eine Karte stattfindet (`map` , `find` , `forEach` usw.), sind die Elemente der Sammlung `tuples` . Der Funktionsparameter kann entweder die Tupel-

Accessoren (`_1` , `_2`) oder eine Teilfunktion mit einem Case-Block verwenden:

```
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
```

Karte und Filter über eine Sammlung

Karte

'Mapping' über eine Sammlung hinweg verwendet die `map` , um jedes Element dieser Sammlung auf ähnliche Weise zu transformieren. Die allgemeine Syntax lautet:

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

Sie können eine anonyme Funktion bereitstellen:

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

Multiplikation ganzzahliger Zahlen mit zwei

```
// 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` wird verwendet, wenn Sie bestimmte Elemente einer Sammlung ausschließen oder ausfiltern möchten. Wie bei `map` nimmt die allgemeine Syntax eine Funktion an, aber diese Funktion muss einen `Boolean` :

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

Sie können eine anonyme Funktion direkt bereitstellen:

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

Paarnummern prüfen

```
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)
```

Weitere Karten- und Filterbeispiele

```
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 )
```

Einführung in die Scala-Sammlungen

Das Scala Collections-Framework ist laut seinen Autoren einfach zu bedienen, präzise, sicher, schnell und universell.

Das Framework besteht aus Scala- Merkmalen , die als Bausteine für das Erstellen von Sammlungen konzipiert wurden. Weitere Informationen zu diesen Bausteinen finden Sie in der offiziellen Übersicht der Scala-Sammlungen .

Diese integrierten Sammlungen sind in unveränderliche und veränderliche Pakete unterteilt. Standardmäßig werden die unveränderlichen Versionen verwendet. Beim Erstellen einer `List()` (ohne etwas zu importieren) wird eine *unveränderliche* Liste erstellt.

Eine der leistungsfähigsten Funktionen des Frameworks ist die konsistente und

benutzerfreundliche Oberfläche für gleichgesinnte Sammlungen. Das Summieren aller Elemente in einer Collection ist beispielsweise für Listen, Sets, Vektoren, Seqs und Arrays gleich:

```
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
```

Diese gleichgesinnten Typen erben von der `Traversable` Eigenschaft.

> Es ist jetzt eine bewährte Methode, `Vector` anstelle von `List` da die Implementierungen eine bessere Leistung bieten. Performance-Eigenschaften finden Sie hier . `Vector` kann überall verwendet werden, wo `List` verwendet wird.

Verfahrbare Typen

Auflistungsklassen mit der Eigenschaft " `Traversable` implementieren `foreach` und erben viele Methoden zum Ausführen allgemeiner Operationen für Auflistungen, die alle identisch funktionieren. Die häufigsten Vorgänge sind hier aufgelistet:

- Map - `map` , `flatMap` und `collect` erzeugen neue Sammlungen, indem auf jedes Element der ursprünglichen Sammlung eine Funktion `flatMap` wird.

```
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")
```

- Konvertierungen - `toList` , `toArray` und viele andere Konvertierungsvorgänge ändern die aktuelle Sammlung in eine spezifischere Art der Sammlung. Dies sind normalerweise Methoden, denen 'to' vorangestellt wird, und der spezifischere Typ (dh 'toList' konvertiert in eine `List`).

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

- Größeninfo - `isEmpty` , `nonEmpty` , `size` und `hasDefiniteSize` sind alle Metadaten über das Set. Dies ermöglicht bedingte Operationen für die Sammlung oder für Code, um die Größe der Sammlung zu bestimmen, einschließlich, ob sie unendlich oder diskret ist.

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

- Elementabruf - `head` , `last` , `find` und ihre `Option` werden verwendet, um das erste oder letzte Element abzurufen oder ein bestimmtes Element in der Auflistung zu suchen.

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

- Abrufvorgänge für Unterauflistungen - `filter` , `tail` , `slice` , `drop` und andere Vorgänge ermöglichen die Auswahl von Teilen der Sammlung, die weiter bearbeitet werden können.

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

- Unterteilungsoperationen - `partition` , `splitAt` , `span` und `groupBy` die aktuelle Sammlung in verschiedene Teile `groupBy` .

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

- Elementtests - `exists` und `count` sind Vorgänge, mit denen diese Sammlung `forall` wird, um `forall` , ob sie ein Vergleichselement erfüllt.

```
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` , `foldRight` (`:\`), `reduceLeft` und `reduceRight` werden verwendet, um binäre Funktionen auf aufeinanderfolgende Elemente in der Auflistung anzuwenden. Hier finden Sie Beispiele für Faltblätter und hier für reduzierte Beispiele .

Falten

Die `fold` Methode durchläuft eine Sammlung, wobei ein anfänglicher Akkumulatorwert verwendet wird und eine Funktion angewendet wird, die jedes Element zum erfolgreichen Aktualisieren des Akkumulators verwendet:

```
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
```

Im obigen Beispiel wurde eine anonyme Funktion für `fold()` bereitgestellt. Sie können auch eine benannte Funktion verwenden, die zwei Argumente akzeptiert. Wenn man dies in mir trägt, kann das obige Beispiel folgendermaßen umgeschrieben werden:

```
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
```

Das Ändern des Anfangswerts wirkt sich auf das Ergebnis aus:

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

Die `fold` Methode hat zwei Varianten - `foldLeft` und `foldRight` .

`foldLeft()` durchläuft von links nach rechts (vom ersten Element der Sammlung bis zum letzten in dieser Reihenfolge). `foldRight()` iteriert von rechts nach links (vom letzten bis zum ersten Element). `fold()` iteriert von links nach rechts wie `foldLeft()` . In der Tat ruft `fold()` tatsächlich `foldLeft()` .

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

`fold()` , `foldLeft()` und `foldRight()` geben einen Wert zurück, der denselben Typ hat wie der ursprüngliche Wert. Im Gegensatz zu `foldLeft()` und `foldRight()` kann der für `fold()` angegebene Anfangswert jedoch nur vom selben Typ oder einem Supertyp des Typs der Sammlung sein.

In diesem Beispiel ist die Reihenfolge nicht relevant. Sie können `fold()` in `foldLeft()` oder `foldRight()` und das Ergebnis bleibt gleich. Die Verwendung einer auftragsempfindlichen Funktion ändert die Ergebnisse.

Im Zweifelsfall ziehen Sie `foldLeft()` vor `foldRight()` . `foldRight()` ist weniger performant.

Für jeden

`foreach` ist unter den Iteratoren der Sammlungen ungewöhnlich, da es kein Ergebnis zurückgibt. Stattdessen wird auf jedes Element eine Funktion angewendet, die nur Nebeneffekte hat. Zum Beispiel:

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

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

Die an `foreach` gelieferte Funktion kann einen beliebigen Rückgabetyp haben, das Ergebnis wird jedoch verworfen . In der Regel wird `foreach` verwendet, wenn Nebenwirkungen erwünscht sind. Wenn Sie Daten transformieren möchten, sollten Sie `map` , `filter` , ein `for comprehension` oder eine andere Option verwenden.

Beispiel für das Verwerfen von Ergebnissen

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

Reduzieren

Die Methoden `reduceRight reduce()` , `reduceLeft()` und `reduceRight` (ähnlich `reduceLeft()` ähneln Falten. Die zum Reduzieren übergebene Funktion nimmt zwei Werte und liefert einen dritten Wert. Wenn Sie mit einer Liste arbeiten, sind die ersten beiden Werte die ersten beiden Werte in der Liste. Das Ergebnis der Funktion und der nächste Wert in der Liste werden dann erneut auf die Funktion angewendet und ergeben ein neues Ergebnis. Dieses neue Ergebnis wird mit dem nächsten Wert der Liste usw. angewendet, bis keine Elemente mehr vorhanden sind. Das Endergebnis wird zurückgegeben.

```
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
```

Es gibt einige Unterschiede in der Funktionsweise der Reduktionsfunktionen im Vergleich zu den Falzfunktionen. Sie sind:

1. Die Reduzierfunktionen haben keinen anfänglichen Speicherwert.
2. Reduktionsfunktionen können bei leeren Listen nicht aufgerufen werden.
3. Reduktionsfunktionen können nur den Typ oder den Supertyp der Liste zurückgeben.

