Scala tutorial

Igor Kraskevich

Published
with GitBook

Table of Contents

Scala tutorial

This book provides a tutorial on the basics of the Scala programming language and an overview of several libraries and frameworks.

Tools and environment

Command line tools

- Scala has an interactive interpreter(REPL). It can be launched using the `scala` command. It allows to type in expressions and evaluate them immediately.

  ```
  scala> 2 + 3
  res0: Int = 5

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

- Scala source file can be compiled using the `scalac` command and the resulting class file can be executed using the `scala` command with a name of this class file as an argument. Here is how it works:

  1. Let's create a file called Main.scala with the following content:

     ```
     object Main extends App {
       println("Hello, Scala!")
     }
     ```

  2. Now let's compile it using the Scala compiler: `scalac Main.scala`

  3. Finally, we can run it:

     ```
     Temp$ scala Main
     Hello, Scala!
     ```

- sbt. A build tool for Java and Scala.

IDEs for Scala

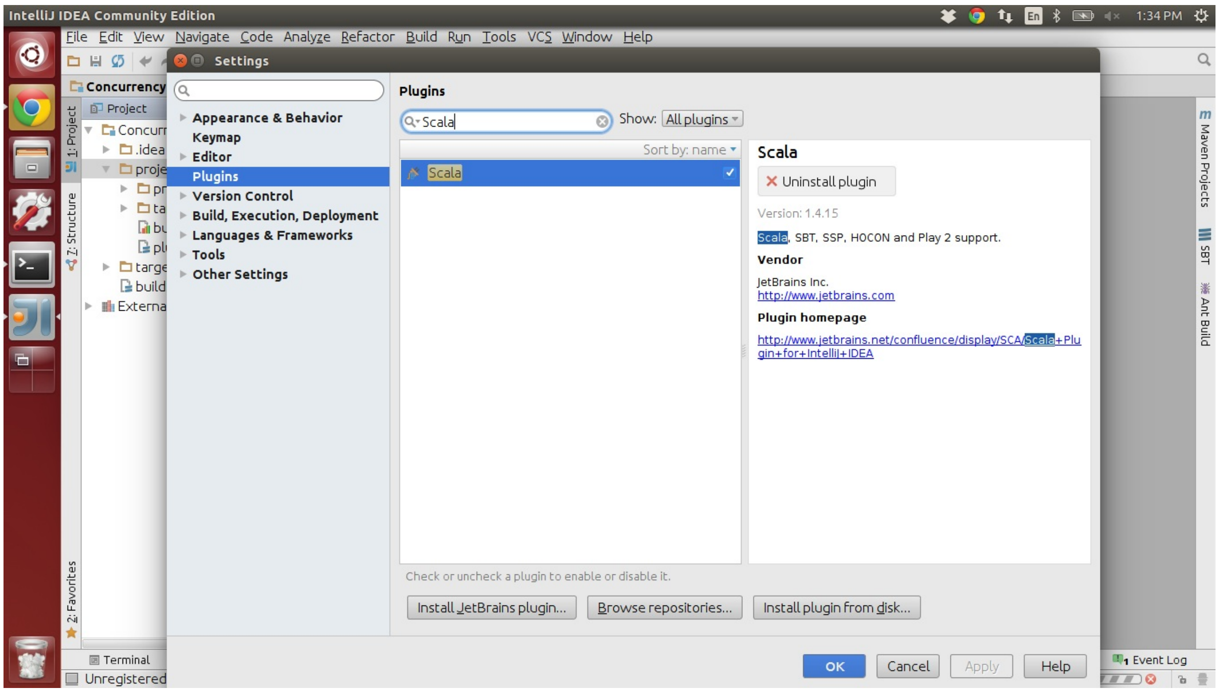
- Intellij IDEA with a Scala plugin. It supports sbt, autocompletion, debugging, Scala worksheets and more.
- Scala IDE for Eclipse. It provides a Scala-aware debugger, sytnax highlighting, autocompletion and a lot of other things.
- There are also Scala plugins for the Netbeans IDE.

IntelliJ IDEA

I use the IntelliJ IDEA with the Scala plugin, so I will describe it here in more details.

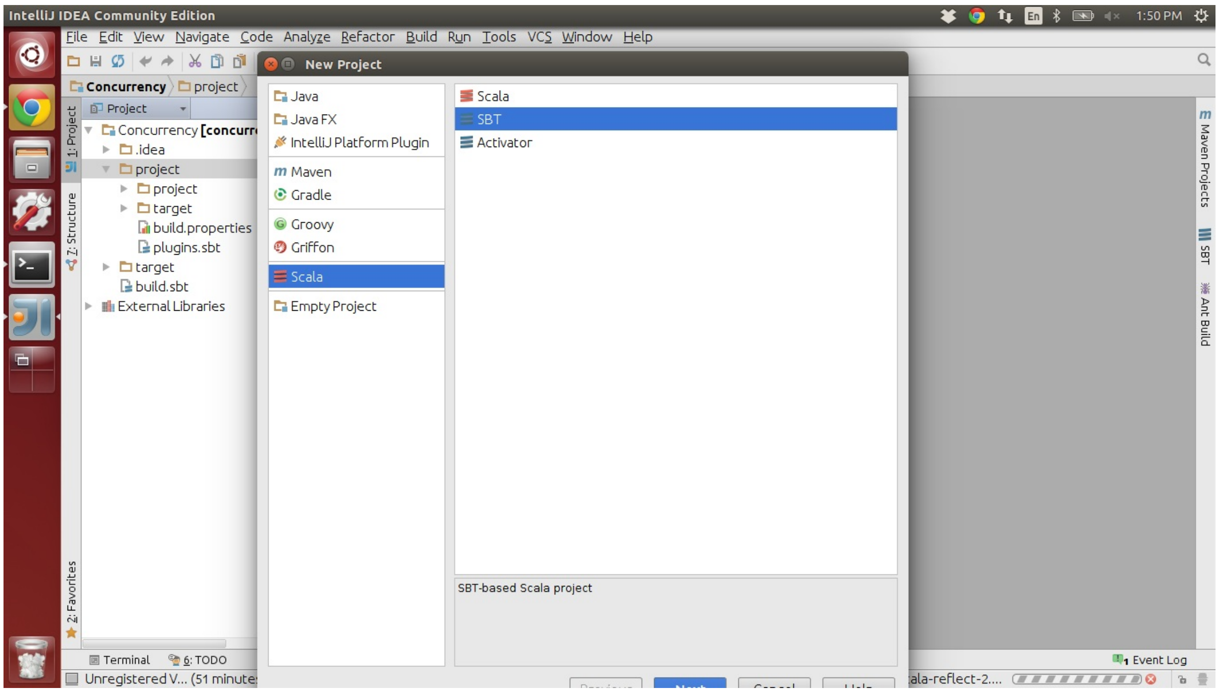
Getting started

1. To get started, you need to install the IntelliJ IDEA. You can download a free version from the official website.
2. The latest version of this IDE suggests to install the Scala plugin automatically on the first launch. If you haven't done that or you use an older version of the IDEA, you can always go to the File -> Settings -> Plugins section and find it there:

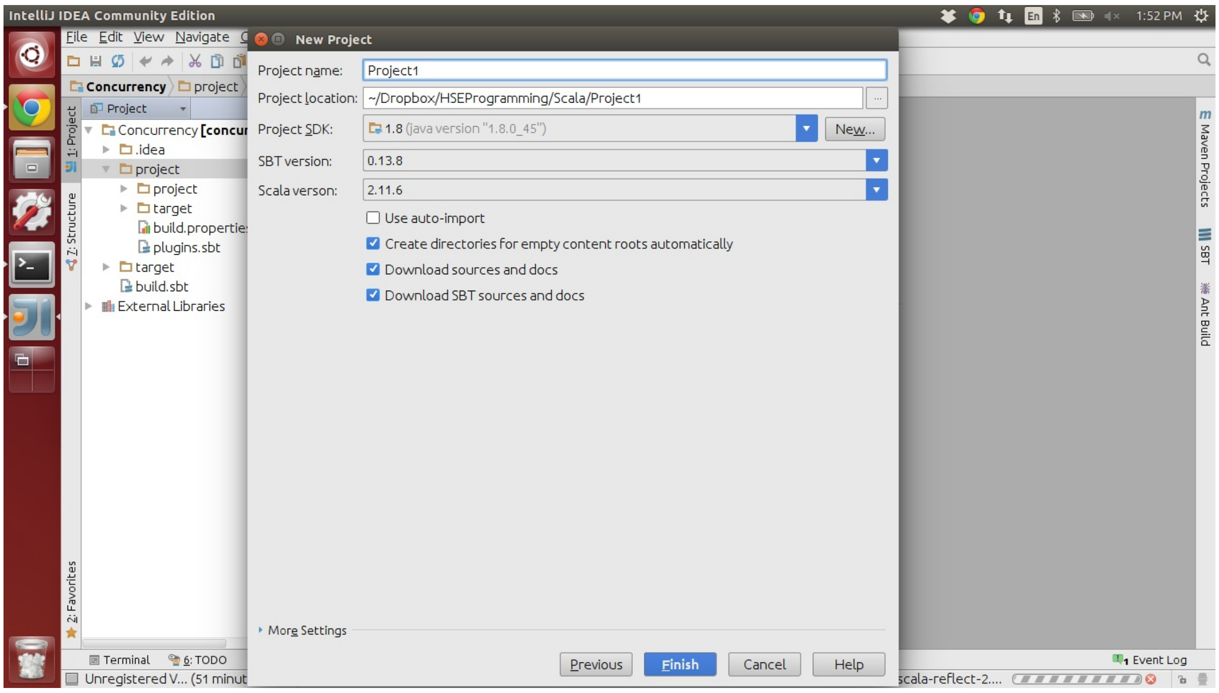


Creating a project.

1. To create a new Scala project, you can go the File -> New -> Project menu and select Scala -> SBT option:



2. After cliking the next button, you will be able to choose the name and the location of a new project. When you are done with that, click the finish button to open the new project:

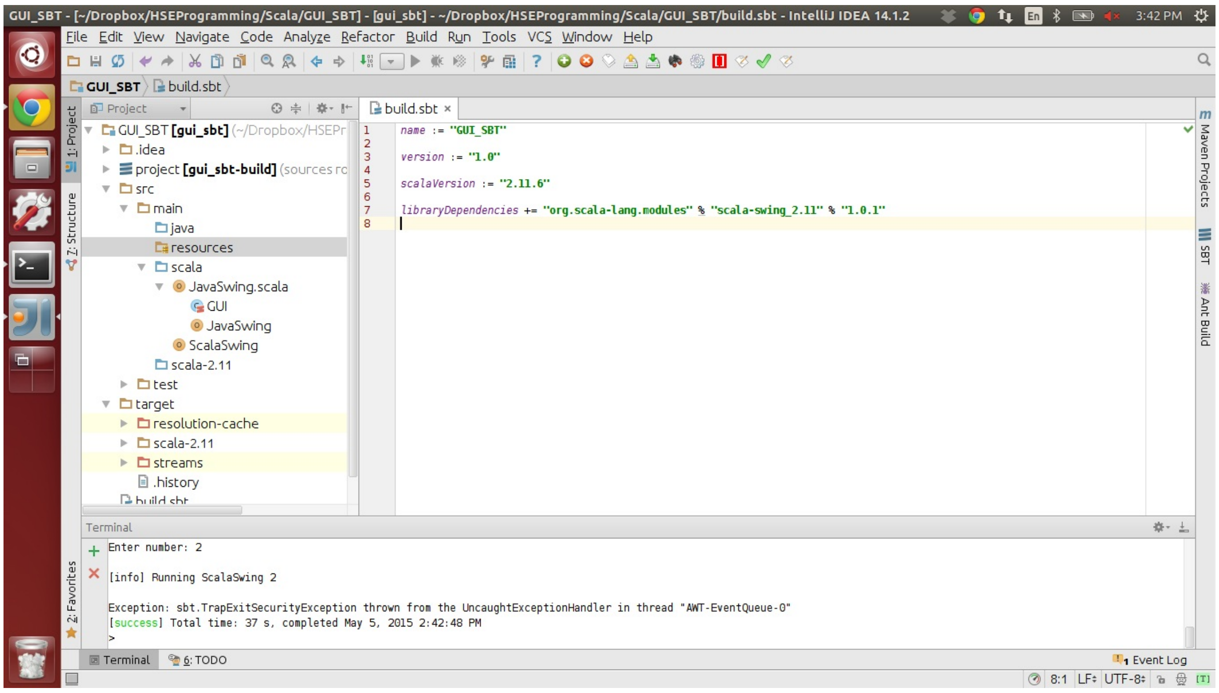


Project name: Project1
Project location: ~/Dropbox/HSEProgramming/Scala/Project1
Project SDK: 1.8 (java version "1.8.0_45")
SBT version: 0.13.8
Scala verson: 2.11.6
[ ] Use auto-import
[✓] Create directories for empty content roots automatically
[✓] Download sources and docs
[✓] Download SBT sources and docs

Working with a project

1. Source files. Scala source files are stored in the src/main/scala directory be default. Test files are should be located in the src/test/scala folder.

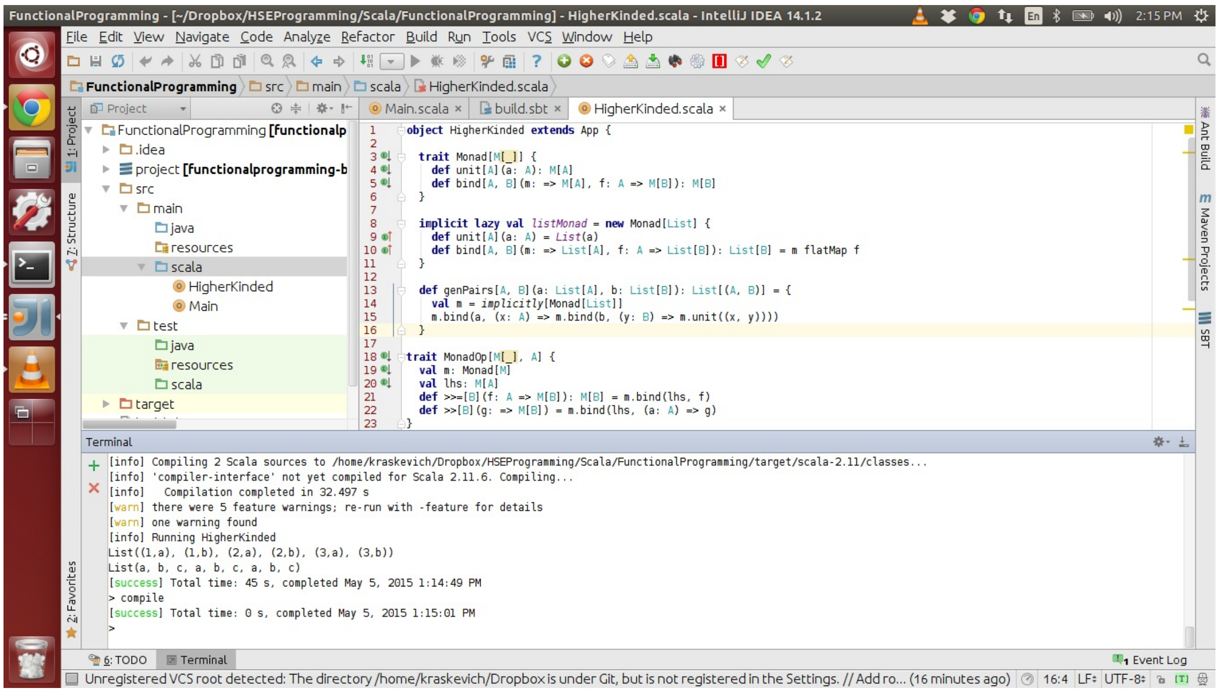
2. Adding dependencies. To add a new dependency, you can edit the build.sbt file directly:


GUI_SBT - [~/Dropbox/HSEProgramming/Scala/GUI_SBT] - [gui_sbt] - ~/Dropbox/HSEProgramming/Scala/GUI_SBT/build.sbt - IntelliJ IDEA 14.1.2
build.sbt
Terminal

