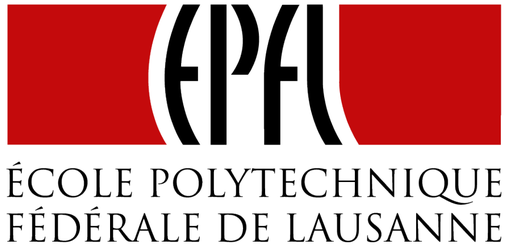


Combinators on Futures

Principles of Reactive Programming

Erik Meijer

Futures recap

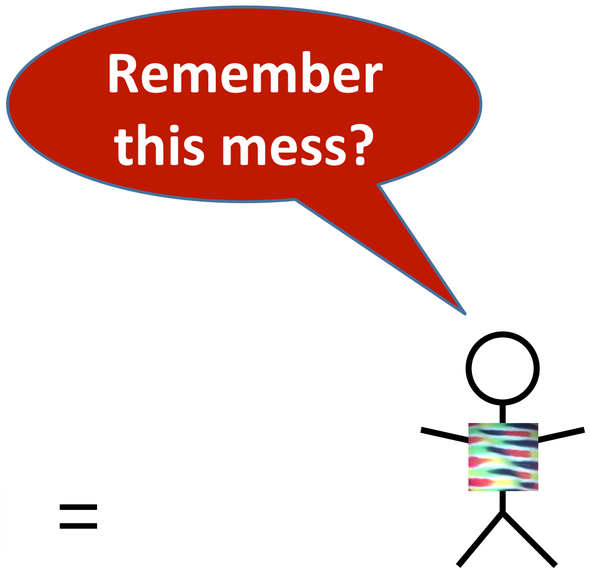
```
trait Awaitable[T] extends AnyRef {
  abstract def ready(atMost: Duration): Unit
  abstract def result(atMost: Duration): T
}

trait Future[T] extends Awaitable[T] {
   def filter(p: T⇒Boolean): Future[T]
   def flatMap[S](f: T⇒ Future[S]): Future[U]
   def map[S](f: T⇒S): Future[S]
   def recoverWith(f: PartialFunction[Throwable, Future[T]]): Future[T]
}

object Future {
  def apply[T](body : ⇒T): Future[T]
}
```

Sending packets using futures

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()
packet onComplete {
  case Success(p) ⇒ {
    val confirmation: Future[Array[Byte]] =
      socket.sendToEurope(p)
  }
  case Failure(t) ⇒ …
}
```



Flatmap to the rescue

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()
val confirmation: Future[Array[Byte]] =
  packet.flatMap(p ⇒ {
      socket.sendToEurope(p)
  })
```

Sending packets using futures under the covers

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.imaginary.Http._

object Http {
  def apply(url: URL, req: Request): Future[Response] =
    {… runs the http request asynchronously …}
}

def sendToEurope(packet: Array[Byte]): Future[Array[Byte]] =
  Http(URL("mail.server.eu"), Request(packet))
    .filter(response ⇒ response.isOK)
    .map(response ⇒ response.toByteArray)
```

Sending packets using futures robustly (?)

```
def sendTo(url: URL, packet: Array[Byte]): Future[Array[Byte]] =
  Http(url, Request(packet))
    .filter(response ⇒ response.isOK)
    .map(response ⇒ response.toByteArray)

def sendToAndBackup(packet: Array[Byte]):
                                    Future[(Array[Byte], Array[Byte])] = {

  val europeConfirm = sendTo(mailServer.europe, packet)

  val usaConfirm = sendTo(mailServer.usa, packet)

  europeConfirm.zip(usaConfirm)
}
```

Send packets using futures robustly

```
def recover(f: PartialFunction[Throwable,T]): Future[T]

def recoverWith(f: PartialFunction[Throwable,Future[T]]): Future[T]
```

Send packets using futures robustly

```
def sendTo(url: URL, packet: Array[Byte]): Future[Array[Byte]] =
  Http(url, Request(packet))
    .filter(response ⇒ response.isOK)
    .map(response ⇒ response.toByteArray)

def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] =
  sendTo(mailServer.europe, packet) recoverWith {
    case europeError ⇒ sendTo(mailServer.usa, packet) recover {
      case usaError  ⇒ usaError.getMessage.toByteArray
    }
  }
```

Better recovery with less matching

```
def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] =
  sendTo(mailServer.europe, packet) recoverWith {
    case europeError ⇒ sendTo(mailServer.usa, packet) recover {
      case usaError  ⇒ usaError.getMessage.toByteArray
    }
  }

def fallbackTo(that: ⇒Future[T]): Future[T] = {
  … if this future fails take the successful result
    of that future …
  … if that future fails too, take the error of
    this future …
}
```

Better recovery with less matching

```
def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] =
  sendTo(mailServer.europe, packet) fallbackTo {
    sendTo(mailServer.usa, packet)
  } recover {
    case europeError ⇒ europeError.getMessage.toByteArray
  }

def fallbackTo(that: ⇒Future[T]): Future[T] = {
  … if this future fails take the succcessful result
    of that future …
  … if that future fails too, take the error of
    this future …
}
```

Fallback implementation

```
def fallbackTo(that: ⇒Future[T]): Future[T] = {
  this recoverWith {
    case _ ⇒ that recoverWith { case _ ⇒ this }
  }
}
```

Quiz

```
object Try {
  def apply(f: Future[T]): Future[Try[T]] = { … }
}
```

```
(a) f onComplete { x ⇒ x }
(b) f recoverWith { case t ⇒ Future.failed(t) }
(c) f.map(x ⇒ Try(x))
(d) f.map(s⇒Success(s)) recover { case t ⇒ Failure(t) }
```

Asynchronous where possible, blocking where necessary

```
trait Awaitable[T] extends AnyRef {
  abstract def ready(atMost: Duration): Unit
  abstract def result(atMost: Duration): T
}

trait Future[T] extends Awaitable[T] {
   def filter(p: T⇒Boolean): Future[T]
   def flatMap[S](f: T⇒ Future[S]): Future[U]
   def map[S](f: T⇒S): Future[S]
   def recoverWith(f: PartialFunction[Throwable, Future[T]]): Future[T]
}
```

Asynchronous where possible, blocking where necessary

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()
val confirmation: Future[Array[Byte]] =
  packet.flatMap(socket.sendToSafe(_))

val c = Await.result(confirmation, 2 seconds)
println(c.toText)
```

Duration

```
import scala.language.postfixOps

object Duration {
  def apply(length: Long, unit: TimeUnit): Duration
}

val fiveYears = 1826 minutes
```