Sammlungen online lesen: https://riptutorial.com/de/scala/topic/686/sammlungen

Kapitel 40: Scala einrichten

Examples

Unter Linux über dpkg

Bei Debian-basierten Distributionen, einschließlich Ubuntu, können Sie die `.deb` Installationsdatei am einfachsten verwenden. Gehen Sie zur Scala-Website . Wählen Sie die Version, die Sie installieren möchten, scrollen Sie nach unten und suchen Sie nach `scala-xxxdeb` .

Sie können die Scala Deb von der Kommandozeile aus installieren:

```
sudo dpkg -i scala-x.x.x.deb
```

Um zu überprüfen, ob es korrekt installiert ist, geben Sie an der Eingabeaufforderung des Terminals Folgendes ein:

```
which scala
```

Die zurückgegebene Antwort sollte derjenigen entsprechen, die Sie in Ihre PATH-Variable eingefügt haben. So überprüfen Sie, ob Scala funktioniert:

```
scala
```

Dies sollte die Scala REPL starten und die Version melden (die wiederum der heruntergeladenen Version entsprechen sollte).

Ubuntu-Installation über manuellen Download und Konfiguration

Laden Sie Ihre bevorzugte Version von Lightbend mit `curl` herunter:

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Entpacken Sie die `tar` - Datei in `/usr/local/share` oder `/opt/bin` :

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Fügen Sie den `PATH` zu `~/.profile` oder `~/.bash_profile` oder `~/.bashrc` indem Sie diesen Text in eine dieser Dateien `~/.bashrc` :

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

Um zu überprüfen, ob es korrekt installiert ist, geben Sie an der Eingabeaufforderung des

Terminals Folgendes ein:

```
which scala
```

Die zurückgegebene Antwort sollte derjenigen entsprechen, die Sie in Ihre `PATH` Variable `PATH` . So überprüfen Sie, ob `scala` funktioniert:

```
scala
```

Dies sollte die Scala REPL starten und die Version melden (die wiederum der heruntergeladenen Version entsprechen sollte).

Mac OSX über Macports

Öffnen Sie auf Mac OSX-Computern mit installierten MacPorts ein Terminalfenster und geben Sie Folgendes ein:

```
port list | grep scala
```

Hier werden alle verfügbaren Scala-Pakete aufgelistet. So installieren Sie eine (in diesem Beispiel die Version 2.11 von Scala):

```
sudo port install scala2.11
```

(Die Version `2.11` kann sich ändern, wenn Sie eine andere Version installieren möchten.)

Alle Abhängigkeiten werden automatisch installiert und der Parameter `$PATH` wird aktualisiert. Um zu überprüfen, dass alles funktioniert hat:

```
which scala
```

Dadurch wird der Pfad zur Scala-Installation angezeigt.

```
scala
```

Dies öffnet die Scala REPL und meldet die installierte Versionsnummer.

Scala einrichten online lesen: https://riptutorial.com/de/scala/topic/2921/scala-einrichten

Kapitel 41: Scala.js

Einführung

`Scala.js` ist eine Schnittstelle von `Scala` , die `JavaScript` kompiliert, das am Ende außerhalb der `JVM` . Es hat Vorteile wie starke Typisierung, Code-Optimierung zur Kompilierzeit und vollständige Interoperabilität mit JavaScript-Bibliotheken.

Examples

console.log in Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

Fettpfeilfunktionen

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

Einfache Klasse

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

Sammlungen

```
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}"
```

DOM manipulieren

```
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)
}
```

Verwendung mit SBT

Sbt-Abhängigkeit

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

Laufen

```
sbt run
```

Laufen mit kontinuierlicher Kompilierung:

```
sbt ~run
```

In eine einzige JavaScript-Datei übersetzen:

```
sbt fastOptJS
```

Scala.js online lesen: https://riptutorial.com/de/scala/topic/9426/scala-js

Kapitel 42: Scaladoc

Syntax

- Geht über Methoden, Felder, Klassen oder Pakete.
- Beginnt mit `/**`
- Jede Zeile hat einen Startvorgang `*` mit den Kommentaren
- Endet mit `*/`

Parameter

Parameter	Einzelheiten
Klassenspezifisch	_
`@constructor detail`	Erläutert den Hauptkonstruktor der Klasse
Methodenspezifisch	_
`@return detail`	Details dazu, was auf der Methode zurückgegeben wird.
Methoden-, Konstruktor- und / oder Klassen-Tags	_
`@param x detail`	Details zum Werteparameter `x` in einer Methode oder einem Konstruktor.
`@tparam x detail`	Details zum Typparameter `x` einer Methode oder eines Konstruktors.
`@throws detail`	Welche Ausnahmen können geworfen werden.
Verwendungszweck	_
`@see detail`	Verweise auf andere Informationsquellen.
`@note detail`	Fügt eine Notiz für Vor- oder Nachbedingungen oder andere wichtige Einschränkungen oder Erwartungen hinzu.
`@example detail`	Bietet Beispielcode oder zugehörige Beispieldokumentation.
`@usecase detail`	Stellt eine vereinfachte Methodendefinition bereit, wenn die vollständige Methodendefinition zu komplex oder zu laut ist.

Parameter	Einzelheiten
Andere	_
`@author detail`	Bietet Informationen zum Autor der folgenden.
`@version detail`	Stellt die Version bereit, zu der dieser Teil gehört.
`@deprecated detail`	Markiert die folgende Entität als veraltet.

Examples

Einfache Scaladoc-Methode

```
/**
  * 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
}
```

Scaladoc online lesen: https://riptutorial.com/de/scala/topic/4518/scaladoc

Kapitel 43: Scalaz

Einführung

Scalaz ist eine Scala-Bibliothek zur funktionalen Programmierung.

Es stellt rein funktionale Datenstrukturen bereit, die die aus der Scala-Standardbibliothek ergänzen. Es definiert einen Satz `Functor` Typklassen (z. B. `Functor` , `Monad`) und entsprechende Instanzen für eine große Anzahl von Datenstrukturen.

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]]
```

Scalaz online lesen: https://riptutorial.com/de/scala/topic/9893/scalaz

Kapitel 44: Selbsttypen

Syntax

- trait Typ {selfId => / *andere Mitglieder können auf* `selfId` *verweisen, falls* `this` *etwas bedeutet* /}
- trait Typ {selfId: OtherType => / * andere Mitglieder können `selfId` und es wird vom Typ `OtherType` * / sein.
- trait Typ {selfId: OtherType1 mit OtherType2 => / * `selfId` ist vom Typ `OtherType1` und `OtherType2` * /

Bemerkungen

Wird oft mit dem Kuchenmuster verwendet.

Examples

Einfaches Selbsttyp-Beispiel

Self-Typen können in Merkmalen und Klassen verwendet werden, um Einschränkungen für die konkreten Klassen zu definieren, mit denen sie gemischt werden. Es ist auch möglich , eine andere Kennung für die verwenden `this` unter Verwendung dieser Syntax (nützlich , wenn äußere Objekt aus einem inneren Objekt verwiesen werden).

Angenommen, Sie möchten einige Objekte speichern. Dazu erstellen Sie Schnittstellen für die Speicherung und fügen einem Container Werte hinzu:

```
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.
  }
}
```

Auf diese Weise befinden sich diese nicht in derselben Objekthierarchie, aber `PermanentStorage` kann nicht implementiert werden, ohne auch `Container` implementieren.

Selbsttypen online lesen: https://riptutorial.com/de/scala/topic/4639/selbsttypen

Kapitel 45: Streams

Bemerkungen

Streams werden faul bewertet, dh sie können zur Implementierung von Generatoren verwendet werden, die auf Abruf ein neues Element des angegebenen Typs bereitstellen oder "erzeugen" und nicht vor der Tatsache. Dadurch wird sichergestellt, dass nur die erforderlichen Berechnungen ausgeführt werden.

Examples

Verwenden eines Streams zum Erzeugen einer zufälligen Sequenz

`genRandom` erstellt einen Stream von Zufallszahlen, der bei jedem `genRandom` eine Chance von eins zu vier hat.