```
name := "GUI_SBT"

version := "1.0"

scalaVersion := "2.11.6"

libraryDependencies += "org.scala-lang.modules" % "scala-swing_2.11" % "1.0.1"
```

```
Enter number: 2

[info] Running ScalaSwing 2

Exception: sbt.TrapExitSecurityException thrown from the UncaughtExceptionHandler in thread "AWT-EventQueue-0"
[success] Total time: 37 s, completed May 5, 2015 2:42:48 PM
>
```

3. The easiest way to compile and run the project is to launch a terminal inside the IDE(Alt+F12 hotkey in Linux) and use sbt commands: "compile" for complining the entire project, "test" for compiling and running all tests and "run" for compiling and running the project:

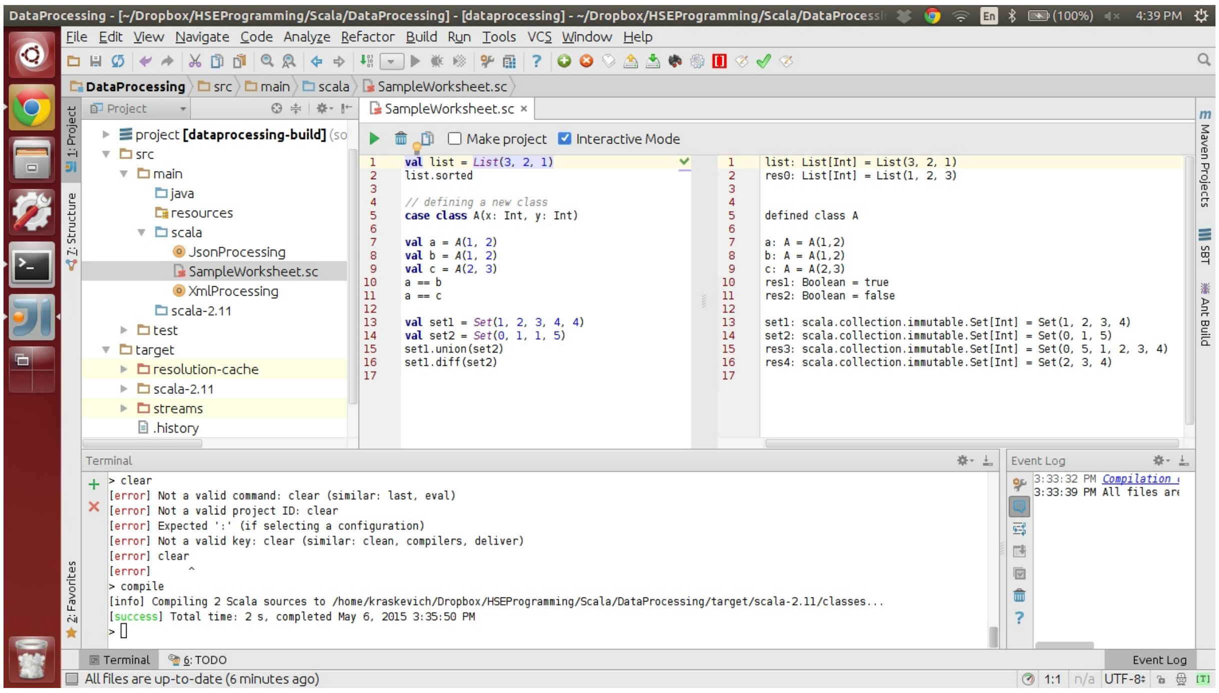

FunctionalProgramming - [~/Dropbox/HSEProgramming/Scala/FunctionalProgramming] - HigherKinded.scala - IntelliJ IDEA 14.1.2
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
FunctionalProgramming › src › main › scala › HigherKinded.scala
Main.scala
build.sbt
HigherKinded.scala
FunctionalProgramming [functionalp
.idea
project [functionalprogramming-b
src
main
java
resources
scala
HigherKinded
Main
test
java
resources
scala
target
Terminal
6: TODO
Terminal
Event Log

```scala
object HigherKinded extends App {

  trait Monad[M[_]] {
    def unit[A](a: A): M[A]
    def bind[A, B](m: => M[A], f: A => M[B]): M[B]
  }

  implicit lazy val listMonad = new Monad[List] {
    def unit[A](a: A) = List(a)
    def bind[A, B](m: => List[A], f: A => List[B]): List[B] = m flatMap f
  }

  def genPairs[A, B](a: List[A], b: List[B]): List[(A, B)] = {
    val m = implicitly[Monad[List]]
    m.bind(a, (x: A) => m.bind(b, (y: B) => m.unit((x, y))))
  }

trait MonadOp[M[_], A] {
  val m: Monad[M]
  val lhs: M[A]
  def >>=[B](f: A => M[B]): M[B] = m.bind(lhs, f)
  def >>[B](g: => M[B]) = m.bind(lhs, (a: A) => g)
}
```

```
[info] Compiling 2 Scala sources to /home/kraskevich/Dropbox/HSEProgramming/Scala/FunctionalProgramming/target/scala-2.11/classes...
[info] 'compiler-interface' not yet compiled for Scala 2.11.6. Compiling...
[info]   Compilation completed in 32.497 s
[warn] there were 5 feature warnings; re-run with -feature for details
[warn] one warning found
[info] Running HigherKinded
List((1,a), (1,b), (2,a), (2,b), (3,a), (3,b))
List(a, b, c, a, b, c, a, b, c)
[success] Total time: 45 s, completed May 5, 2015 1:14:49 PM
> compile
[success] Total time: 0 s, completed May 5, 2015 1:15:01 PM
>
```

Unregistered VCS root detected: The directory /home/kraskevich/Dropbox is under Git, but is not registered in the Settings. // Add ro... (16 minutes ago) 16:4 LF: UTF-8

4. Worksheets. A worksheet is a file which is evaluated every time it is saved and the result of each expression is avialable in the right column next to it. It combines advantages of a REPL(interactivity and ability to make quick changes) and a standard source file(auto completion, syntax highlighting, the ease to make changes in an abitrary place, not just the last expression). To create a worksheet, you need to right click the directory with source files and choose the new file -> scala worksheet option. Here is how it looks like:


DataProcessing - [~/Dropbox/HSEProgramming/Scala/DataProcessing] - [dataprocessing] - ~/Dropbox/HSEProgramming/Scala/DataProcess
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
DataProcessing › src › main › scala › SampleWorksheet.sc
SampleWorksheet.sc
project [dataprocessing-build]
src
main
java
resources
scala
JsonProcessing
SampleWorksheet.sc
XmlProcessing
scala-2.11
test
target
resolution-cache
scala-2.11
streams
.history
Make project
Interactive Mode
Terminal
Event Log
6: TODO

