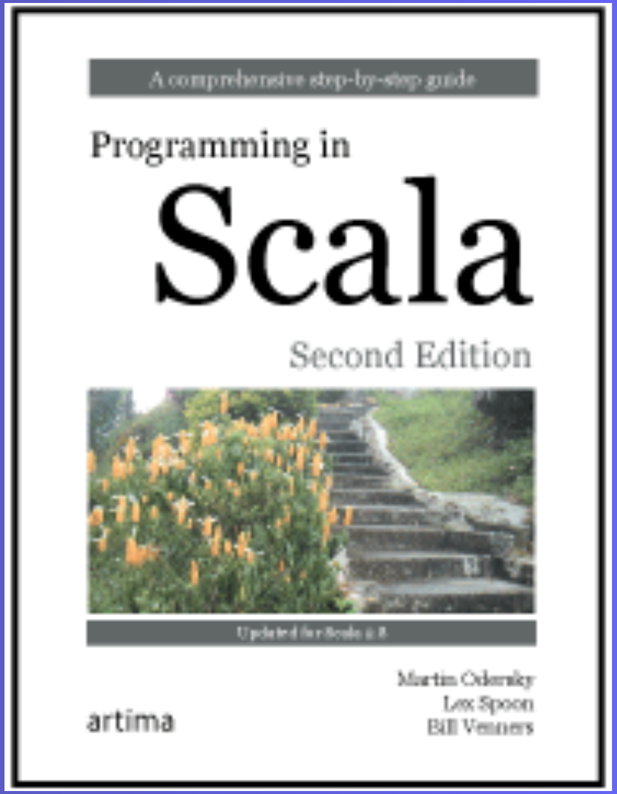
artima

Stairway to Scala - Flight 13

Partial functions and actors

Bill Venners
Dick Wall

www.artima.com



Flight 13 goal

Look at partial functions and get a taste of Scala's actors library.

Partial functions

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}

scala> withDefault(Some(10))
res28: Int = 10

scala> withDefault(None)
res29: Int = 0
```

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() {
    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() {
    for (i <- 1 to 5) {
      println("To be or not to be.")
      Thread.sleep(1000)
    }
  }
}
```

Independent actors

```
scala> SillyActor.start(); 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
```

Can treat native threads as actors

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

scala> self ! "hello"

scala> self.receive { case x => x }
res6: Any = hello

scala> self.receiveWithin(1000) { case x => x } // wait a sec!
res7: Any = TIMEOUT
```

An actor that calls react

```
object NameResolver extends Actor {
  import java.net.{InetAddress, UnknownHostException}

  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
        act()
      case "EXIT" =>
        println("Name resolver exiting.")
        // quit
      case msg =>
        println("Unhandled message: "+ msg)
        act()
    }
  }
```

An actor that calls react (cont.)

```
  def getIp(name: String): Option[InetAddress] = {
    try {
      Some(InetAddress.getByName(name))
    } catch {
      case _:UnknownHostException => None
    }
  }
}
```

Using the name resolver

```
scala> NameResolver.start()
res0: scala.actors.Actor = NameResolver\$@90d6c5

scala> NameResolver ! ("www.scala-lang.org", self)

scala> self.receiveWithin(0) { case x => x }
res2: Any = Some(www.scala-lang.org/128.178.154.102)

scala> NameResolver ! ("wwwwww.scala-lang.org", self)

scala> self.receiveWithin(0) { case x => x }
res4: Any = None
```

Can use loop

```
def act() {
  loop {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
      case "EXIT" =>
        exit()
      case msg =>
        println("Unhandled message: " + msg)
    }
  }
}
```

Round Trip Alternatives

```
// Alternative to the act() definition
def act() {
  while(true) {
    receive {
      case "EXIT" => // now has to be first (String match)
        println("Name resolver exiting.")
        exit()
      case name: String =>  // no need to send source actor
        reply(getIp(name))   // reply sends message back
      case msg =>
        println("Unhandled message: "+ msg)
    }
  }
}
```

!?, !! and Futures

```
val results = NameResolver !? "http://www.javaposse.com"
```
(Blocking - returns results directly)

```
val results = NameResolver !? (100, "http://www.javaposse.com")
```
(Waits for 100ms, returns None if no response in that time)

```
val future = NameResolver !! "http://www.javaposse.com"
```
Returns a Future[Any]

```
future.isSet
```
Returns true if result is ready, false otherwise

```
future.apply()
```
Returns Some(results) if ready, None otherwise

Actor Lifecycle (Simplified)

```
scala> NameResolver.getState
res0: scala.actors.Actor.State.Value = New

scala> NameResolver.start()
res1: scala.actors.Actor = NameResolver$@84a6b9

scala> NameResolver.getState
res2: scala.actors.Actor.State.Value = Blocked
```

Could also be Suspended, Running, etc.

Actor Lifecycle (Continued)

```
scala> NameResolver ! "EXIT"
Name resolver exiting.

scala> NameResolver.getState
res4: scala.actors.Actor.State.Value = Terminated

scala> NameResolver.restart()

scala> NameResolver.getState
res6: scala.actors.Actor.State.Value = Blocked
```