```
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.
```

Beachten Sie das `#::` *-Konstrukt* , das *langsam rekonstruiert* : Da es die aktuelle Zufallszahl einem Stream voranstellt, wertet es den Rest des Streams nicht aus, bis er durchlaufen wird.

Unendliche Streams über Rekursion

Streams können aufgebaut werden, die sich selbst referenzieren und somit unendlich rekursiv werden.

```
// 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 diesem Zusammenhang ist der Unterschied zwischen Var, Val und Def interessant. Als `def` jedes Element jedes Mal neu berechnet, wenn es referenziert wird. Als `val` jedes Element beibehalten und wiederverwendet, nachdem es berechnet wurde. Dies kann demonstriert werden, indem bei jeder Berechnung ein Nebeneffekt erstellt wird.

```
// 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
```

Dies erklärt auch , warum die Zufallszahl - `Stream` nicht als Arbeit `val` .

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

Unendlich selbstreferenzierender 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, ?)")
```

Streams online lesen: https://riptutorial.com/de/scala/topic/3702/streams

Kapitel 46: String Interpolation

Bemerkungen

Diese Funktion ist in Scala 2.10.0 und höher verfügbar.

Examples

Hallo String Interpolation

Der **s-** Interpolator erlaubt die Verwendung von Variablen innerhalb eines Strings.

```
val name = "Brian"
println(s"Hello $name")
```

druckt "Hallo Brian" auf die Konsole, wenn er ausgeführt wird.

Formatierte String-Interpolation mit dem f-Interpolator

```
val num = 42d
```

Mit *f werden* zwei Nachkommastellen für `num` *gedruckt*

```
println(f"$num%2.2f")
42.00
```

Drucken Sie `num` wissenschaftlicher Notation mit *e*

```
println(f"$num%e")
4.200000e+01
```

`num` in hexadezimal mit *a* `num`

```
println(f"$num%a")
0x1.5p5
```

Weitere Formatzeichenfolgen finden Sie unter
https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

Ausdruck in String-Literalen verwenden

Sie können geschweifte Klammern verwenden, um Ausdrücke in String-Literale zu interpolieren:

```
def f(x: String) = x + x
val a = "A"
```

```
s"${a}"    // "A"
s"${f(a)}" // "AA"
```

Ohne die geschweiften Klammern interpoliert scala den *Bezeichner nur* nach dem `$` (in diesem Fall `f`). Da es keine implizite Konvertierung von `f` in einen `String` gibt, ist dies in diesem Beispiel eine Ausnahme:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

Benutzerdefinierte String-Interpolatoren

Es ist möglich, zusätzlich zu den integrierten Stringinterpolatoren eigene Zeichenfolgen zu definieren.

```
my"foo${bar}baz"
```

Wird vom Compiler erweitert um:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` hat keine `my` Methode und kann daher durch implizite Konvertierung bereitgestellt werden. Eine benutzerdefinierte Interpolator mit dem gleichen Verhalten wie das Builtin `s` Interpolator würde dann wie folgt umgesetzt werden:

```
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
  }
}
```

Und die Interpolation, die `my"foo${bar}baz"` hätte:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Beachten Sie, dass die Argumente oder der Rückgabetyp der Interpolationsfunktion nicht eingeschränkt sind. Dies führt uns zu einem dunklen Pfad, auf dem die Interpolationssyntax kreativ zum Erstellen beliebiger Objekte verwendet werden kann, wie im folgenden Beispiel veranschaulicht:

```
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
```

Stringinterpolatoren als Extraktoren

Es ist auch möglich, die String-Interpolationsfunktion von Scala zu verwenden, um aufwendige Extraktoren (Pattern-Matcher) zu erstellen, wie sie vielleicht am bekanntesten in der Quasiquoten-API von Scala-Makros verwendet werden.

Da `n"p0${i0}p1"` in den `new StringContext("p0", "p1").n(i0)` ist `new StringContext("p0", "p1").n(i0)` , ist es nicht überraschend, dass die Extraktorfunktionalität durch die implizite Konvertierung von `StringContext` in eine Klasse mit aktiviert wird Eigenschaft `n` eines Typs, der eine nicht `unapply` oder `unapplySeq` Methode definiert.

Betrachten Sie als Beispiel den folgenden Extraktor, der `StringContext` extrahiert, indem Sie einen regulären Ausdruck aus den `StringContext` Teilen `StringContext` . Wir können dann den Großteil des schweren `unapplySeq` an die `unapplySeq` Methode `unapplySeq` , die von der resultierenden scala.util.matching.Regex bereitgestellt wird:

```
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 _ => ???
}
```

Beachten Sie, dass das `path` auch eine `apply` Methode definieren kann `apply` um sich auch als regulärer Interpolator zu verhalten.

Raw String Interpolation

Sie können das **Roh** - Interpolator verwenden , wenn Sie eine Zeichenfolge gedruckt werden soll , wie und ohne Entkommen von Literalen.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Mit dem Einsatz des **Roh** - Interpolator, sollten Sie die folgenden gedruckt in der Konsole angezeigt:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Ohne das **Roh** - Interpolator, `\n` und `\t` entgangen worden wäre.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Drucke:

```
Hello World In English And French
English:       Hello World
French:        Bonjour Le Monde
```

String Interpolation online lesen: https://riptutorial.com/de/scala/topic/1629/string-interpolation

Kapitel 47: Symbol Literals

Bemerkungen

Scala verfügt über ein Konzept von **Symbolen** - Zeichenfolgen, die *intern sind* , das heißt: Zwei Symbole mit demselben Namen (dieselbe Zeichenfolge) beziehen sich im Gegensatz zu Zeichenfolgen während der Ausführung auf dasselbe Objekt.

Symbole sind ein Merkmal vieler Sprachen: Lisp, Ruby und Erlang und mehr. In Scala sind sie jedoch von relativ geringem Nutzen. Gute Eigenschaft, trotzdem zu haben.

Benutzen:

Wörtliche Anfang mit einem einfachen Anführungszeichen `'` , gefolgt von einer oder mehreren Ziffern, Buchstaben oder Unter Partituren _ ist ein Symbol wörtlich zu nehmen . Das erste Zeichen ist eine Ausnahme, da es sich nicht um eine Ziffer handeln kann.

Gute definitionen:

```
'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")
```

Schlechte Definitionen:

```
'8'th_division
'94_pattern
'bad-format
```

Examples

Zeichenfolgen in case-Klauseln ersetzen

Nehmen wir an, wir haben mehrere Datenquellen, die *Datenbank, Datei, Eingabeaufforderung* und *Argumentliste enthalten* . Abhängig von der gewählten Quelle ändern wir unseren Ansatz:

```
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"))
```

```
}
```

Wir hätten `String` anstelle von `Symbol` . Dies ist jedoch nicht der Fall, da keine der Funktionen von Strings in diesem Zusammenhang nützlich ist.

Dies macht den Code einfacher und weniger fehleranfällig.

Symbol Literals online lesen: https://riptutorial.com/de/scala/topic/6419/symbol-literals

Kapitel 48: synchronisiert

Syntax

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

für ein Objekt synchronisieren

`synchronized` ist ein einfaches Parallelitätskonstrukt auf niedriger Ebene, mit dem verhindert werden kann, dass mehrere Threads auf dieselben Ressourcen zugreifen. Einführung in die JVM unter Verwendung der Java-Sprache .