```scala
val list = List(3, 2, 1)
list.sorted

// defining a new class
case class A(x: Int, y: Int)

val a = A(1, 2)
val b = A(1, 2)
val c = A(2, 3)
a == b
a == c

val set1 = Set(1, 2, 3, 4, 4)
val set2 = Set(0, 1, 1, 5)
set1.union(set2)
set1.diff(set2)
```

```
list: List[Int] = List(3, 2, 1)
res0: List[Int] = List(1, 2, 3)


defined class A

a: A = A(1,2)
b: A = A(1,2)
c: A = A(2,3)
res1: Boolean = true
res2: Boolean = false

set1: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
set2: scala.collection.immutable.Set[Int] = Set(0, 1, 5)
res3: scala.collection.immutable.Set[Int] = Set(0, 5, 1, 2, 3, 4)
res4: scala.collection.immutable.Set[Int] = Set(2, 3, 4)
```

```
> clear
[error] Not a valid command: clear (similar: last, eval)
[error] Not a valid project ID: clear
[error] Expected ':' (if selecting a configuration)
[error] Not a valid key: clear (similar: clean, compilers, deliver)
[error] clear
[error]      ^
> compile
[info] Compiling 2 Scala sources to /home/kraskevich/Dropbox/HSEProgramming/Scala/DataProcessing/target/scala-2.11/classes...
[success] Total time: 2 s, completed May 6, 2015 3:35:50 PM
>
```

```
3:33:32 PM Compilation
3:33:39 PM All files are
```

All files are up-to-date (6 minutes ago) 1:1 n/a UTF-8

Compilation speed

Scala compiler is rather slow. It doesn't matter if you use an IDE, sbt or run it from the command line directly.

Using Java libraries.

Java code can be called from Scala directly. It means that Java libraries can be used as is.

Standard types in Scala

1. Everything in Scala is an object. It supports the same data types as primitives in Java(`Int`, `Boolean`, `Char` and so on), but they are treated as objects.
2. It also has a rich set of mutable and immutable collections. Moreover, it is possible to use all classes from Java standrard library.
3. Another widely used data type in Scala is a tuple.
4. There is no `void` type in Scala. `Unit` is used instead of it.

The following code snippet shows how to create and use values of several standard data types:

```
object StandardTypes {

  def main(args: Array[String]): Unit = {
    // Numbers. Type annotations are used deliberately.
    val i: Int = 2
    println(i)
    val long: Long = 100000000000000L
    println(long)
    val d: Double = 3.14
    println(d)
    // A String
    val s: String = "A string"
    println(s)
    // A Boolean
    val b: Boolean = true
    println(b)
    // Tuples.
    val t1: (String, Int) = new Tuple2("string", 1)
    println(t1)
    val t2: (Int, Int, Boolean) = new Tuple3(1, 2, false)
    println(t2)
    // An array.
    val a: Array[Int] = new Array(3)
    a(0) = 1
    a(1) = 10
    a(2) = 100
    println(a(0) + " " + a(1) + " " + a(2))
    // A list.
    val list: List[Int] = List(1, 2, 3)
    println(list)
  }
}
```

Basic statements and expressions

1. Conditional expression(`if`-`else`). In Scala, `if` is an expression, not a statememt. However, it is allowed to omit the `else` branch.

   ```
   // If is an expression, not a statement.
   def doubleIfEven(n: Int) =
    if (n % 2 == 0)
      n * 2
    else
      n
   ```

2. `while` and `do`-`while` loops. They behave in the same way as in traditional imperative languages:

   ```
     // while loop.
     def whileLoop(size: Int): Unit = {
       var i = 0
       val a = new Array[Int](size)
   ```

```
    while (i < a.length) {
      a(i) = i
      i = i + 1
    }
    a.foreach(println)
  }

  // do-while loop
  def doWhileLoop(size: Int): Unit = {
    var i = 0
    val a = new Array[Int](size)
    do {
      a(i) = i
      i = i + 1
    } while (i < a.length)
    a.foreach(println)
  }
```

3. `for` loop in Scala is very different from a `for` loop in traditional imperative programming languages. It actually binds monads using the `flatMap` method(similar to a `do` block in Haskell), but it uses duck typing(that is, it is possible to bind different monads as long as the result of desugaring type checks. A monad in Scala is just a pattern, it is not a part of the language). Here are a few examples of the `for` loop:

```
// A for loop with filters and an assigment.
val names = List("John", "Alex", "Joe", "VeryLongName")
for {
  name <- names
  if !name.startsWith("A")
  if name.length <= 4
  upperCase = name.toUpperCase
} println(upperCase)

// Binding a List and an Option.
// It is possible because of an implicit conversion
// from Option[A] to Iterable[A].
val list = List(1, 2, 3)
val opt = Some(5)
for {
  x <- list
  y <- opt
} yield (x + y)
```

4. `case` expression allows to perform pattern matching on an arbitrary value. To support "deep" pattern matching, the class should implement the `unapply` and/or `unapplySeq` method. The following method uses pattern matching on a pair of Ints:

```
def matchPair(p: (Int, Int)): String =
  p match {
    case (1, 1) => "one one"
    case (2, 3) => "two three"
    case (_, _) => "something else"
  }
```

Actually, it is not necessary to implement the `unapply` method explicitly. `case class` is a convinient way to make the compiler do this job for you:

```
// StringPair is a case class, so we can pattern match against it.
case class StringPair(first: String, second: String)

def main(args: Array[String]): Unit = {
  val pair1 = new StringPair("a", "b")
  val pair2 = new StringPair("c", "d")
  val pairs = List(pair1, pair2)
  for (pair <- pairs)
    // pair1 matches the first pattern, pair2 doesn't.
```

```
        pair match {
          // "deep" pattern matching
          case StringPair("a", "b") => println("matches")
          case _ => println("does mot match")
        }
    }
```

val vs var

There are two types of variables in Scala: `val` and `var`. `val` is an immutable reference(that is, it cannot be reassigned, but the object it references can still change its state). `var` is mutable(it can be reassigned). The following code shows the difference:

```
val a = new Array[Int](2)
// Fine, val means that a reference is immutable, the object itself can mutate.
a(0) = 2
// a = new Array[Int](2) Compilation error: reassignment to a val.

var b = new Array[Int](2)
b(0) = 2 // Fine.
b = new Array[Int](2) // Fine. A var can be reassigned.
```

Exception handling

It is very similar to Java, even though the syntax is a little bit different. Here is a small example of the `try-catch-finally` block:

```
try {
  val s: String = null
  println(s.length) // throws a NullPointerException
} catch {
  // Instead of multiple catch clauses for different types of exceptions,
  // there one catch clause which uses pattern matching.
  case e: NullPointerException => println(e)
  case _ => println("Unknown exception")
} finally {
  println("finally block")
}
```

Evaluation model

Strict evaluation is used in Scala by default. Let's take a look at this code snippet:

```
object Evaluation extends App {

  // A simple method which adds a number to itself and returns the
  // result.
  def callByValue(a: Int): Int = a + a

  def s = callByValue({
    println("Executing this block")
    2
  })

  println(s)
}
```

If you run it, you'll see that it prints "Executing this block" once and then it prints 4. Nothing special here.

However, Scala also supports call-by-name evaluation strategy. Let's modify the code a little bit and run it again:

```
object Evaluation extends App {

  // A simple method which adds a number to itself and returns the
  // result.
  // a is passed by name.
  def callByName(a: => Int): Int = a + a

  def s = callByName({
    println("Executing this block")
    2
  })

  println(s)
}
```

This time it prints "Executing this block" twice(the sum is still 4, of course). When an argument is passed by name, it is not evaluated immediately. Instead, it is evaluated every time it is referenced inside the body of the method. Call-by-name can be useful if the argument's value might not be needed and its evaluation is relatively expensive. It can also be used for implementing infinite data structures(for instance, there is a standard Stream class).

Scala does not have lazy evaluation, but it supports lazy initialization. Marking a `val` with a `lazy` keyword delays its evaluation until it is needed(it is fully thread-safe). Here is an example of lazy initialization:

```
object LazyVal extends App {

  // Initialzied right here.
  val eagerVal = {
    println("eager initialization")
    1
  }

  // Lazy vals are not initialized until their value is necessary.
  lazy val lazyVal = {
    println("lazy initialization")
    2
  }

  println(eagerVal)
  // lazyVal is initialized here.
  println(lazyVal)
}
```

This code prints

```
eager initialization
1
lazy initialization
2
```

Type System

Type Inference

Scala compiler can infer types and type parameters. That's why explicit type annotations are not always required. Here are a few examples:

```
val i = 2 // inferred type: Int
val s = "abc" // inferred type: String
val a = new Array[Int](2) // infered type: Array[Int]
def f(x: String) = "Hello, " + x + "!" // inferred type of the return value: String
```

However, its capabilities are limited. Scala supports only local type inference(which is far less powerful than Hindley-Milner). Moreover, the presense of subtyping makes things pretty complicated, so sometimes it can be a good idea to use explicit type annotations even when they optional to make sure that the inferred type is not different from the one you expect to see.

Classes

A class in Scala is not very different from a class in Java(there are no static members, but similar effect can be achieved using a companion-object). We can define classes and instantiate them in the following way:

```
class MyClass(val a: Int, val b: Int) {
  def sum = a + b
  def sumSquared = a * a + b * b
}

val myInstance = new MyClass(2, 3)
```

Like Java, Scala supports only single inheritance for classes.

Objects

The `object` keyword is used for defining singletons. Objects cannot be instantiated. Their members can be accessed directly using the name of the `object`(it is similar to `static` members in Java).

```
object Math {
  val pi = 3.14
  def sum(a: Int, b: Int) = a + b
  def product(a: Int, b: Int) = a * b
}

Math.sum(2, 3)
Math.product(5, 6)
Math.pi
```

Traits

A `trait` in Scala is similar to a Java `interface`, but its methods can have a default implementation. Traits are usually used for creating mixins. A class can extend a trait(and vice versa). Here is a relatively large example of using traits:

```
object Traits {

  /**
```

```
     * An actor class. Instances of this class represent an entity that
     * can perform an arbitrary action.
     */
    abstract class Actor {
      def act(): Unit
    }

    /**
     * A trait that allows to add loggers to a class.
     */
    trait Observable {
      // Structural type system is supported in Scala.
      type Logger = { def log(message: String): Unit }

      private var loggers = List[Logger]()

      def addLogger(logger: Logger): Unit = loggers ::= logger

      def notifyLoggers(message: String): Unit = loggers.foreach(_.log(message))
    }

    /**
     * A subtype of the Logger type from the Observable trait.
     */
    class ConsoleLogger {
      def log(message: String) = println(message)
    }

    /**
     * A trait which adds logging to an actor.
     */
    trait ObservableActor extends Actor with Observable {
      abstract override def act() = {
        notifyLoggers("act method called")
        // super is bound when this trait is mixed with a concrete class.
        // It will call an appropriate method for any subclass of Actor.
        super.act()
      }
    }

    /**
     * A subclass of an Actor which increments a counter and prints
     * its value every time the act method is called.
     */
    class Counter extends Actor {
      var counter = 0

      override def act() = {
        counter = counter + 1
        println(counter)
      }
    }

