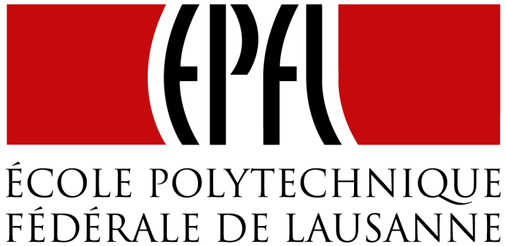


Latency as an Effect (2/2)

Principles of Reactive Programming

Erik Meijer

Monads guide you through the happy path

```
Future[T]
```

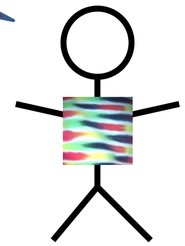
A monad that handles exceptions and **latency**.

Futures asynchronously notify consumers

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

trait Future[T] {
  def onComplete(callback: Try[T] ⇒ Unit)
    (implicit executor: ExecutionContext): Unit
}
```

We will totally ignore execution contexts

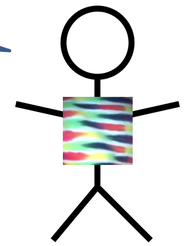


Futures asynchronously notify consumers

```
trait Future[T] {
   def onComplete(callback: Try[T] => Unit)
      (implicit executor: ExecutionContext): Unit
}
```

callback needs
to use pattern matching

```
ts match {
   case Success(t) =>
onNext(t)
   case Failure(e) =>
onError(e)
```

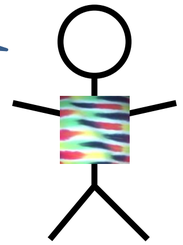


Futures asynchronously notify consumers

```
trait Future[T] {
  def onComplete(callback: Try[T] => Unit)
    (implicit executor: ExecutionContext): Unit
}
```

boilerplate code

```
ts match {
  case Success(t) =>
onNext(t)
  case Failure(e) =>
onError(e)
}
```



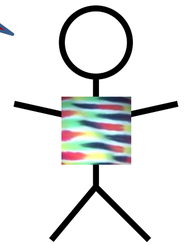
Futures alternative designs

```
trait Future[T] {
  def onComplete
      (success: T => Unit, failed: Throwable =>
Unit): Unit

  def onComplete(callback: Observer[T]): Unit
}

trait Observer[T] {
  def onNext(value: T): Unit
  def onError(error: Throwable): Unit
}
```

An *object* is a closure with multiple methods. A *closure* is an object with a single method.



Futures asynchronously notify consumers

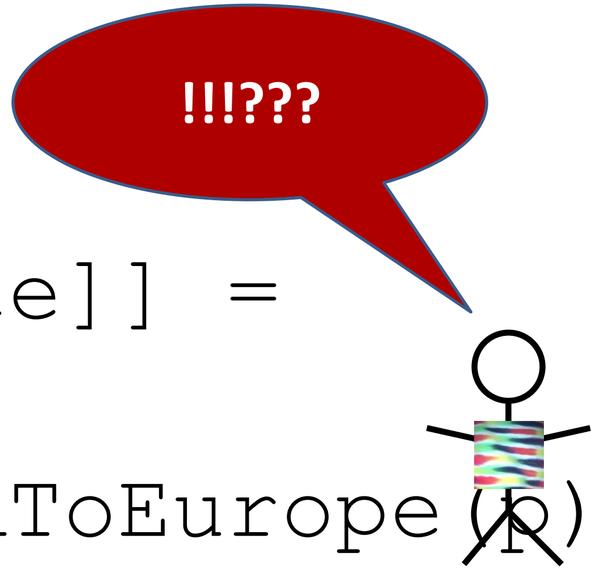
```
trait Future[T] {
  def onComplete(callback: Try[T] => Unit)
    (implicit executor: ExecutionContext): Unit
}

trait Socket {
  def readFromMemory(): Future[Array[Byte]]
  def sendToEurope(packet: Array[Byte]):
Future[Array[Byte]]
}
```

Send packets using futures I

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

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

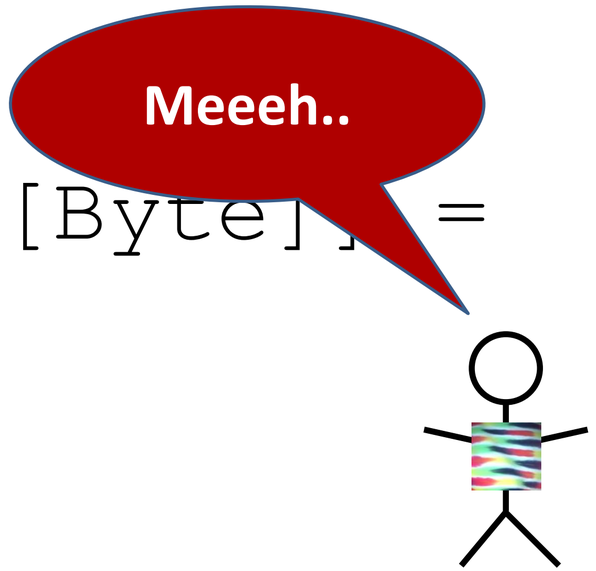


Send packets using futures II

```
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) => …
}
```



Creating Futures

```
// Starts an asynchronous computation
// and returns a future object to which you
// can subscribe to be notified when the
// future completes

object Future {
  def apply(body: =>T)
    (implicit context: ExecutionContext):
                                    Future[T]
}
```

Creating Futures

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.serializer._

val memory = Queue[EMailMessage](
  EMailMessage(from = “Erik”,   to = “Roland”),
  EMailMessage(from = “Martin”, to = “Erik”),
  EMailMessage(from = “Roland”, to = “Martin”))

def readFromMemory(): Future[Array[Byte]] = Future {
  val email = queue.dequeue()
  val serializer = serialization.findSerializerFor(email)
  serializer.toBinary(email)
}
```