```
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
}
```

Im Falle von `object` s synchronisiert es sich möglicherweise mit der Klasse des Objekts, nicht mit der Singleton-Instanz.

implizit auf dieses synchronisieren

```
 /* 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`
    */
 }
```

synchronisiert online lesen: https://riptutorial.com/de/scala/topic/3371/synchronisiert

Kapitel 49: Teilfunktionen

Examples

Zusammensetzung

Teilfunktionen werden häufig verwendet, um eine Gesamtfunktion in Teilen zu definieren:

```
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 dieser Verwendung werden die Teilfunktionen in der Reihenfolge der Verkettung mit der `orElse` Methode versucht. Normalerweise wird eine abschließende Teilfunktion bereitgestellt, die alle verbleibenden Fälle erfüllt. Insgesamt wirkt die Kombination dieser Funktionen als Gesamtfunktion.

Dieses Muster wird normalerweise verwendet, um Bedenken zu trennen, bei denen eine Funktion einen Dispatcher effektiv für unterschiedliche Codepfade einsetzen kann. Dies ist zum Beispiel bei der Empfangsmethode eines Akka-Schauspielers üblich .

Verwendung mit "Collect"

Während Teilfunktionen oft als bequeme Syntax für Gesamtfunktionen verwendet werden, indem eine abschließende Platzhalterübereinstimmung (`case _`) eingefügt wird, ist bei einigen Methoden deren Parteilichkeit der Schlüssel. Ein sehr verbreitetes Beispiel in idiomatischer Scala ist die `collect` Methode, die in der Scala-Sammlungsbibliothek definiert wird. Teilfunktionen ermöglichen hier die gemeinsamen Funktionen der Untersuchung der Elemente einer Sammlung, um diese abzubilden und / oder zu filtern, damit sie in einer kompakten Syntax vorkommen.

Beispiel 1

Angenommen, wir haben eine Quadratwurzelfunktion, die als Teilfunktion definiert ist:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

Wir können es mit dem `collect` Combinator aufrufen:

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

effektiv die gleiche Operation ausführen wie:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

Beispiel 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")
```

Im obigen Beispiel sind einige Punkte zu beachten:

- Die linke Seite jedes Mustermatches wählt effektiv Elemente aus, die verarbeitet und in die Ausgabe aufgenommen werden sollen. Jeder Wert, der keinen passenden `case` wird einfach weggelassen.
- Die rechte Seite definiert die anwendungsspezifische Verarbeitung.
- Pattern Matching bindet Variable zur Verwendung in Guard-Anweisungen (`if` Klauseln) und auf der rechten Seite.

Grundlegende Syntax

Scala verfügt über eine spezielle Funktion , die als Teilfunktion bezeichnet wird und die normalen Funktionen erweitert. `PartialFunction` bedeutet, dass eine `PartialFunction` Instanz überall dort eingesetzt werden kann, wo `Function1` erwartet wird. Teilfunktionen können anonym mit der `case` der Mustererkennung verwendeten Fallsyntax definiert werden :

```
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)
```

Wie im Beispiel zu sehen ist, muss eine Teilfunktion nicht über die gesamte Domäne des ersten

Parameters definiert werden. Eine Standard-Instanz von `Function1` wird als *total angenommen* , was bedeutet, dass sie für jedes mögliche Argument definiert ist.

Verwendung als Gesamtfunktion

Teilfunktionen sind in idiomatischen Scala sehr häufig. Sie werden häufig für ihre praktische `case` , um die Gesamtfunktionen über die Merkmale zu definieren:

```
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)
```

Dadurch wird die zusätzliche Syntax einer `match` in einer normalen anonymen Funktion gespeichert. Vergleichen Sie:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

Sie wird auch häufig verwendet, um eine Parameterzerlegung mit Hilfe von Pattern Matching durchzuführen, wenn ein Tupel oder eine Fallklasse an eine Funktion übergeben wird:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

Verwendung zum Extrahieren von Tupeln in einer Kartenfunktion

Diese drei Kartenfunktionen sind gleichwertig. Verwenden Sie also die Variante, die Ihr Team für lesbar hält.

```
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" })
```

Die Teilfunktion **muss mit allen Eingaben übereinstimmen** : Jeder Fall, der nicht übereinstimmt, löst zur Laufzeit eine Ausnahme aus.

Teilfunktionen online lesen: https://riptutorial.com/de/scala/topic/1638/teilfunktionen

Kapitel 50: Testen mit ScalaCheck

Einführung

ScalaCheck ist eine in Scala geschriebene Bibliothek, die zum automatisierten, objektbasierten Testen von Scala- oder Java-Programmen verwendet wird. ScalaCheck wurde ursprünglich von der Haskell-Bibliothek QuickCheck inspiriert, hat sich aber auch für sich entschieden.

ScalaCheck hat keine externen Abhängigkeiten außer der Scala-Laufzeitumgebung und funktioniert hervorragend mit sbt, dem Scala-Build-Tool. Es ist auch vollständig in die Test-Frameworks ScalaTest und specs2 integriert.

Examples

Scalacheck mit Scalatest und Fehlermeldungen

Beispiel für die Verwendung von Scalacheck mit Scalatest. Im Folgenden haben wir vier Tests:

- "zeige Passbeispiel" - es geht
- "einfaches Beispiel ohne benutzerdefinierte Fehlermeldung anzeigen" - nur fehlgeschlagene Nachricht ohne Details, `&&` boolescher Operator wird verwendet
- "Beispiel mit Fehlernachrichten bei Argument anzeigen" - Fehlermeldung bei Argument (`"argument" |:` :) Props.all-Methode wird anstelle von `&&`
- "Beispiel mit Fehlermeldungen beim Befehl anzeigen" - Fehlermeldung beim Befehl (`"command" |:` :) Props.all-Methode wird anstelle von `&&`

```
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)
```

```
    }

  }
}
```

Die Ausgabe (Fragmente):

```
[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
```

Testen mit ScalaCheck online lesen: https://riptutorial.com/de/scala/topic/9430/testen-mit-scalacheck

Kapitel 51: Testen mit ScalaTest

Examples

Hallo Weltspezifikationstest

Erstellen Sie eine Testklasse im Verzeichnis `src/test/scala` in einer Datei namens `HelloWorldSpec.scala` . Fügen Sie dies in die Datei ein:

```
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 ("")
  }
}
```

- In diesem Beispiel werden `FlatSpec` und `Matchers` , die Bestandteil der ScalaTest-Bibliothek sind.
- `FlatSpec` ermöglicht das `FlatSpec` Tests im **BDD-** Stil **(Behavior-Driven Development)** . In diesem Stil wird ein Satz verwendet, um das erwartete Verhalten einer bestimmten Codeeinheit zu beschreiben. Der Test bestätigt, dass der Code diesem Verhalten entspricht. Weitere Informationen finden Sie in der Dokumentation .

Spec Test Cheatsheet

Konfiguration

Bei den folgenden Tests werden diese Werte für die Beispiele verwendet.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

Typcheck

So überprüfen Sie den Typ für einen bestimmten `val` :

```
helloWorld shouldBe a [String]
```

Beachten Sie, dass die Klammern hier verwendet werden, um den Typ `String` .

Gleiche Kontrolle

Gleichheit prüfen:

```
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")
```

Nicht gleich prüfen

Um Ungleichheit zu testen:

```
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
```

Längenprüfung

Länge und / oder Größe überprüfen:

```
helloWorld should have length 11
helloWorldList should have size 2
```

Ausnahmen prüfen

So überprüfen Sie den Typ und die Nachricht einer Ausnahme:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

Binden Sie die ScalaTest Library in SBT ein

`build.sbt` mit SBT die Bibliotheksabhängigkeit hinzu , und fügen Sie `build.sbt` zu `build.sbt` :

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

Weitere Informationen finden Sie auf der ScalaTest-Website .

Testen mit ScalaTest online lesen: https://riptutorial.com/de/scala/topic/5506/testen-mit-scalatest

Kapitel 52: Tuples

Bemerkungen

Warum sind Tupel auf Länge 23 beschränkt?

Tupel werden vom Compiler als Objekte neu geschrieben. Der Compiler hat Zugriff auf `Tuple1` durch `Tuple22` . Diese willkürliche Grenze wurde von Sprachdesignern festgelegt.

Warum zählen Tupflängen von 0?

Ein `Tuple0` entspricht einer `Unit` .

Examples

Einen neuen Tupel erstellen

Ein Tupel ist eine heterogene Sammlung von zwei bis zweiundzwanzig Werten. Ein Tupel kann mit Klammern definiert werden. Für Tupel der Größe `2` (auch als "Paar" bezeichnet) gibt es eine Pfeilsyntax.