    /**
     * A subclass of an Actor that prints a "Hello, world!" message to the
     * screen when the act method is called.
     */
    class HelloWorldPrinter extends Actor {
      def act() = println("Hello, world!")
    }

    def main(args: Array[String]) = {
      // The ObservableActor trait can be mixed with any subclass of an Actor.
      val counter = new Counter with ObservableActor
      counter.act()
      counter.addLogger(new ConsoleLogger())
      counter.act()
      val printer = new HelloWorldPrinter with ObservableActor
      printer.addLogger(new ConsoleLogger())
      printer.addLogger(new AnyRef {
        def log(message: String) = println("Another logger: " + message)
      })
      printer.act()
    }
  }
```

There is one more intereseting feature that we can in this example: structural subtyping. The `Logger` type in the `Observable` trait is defined in terms of an interface it should implement and any type which has all the required methods can be used as `Logger`. It does not need to extend the `Logger` explicitly.

This example also demonstrates that it is possible to mix multiple traits at the same time(so they provide a form of multiple inheritance).

Generics

Scala supports parametric polymorphism using generics, but it is more complex than in Java(for example, it is possible to explicitly specify the variance of type parameters). However, basic generics use is rather straightforward:

```
// A pair of two elements of arbitrary types.
class MyPair[A, B](private val a: A, private val b: B) {
  def first: A = a
  def second: B = b
}

// Creating an instance of a generic class.
val pair = new MyPair[Int, String](1, "abc")
```

It is possible to use generics with any types in Scala(including `Int`, `Char`, `Boolean` and so on) because there are no primitives.

Existential types

Exestintial types in Scala are somewhat similar to wildcards in Java, but they more powerful. They can have both lower and upper bounds. They can be used for interacting with raw Java types.

More

Scala type system is complicated(it supports several other things that were not shown here, such as abstract types and implicit conversions), but it is possible to use this language efficiently without knowing all the details of its type system.

Functional programming in Scala

First-class functions

Functions in Scala are first-class citizens of the language. Here are a few examples:

```
// A function literal.
val double = (x: Int) => x * 2

// A closure.
def sumOfRange(low: Int, high: Int) = {
  // Explicit tail recursion annotation. It ensure that the
  // tail call is optimized.
  @tailrec
  def sumHelper(i: Int, acc: Int): Int = {
    if (i > high)
      acc
    else
      sumHelper(i + 1, acc + i)
  }
  sumHelper(low, 0)
}

// Partial application.
def prod(a: Int, b: Int): Int = a * b
def prod100: Int => Int = prod(100, _)

// Currying.
val sum = (a: Int, b: Int) => a + b
def curriedSum = sum.curried
// This is how we apply arguments to a curried function.
def sum100: Int => Int = curriedSum(100)

// Higher-order functions.
// A composition of two functions((f * g) x = (f(g(x))))
def compose[A, B, C](f: B => C, g: A => B): A => C = (x: A) => f(g(x))
```

Immutable collections and operations on them

Scala's standard library has a rich set of mutable and immutable collections. Here, I will concentrate on the immutable ones. One of the basic data structures in functional programming is a list. Scala's `List` supports a bunch of standard operations, such as `map`, `filter` and others. Let's take a look at a few examples:

```
  // Creating a new list using the apply method of the List companion object.
  val xs = List(1, 2, 3, 4, 5)
  // Map function is usually used for transforming a list.
  val squared = xs.map(x => x * x) // squared == List(1, 4, 9, 16, 25)
  // Filter function is used to filter out the elements that do not
  // satisfy the given predicate.
  val odd = xs.filter(x => x % 2 == 1) // odd == List(1, 3, 5)
  // If we need a more general operations, we can use a left or a right
  // fold.
  val prod = xs.foldLeft(1)(_ * _) // prod == 120
```

There is also a `foreach` method(which simply applies the given function to each element of the list and ignores the result), but I do not show it here because it is impure(it can be useful only with a function that has side effects).

A big advantage of functional programming compared to traditional object-oriented approach is composability. We can easily preform a series of different transformations on a list. For instance, this piece of code squares all elements of the list, removes even numbers and then computes their product:

```
xs.map(x => x * x).filter(x => x % 2 == 0).foldLeft(1)(_ * _)
```

We can also zip multiple lists together using the `zip` method to obtain a sequence of pairs. This example shows how to get a list of lowercase latin letters with their positions in the alphabet:

```
(1 to 26).zip('a' to 'z') // (1, 'a'), (2, 'b'), ..., (26, 'z')
```

It is important to note that lists in Scala are not lazy, so they cannot be infinite.

Scala also has an immutable `Set`, `Map` and `Vector`.

Infinite data structures.

Even though Scala is strict by default, it supports call-by-name evaluation strategy, too. It means that one can implement infinite data structures in Scala. Let's take a look at one of them: the `Stream` (it is a part of Scala's standard library). Streams are similar to lists, but they are lazy and can be infinite. Here is an example of a classical Fibonacci sequence implementation which uses streams:

```
 // The definition of Fibonacci numbers.
 // If streams were strict, it would never terminate.
 val fs: Stream[BigInt] =
    BigInt(1) #:: BigInt(1) #:: fs.zip(fs.tail).map(f => f._1 + f._2)

  // Prints all Fibonacci numbers that are less than or equal to 100.
  println(fs.takeWhile(_ <= 100).toList)
```

Now let's generate prime numbers in a functional way:

```
  // An infinite range [from...].
  def rangeFrom(from: BigInt): Stream[BigInt] = from #:: rangeFrom(from + 1)
  // A stream of positive integers 1, 2, ...
  val positiveNumbers = rangeFrom(1)
  // A stream of prime numbers
  val primes = positiveNumbers.filter(_.isProbablePrime(50))
  // Now we can take the first N primes do something with them(just print
  // in this case).
  println(primes.take(20).toList)
```

We can do the same thing with the Eratosthenes sieve:

```
  // This implementation is not efficient.
  // There is a faster implementation which still uses infinite streams,
  // but it is much more sophisticated.
  def sieve(s: Stream[BigInt]): Stream[BigInt] = {
    val h = s.head
    h #:: sieve(s.tail.filter(_ % h != BigInt(0)))
  }

  val primes2 = sieve(rangeFrom(2))
  // Prints all primes below 100.
  println(primes2.takeWhile(_ <= 100).toList)
```

One important note: a stream does not require any special treatment by the compiler. In fact, we could implement our own version of a stream if there weren't one in the standard library. Let's do it, anyway:

```
abstract sealed class MyStream[+A] {
  def head: A
  def tail: MyStream[A]
  def zip[B](s: MyStream[B]): MyStream[(A, B)]
  def filter(pred: A => Boolean): MyStream[A]
  def take(n: Int): MyStream[A]
  def takeWhile(pred: A => Boolean): MyStream[A]
  def map[B](f: A => B): MyStream[B]
  def toList: List[A]
}

// An empty stream.
object Nil extends MyStream[Nothing] {
  override def head = throw new UnsupportedOperationException("head of a Nil")
  override def tail = throw new UnsupportedOperationException("tail of a Nil")
  override def filter(pred: Nothing => Boolean) = Nil
  override def take(n: Int) = Nil
  override def takeWhile(pred: Nothing => Boolean) = Nil
  override def zip[B](s: MyStream[B]) = Nil
  override def map[B](f: Nothing => B) = Nil
  override def toList = List()
}

// A stream as a pair (head, another stream(a tail)).
class Cons[+A](h: => A, t: => MyStream[A]) extends MyStream[A] {
  override lazy val head = h
  // Tail must be evaluated lazily.
  override lazy val tail = t

  override def zip[B](s: MyStream[B]): MyStream[(A, B)] =
    if (s == Nil)
      Nil
    else
      Cons((head, s.head), tail.zip(s.tail))

  override def filter(pred: A => Boolean) =
    if (pred(head))
      Cons(head, tail.filter(pred))
    else
      tail.filter(pred)

  override def take(n: Int) =
    if (n <= 0)
      Nil
    else
      Cons(head, tail.take(n - 1))

  override def takeWhile(pred: A => Boolean) =
    if (pred(head))
      Cons(head, tail.takeWhile(pred))
    else
      Nil

  override def map[B](f: A => B) = Cons(f(head), tail.map(f))

  override def toList = head :: tail.toList
}

object Cons {
  def apply[A](h: => A, t: => MyStream[A]) = new Cons[A](h, t)
}
```

The idea is very simple: we can pass a head and a tail by name to the `Cons` constructor and memoize their values using `lazy val`s(it is important that the tail is lazy, but the head can be passed by value, too). Once we have head and tail working properly, the implementation of other methods is straightforward. Now we can rewrite all the examples with prime and Fibonacci numbers shown above using this version of a stream instead of the standard one:

```
val fs: MyStream[BigInt] =
  Cons(BigInt(1), Cons(BigInt(1), fs.zip(fs.tail).map(f => f._1 + f._2)))

println(fs.takeWhile(_ <= 100).toList)
```

```
    def rangeFrom(from: BigInt): MyStream[BigInt] =
      Cons(from, rangeFrom(from + 1))

    val positiveNumbers = rangeFrom(1)
    val primes = positiveNumbers.filter(_.isProbablePrime(50))
    println(primes.take(20).toList)

    def sieve(s: MyStream[BigInt]): MyStream[BigInt] = {
      val h = s.head
      Cons(h, sieve(s.tail.filter(_ % h != BigInt(0))))
    }

    val primes2 = sieve(rangeFrom(2))
    println(primes2.takeWhile(_ <= 100).toList)
```

If we run this code, we'll see that it does the same thing as before.

Variant types

Let's assume that we want to write a simple calculator that supports constants, addition, subtraction and mulitplication. A standard way to do it in a functional programming language is to define an expression as a variant type and use pattern matching in the `eval` function. Here is how we could do it in Haskell:

```
// Expression is either a constant, a sum, a difference
// or a product of two other expressions.
data Expr a =
    Const a |
    Plus (Expr a) (Expr a) |
    Minus (Expr a) (Expr a) |
    Mult (Expr a) (Expr a) deriving (Read, Show)

eval :: (Num a) => Expr a -> a
eval (Const a) = a
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
eval (Mult a b) = eval a * eval b
```

Unfortunately, it is not possible to define variant types in Scala directly. A typical way to model them is to use an `abstract sealed` class and a case subclass for each option. Let's implement the same thing in Scala:

```
abstract sealed class Expr[A]

case class Const[A](a: A) extends Expr[A]

case class Plus[A](a: Expr[A], b: Expr[A]) extends Expr[A]

case class Minus[A](a: Expr[A], b: Expr[A]) extends Expr[A]

case class Mult[A](a: Expr[A], b: Expr[A]) extends Expr[A]

def eval[A](expr: Expr[A])(implicit op: Numeric[A]): A =
  expr match {
    case Const(a) => a
    case Plus(a: Expr[A], b: Expr[A]) => op.plus(eval(a), eval(b))
    case Minus(a: Expr[A], b: Expr[A]) => op.minus(eval(a), eval(b))
    case Mult(a: Expr[A], b: Expr[A]) => op.times(eval(a), eval(b))
  }
```

It is more verbose, but it works.

Disadvantages

Even though Scala supports functional programming paradigm, it is not purely functional. For instance, there is no way to enforce a lack of side effects as it is always possible to insert an impure statement(such as a `println`) in an arbitrary place.

Moreover, a lot of functional programming concepts that are actively used in Scala are just patterns, not first-level abstractions(for example, the `for` loop actually binds monads but there is no such thing as a monad in Scala). There is a scalaz libary for functional programming in Scala, which provides a bunch of basic typeclasses and purely functional data structeres that are not present in the standrad library.

Testing

1. Using ScalaTest framework, it is possible to write traditional unit-tests. If you are familiar with JUnit, you will easily learn how to do in Scala. Let's take a look at a small example which shows how to write assertions and check that a block of code throws a specific exception.

   ```
   // The class that contains tests should extend the FunSuite class.
   class UnitTestList extends FunSuite {

