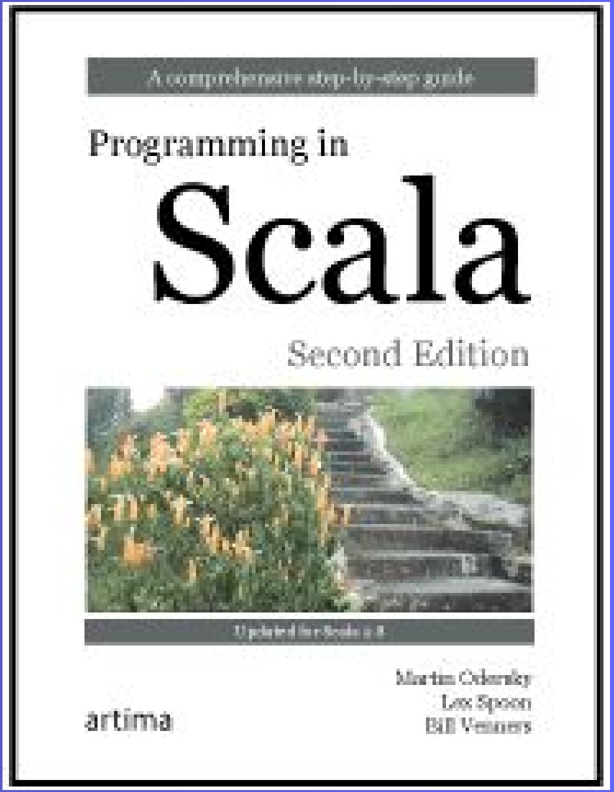
Stairway to Scala - Flight 13

Partial functions and actors

Bill Venners

Dick Wall

escalatesoft.com



Flight 13 goal

Look at partial functions and get a taste of the Akka actors library.

Partial functions

A partial function is a function that takes 1 parameter that may be defined for only a subset of its possible inputs.

Example: square root defined only for positive Doubles

Conceptual partial function

```
val sqrt: Function1[Double, Double] = { d =>
    require(d >= 0, s"No sensible result for $d")
    scala.math.sqrt(d)
}

scala> sqrt(2.0)
res62: Double = 1.4142135623730951

scala> sqrt(-2.0)
java.lang.IllegalArgumentException: requirement failed: No
sensible result for -2.0
 ...
```

Actual partial functions

```
val sqrt: PartialFunction[Double, Double] = {
  case d if d >= 0 => scala.math.sqrt(d)
  case d =>
     throw new Exception("No sensible result for " + d)
}
```

```
scala> sqrt(2.0)
res62: Double = 1.4142135623730951

scala> sqrt(-2.0)
java.lang.Exception: No sensible result for -2.0
 ...
```

Partial function literals

```
  val sqrt: PartialFunction[Double, Double] = {
    case d if d >= 0 => scala.math.sqrt(d)
  }

scala> sqrt(2.0)
res62: Double = 1.4142135623730951

scala> sqrt(-2.0)
scala.MatchError: -2.0 (of class java.lang.Double)
 ...
```

It really is a *partial* function

```
val second: (List[Int] => Int) = {
  case x :: y :: _ => y
}
```

```
warning: match is not exhaustive!
missing combination            Nil
```

3-element list works, empty list does not

```
scala> second(List(5,6,7))
res24: Int = 6

scala> second(List())
scala.MatchError: List()
    at $anonfun$1.apply(<console>:17)
    at $anonfun$1.apply(<console>:17)
```

isDefinedAt

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y
}

scala> second.isDefinedAt(List(5,6,7))
res30: Boolean = true

scala> second.isDefinedAt(List())
res31: Boolean = false
```

How it's compiled

```
{ case x :: y :: _ => y }

new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}
```

An actor's act method

```
import scala.actors._

object SillyActor extends Actor {
  def act(): Unit = {
    for (i <- 1 to 5) {
      println("I'm acting!")
      Thread.sleep(1000)
    }
  }
}
```

Start an actor with start()

```
scala> SillyActor.start()
I'm acting!
res4: scala.actors.Actor = SillyActor\$@1945696

scala> I'm acting!
I'm acting!
I'm acting!
I'm acting!
```

Each actor runs independently

```
import scala.actors._

object SeriousActor extends Actor {
  def act(): Unit = {
    for (i <- 1 to 5) {
      println("To be or not to be.")
      Thread.sleep(1000)
    }
  }
}
```

Independent actors

```
scala> SillyActor.restart(); SeriousActor.start()
res3: scala.actors.Actor = seriousActor\$@1689405

scala> To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
```

The actor method

```
scala> import scala.actors.Actor._

scala> val seriousActor2 = actor {
        for (i <- 1 to 5) {
           println("That is the question.")
          Thread.sleep(1000)
         }
      }

scala> That is the question.
That is the question.
That is the question.
That is the question.
That is the question.
```

Sending a message

```
scala> SillyActor ! "hi there"

val echoActor = actor {
  while (true) {
    receive {
      case msg =>
        println("received message: " + msg)
    }
  }
}

scala> echoActor ! "hi there"
received message: hi there
```

An actor has an "inbox"

- Actor will only process messages matching one of the cases passed to receive

```
scala> val intActor = actor {
        receive {
          case x: Int => // I only want Ints
            println("Got an Int: "+ x)
        }
      }

scala> intActor ! "hello"
scala> intActor ! math.Pi
scala> intActor ! 12
Got an Int: 12
```

Including/Importing Akka

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

libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.4.1"
```

```
import akka.actor._
import akka.dispatch._
import akka.util.Timeout
import scala.concurrent.duration._
import scala.concurrent.Await   // Use sparingly
import akka.pattern._
```

Creating an Akka Actor

```
class Reflector extends Actor {

    def receive = {
        case s: String => sender ! s.reverse
        case b: Boolean => sender ! (!b)
        case i: Int => sender ! (i * -1)
    }
}

val system = ActorSystem("ReflectorSystem")
val reflector = system.actorOf(Props(new Reflector), name = "reflector")
```

Using the Actor

```
scala> implicit val timeout = Timeout(5 seconds)

scala> val f1 = reflector ? 6
f1: akka.dispatch.Future[Any] = akka.dispatch.DefaultPromise@671d0d

scala> f1.value
res1: Option[scala.util.Try[Any]] = Some(Success(-6))

scala> Await.result(f1, timeout.duration)
res2: Any = -6

scala> val f2 = reflector ? 3.4
f2: akka.dispatch.Future[Any] = akka.dispatch.DefaultPromise@1b3cedc

scala> f2.value
res3: Option[scala.util.Try[Any]] = Some(Failure(akka.pattern.
AskTimeoutException))
```

Back to the Future!

```
scala> val f3 = f1.map { case i: Int => i * -7 }
f3: akka.dispatch.Future[Int] = akka.dispatch.DefaultPromise@139e9f8

scala> f3.value
res4: Option[scala.util.Try[Int]] = Some(Success(42))

scala> f3.<hit tab>
andThen         apply         asInstanceOf
failed          fallbackTo    filter
flatMap         foreach       isCompleted
isInstanceOf    map           mapTo
onComplete      onFailure     onSuccess
ready           recover       recoverWith
result          toString      value
withFilter      zip
```

In the Future, Everything Will Be Better

```
scala>  f3.onSuccess {
          case i: Int => println("it worked %d".format(i))
         }
res5: f3.type = akka.dispatch.DefaultPromise@139e9f8
scala> it worked 42

scala> import scala.concurrent._

scala> import ExecutionContext.Implicits.global

scala> implicit val ec = ExecutionContext.defaultExecutionContext(system)

scala> val future = Future { Thread.sleep(20000); "Hello, World" }

scala> future.onSuccess { case s: String => println (s) }
res6: future.type = akka.dispatch.DefaultPromise@8e86bd
scala> Hello, World
```

I Never Promised You A Perfect Future

```
scala> val yayy = Promise.successful("It worked!")
yayy: akka.dispatch.Promise[java.lang.String] = akka.dispatch.
KeptPromise@27b86c

scala> yayy.future.value
res7: Option[scala.util.Try[String]] = Some(Success(It worked!))

scala> val nayy = Promise.failed(new IllegalStateException("Can't do it
Captain!"))
nayy: akka.dispatch.Promise[Nothing] = akka.dispatch.KeptPromise@149c332

scala> nayy.future.value
res8: Option[scala.util.Try[Nothing]] = Some(Failure(java.lang.
IllegalStateException: Can't do it Captain!))
```

http://docs.scala-lang.org/sips/pending/futures-promises.html

Exercises for Flight 13