```
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` ist ein Tupel der Größe zwei. Um auf die Elemente eines Tupels `._1` , verwenden Sie `._1` bis `._22` . Mit `x._1` können wir `x._1` auf das erste Element des `x` Tupels zugreifen. `x._2` greift auf das zweite Element zu. Eleganter können Sie Tupel-Extraktoren verwenden .

Die Pfeilsyntax zum Erstellen von Tupeln der Größe zwei wird hauptsächlich in Maps verwendet. Hierbei handelt es sich um Sammlungen von `(key -> value)` -paaren:

```
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))
```

Die Syntax für das Paar in der Map ist die Pfeilsyntax. Damit wird klar, dass 1 der Schlüssel und a der mit diesem Schlüssel verknüpfte Wert ist.

Tupel in Sammlungen

Tupel werden häufig in Sammlungen verwendet, müssen jedoch auf bestimmte Weise gehandhabt werden. Geben Sie beispielsweise die folgende Liste von Tupeln an:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Es kann naheliegend sein, die Elemente durch implizites Entpacken des Tupels zusammenzufügen:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

Dies führt jedoch zu folgendem Fehler:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala kann die Tupel auf diese Weise nicht implizit auspacken. Wir haben zwei Möglichkeiten, diese Karte zu reparieren. Die erste besteht darin, die Positionszugriffe `_1` und `_2` :

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

Die andere Option ist die Verwendung einer `case` Anweisung zum Entpacken der Tupel mithilfe von Mustervergleich:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

Diese Einschränkungen gelten für alle Funktionen höherer Ordnung, die auf eine Sammlung von Tupeln angewendet werden.

Tuples online lesen: https://riptutorial.com/de/scala/topic/4971/tuples

Kapitel 53: Typ Inferenz

Examples

Lokale Typinferenz

Scala verfügt über einen leistungsfähigen, in die Sprache integrierten Typinterferenzmechanismus. Dieser Mechanismus wird als "lokale Typinferenz" bezeichnet:

```
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
```

Der Compiler kann den Typ der Variablen aus dem Initialisierungsausdruck ableiten. Ebenso kann der Rückgabetyp von Methoden weggelassen werden, da sie dem vom Methodenhauptteil zurückgegebenen Typ entsprechen. Die obigen Beispiele entsprechen den unten angegebenen expliziten Typdeklarationen:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

Typ Inferenz und Generics

Der Scala-Compiler kann auch Typparameter ableiten, wenn polymorphe Methoden aufgerufen werden oder wenn generische Klassen instanziiert werden:

```
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")
```

Die obige Form der Typeninferenz ähnelt dem in Java 7 eingeführten Diamond Operator .

Einschränkungen für Inferenz

Es gibt Szenarien, in denen die Scala-Typinferenz nicht funktioniert. Zum Beispiel kann der Compiler nicht auf die Art der Methodenparameter schließen:

```
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
```

Der Compiler kann nicht auf den Rückgabetyp rekursiver Methoden schließen:

```
// 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)
```

Keine Schlüsse ziehen

Basierend auf diesem Blogeintrag .

Angenommen, Sie haben eine Methode wie diese:

```
  def get[T]: Option[T] = ???
```

Wenn Sie versuchen, ihn ohne Angabe des generischen Parameters aufzurufen, wird `Nothing` abgeleitet, was für eine tatsächliche Implementierung nicht sehr nützlich ist (und das Ergebnis ist nicht nützlich). Mit der folgenden Lösung kann der `NotNothing` Kontext gebunden die Verwendung der Methode verhindern, ohne den erwarteten Typ anzugeben (in diesem Beispiel wird `RuntimeClass` ebenfalls ausgeschlossen, da `ClassTags` nicht `Nothing` , aber `RuntimeClass` wird abgeleitet):

```
@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]] = ???
}
```

Verwendungsbeispiel:

```
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()
```

```
}
```

Typ Inferenz online lesen: https://riptutorial.com/de/scala/topic/4918/typ-inferenz

Kapitel 54: Typabweichung

Examples

Kovarianz

Das Symbol `+` kennzeichnet einen Typparameter als *Kovariante* - hier heißt es " `Producer` ist Kovariante auf `A` ":

```
trait Producer[+A] {
  def produce: A
}
```

Ein kovarianter Parameter kann als "Ausgabe" -Typ betrachtet werden. Die Markierung `A` als Kovariante besagt, dass `Producer[X] <: Producer[Y]` vorausgesetzt, dass `X <: Y` Zum Beispiel ist ein `Producer[Cat]` ein gültiger `Producer[Animal]` , da alle produzierten Katzen auch gültige Tiere sind.

Ein kovarianter Parameter kann nicht an einer anderen Position (Eingabeposition) angezeigt werden. Das folgende Beispiel wird nicht kompiliert, da wir behaupten, dass `Co[Cat] <: Co[Animal]` , aber `Co[Cat] def handle(a: Cat): Unit` die kein `Animal` so behandeln kann, wie es von `Co[Animal]` verlangt wird!

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

Eine Möglichkeit, mit dieser Einschränkung umzugehen, besteht in der Verwendung von Typparametern, die durch den kovarianten Typparameter begrenzt werden. Im folgenden Beispiel wissen wir, dass `B` ein Supertyp von `A` . Wenn Sie also die `Option[X] <: Option[Y]` für `X <: Y def getOrElse[B >: X](b: => B): B` , wissen wir, dass die `Option[X] def getOrElse[B >: X](b: => B): B` jeden Supertyp von `X` akzeptiert - welche die von `Option[Y]` geforderten Supertypen von `Y` umfasst:

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

Invarianz

Standardmäßig sind alle Typparameter invariant - mit `trait A[B]` sagen wir, dass " `A` für `B` invariant ist". Dies bedeutet, dass wir bei zwei Parametrisierungen `A[Cat]` und `A[Animal]` keine Sub- / Superklassen-Beziehung zwischen diesen beiden Typen behaupten - es gilt nicht, dass `A[Cat] <: A[Animal]` oder `A[Cat] >: A[Animal]` unabhängig von der Beziehung zwischen `Cat` und `Animal` .

Varianzanmerkungen geben uns die Möglichkeit, eine solche Beziehung zu deklarieren, und legen

Regeln für die Verwendung von Typparametern fest, damit die Beziehung gültig bleibt.

Verstöße

Das `-` Symbol kennzeichnet einen Typparameter als *kontravariant* - hier sagen wir " `Handler` ist kontravariant für `A` ":

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

Ein kontravarianter Parameter kann als "Eingabe" -Typ betrachtet werden. Wenn `A` als kontravariant markiert wird, wird behauptet, dass `Handler[X] <: Handler[Y]` vorausgesetzt, dass `X >: Y` Ein `Handler[Animal]` ist beispielsweise ein gültiger `Handler[Cat]` , da ein `Handler[Animal]` auch mit Katzen umgehen muss.

Ein kontravarianter Typparameter kann nicht an der kovarianten Position (Ausgabe) angezeigt werden. Das folgende Beispiel wird nicht kompiliert, da wir behaupten, dass ein `Contra[Animal] <: Contra[Cat]` , ein `Contra[Animal]` jedoch `def produce: Animal` dem nicht garantiert wird, dass es gemäß `Contra[Cat]` Katzen produziert!

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Beachten Sie jedoch: Um die Auflösung zu überlasten, invertiert die Widersprüchlichkeit auch die Spezifität eines Typs auf den Parameter für den kontravarianten Typ. - `Handler[Animal]` wird als "spezifischer" als `Handler[Cat]` .