     // This how a test is defined.
     test("an empty list is empty") {
       // This is how assertions are written.
       assert(List().isEmpty)
     }

     test("a head of a list of one element is this element") {
       val elem = 1
       assert(List[Int](elem).head == elem)
     }

     test("a size of list of two elements is two") {
       val list = List[Int](1, 2)
       assert(list.size == 2)
     }

     test("head of an empty list throws an exception") {
       // This is how to check that a piece of code throws a specific exception.
       intercept[NoSuchElementException] {
         List().head
       }
     }

     test("tail of an empty list throws an exception") {
       intercept[UnsupportedOperationException] {
         List().tail
       }
     }

     test("concatenation of two lists") {
       val list1 = List(1, 2)
       val list2 = List(3)
       assert(list1 ++ list2 == List(1, 2, 3))
     }
   }
   ```

   Note that ScalaTest is not a part of the Scala standard library. To use it, you need to add the following dependency to the build.sbt file:

   `libraryDependencies += "org.scalatest" % "scalatest_2.11" % "2.2.4" % "test"`

2. ScalaTest also supports specification-based testing. Here is the same example rewritten in specification-style:

   ```
   // The class that contains tests should extend the FlatSpec class.
   class SpecTestList extends FlatSpec {

     // Here is how a test is defined.
     "An empty list" should " be empty" in {
       // Assertions are written in the same way as in the unit-tests example.
       assert(List().isEmpty)
     }

     "A head of List(1)" should "be equal to 1" in {
       val elem = 1
       assert(List(elem).head == elem)
     }

     "A list of two elements" should "have size equal to 2" in {
       assert(List(1, 2).size == 2)
   ```

```
    }

    "A head of an empty list" should "throw a NoSuchElementException" in {
      // Checking that a block of code throws an exception is done in the
      // same way as in the unit-tests example.
      intercept[NoSuchElementException] {
        List().head
      }
    }

    "A tail of an empty list" should "throw a UnsupportedOperationException" in {
      intercept[UnsupportedOperationException] {
        List().tail
      }
    }

    "Concatenation of two lists" should "contain elements from both of them" in {
      val list1 = List(1, 2)
      val list2 = List(3)
      assert(list1 ++ list2 == List(1, 2, 3))
    }
  }
```

It looks like unit-testing, but it emphasizes the desired behavior of the system instead of the process of writing the tests themselves.

3. A more intersting type of testing which is not common for traditional imperative programming languages is properties-based testing. The idea behind it is to specify properties that must always hold true and let the library generate a large amount of random examples to make sure that these properties are satisified. The advantage of this approach is that there is no need to come up with test data and the total number of checked inputs is much larger than what you can test by hand(or using a traditional XUnit framework). This libarary can generate random instances of a bunch of standard types including collections. To use this library, the following dependency should be added to the build.sbt file: `libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.12.2" % "test"` Let's take a look at a simple example:

```
  // The class that contains properties should extend the Properties class.
  class PropertyTestList extends Properties("List") {

    // This how a property is defined.
    property("cons of head and tail is equal to the list itself") = forAll {
      (a: List[Int]) => (!a.isEmpty) ==> {
        a.head :: a.tail == a
      }
    }

    property("size of concatenation is equal to sum of sizes") = forAll {
      (a: List[Int], b: List[Int]) => {
        a.size + b.size == (a ++ b).size
      }
    }

    property("head of cons(x, xs) is x") = forAll {
      (x: Int, xs: List[Int]) => {
        (x :: xs).head == x
      }
    }

    // One more way to define a property.
    // But is not checked automatically when defined this way.
    val propLength = forAll {
      (a: List[Int]) => (!a.isEmpty) ==> {
        a.size == a.tail.size + 1
      }
    }

    val propHeadOfTail = forAll {
      (a: List[Int]) => (a.size >= 2) ==> {
        a.tail.head == a(1)
      }
    }
    property("length of tail + 1 is equal to the length of the list") = propLength

    property("head of tail is the second element") = propHeadOfTail

    // One can also define combination of different properties.
    property("length and head of tail") = propLength && propHeadOfTail

    property("length or head of tail") = propLength || propHeadOfTail

  }
```

If we run this test suite, the libarary will generate a large amount of random lists and report whether the tests have passed or not. It has several useful additional features such as counterexample minimization(once it finds an input which falsifies a property, it tries to find a smaller example that makes it fail).

One of the functional programming virtues is composabilty. Let's take a look at a more sophisticated example which shows how to construct a generator for a user-defined type by composing other, already-defined generators. It allows us to check properties which take instances of a user-defined type as an argument. Let's start with creating a custom data type. It is a binary tree in this case:

```
abstract class BinaryTree[A] {
  def size: Int
  def add(elem: A)(implicit ord: Ordering[A]): BinaryTree[A]
  def contains(elem: A)(implicit ord: Ordering[A]): Boolean
  def toList: List[A]
}
```

The implementation of the tree does not matter for testing, so I will omit it. Now we need to write a generator:

```
// Given a standard generator for a list we, can easily define
// a generator for the binary tree.
def fromList[A](xs: List[A])(implicit ord: Ordering[A]): BinaryTree[A] =
  xs.foldLeft(BinaryTree.empty[A])((acc: BinaryTree[A], x) => acc.add(x))

// A custom generator for binary trees.
def genTree: Gen[BinaryTree[Int]] = arbitrary[List[Int]] map fromList[Int]

implicit lazy val arbTree: Arbitrary[BinaryTree[Int]] = Arbitrary(genTree)
```

Once we can generate arbitrary binary trees, we may specify a set of properties we want to check:

```
class PropertyTestBinaryTree extends Properties("BinaryTree") {

  // A code for the generator which was
  // shown in the previous snippet goes here.
  ...

  property("size of a tree is equal to the length of a toList list") = forAll {
    (t: BinaryTree[Int]) => t.size == t.toList.size
  }

  property("a tree removes duplicates and sorts elements") = forAll {
    (xs: List[Int]) =>
      val t = fromList(xs)
      val sorted = xs.distinct.sorted
      t.toList == sorted
  }

  property("a newly added element is always in a tree") = forAll {
    (x: Int, t: BinaryTree[Int]) =>
      t.add(x).contains(x)
  }

  property("a trees contains all elements of a list it was made from") =
    forAll {
      (xs: List[Int]) =>
        val t = fromList(xs)
        xs.forall(x => t.contains(x))
    }
}
```

Here, we have easily created a generator for binary trees using a generator for lists. We can compose several generators together using either the `for` loop or by calling the `flatMap` method directly:

```
// Given a generator for A, B and C, we can create a generator for (A, B, C).
def genTuple1[A, B, C](implicit arbA: Arbitrary[A], arbB: Arbitrary[B],
    arbC: Arbitrary[C]): Gen[Tuple3[A, B, C]] =
  for {
    a <- arbA.arbitrary
    b <- arbB.arbitrary
    c <- arbC.arbitrary
  } yield (a, b, c)

def genTuple2[A, B, C](implicit arbA: Arbitrary[A], arbB: Arbitrary[B],
                       arbC: Arbitrary[C]): Gen[Tuple3[A, B, C]] =
  arbA.arbitrary.flatMap(a => arbB.arbitrary.flatMap(b =>
    arbC.arbitrary.map(c => (a, b, c))))
```

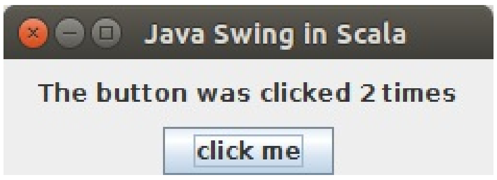
GUI in Scala

Java Swing

It is possible to use Java Swing toolkit directly. Here is an example of a simple application which has a button and a shows the number of times it was clicked:

```
import java.awt.{Dimension, Component, EventQueue}
import java.awt.event.{ActionEvent, ActionListener}
import javax.swing._

object JavaSwing {
  def main (args: Array[String]): Unit = {
    EventQueue.invokeLater(new GUI())
  }
}

class GUI extends Runnable {

  val verticalGap = 10

  var clicksCount = 0
  var frame: JFrame = null
  var label: JLabel = null
  var button: JButton = null

  /**
   * Creates a new frame and adds a label and a button to it.
   */
  override def run(): Unit = {
    frame = new JFrame("Java Swing in Scala")
    frame.setLayout(new BoxLayout(frame.getContentPane, BoxLayout.Y_AXIS))
    frame.add(Box.createRigidArea(new Dimension(0, verticalGap)))
    addLabel()
    frame.add(Box.createRigidArea(new Dimension(0, verticalGap)))
    addButton()
    frame.pack()
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
    frame.setVisible(true)
  }

  /**
   * Adds a new label to the frame.
   */
  def addLabel(): Unit = {
    label = new JLabel("The button has not been clicked yet")
    label.setAlignmentX(Component.CENTER_ALIGNMENT)
    frame.add(label)
  }

  /**
   * Adds a new button to the frame.
   */
  def addButton(): Unit = {
    button = new JButton("click me")
    button.setAlignmentX(Component.CENTER_ALIGNMENT)
    frame.add(button)
    button.addActionListener(new ActionListener {
      override def actionPerformed(e: ActionEvent): Unit = {
        clicksCount = clicksCount + 1
        label.setText("The button was clicked " + clicksCount + " "
          + getTimeOrTimes(clicksCount))
      }
    })
  }

  def getTimeOrTimes(n: Int): String =
    if (clicksCount == 1)
      "time"
    else
      "times"
}
```

Here is how it looks like:



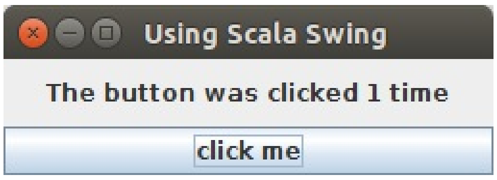
The button was clicked 2 times

Scala Swing

Scala Swing is a wrapper of Java Swing for Scala. It has similar classes hierarchy and layout managers. Let's take a look at the same application implemented using Scala Swing:

```
import scala.swing.Swing._
import scala.swing._

object ScalaSwing extends SimpleSwingApplication {

  val verticalGap = 10

  def top = new MainFrame {
    // The title of the frame.
    title = "Using Scala Swing"

    var clicksCount = 0

    // A label with the number of button clicks.
    val label = new Label("The button has not been clicked yet")
    // A button which changes the text of the label when it is clicked.
    val button = new Button {
      text = "click me"
      action = Action("click me") {
        clicksCount = clicksCount + 1
        label.text = "The button was clicked " + clicksCount + " " +
          getTimeOrTimes(clicksCount)
      }
    }

    contents = new BoxPanel(Orientation.Vertical) {
      contents += VStrut(verticalGap)
      contents += new BorderPanel{ add(label, BorderPanel.Position.Center)}
      contents += VStrut(verticalGap)
      contents += new BorderPanel{ add(button, BorderPanel.Position.Center)}
    }

    def getTimeOrTimes(n: Int): String =
      if (clicksCount == 1)
        "time"
      else
        "times"
  }
}
```

Let's run the code and see what we get:



The button was clicked 1 time

Scala Swing allows to write ideomatic Scala code for GUI applications, but it has several disadvantages compared to Java Swing. For instance, its documentation is not full and it is harder to find tutorials on it.

Advanced features of the type system

Typeclasses

In traditional object-oriented programming languages it is impossible to add a new method to an existing class without modifying its source code. One the other hand, type classes are much more flexible(it is easy to make an existing type an instance of a new typeclass and it is also easy to make a new type an instance of an existing typeclass). So is it possible to create something similar to type classes in Scala? Yes, it is, even though type classes are not a part of the language. Let's take a look at this example:

```
 // Here we define a new trait, which represents a type class.
 trait MyPlus[A] {
    def plus(x: A, y: A): A
  }

  // Here we have a function which takes a value
  // of a type which is an instance of this type class.
  def addThreeTimes[A: MyPlus](a: A): A = {
    val p = implicitly[MyPlus[A]]
    p.plus(a, p.plus(a, a))
  }
```

The cool thing about it is that we can make an exisiting type an instance of a newly-defined type class so that the `addThreeTimes` function can operate on values of this type. Here we do it with a List:

```
// Making a List[A] an instance of the MyPlus type class.
implicit def myPlusList[A]: MyPlus[List[A]] = new MyPlus[List[A]] {
  def plus(xs: List[A], ys: List[A]) = xs ++ ys
}

// Now the addThreeTime function can work with lists.
addThreeTimes(List(1, 2)) // List(1, 2, 1, 2, 1, 2)
```

Method injection

It is actaully possible to go further and add a method to a class without accessing its source code. Let's continue the example show above:

```
// We create a new trait and provide an implicit conversion
// function. Any type which is an instance of the MyPlus
// typeclass automatically gets the plusOp method.
trait PlusOp[A] {
  val op: MyPlus[A]
  val lhs: A
  def plusOp(rhs: A) = op.plus(lhs, rhs)
}

implicit def toPlusOp[A: MyPlus](a: A): PlusOp[A] = new PlusOp[A] {
  override val op = implicitly[MyPlus[A]]
  override val lhs = a
}
```

Now we can use the `plusOp` method on lists without writing any additional code:

```
// Using infix notation
List(1, 2, 3) plusOp List(4, 5) // List(1, 2, 3, 4, 5)
// Using standard notation.
List(1, 2, 3).plusOp(List(4, 5)) // List(1, 2, 3, 4, 5)
```

Higher-kinded polymorphism

Let's start with defining a `Monad` typeclass(or a trait, in terms of Scala):

```
trait Monad[M[_]] {
  def unit[A](a: A): M[A]
  def bind[A, B](m: => M[A], f: A => M[B]): M[B]
}
```

A `Monad` is abstract over a type `M` which, in turn, is abstract over a proper type(or, put it another way, it is abstract over a type constructor). It has a kind `* -> *`, which makes it a higher-kinded type. Let's play with it a little bit more:

```
// A method injection techinque as described above.
trait MonadOp[M[_], A] {
  val m: Monad[M]
  val lhs: M[A]
  def >>=[B](f: A => M[B]): M[B] = m.bind(lhs, f)
  def >>[B](g: => M[B]) = m.bind(lhs, (a: A) => g)
}

implicit def toMonadOp[M[_]: Monad, A](monad: M[A]): MonadOp[M, A] =
  new MonadOp[M, A] {
    override val m = implicitly[Monad[M]]
    override val lhs = monad
  }

// Makes a List an instance of the monad typeclass.
implicit lazy val listMonad = new Monad[List] {
  def unit[A](a: A) = List(a)
  def bind[A, B](m: => List[A], f: A => List[B]): List[B] = m flatMap f
}

// Now we can write code like this:
println(List(1, 2, 3) >>=
  (x => List("a", "b") >>= (y => implicitly[Monad[List]].unit(x, y))))

println(List(1, 2, 3) >> List("a", "b", "c"))
```

This code behaves like the standard `flatMap` method of a `List`, but unlike the standard `for` comprehension, it is not ducked typed anymore. A monad was made a first-level abstraction of the language. I would like to highlight one inconvenience that arose here: we have to write `implicitly[Monad[List]].unit(x, y)` rather then `unit(x, y)` because Scala does not support dispatch on a type of the return value.

This example shows that it is possible to implement many functional programming abstractions in Scala, but it requires writing more code than in a purely functional programming language like Haskell.

Do these examples have any practical use?

Yes, they do. Many third-party Scala libraries use the notion of type classes(directly or indirectly), so one needs to understand what it is in order to use these libraries efficiently.

Data processing in Scala

XML

1. To gets started with xml processing in Scala, you need to add the scala-xml library to the project. In sbt, the following dependencie should be added to the build.sbt file: `libraryDependencies += "org.scala-lang.modules" %% "scala-xml" % "1.0.4"`

2. Creating a document. Scala xml library can work with xml literals. It means that one can create an xml document in Scala by writing xml code directly:

```
val elem =
  <books_list name="my books">
    <book id="1" name="book1">The first book</book>
    <book id="2" name="book2">The second book</book>
  </books_list>
```

3. We usually need to process xml documents stored in a file or passed as a string. The `loadString` method creates an xml element from a string. This code does as the same thing as the snippet shown above:

```
import scala.xml.XML.loadString

...

val doc = """<books_list name="my books">""" +
          """<book id="1" name="book1">The first book</book>""" +
          """<book id="2" name="book2">The second book</book>""" +
          """</books_list>"""
val elem = loadString(doc)
```

4. Selectors. To get a sequence of direct children of the given element with a specific tag one can use the `\` method. For example, the expression `val elem \ "book"` returns a sequence of two element(the first book and the second book). This sequence can be processed using a bunch of standard methods, such as `map` and `filter`. Here is how we can find names of all books:

```
println((elem \ "book") map (_ \"@name")) // List(book1, book2)
```

   The `\"@attr"` method selects the specified attribute of the element.
   Note that the `\` method works with direct children of the element. To access all descendants, one should use the `\\` method. Let's see the difference:

```
val elem =
  <outer>
    <inner>The first inner element</inner>
    <middle>
      <inner>The second inner element</inner>
    </middle>
  </outer>

println(elem \ "inner")
// <inner>The first inner element</inner>

println(elem \\ "inner")
// <inner>The first inner element</inner><inner>The second inner element</inner>
```

5. Mixing Scala code and xml literals. Curly brackets can be used to escape a part of an xml literal. Here is a small

example:

```
val firstName = "First name"
val lastName = "Last name"
val name =
  <name>
    <first_name>{firstName}</first_name>
    <last_name>{lastName}</last_name>
  </name>
// The result is:
// <name>
//   <first_name>First name</first_name>
//   <last_name>Last name</last_name>
// </name>
```

JSON

In this tutorial I will use the play-json library. It can be downloaded independently from the play framework.

1. To use this library one should add the following dependency to the build.sbt file: `libraryDependencies += "com.typesafe.play" %% "play-json" % "2.4.0-M3"`

2. Creating JSON values. One way to create a JSON value is to use the `Json.parse` method. It takes a string and returns a `JsValue`:

   ```
   val sampleValue = Json.parse(
     """{ "numbers": [1, 2, 3], "full": true, "ref": null}""")
   ```

   Another way is to use constructors explicitly:

   ```
   val theSameValue = JsObject(Seq(
     "numbers" -> JsArray(Seq(JsNumber(1), JsNumber(2), JsNumber(3))),
     "full" -> JsBoolean(true),
     "ref" -> JsNull
   ))
   ```

3. Serialization. This library can serialize values of many standard types(`Int` , `Double` , `Boolean` , `String` and so on). It is also possible to serialize values of a custom type `T` by defining an implicit converter to the `Writes[T]` type. Let's create a small class and make it serializable:

   ```
   class MyClass(val numbers: Array[Int], val full: Boolean)

   implicit val writer = new Writes[MyClass] {
     def writes(myInstance: MyClass): JsValue = JsObject(Seq(
       "numbers" -> Json.toJson(myInstance.numbers),
       "full" -> Json.toJson(myInstance.full)
     ))
   }

   val myInstance = new MyClass(Array(1, 2, 3), false)
   val serialized = Json.toJson(myInstance) // {"numbers":[1,2,3],"full":false}
   ```

   Note that `Array` s of this type become serializable automatically.

4. Traversing a JSON value. The `\` method allows to find the value of a field by its name in a JsObject: `JsObject(Seq("foo" -> JsString("bar"))) \ "foo" // returns "bar"`

   The `\\` method does a similar thing, but it traverses the entire strucutre of the object recursively.

`JsArray`'s elements can be accessed by an index in the following way(indices are zero-based):

`JsArray(Seq(JsNumber(1), JsNumber(2), JsNumber(3)))(0) // returns JsNumber(1)`

5. Printing a JSON value. There are several methods for converting a JsValue to a String. For instance, `Json.stringify` returns a compact representation and `Json.prettyPrint` returns a human-readable version.

6. Deserialization. A JsValue can be converted to any type which has an implicit converter to the `Reads[T]` type. It is defined for a wide range of standard types. Here is how we can make the `MyClass` class deserializable:

```
// First, we need to define a converter.
// A typical way to read a value of a custom type
// is to combine readers for standard types.
// Here we combine a reader for an array of ints
// and a boolean.
implicit val reader: Reads[MyClass] = {
  val builder = (JsPath \ "numbers").read[Array[Int]] and
    (JsPath \ "full").read[Boolean]
  builder((array, bool) => new MyClass(array, bool))
}

// Now we can convert JsValues to instances of MyClass.
val fromJson = JsObject(Seq(
  "numbers" -> JsArray(Seq(JsNumber(1), JsNumber(2), JsNumber(3))),
  "full" -> JsBoolean(true)
)).as[MyClass]
```

If we need to have a better error handling, we can use the `validate` method instead of `as`(it also converts a json value to a value of a specific type, but it captures the error information if the parsing fails).

Raw text

1. Like most of the other modern programming languages, Scala supports regular expressions. This small code snippet shows basic ways to use them:

```
import scala.util.parsing.combinator.RegexParsers

object SampleParsers extends RegexParsers {
  // Creating parsers using regular expressions.

  // An integer decimal number.
  def number: Parser[String] = """-?(\d)+""".r
  // An identifier.
  def id: Parser[String] = """([a-z]|[A-z]|_)([a-z]|[A-z]|[0-9]|_)*""".r

  // Let's check how they work.
  def main(args: Array[String]): Unit = {
    // Successfully parses -123 number.
    // The last character remains untouched.
    println(parse(number, "-123a"))
    // This one fails.
    println(parseAll(number, "-123a"))
    // Both versions succeed.
    println(parse(number, 11))
    println(parseAll(number, 11))
    // Sucessfully parses identifiers.
    println(parse(id, "_123"))
    println(parse(id, "x1"))
  }
}
```

The difference between `parse` and `parseAll` is that the former succeeds if it matches any prefix of the given string while the latter succeeds if and only if it matches the entire string. These methods return a value of type `ParseResult` which can be a `Success`, a `Failure` or an `Error`(we can pattern match againts it).

2. It is also possible to combine parsers to obtain a new, more complex ones. Here are a few commonly used combinators:

```
import scala.util.parsing.combinator.RegexParsers

object SampleParsers extends RegexParsers {
  // An integer decimal number.
  def number: Parser[String] = """-?(\d)+""".r
  // An identifier.
  def id: Parser[String] = """([a-z]|[A-z]|_)([a-z]|[A-z]|[0-9]|_)*""".r

  // A number followed by an identifier.
  // p1 ~ p2 means p1 followed by p2.
  def numberThenId = number ~ id

  // p1 | p2 succeeds if p1 or p2 succeeds.
  // p1 is checked first.
  def numberOrId = number | id

  // p1.* matches zero or more repetitions of p1.
  def repeatedNumber = number.*

  // p1.? matches zero or one repetition of p1.
  def maybeNumber = number.?

  def main(args: Array[String]): Unit = {
    // Succeeds.
    println(parseAll(numberThenId, "123 x"))
    // Fails. The order is important.
    println(parseAll(numberThenId, "x 123"))
    // Fails. Both a number and an id must be present.
    println(parseAll(numberThenId, "123"))

    // The two following examples succeed.
    println(parseAll(numberOrId, "-11"))
    println(parseAll(numberOrId, "foo"))

    // Succeeds. * matches any number of repetitions.
    // Leading whitespaces are ignored by all parses by default.
    println(parseAll(repeatedNumber, "1 2 3"))

    // Succeeds.
    println(parseAll(maybeNumber, ""))
    println(parseAll(maybeNumber, "1"))
    // Fails. ? matches only zero or one repetition.
    println(parseAll(maybeNumber, "1 2"))
  }
}
```

Concurrency in Scala

Using Java standard library

All concurrency primitives and classes from the Java standard library are available in Scala. It also inherits the Java memory model. However, traditional imperative approach to concurrency has sevaral disadvantages, such as dealing with shared mutable state. Here is a quick overview of the Java-style concurrency in Scala:

1. Memory model. Scala runs on the JVM and its memory model is the same as in Java. This documentation contains a detailed description of the Java memory model.

2. Creating threads directly. To create a thread directly, one can pass a new `Runnable` to a `Thread` constructor. This program creates 4 threads that print specific messages. If you run it multiple times, you will see that messages are printed in a non-deterministic order(theoretically, there are no guarentees about their order at all). A `CountDownLatch` is used to make all the threads start printing messages at roughly the same time.

```
import java.util.concurrent.CountDownLatch

object SimpleThreads extends App {

  // A countdown latch will make all the threads wait before starting
  // printing messages.
  val startLatch = new CountDownLatch(1)

  // Creating a class that implements Runnable.
  class Printer(val message: String, val iterations: Int) extends Runnable {
    def run(): Unit = {
      startLatch.await()
      for (i <- 0 until iterations)
        println(message)
    }
  }

  val iterations = 5

  // Creating a list of threads that will print different messages once
  // they are started.
  // The threads are not started automatically.
  val threads = List(
    new Thread(new Printer("the first message", iterations)),
    new Thread(new Printer("the second message", iterations)),
    new Thread(new Printer("another message", iterations)),
    new Thread(new Printer("one more message", iterations)))

  // Starting the threads.
  threads.foreach(_.start())
  startLatch.countDown()

  // Waiting for their termination.
  threads.foreach(_.join())
}
```

3. The approach shown above have a few disadvantages. For example, if the number of tasks gets too large, creating a thread for each of them becomes infeasible(in most of the JVM implementations, JVM threads are mapped directly to native OS threads). Moreover, there is no way(at least, no easy way) to reuse an existing thread for another task. One can use thread pools from the Java standard library. Let's write the same programming using thread pools:

```
import java.util.concurrent.{CountDownLatch, Executors}

object ThreadPools extends App {

  val startLatch = new CountDownLatch(1)
```

```
    class Printer(val message: String, val iterations: Int) extends Runnable {
      def run(): Unit = {
        startLatch.await()
        for (i <- 0 until iterations)
          println(message)
      }
    }

    val iterations = 5

    // Creating a list of tasks. We don't create a threads here.
    val tasks = List(
      new Printer("the first message", iterations),
      new Printer("the second message", iterations),
      new Printer("another message", iterations),
      new Printer("one more message", iterations))

    // Using a factory method to create a new thread pool.
    val pool = Executors.newFixedThreadPool(tasks.length)

    // Now we can submit tasks to the pool.
    tasks.foreach(task => pool.execute(task))
    startLatch.countDown()

    // To avoid resource leaks, we need to shutdown the pool when we are done.
    pool.shutdown()
  }
```

Notice that the `CountDownLatch` is not compulsory for both of these examples(they would work fine without it, but it brings "more randomness" to the order of the printed messages on my machine). There is a bunch of other thread pool types that support more complex functionality, such as scheduling tasks or automatically adjusting the number of worker threads based on the number of tasks or hardware configuration.

4. Access to a shared mutable state requires synchronization. Unlike Java, Scala does not have the `synchronized` keyword, but the `synchronized` method of the `AnyRef` class has similar semantics. For instance, this Java code: `synchronized(this) {...}` can be rewritten in Scala as `this.synchronized {...}`.

5. All other synchronization primitives from the Java standard library(locks, count down latches and so on) are available in Scala.

6. The `@volatile`annotation in Scala has the same semantics as the `volatile` keyword in Java.

Thread safe data structures

Immutable collections are thread safe by default and do not require any additional synchronization. If you need to use a mutable thread safe collection in Scala, you can either mix in a special trait(such as `SynchronizedMap` and `SynchronizedSet`) to an existing non-thread safe collection or use an efficient thread safe collection from the Java standrard library.

Actors

Communication through a shared mutable state is not the only way to write concurrent programs. Akka system is based on actors, which communicate using asynchronous message passing. Here I'll show how to use it.

1. To use Akka actors in a project, the following dependencies should be added to the build.sbt file:

```
resolvers +=
  "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "com.typesafe.akka" % "akka-actor_2.11" % "2.3.11"
```

2. Let's write a small program which uses Akka actors and walk through it step-by-step:

```
import akka.actor.{Props, ActorSystem, Actor}

object Main extends App {
  // While it is possible to pass any object to an Actor, it is a good
  // a practice to use case classes/case objects for each type of a
  // message.
  case object SayHello

  case object StopSystem

  class HelloWorldPrinter extends Actor {
    def receive = {
      case SayHello => println("Hello, world!")
      case StopSystem => context.system.shutdown()
      case _ => println("Unknown message")
    }
  }

  val system = ActorSystem("Main")
  val helloWorldPrinter = system.actorOf(Props[HelloWorldPrinter])
  helloWorldPrinter ! SayHello
  helloWorldPrinter ! "another message"
  helloWorldPrinter ! StopSystem
}
```

The first thing we do in this code is creating a custom `case object` for each type of a message.

After that, we define a subclass of the `Actor` to create an actor with custom behavior. It processes recieved messages using the `recieve` method. Inside this method, we use pattern matching to distinguish messages of different types. Note that pattern matching doesn't need to be exhaustive.

Anything can be passed as message(its type is `Any`). `!` is just a method which sends it to a specific actor. In this example, the first message we send is `SayHello`. The `helloWorldPrinter` prints `Hello, world!` when it recieves this message. When it gets `"another message"`, it prints the `Unknown message` line.

To create a new system, one can use the apply method of the `ActorSystem` object. To create a new `Actor`, the `actorOf` method should be used. It is not possible to create a new actor by writing `new HelloWorldPrinter()`. It would compile, but it would throw an error at runtime.

It is necessary to shut the system down when we are done. It will run indefinitely if we don't do this. That's why we send one more message: `StopSystem` to the `helloWorldActor` at the end of the program. When it receives this message, it shuts the entire system down. In more complex cases, it might be not clear when the system should be stopped. A typical way to deal with it is to have an Actor which knows when all the work is done and shuts the system down.

3. Let's learn more about the behavior of the actors.

   - An actor does not necessary have a unique thread. It allows to create more actors than threads, but it means that we should be very careful with blocking operations(or, ideally, avoid them) because if an actor blocks, the entire thread it runs on blocks, too.
   - It is guaranteed that messages sent from one actor to the same destination arrive in a right order. Nothing else is guaranteed(that is, messages sent to different destinations can arrive in an arbitrary order).
   - Messages are sent asynchronously, but each actor is guaranteed to process at most one message at a time(that is, the `receive` method is invoked by at most one thread at a time).
   - Messages are published in a thread-safe manner.
   - Does it mean that we can completely forget about thread safety? No, of course it doesn't. If we publish a reference to a mutable non-thread safe data structure through a message, the program becomes incorrect. That's why

immutable state should be preferred(it is safe to publish a mutable reference to an immutable data structure, but not the other way round). However, it is safe to have mutable state which is fully contained within one actor(as long as it does not leak to the outer world).

4. Here are a few more useful methods: `sender` allows to refer to the sender of the message in the `recieve` method and `self` can be used to obtain a self-reference to an actor(it is important that `!` is a method of the `ActorRef` class, not the `Actor`, so `this ! message` would not compile. `self ! message` should be used instead).

5. Let's see how to create an Actor that has a constructor with parameters:

```
// An actor which has a constructor with two parameters.
class ActorWithParameters(val anInt: Int, val aString: String) extends Actor {
  def receive = ???
}

// The following code obviously does not do what we want.
// Unfortunately, is DOES compile but it throws an exception
// at runtime.
// val actorWithParameters = system.actorOf(Props[ActorWithParameters])

// This is a proper way to do it.
// Here we create an instance of the ActorWithParameters with
// anInt = 1 and aString = "foo".
val props = Props(classOf[ActorWithParameters], 1, "foo")
val actorWithParameters = system.actorOf(props)
```

6. Dealing with an actor that need to change its behavior. Let's assume that we have an actor which needs to react to messages differently depending on its current state. The most obvious way(and usaully not a good one) to implement it is to maintain some kind of state variable and a use a nested `case` or `if` expression inside the `receive` method. Something like this:

```
class StatefulActor extends Actor {
  // I will use an Int constant to specify the current state for brevity.
  var state = 0

  def receive = {
    case message1 =>
      state match {
        case 0 => ...
        case 1 => ...
        ...
        case k => ...
      }
    ...
    case messageN =>
      state match {
        case 0 => ...
        case 1 => ...
        ...
        case k => ...
      }
  }
}
```

This code looks pretty ugly and it is error-prone. Is there a better way to do it? Yes, there is. `become` and `unbecome` methods help to manage the state in a clear and concise way. Here is a small example:

```
case object WakeUp

case class Point(x: Double, y: Double)

case object GoToSleep

class StatefulActor extends Actor {
  // Initially the actor is sleeping.
```

```
    def receive = sleep

    // In this state it waits to be awaken.
    def sleep: Receive = {
      // Once it gets a WakeUp message, it changes its state.
      case WakeUp => context.become(compute)
    }

    // In this state it can perform some computations until it receives a
    // a message to go back to sleep.
    def compute: Receive = {
      case Point(x, y) => // do some useful computations
      case GoToSleep => context.become(sleep)
    }
  }
```

This one looks much better. `unbecome` is not shown in this example(it simply changes the behavior to the previous one).

7. Sometimes we need to stop an actor without shutting the entire system down. One way to do it is to call the `context.stop` method. Another way is to send it a `PoisionPill`(just like any normal message). In both cases, the actor stops receiving new messages but keeps processing already enqueued ones.

Summary

Actor-based concurrency model has several advantages compared to traditional approach which is based on shared state. It is easier to reason about(the mutable state can be fully encapsulated within an actor). It offers more transparent means of communication(messages are always send explicitly. Parts of the internal state of an actor cannot "accidently" become shared). It is possible to create a lot of short-lived actors efficiently because they are not mapped to native OS threads directly(unlike JVM threads which are rather expensive). However, it is not perfect. Publishing a reference to a non-thread safe object through a message can make the program incorrect(that why immutability should be preferred). The Akka impelentation of the actors sacrifices type-safety for flexibilty(the fact that a message has a type `Any` can lead to bugs that cannot be caught at compile time, even though pattern matching is fully typesafe), which can cause errors that are caught at runtime only(for instance, in the point 3 and 5 we can see an improper way to create an actor which compiles successfully).

Comparison between Scala and other programming languages

This chapter provides a comparison between Scala and several other popular programming languages. Some of the points made here are fully opinion-based.

I refer to a concrete implementation rather than a language itself in an abstract sense when I say that one language is faster than the other one or when I discuss the ability to interact with various resources and the code written in other programming language. I will use terms language/its implementation interchangeably here, even though it is not technically correct.

Scala and Java

Similarities

Both Scala and Java run a on the JVM, which makes them available on any platform that supports the JVM. It also makes all Java libraries available in Scala. They both have a good support for the object-oriented programming.

Advantages of Scala

1. Scala has a more powerful type system. For instance, higher-order polymorphism is supported in Scala, but not in Java. In addition, traits provide a restricted form of multiple inheritance and make several design patterns(such as dependency injection) very easy to implement. Java 8 also supports streams with similar methods, but they are not that convenient to use(it is always necessary to explicitly convert a collection to a stream and back).
2. Scala has a more concise syntax. Combined with partial function application, anonymous functions(notice that they are available in Java 8) and a bunch of higher-order methods for standard collections(such as map, fold, filter and so on) it allows to write much shorter code.
3. Type inference. In many cases it helps to avoid explicit type annotations, reducing the amount of noise and repetitions in the code.
4. Uniformity of primitives and objects. Values of types like Int, Double and others are treated as objects, so there is no need to convert them to a wrapped version to call a method on them.
5. Immutability. Scala's standard library has a better support of efficient immutable collections, which can be very helpful to reason about the correctness of a program(especially in a multithreaded environment).
6. Call-by-name evaluation model is available in Scala, but not in Java. It allows to implement control flow structures, such as conditional expressions and loops, as ordinary functions. It also makes possible to implement infinite data structures in a concise manner.
7. Lazy values make thread-safe lazy initialization a part of the language and reduce the amount of code the programmer needs to write(and the amount of possible bugs) compared to Java.
8. Scala also brings a bunch of concepts from functional programming, such as type classes(they are not a part of the language, but they can be easily modeled with case classes and inheritance and widely used in various libraries), currying and partial application, that help to look at standard problems from a completely different perspective and improve the design of an application or a library.
9. Operator overloading. Scala treats operators as methods, so it is possible to define operators for custom types. It makes the code more readable and natural in case of, for example, arbitrary precision integers. It is not possible to do the same thing in Java.
10. Implicit conversions and implicit parameters. They allow to reduce the amount of typing required in some cases. They also help to express several functional programming concepts in a concise manner(such as type classes).
11. Scala's syntax is convenient for creating domain-specific languages, which is not the case for Java.
12. Lack of checked exceptions. This one is definitely controversial, but I believe that it is better not to have them.
13. Some erroneous patterns are strongly discouraged in Scala. For example, the `Option` type is normally used instead of the `null` reference. This idioms started to make their way into Java(for instance, there is an `Option` class in Java 8).
14. Scala has a REPL. It allows to write and test code in an interactive manner. It's very useful when you want to experiment with your code.

Disadvantages of Scala

1. Performance. Sometimes Scala can be significantly slower than Java and the performance is less predictable(for instance, Int can be represented as a primitive or an object inside the JVM depending on the way it is used. The same is true for arrays).
2. Syntactic sugar. It is rather easy to go too far and write an unreadable code(by mixing infix/postfix method calls, partial application and other features of the language). There are a lot of different ways to write the same thing is Scala. It makes the discipline and a strict adherence to a particular code style very important for successful collaboration with other developers. It is not such a big problem in Java because it is a simpler language and provides less ways to make your code obscure.
3. Compilation time. Scala compiler is rather slow even when the source file is small. It can get pretty annoying if you need to recompile your project frequently.
4. Unstable public API. The API of the standard library and third-party libraries changes quite often and some changes completely break the old code. Choosing the right version of each library can make building your project rather complicated. Java does a much better job in terms of backward compatibility.
5. Documentation. Some of parts of the standard library(for example, Scala Swing, are not fully documented and it does create problems in practice). The same is true for many third-party libraries. Conversely, the Java standard library is documented very well and it is easy to find good tutorials on almost any part of it.
6. A complicated type system. While being more powerful, Scala type system is harder to understand. The ability to specify variance for type parameters can make things confusing. Combined with subtyping, higher-order polymorphism, pattern matching, type erasure and type inference it can lead to hard-to-understand compilation errors. However, it is usually not that bad in practice. Anyway, Java type system is simpler.
7. A mix of the functional and the object-oriented paradigm. To understand how they interact with each other, one needs to have a good understanding of both. If you know Java and some purely functional programming language(like Haskell), it is not a problem. However, if you are not familiar with the functional or the object-oriented paradigm, their mix can be very confusing.
8. Implicit conversions and implicit parameters. Yes, they do make the language more powerful. But they also can make the code hard to understand. The rules for applying them are not straight forward when multiple conversions are available.

Summary

So which of them is better? It is hard to give a precise answer, but I'd recommend to use Scala instead of Java unless there are good reasons not to do so. Moreover, compiled Java code can be called from Scala directly, so switching to it does not involve throwing away the existing code base.

Scala and C++

Similarities

It looks like Scala and C++ are very different, but they definitely have one thing in common(besides the support of object-oriented programming): they support multiple paradigms and provide a bunch of different(quite often, obscure) ways to implement the same thing. Both Scala and C++ have good libraries for different purposes.

Advantages of Scala

1. Scala is a higher-level language. It is safer and allows to write more concise code, which makes a programmer more productive.
2. Automatic memory management prevents a wide range of bugs.
3. C++ templates can cause strange parsing(and other) errors(at least if you don't know them well). A basic parametric polymorphism is definitely easier to use in Scala.
4. Scala has a stronger support of the functional paradigm.
5. Scala code can be executed in an interactive mode using a REPL. None of the popular C++ implementations provides

any means of interactivity.
6. Because of the dynamic linking, Scala usually requires less recompilation when changes are made. It can be important because both Scala and C++ compilers are relatively slow.
7. Scala is more strict in a sense that a compiler gives more guarantees about the program that typechecks. For example, it is not possible to "accidently" cast Long to Int or to use an assigment instead of a comparison inside the condition of the `if` expression. Some of these bugs can be detected by a C++ compiler and reported as warnings when appropriate options are turned on, but I prefer to use a language which completely forbids them.

Disadvantages of Scala

1. C++ is more time and memory efficient(this statement is not very precise: a language can't be more or less efficient than the other one. Only its implementation can. Of course, I'm comparing popular C++ implementations(g++, clang) with the standard Scala implementation here).
2. C++ provides a low-level access to various resources(which can cause more bugs, but it can be crucial if this level of control is necessary).
3. It also supports more convenient non-memory resources management(with deterministic destructors and RAII idiom) compared to Scala's `try-catch-finally` blocks.

Summary

If you start a new project, I'd strongly recommend to prefer Scala to C++ unless you need very high performance(in general, Scala does a pretty good job), a very small memory footprint or a direct access to low level resources.

Scala and Haskell

Similarities

Scala and Haskell support functional programming much better than most of the other popular programming languages(I would not call Scala a functional language, though). They both can be used for creating domain-specific languages. Scala and Haskell provide a good support for parsing various data formats. They also have similar problems: their build systems can be painful to use and many libraries do not have good documentation and tutorials.

Advantages of Scala

1. Scala is easier to learn for people who come from an imperative background(that is, most of the programmers nowadays). It resembles Java in many aspects and does not require you to fully dive into functional programming from the very beginning. This point has nothing to do with the internal quality of these language, but it can be important in real life.
2. It runs on a JVM, which makes it portable. It is not the case with Haskell.
3. Scala is strict by default, which makes its performance and memory consumption more predictable. Quite often, Haskell code requires profiling and adding strictness to make it run fast.
4. Arrays are easy to use in Scala. They are not that easy to use in Haskell(mutable arrays are available only inside the `ST` and `IO` monads). Taking into account that arrays(with `O(1)` access and update operations) play an important role in many efficient algorithms, it makes Scala easier to use to implement some of them. Purely functional counterparts often either have a worse time/space complexity or at least a larger constant and can be much more complicated. Scala does not push you to keep everything purely functional, but Haskell does.
5. Scala allows to use existing Java code.

Disadvantages of Scala

1. Haskell has a more powerful type inference. In most of the cases, type annotations are not required at all.
2. There are no problems that arise from mixing subtyping and parametric polymorphism in Haskell as it doesn't have subtyping.

3. Haskell has a clear separation between pure and impure code(I assume that we do not use backdoors such as `unsafePerformIO`). There is no way to restrict side-effects in Scala.
4. Haskell supports functional programming concepts directly. For example, type classes and ADTs(algebraic data types) are a part of the language in Haskell. They have to be modeled in Scala. It usually results in more code and the code itself becomes more complicated.
5. In Haskell, many concepts a part of the language rather than just a design pattern. For instance, in Scala the `for` comprehension binds monads, but they are not called this way and this syntactic construct is ducked-typed. Combined with implicit conversions, it can cause troubles(even though doesn't usually happen in practice). In Haskell, the `do` block is type checked normally. In general, Haskell has less "magical" syntax which makes it easier to understand and prevents a class of errors.
6. In general, Haskell feels more natural and more pure. The main reason for it, in my opinion, is that it does not mix different paradigms. Obviously, it is very difficult to put a bunch of completely different things into one language and keep it uniform.
7. If you are interested in type theory, Haskell(with GHC extensions) gives more opportunities to experiment with types. Some of these extensions are also useful in practice to write more secure code or get more guarantees from the compiler(a typical example here is a simple interpreter implemented with GADTs which does not typechecks if the it is not correct).
8. The point 4 from the previous list can also be viewed as an advantage of Haskell. By pushing you to write purely functional code, it forces you too switch to functional paradigm much faster. It is easy to use Scala as Java with slightly different syntax. You cannot do that with Haskell.
9. One can call C code from Haskell(but it is not as straight forward as using Java code from Scala).
10. Being compiled into machine code(I refer to the GHC implementation of Haskell here), Haskell can be more efficient than Scala.
11. Haskell allows to write more concise code(due to more powerful type inference and more convenient syntax for currying/uncurrying/partial application/function composition).
12. Haskell supports a dispatch on a type of the return value. Scala doesn't. That's why implementing some functional programming concepts requires finding workarounds in Scala(a typical example is a monad).

Summary

Which one of them would I recommend? If you are not bound by any existing code and you are ready to fully dive into functional programming(and ready to learn a bunch completely of new(assuming that you don't have prior functional programming/type theory/category theory experience) concepts), I'd suggest using Haskell. It is definitely a better choice if you want to take a look at programming from a completely different point of view and it is more uniform because it does not mix paradigms. However, Scala can be a better choice in practice if you have mostly worked with "traditional" programming languages like Java, C# or C++(and imperative paradigm in general) as it is more similar to them(it is possible to "gently" switch from Java to Scala).