Da Methoden für Typparameter nicht überladen werden können, wird dieses Verhalten im Allgemeinen nur beim Auflösen impliziter Argumente problematisch. Im folgenden Beispiel wird `ofCat` niemals verwendet, da der Rückgabetyp `ofAnimal` spezifischer ist:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

Dieses Verhalten wird derzeit geplant , um in dotty zu ändern , und deshalb (als Beispiel) `scala.math.Ordering` von seinem Typ Parameter invariant ist `T` . Eine Problemumgehung besteht darin, Ihre Typklasse invariant zu machen und die implizite Definition für den Fall, für den sie auf Unterklassen eines bestimmten Typs angewendet werden soll, zu parametrisieren:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

Kovarianz einer Sammlung

Da Sammlungen in ihrem Elementtyp * normalerweise kovariant sind, kann eine Sammlung eines Untertyps übergeben werden, an dem ein Supertyp erwartet wird:

```
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
```

Es mag nicht magisch erscheinen, aber die Tatsache, dass ein `Seq[Dog]` von einer Methode akzeptiert wird, die ein `Seq[Animal]` erwartet, ist das gesamte Konzept eines höherwertigen Typs (hier: `Seq`), der in seinem Typparameter kovariant ist.

* Ein Gegenbeispiel ist das Set der Standardbibliothek

Kovarianz bei einer unveränderlichen Eigenschaft

Es gibt auch eine Möglichkeit, dass eine einzelne Methode ein kovariantes Argument akzeptiert, anstatt das gesamte Merkmal kovariant zu haben. Dies kann notwendig sein, weil Sie `T` in einer kontravarianten Position verwenden möchten, es aber dennoch kovariant sind.

```
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 {})
}
```

Typabweichung online lesen: https://riptutorial.com/de/scala/topic/1651/typabweichung

Kapitel 55: Typparameterisierung (Generics)

Examples

Der Optionstyp

Ein schönes Beispiel für einen parametrisierten Typ ist der Optionstyp . Dies ist im Wesentlichen nur die folgende Definition (mit mehreren weiteren Methoden, die für den Typ definiert sind):

```
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")
}
```

Wir können auch sehen, dass dies eine parametrisierte Methode `fold` , die etwas vom Typ `B` zurückgibt.

Parametrisierte Methoden

Der Rückgabetyp einer Methode kann vom *Typ* des Parameters abhängen. In diesem Beispiel ist `x` der Parameter, `A` der *Typ* von `x` , der als *Typparameter bezeichnet wird* .

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala verwendet die Typinferenz , um den Rückgabetyp zu bestimmen, der bestimmt, welche Methoden für den Parameter aufgerufen werden können. Daher ist Vorsicht geboten: Folgendes ist ein Fehler bei der Kompilierung, da `*` nicht für jeden Typ `A` :

```
def g[A](x: A): A = 2 * x  // Won't compile
```

Generische Sammlung

Definieren der Liste der Ints

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

Was aber, wenn wir die Liste von Boolean, Double usw. definieren müssen?

Generische Liste definieren

```
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")
}
```

Typparameterisierung (Generics) online lesen:
https://riptutorial.com/de/scala/topic/782/typparameterisierung--generics-

Kapitel 56: Überladung des Bedieners

Examples

Benutzerdefinierte Infix-Operatoren definieren

In Scala-Operatoren (wie `+` , `-` , `*` , `++` usw.) sind dies nur Methoden. Zum Beispiel kann `1 + 2` als `1.+(2)` . Diese Arten von Methoden werden als *"Infixoperatoren" bezeichnet* .

Dies bedeutet, dass benutzerdefinierte Methoden für Ihre eigenen Typen definiert werden können, wobei diese Operatoren wiederverwendet werden:

```
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)
  }
}
```

Diese als Methoden definierten Operatoren können wie folgt verwendet werden:

```
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
```

Beachten Sie, dass Infix-Operatoren nur ein einziges Argument haben können. Das Objekt vor dem Operator ruft seinen eigenen Operator für das Objekt nach dem Operator auf. Als Infix-Operator kann jede Scala-Methode mit einem einzigen Argument verwendet werden.

> Dies sollte mit Parcimony verwendet werden. Es wird im Allgemeinen nur dann als bewährte Methode angesehen, wenn Ihre eigene Methode genau das tut, was Sie von diesem Operator erwarten würden. Verwenden Sie im Zweifelsfall eine konservativere Benennung, z. B. `add` anstelle von `+` .

Benutzerdefinierte unäre Operatoren definieren

Unäre Operatoren können definiert werden, indem dem Operator `unary_` . Unäre Operatoren sind auf `unary_+` , `unary_-` , `unary_!` und `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)
  }
}
```

Der unäre Operator kann wie folgt verwendet werden:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> Dies sollte mit Parcimony verwendet werden. Das Überladen eines unären Operators mit einer Definition, die nicht erwartet wird, kann zu Code-Verwirrung führen.

Überladung des Bedieners online lesen: https://riptutorial.com/de/scala/topic/2271/uberladung-des-bedieners

Kapitel 57: Umfang

Einführung

Bereich in Scala definiert, von wo aus ein Wert (`def` , `val` , `var` oder `class`) aufgerufen werden kann.

Syntax

- Erklärung
- private erklärung
- private [diese] Deklaration
- private [fromWhere] Deklaration
- geschützte Erklärung
- geschützte [fromWhere] Deklaration

Examples

Öffentlicher (Standard) Bereich

Der Bereich ist standardmäßig `public` , auf den Wert kann von überall zugegriffen werden.

```
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
  }
}
```

Ein privater Umfang

Wenn der Bereich privat ist, kann nur von der aktuellen Klasse oder von anderen Instanzen der aktuellen Klasse auf ihn zugegriffen werden.

```
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
  }
}
```

Ein privater paketspezifischer Umfang

Sie können ein Paket angeben, auf das auf den privaten Wert zugegriffen werden kann.

```
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
  }
}
```

Objekt privater Umfang

Der restriktivste Bereich ist der Bereich *"Objekt-privat"* , der nur den Zugriff auf diesen Wert von derselben Instanz des Objekts ermöglicht.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

Geschützter Umfang

Der geschützte Bereich ermöglicht den Zugriff auf den Wert von allen Unterklassen der aktuellen Klasse.

```
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
}
```

Paket geschützter Umfang

Der durch das Paket geschützte Bereich ermöglicht den Zugriff auf den Wert nur von einer

Unterklasse in einem bestimmten Paket.

```
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
  }
}
```

Umfang online lesen: https://riptutorial.com/de/scala/topic/9705/umfang

Kapitel 58: Var, Val und Def

Bemerkungen

Da `val` semantisch statisch ist, werden sie dort, wo sie im Code erscheinen, "in-place" initialisiert. Dies kann zu einem überraschenden und unerwünschten Verhalten führen, wenn es in abstrakten Klassen und Merkmalen verwendet wird.

`PlusOne` , wir möchten ein Merkmal namens `PlusOne` , das eine Inkrementierungsoperation für einen `PlusOne Int` . Da `Int` s unveränderlich sind, ist der Wert plus eins bei der Initialisierung bekannt und wird danach nie geändert. Es ist also semantisch ein `val` . Wenn Sie es auf diese Weise definieren, wird dies zu einem unerwarteten Ergebnis führen.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

Egal welchen Wert `i` Sie konstruieren `IntWrapper` mit, ruft `.incr` auf das zurückgegebene Objekt wird immer 1 zurückkehren Dies liegt daran , die val `incr` *in der Eigenschaft* initialisiert wird, bevor die abgeleitete Klasse, und zu diesem Zeitpunkt `i` hat nur den Standardwert `0` (In anderen Umständen könnte es mit bevölkert werden `Nil` , `null` oder einem ähnlichen Standard.)

Die allgemeine Regel, dann ist die Verwendung zu vermeiden `val` auf einen beliebigen Wert, der auf einem abstrakten Feld abhängt. Verwenden Sie stattdessen `lazy val` , das nicht ausgewertet wird, bis es benötigt wird, oder `def` , das jedes Mal, wenn es aufgerufen wird, ausgewertet wird. Beachten Sie jedoch, dass derselbe Fehler auftritt, wenn der `lazy val` vor Abschluss der Initialisierung durch einen `val` erzwungen wird.

Eine Geige (in Scala-Js geschrieben, aber das gleiche Verhalten gilt) kann hier gefunden werden.

Examples

Var, Val und Def

var

Ein `var` ist eine Referenzvariable, ähnlich wie Variablen in Sprachen wie Java. Verschiedene Objekte können einem `var` frei zugewiesen werden, solange das angegebene Objekt denselben Typ hat, mit dem das `var` deklariert wurde:

```
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"
       ^
```

Hinweis in dem obigen Beispiel der Art des `var` wurde von der der ersten Wertzuweisung gegeben Compiler abgeleitet.

val

Ein `val` ist eine konstante Referenz. Daher kann ein neues Objekt nicht einem bereits zugewiesenen `val` zugewiesen werden.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

Das Objekt, auf das ein `val` zeigt, ist jedoch *nicht* konstant. Dieses Objekt kann geändert werden:

```
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

Ein `def` definiert eine Methode. Eine Methode kann nicht erneut zugewiesen werden.

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

In den obigen Beispielen geben `val y` und `def z` den gleichen Wert zurück. Jedoch ist ein `def` wird

ausgewertet , *wenn es heißt,* während ein `val` oder `var` ausgewertet wird , *wenn er zugeordnet ist.* Dies kann zu unterschiedlichem Verhalten führen, wenn die Definition Nebenwirkungen hat:

```
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
```

Funktionen

Da Funktionen Werte sind, können sie `val` / `var` / `def` zugewiesen werden. Alles andere funktioniert genauso wie oben:

```
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` ist eine Sprachfunktion, bei der die Initialisierung eines `val` verzögert wird, bis zum ersten Mal darauf zugegriffen wird. Danach verhält es sich wie ein normaler `val` .

Fügen Sie das `lazy` Schlüsselwort vor `val` . Verwenden Sie zum Beispiel die 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
```

Dieses Beispiel zeigt die Ausführungsreihenfolge. Wenn die `lazy val` deklariert wird, ist alles , was auf die gespeicherte `foo` Wert ist ein fauler Funktionsaufruf , die noch nicht ausgewertet wurde. Wenn die reguläre `val` gesetzt ist, sehen wir der `println` Anruf ausführen und der Wert wird zugewiesen `bar` . Wenn wir `foo` das erste Mal `println` , sehen wir `println` ausführen - aber nicht, wenn es das zweite Mal ausgewertet wird. Ebenso, wenn `bar` ausgewertet wir nicht sehen `println` ausführen - nur , wenn er erklärt.

Wann "faul" verwendet werden

1. **Die Initialisierung ist rechenintensiv und die Verwendung von `val` ist selten.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` die Berechnung von `tiresomeValue` eine lange Zeit erforderlich, und er wird nicht immer verwendet. Ein `lazy val` erspart unnötige Berechnungen.

2. **Zyklische Abhängigkeiten auflösen**

   Schauen wir uns ein Beispiel mit zwei Objekten an, die während der Instantiierung gleichzeitig deklariert werden müssen:

   ```
   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")
}
```

Ohne das Schlüsselwort `lazy` können die jeweiligen Objekte keine Mitglieder eines Objekts sein. Die Ausführung eines solchen Programms würde zu einer `java.lang.NullPointerException` . Durch die Verwendung von `lazy` kann die Referenz vor der Initialisierung zugewiesen werden, ohne einen uninitialisierten Wert zu befürchten.

Überladen Def

Sie können einen `def` überladen, wenn die Signatur unterschiedlich ist:

```
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'"
```

Dies funktioniert gleich, ob innerhalb von Klassen, Eigenschaften, Objekten oder nicht.

Benannte Parameter

Beim Aufrufen eines `def` können Parameter explizit nach Namen zugewiesen werden. Dies bedeutet, dass sie nicht korrekt angeordnet werden müssen. Definieren Sie beispielsweise `printUs()` als:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Nun kann es unter anderem auf folgende Weise aufgerufen werden:

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

Dies führt dazu `one, two, three` dass in allen Fällen `one, two, three` gedruckt werden.

Wenn nicht alle Argumente benannt werden, stimmen die ersten Argumente überein. Auf ein benanntes Argument darf kein positionelles (nicht benanntes) Argument folgen:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Var, Val und Def online lesen: https://riptutorial.com/de/scala/topic/3155/var--val-und-def

Kapitel 59: Während Schleifen

Syntax

- while (boolean_expression) {Blockausdruck}
- do {block_expression} while (boolean_expression)

Parameter

Parameter	Einzelheiten
`boolean_expression`	Jeder Ausdruck, der als `true` oder `false` ausgewertet wird.
`block_expression`	Jeder Ausdruck oder eine Gruppe von Ausdrücken, die ausgewertet werden, wenn der `boolean_expression` `true` `boolean_expression` .

Bemerkungen

Der primäre Unterschied zwischen , `while` und `do-while` - Schleifen ist , ob sie die Ausführungs `block_expression` , bevor sie überprüfen , ob sie sollten Schleife.

Da `while` und `do-while` Schleifen zum Beenden auf einem Ausdruck basieren, der als `false` ausgewertet wird, ist es häufig erforderlich, dass der veränderliche Status außerhalb der Schleife deklariert und dann innerhalb der Schleife geändert wird.

Examples

Während Schleifen

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Do-While-Schleifen

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
```

```
  println("Line number: " + line)
} while (line < maximum_lines)
```

Die `do` / `while` Schleife wird in der Funktionsprogrammierung selten verwendet, kann jedoch verwendet werden, um die fehlende Unterstützung für das `break` / `continue` Konstrukt zu umgehen, wie in anderen Sprachen zu sehen ist:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Während Schleifen online lesen: https://riptutorial.com/de/scala/topic/650/wahrend-schleifen

Kapitel 60: Wenn Ausdrücke

Examples

Grundlegende If-Ausdrücke

In Scala (im Gegensatz zu Java und den meisten anderen Sprachen) ist `if` ein **Ausdruck** anstelle einer *Anweisung* . Unabhängig davon ist die Syntax identisch:

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

`if` ein Ausdruck sein möchten, können Sie das Ergebnis der Auswertung des Ausdrucks einer Variablen zuweisen:

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

Oben sehen wir, dass der `if` Ausdruck ausgewertet wird und das `result` auf diesen Ergebniswert gesetzt wird.

Der Rückgabetyp eines `if` Ausdrucks ist der **übergeordnete** Typ aller Logikzweige. Dies bedeutet, dass der Rückgabetyp für dieses Beispiel ein `String` . Da nicht alle `if` Ausdrücke einen Wert zurückgeben (z. B. eine `if` Anweisung, die keine `else` Verzweigungslogik hat), ist der Rückgabetyp möglicherweise `Any` :

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

Wenn kein Wert zurückgegeben werden kann (wenn beispielsweise nur Nebeneffekte wie `println` in den logischen Zweigen verwendet werden), `println` der Rückgabetyp `Unit` :

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

`if` Ausdrücke in Scala sind ähnlich wie der ternäre Operator in Java . Aufgrund dieser Ähnlichkeit gibt es in Scala keinen solchen Operator: Er wäre überflüssig.

> Geschweifte Klammern können in einem `if` Ausdruck weggelassen `if` , wenn der Inhalt eine einzelne Zeile ist.

Wenn Ausdrücke online lesen: https://riptutorial.com/de/scala/topic/4171/wenn-ausdrucke

Kapitel 61: XML-Behandlung

Examples

Verschönern oder Pretty-Print-XML

Das `PrettyPrinter` Dienstprogramm druckt XML-Dokumente hübsch. Das folgende Code-Snippet druckt unformatierte XML-Dateien aus:

```
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)
```

Dadurch wird der Inhalt mit einer Seitenbreite von `150` und einer Einrückungskonstante von `4` Leerzeichen ausgegeben:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

Sie können `XML.loadFile("nameoffile.xml")` , um XML aus einer Datei statt aus einem String zu laden.

XML-Behandlung online lesen: https://riptutorial.com/de/scala/topic/1453/xml-behandlung

Kapitel 62: Züge

Syntax

- Merkmal ATrait {...}
- Klasse AClass (...) erweitert ATrait {...}
- Klasse AClass erweitert BClass mit ATrait
- Klasse AClass erweitert ATrait um BTrait
- Klasse AClass erweitert ATrait um BTrait um CTrait
- Klasse ATrait erweitert BTrait

Examples

Stapelbare Modifikation mit Eigenschaften

Sie können Merkmale verwenden, um die Methoden einer Klasse zu ändern, indem Sie die Merkmale stapelbar verwenden.

Das folgende Beispiel zeigt, wie Merkmale aufeinander gestapelt werden können. Die Reihenfolge der Merkmale ist wichtig. Durch die unterschiedliche Reihenfolge der Merkmale wird ein unterschiedliches Verhalten erreicht.

```
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
  }
}
```

Beachten Sie, dass `super` zum Aufrufen der `roll()` -Methode in beiden Merkmalen verwendet wird. Nur so können wir eine stapelbare Modifikation erreichen. Bei stapelbaren Modifikationen wird die

Reihenfolge der Methodenaufrufe durch eine *Linearisierungsregel bestimmt* .

Trait-Grundlagen

Dies ist die grundlegendste Version eines Merkmals 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"
}
```

Da keine übergeordnete Klasse für die Eigenschaft `Identifiable` deklariert ist, geht sie standardmäßig von der `AnyRef` Klasse aus. Da in `Identifiable` keine Definition für `getIdentifier` ist, muss die `Puppy` Klasse diese implementieren. `Puppy` `printIdentification` die Implementierung von `printIdentification` von `Identifiable` .

In der REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

Das Diamantproblem lösen

Das Raute-Problem oder die Mehrfachvererbung wird von Scala mithilfe von Traits behandelt, die Java-Schnittstellen ähneln. Traits sind flexibler als Schnittstellen und können implementierte Methoden enthalten. Dies macht Merkmale ähnlich wie Mixins in anderen Sprachen.

Scala unterstützt keine Vererbung von mehreren Klassen. Ein Benutzer kann jedoch mehrere Merkmale in einer einzigen Klasse erweitern:

```
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
```

Hier `grandChild` erbt von beiden `traitB` und `traitC` , die wiederum beide von erben `traitA` . Die Ausgabe (unten) zeigt auch die Rangfolge bei der Auflösung, welche Methodenimplementierungen zuerst aufgerufen werden:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Beachten Sie, dass bei der Verwendung von `super` zum Aufrufen von Methoden in `class` oder `trait` Linearisierungsregel zur Entscheidung der Aufrufhierarchie ins Spiel kommt. Die Linearisierungsreihenfolge für `grandChild` :

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Any

Unten ist ein anderes Beispiel:

```
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!")
  }
}
```

Dieses Programm druckt:

```
*************
-------------
Hello World!
```

Die Linearisierung für `CustomPrinter` :

CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Drucker -> AnyRef -> Any

Linearisierung

Bei stapelbaren Modifikationen ordnet Scala Klassen und Merkmale in linearer Reihenfolge an, um die Methodenaufrufhierarchie zu bestimmen, die als *Linearisierung bezeichnet wird* . Die Linearisierungsregel wird *nur* für Methoden verwendet, die den Methodenaufruf über `super()` erfordern. Betrachten wir dies anhand eines Beispiels:

```
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)
  }
}
```

Die Linearisierung erfolgt von *hinten nach vorne* . In diesem Fall,

1. First `Shape` wird linearisiert, was wie folgt aussieht:

   ```
   Shape -> AnyRef -> Any
   ```

2. Dann wird `Dotted` linearisiert:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. Als nächstes kommt `Blue` . Normalerweise ist die Linearisierung von `Blue` :

```
Blue -> Color -> Shape -> AnyRef -> Any
```

denn in `MyShape` Linearisierung (*Schritt 2*) ist `Shape -> AnyRef -> Any` bereits erschienen. Daher wird es ignoriert. Die `Blue` Linearisierung ist also:

```
Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
```

4. Schließlich wird `Circle` hinzugefügt und die endgültige Linearisierungsreihenfolge lautet:

    Kreis -> Blau -> Farbe -> Punkt -> Rand -> Form -> AnyRef -> Any

Diese Linearisierungsreihenfolge bestimmt die Aufrufreihenfolge von Methoden, wenn `super` in einer Klasse oder einem Merkmal verwendet wird. Die erste Methodenimplementierung von rechts wird in der Linearisierungsreihenfolge aufgerufen. Wenn `new MyShape().paint("Circle ")` ausgeführt wird, wird `new MyShape().paint("Circle ")` gedruckt:

```
Circle with Blue Color with Dotted Border
```

Weitere Informationen zur Linearisierung finden Sie hier .

Züge online lesen: https://riptutorial.com/de/scala/topic/1056/zuge

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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	Abhängigkeitsspritze	Hoang Ong
3	Anmerkungen	Gábor Bakos, Nathaniel Ford, Thomas Matecki
4	Aufzählungen	Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
5	Benutzerdefinierte Funktionen für Hive	Camilo Sampedro
6	Best Practices	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
7	Betreiber in Scala	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
8	Currying	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890
9	Dynamischer Aufruf	HTNW
10	Einzelne abstrakte Methodentypen (SAM-Typen)	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
11	Extraktoren	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
12	Fallklassen	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
13	Fehlerbehandlung	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos,

		Tzach Zohar
14	Fortsetzungen Bibliothek	dmitry, HTNW
15	Funktion höherer Ordnung	acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini
16	Funktionen	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
17	Für Ausdrücke	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
18	Futures	isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
19	Handling Units (Maßnahmen)	Gábor Bakos
20	Impliziert	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
21	Java-Interoperabilität	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
22	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
23	Klassen eingeben	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
24	Klassen und Objekte	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
25	Makros	gregghz, HTNW, Nathaniel Ford
26	Mit Daten unveränderlich arbeiten	Filippo Vitale
27	Mit Gradle arbeiten	Bianca Tesila, Nathaniel Ford, Rjk
28	Monaden	ipoteka, Nathaniel Ford
29	Musterabgleich	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
30	Optionsklasse	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
31	Pakete	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
32	Parallele Sammlungen	Nathaniel Ford, Shuklaswag
33	Parser-Kombinatoren	Nathaniel Ford
34	Programmierung auf Typebene	J Cracknell
35	Quasiquoten	gregghz
36	Reflexion	Sachin Janani
37	Reguläre Ausdrücke	dmitry, J Cracknell, Nathaniel Ford
38	Rekursion	Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
39	Sammlungen	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
40	Scala einrichten	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk
41	Scala.js	Camilo Sampedro
42	Scaladoc	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
43	Scalaz	chengpohi
44	Selbsttypen	Gábor Bakos, irundaia
45	Streams	jwvh, Nathaniel Ford, Oleg Pyzhcov
46	String Interpolation	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
47	Symbol Literals	ZbyszekKr

48	synchronisiert	Gábor Bakos
49	Teilfunktionen	acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
50	Testen mit ScalaCheck	Andrzej Jozwik
51	Testen mit ScalaTest	Nadim Bahadoor, Nathaniel Ford
52	Tuples	corvus_192, evan.oman, Lawsy, Nathaniel Ford
53	Typ Inferenz	Gábor Bakos, Nathaniel Ford, suj1th
54	Typabweichung	acjay, J Cracknell, Reactormonk
55	Typparameterisierung (Generics)	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
56	Überladung des Bedieners	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
57	Umfang	Camilo Sampedro
58	Var, Val und Def	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick , Shuklaswag, stefanobaghino, ZbyszekKr
59	Während Schleifen	J Cracknell, Nathaniel Ford
60	Wenn Ausdrücke	corvus_192, Nathaniel Ford, ScientiaEtVeritas
61	XML-Behandlung	Nathaniel Ford, Rockie Yang, vsnyc
62	Züge	André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz