ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Monads and Effects (1/2)

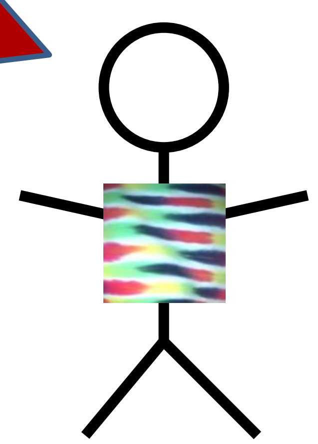
Principles of Reactive Programming

Erik Meijer

Warning

There is no type-checker for PowerPoint yet, hence these slides might contain typos and bugs. Hence, do not take these slides as the gospel or ultimate source of truth.

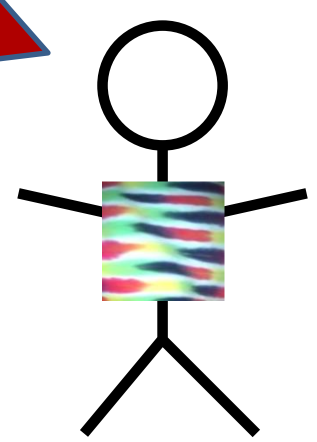
The only artifact you can trust is actual source code.



Warning

When we show code fragments in these lectures we really mean code *fragments*.

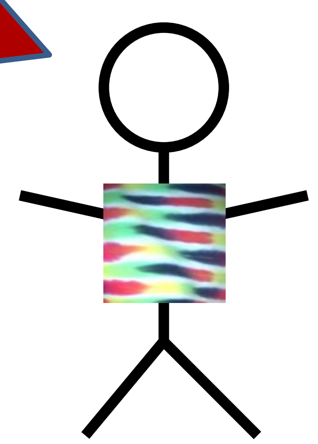
In particular, do not expect to be able to cut & past working code from the slides. You can find running & up-to-date on the GitHub site for this course.



Warning

When we use RxScala in these lectures, we assume version 0.23. Different versions of RxScala might not be compatible.

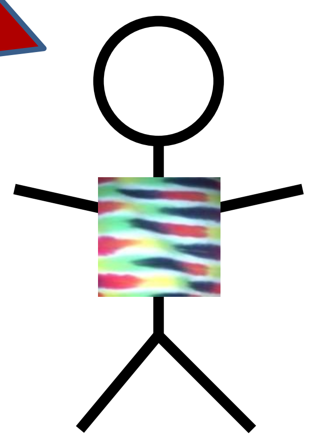
The RxScala method names do not necessarily correspond 1:1 with the underlying RxJava method names.



Warning

When we say "*monad*" in these lectures we mean a generic type with a constructor and a flatMap operator.

In particular, we'll be fast and loose about the monad laws (that is, we completely ignore them).



The Four Essential Effects In Programming

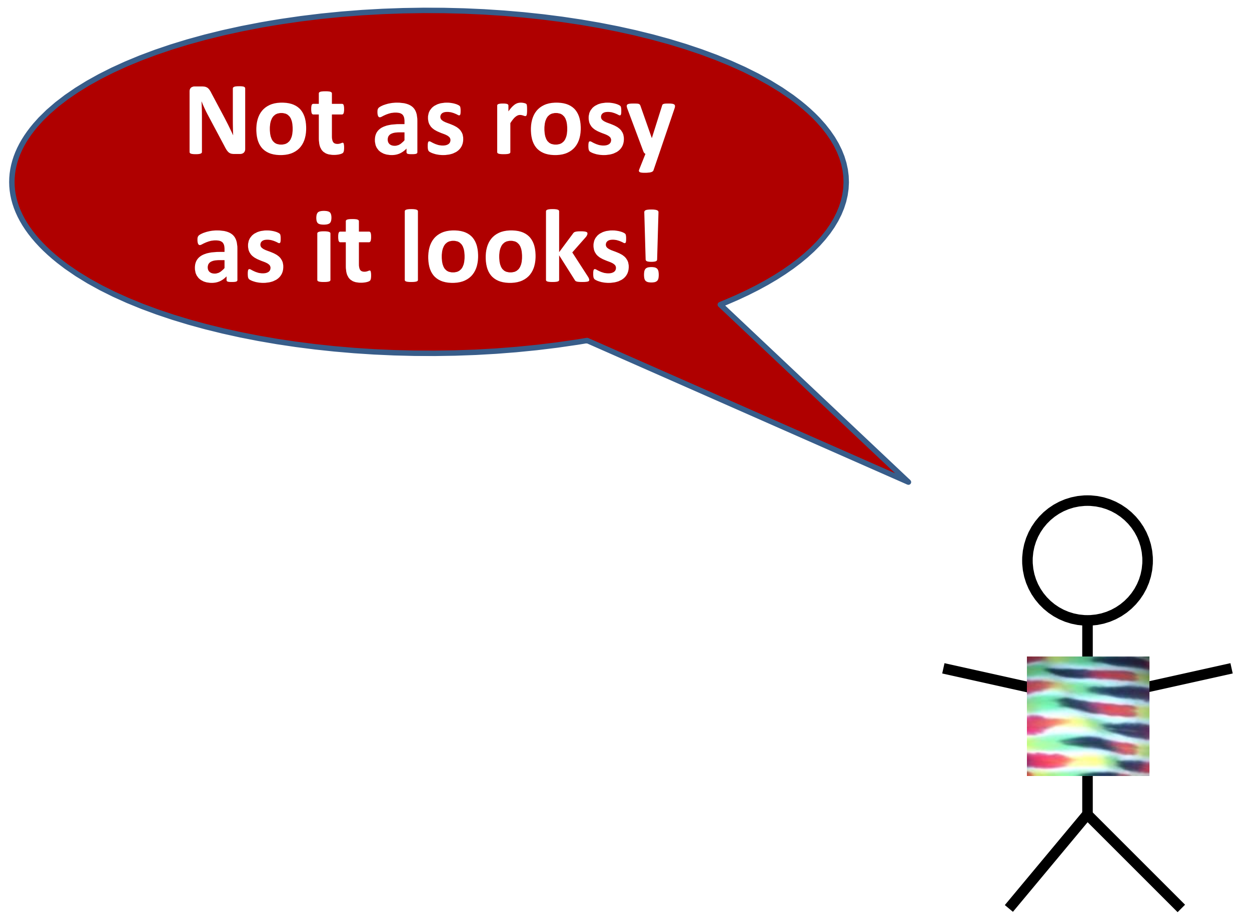
	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

The Four Essential Effects In Programming

	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

A simple adventure game

```
trait Adventure {
   def collectCoins(): List[Coin]
   def buyTreasure(coins: List[Coin]):
Treasure
}
```




```
val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

Actions may fail

```
def collectCoins(): List[Coin] = {
  if (eatenByMonster(this))
    throw new GameOverException(
"Ooops")
  List(Gold, Gold, Silver)
}

val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

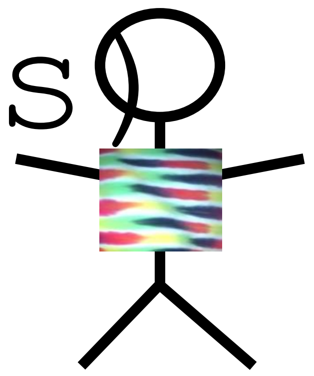
Actions may fail

```
def buyTreasure(coins: List[Coin]):
Treasure = {
   if (coins.sumBy(_.value) < treasureCost)
     throw new GameOverException("Nice try!")
   Diamond
}

val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

Sequential composition of actions that may fail

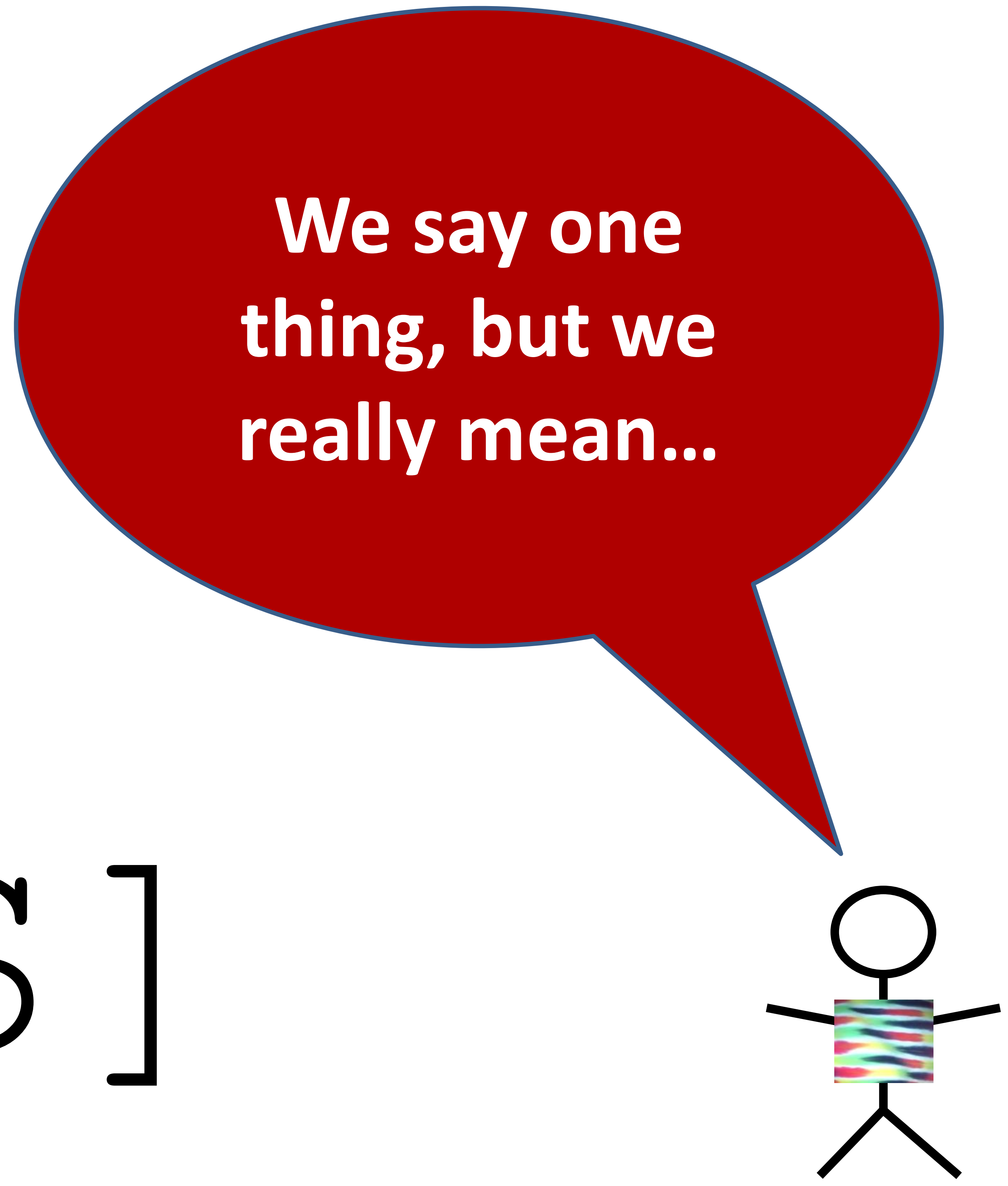
```
val adventure = Adventure()

val coins = adventure.collectCoi
// block until coins are collected
// only continue if there is no exception
val treasure = adventure.buyTreasure(coins)
// block until treasure is bought
// only continue if there is no exception
```

Lets make the happy path and the unhappy path explicit



Expose possibility of failure in the types, honestly

```
T => S

T => Try[S]
```



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Monads and Effects (1/2)

Principles of Reactive Programming

Erik Meijer

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Monads and Effects (2/2)

Principles of Reactive Programming

Erik Meijer

Making failure evident in types

```
abstract class Try[T]
case class Success[T](elem: T) extends Try[T]
case class Failure(t: Throwable)
                        extends Try[Nothing]

trait Adventure {
  def collectCoins(): Try[List[Coin]]
  def buyTreasure(coins: List[Coin]):
                              Try[Treasure]
}
```

Dealing with failure explicitly

```
val adventure = Adventure()

val coins: Try[List[Coin]] =
             adventure.collectCoins()

val treasure: Try[Treasure] = coins match {
  case Success(cs)          =>
             adventure.buyTreasure(cs)
  case failure@Failure(e) => failure
}
```

Higher-order Functions to manipulate Try[T]

```
def flatMap[S](f: T=>Try[S]): Try[S]

def flatten[U <: Try[T]]: Try[U]

def map[S](f: T=>S): Try[T]

def filter(p: T=>Boolean): Try[T]

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

`Try[T]`

A monad that handles **exceptions**.

Noise reduction

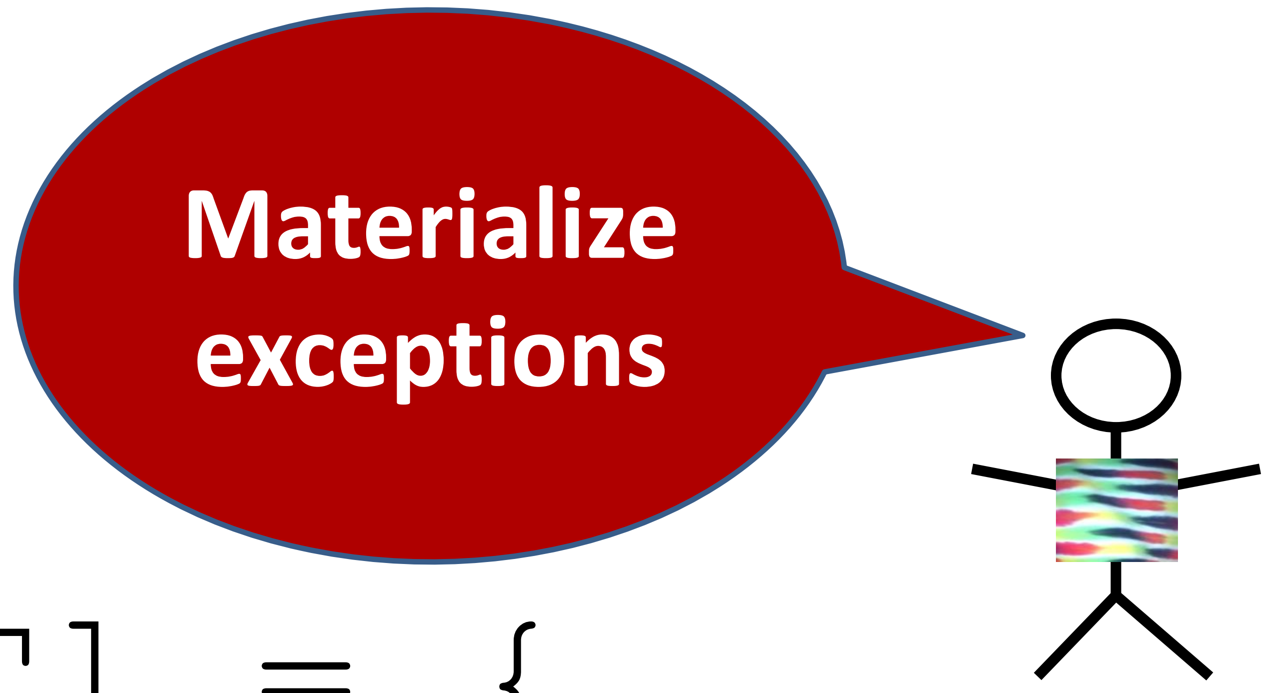
```
val adventure = Adventure()

val treasure: Try[Treasure] =
  adventure.collectCoins().flatMap(
    coins ⇒ {
      adventure.buyTreasure(coins)
  })
```

Using comprehension syntax

```
val adventure = Adventure()

val treasure: Try[Treasure] = for {
  coins    <- adventure.collectCoins()
  treasure <- buyTreasure(coins)
} yield treasure
```

Higher-order Function to manipulate Try[T]

```
def map[S](f: T=>S): Try[S] = this match{
  case Success(value)        => Try(f(value))
  case failure@Failure(t) => failure
}

object Try {
  def apply[T](r: =>T): Try[T] = {
    try { Success(r) }
    catch { case t => Failure(t) }
}
```



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Monads and Effects (2/2)

Principles of Reactive Programming

Erik Meijer

Latency as an Effect (1/2)

Principles of Reactive Programming

Erik Meijer

The Four Essential Effects In Programming

	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

The Four Essential Effects In Programming

	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

Recall our simple adventure game ….

```
trait Adventure {
  def collectCoins(): List[Coin]
  def buyTreasure(coins: List[Coin]):Treasure
}


val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

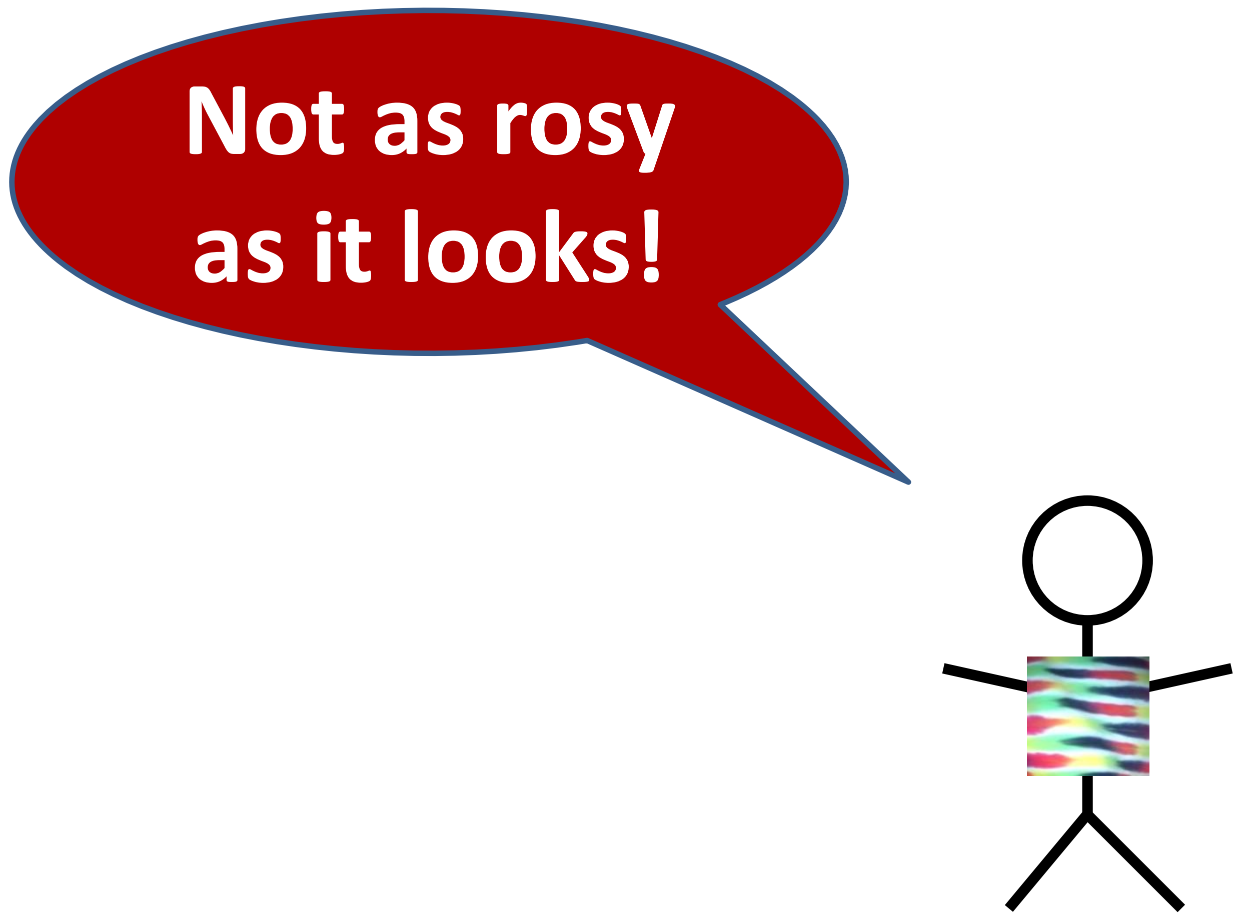
Recall our simple adventure game ….

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

val socket = Socket()
val packet = socket.readFromMemory()
val confirmation =
    socket.sendToEurope(packet)
```

```
trait Adventure {
  def collectCoins(): List[Coin]
  def buyTreasure(coins: List[Coin]): Treasure
}

val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

It is actually very similar to a simple network stack

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


val socket = Socket()
val packet = socket.readFromMemory()
val confirmation = socket.sendToEurope(packet)
```



Timings for various operations on a typical PC

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

http://norvig.com/21-days.html#answers

Sequential composition of actions that take time

```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 50,000 ns
// only continue if there is no exception
val confirmation = socket.sendToEurope(packet)
// block for 150,000,000 ns
// only continue if there is no exception
```

Sequential composition of actions

Lets translate this into human terms.

1 nanosecond

→

1 second (then hours/days/months/years)

Timings for various operations on a typical PC on human scale

execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

Sequential composition of actions

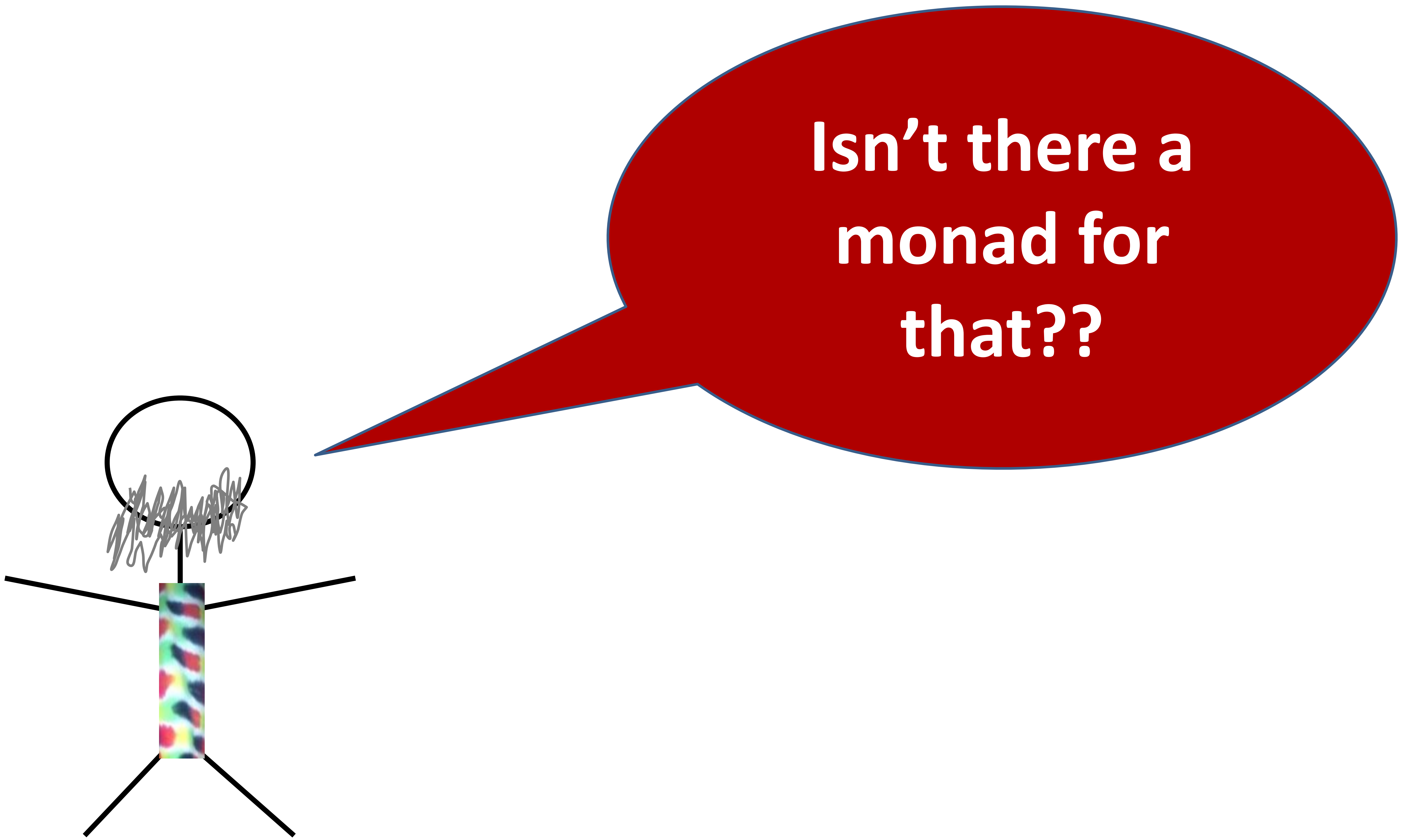
```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 3 days
// only continue if there is no exception
val confirmation = socket.sendToEurope(packet)
// block for 5 years
// only continue if there is no exception
```

Sequential composition of actions

12 months to walk coast-to-coast
3 months to swim across the Atlantic
3 months to swim back
12 months to walk back

Humans are twice as fast as computers!

Sequential composition of actions that take time and fail



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Latency as an Effect (1/2)

Principles of Reactive Programming

Erik Meijer

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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

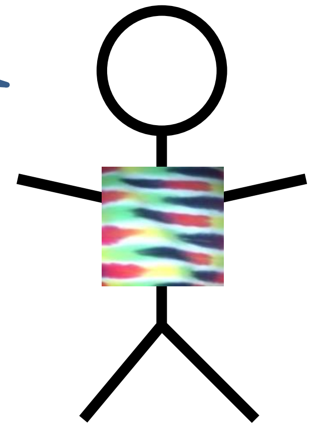
```
import scala.concurrent._
import
scala.concurrent.ExecutionContext.Implicits.global

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

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)
```

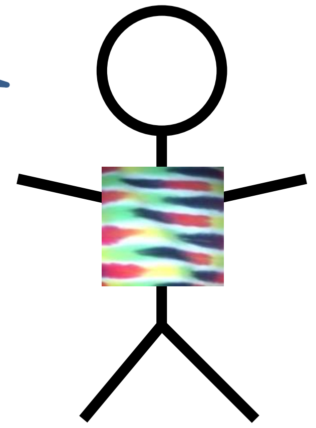


Futures asynchronously notify consumers

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

boilerplate code

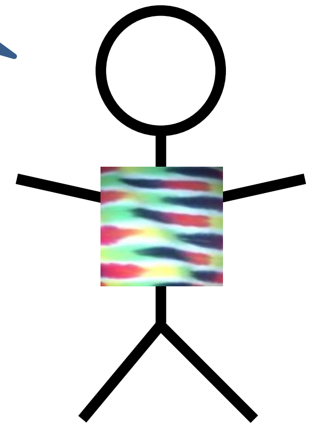
```
ts match {
    case Success(t) =>
onNext(t)
    case Failure(e) =>
onError(e)
}
```



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.

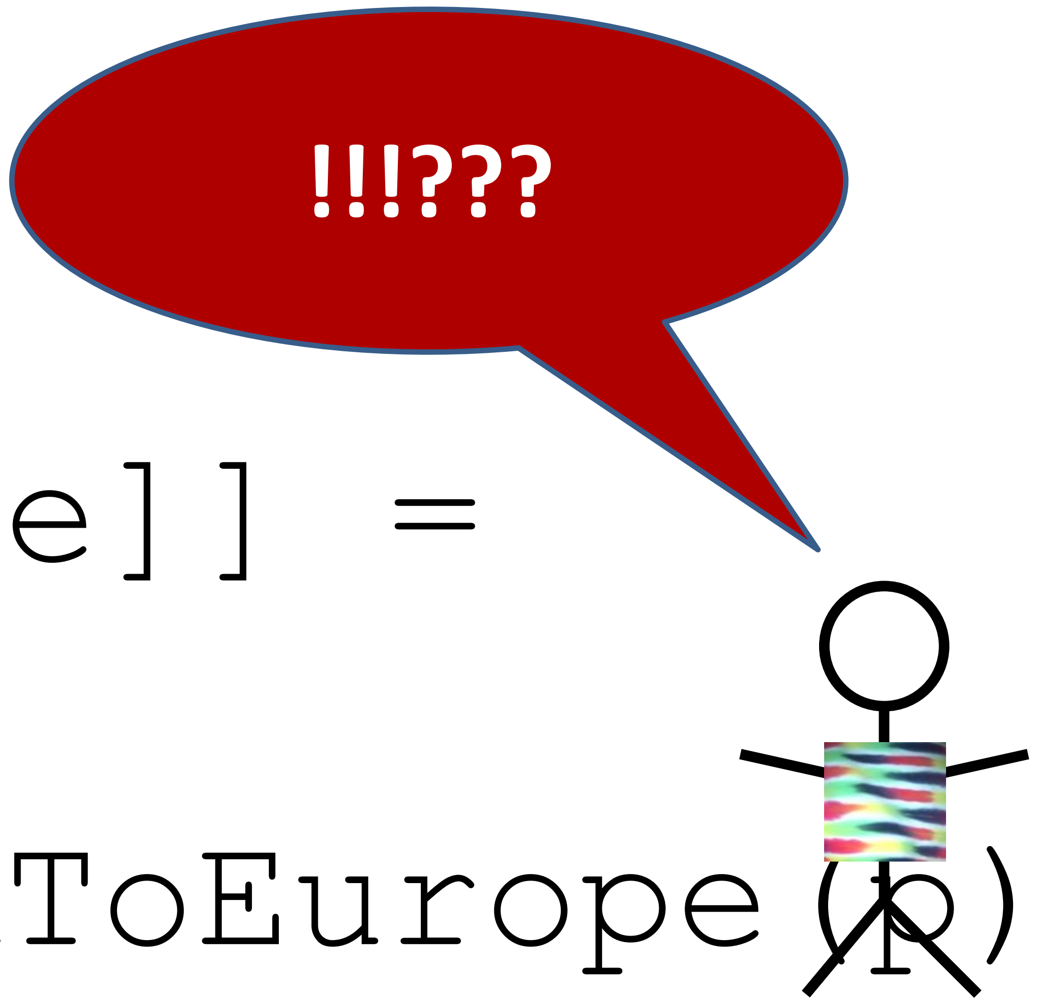


Futures asynchronously notify consumers

```
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]]
}
```

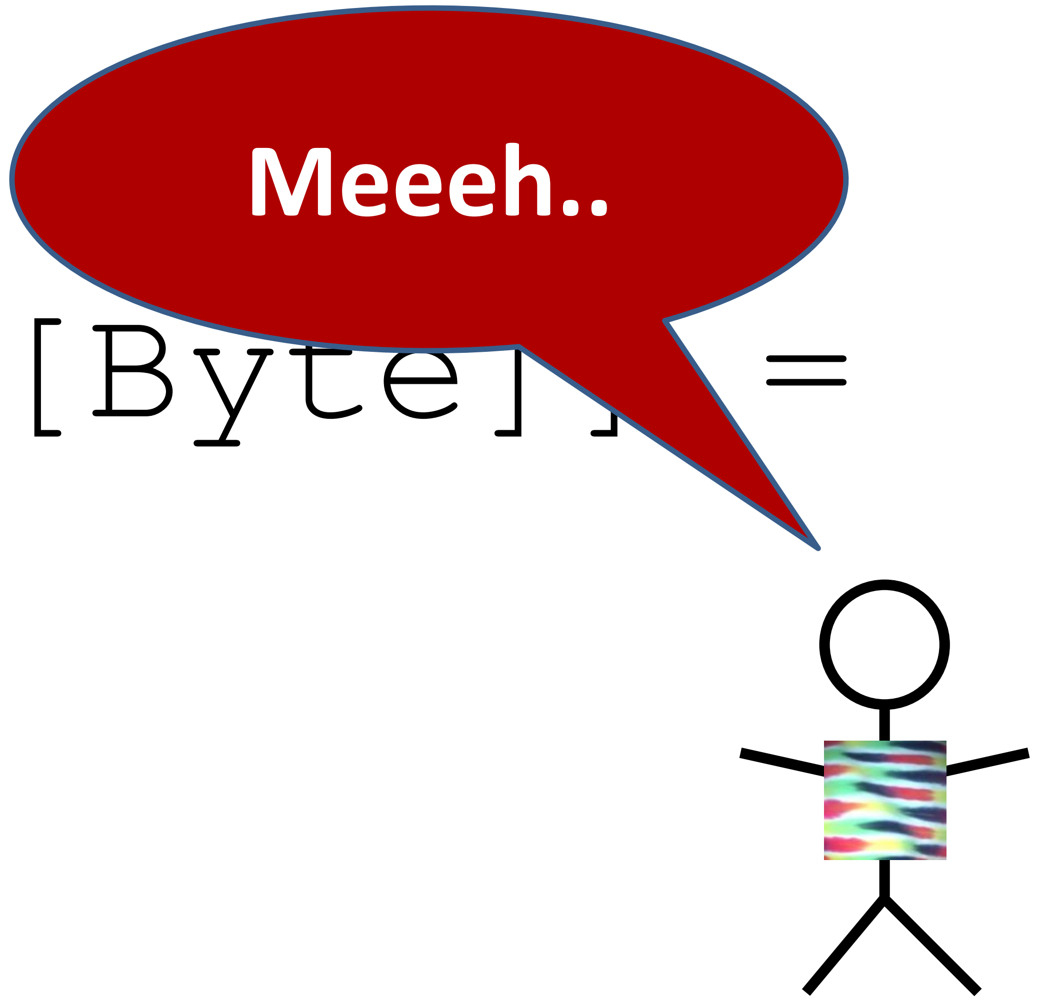
Send packets using futures I

```scala
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

```scala
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)
}
```

Combinators on Futures (1/2)

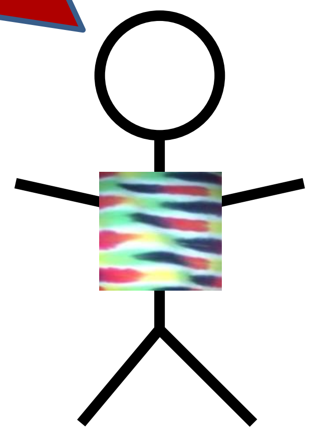
Principles of Reactive Programming

Erik Meijer

Futures recap

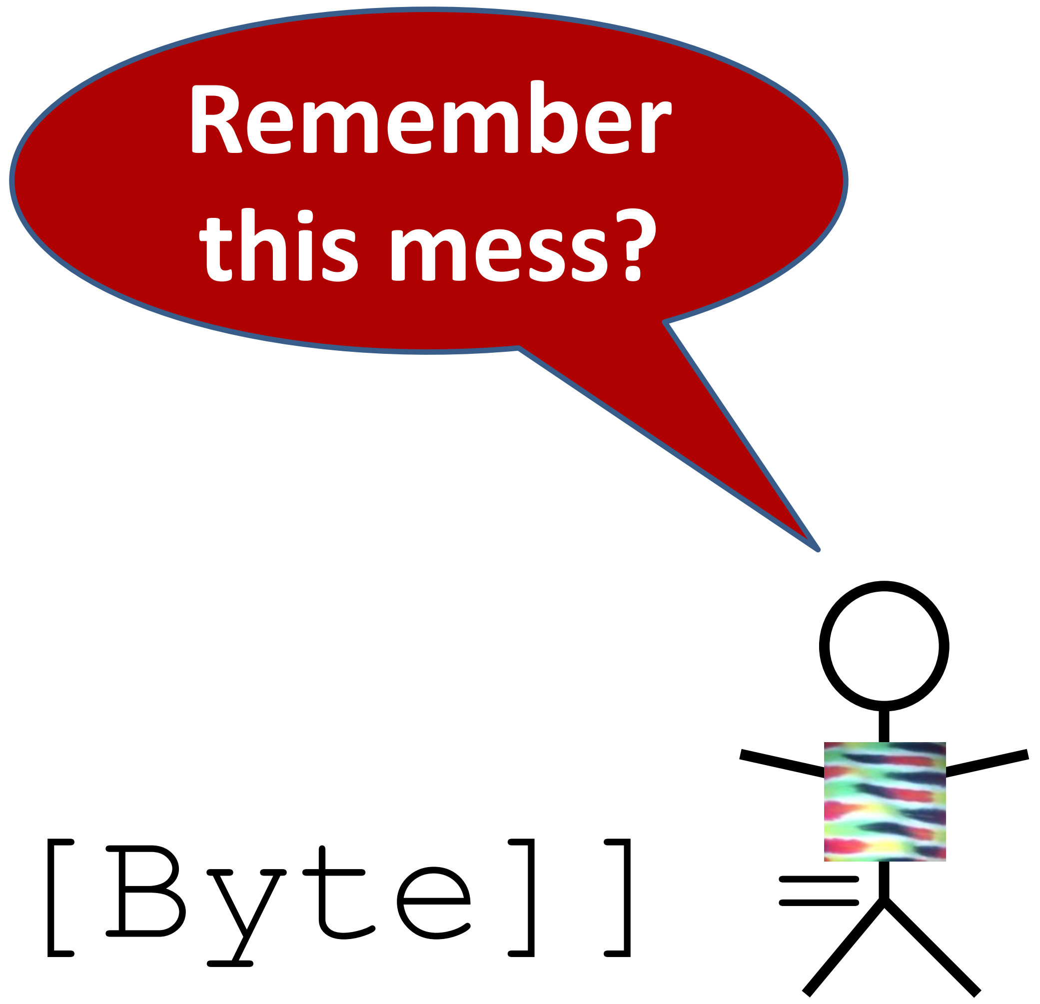
```
trait Awaitable[T] extends AnyRef {
  abstract def ready(atMost: Duration):
  abstract def result(atMost: Duration
}

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]
}
```

All these methods take an implicit execution context



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) ⇒ …
}
```

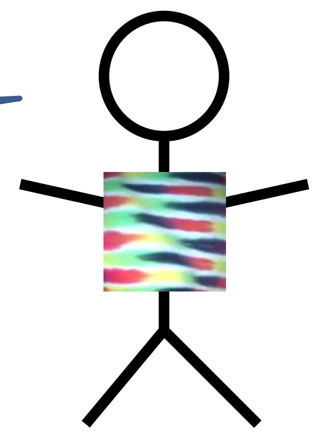


Flatmap to the rescue

```
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

```scala
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.toByte
```

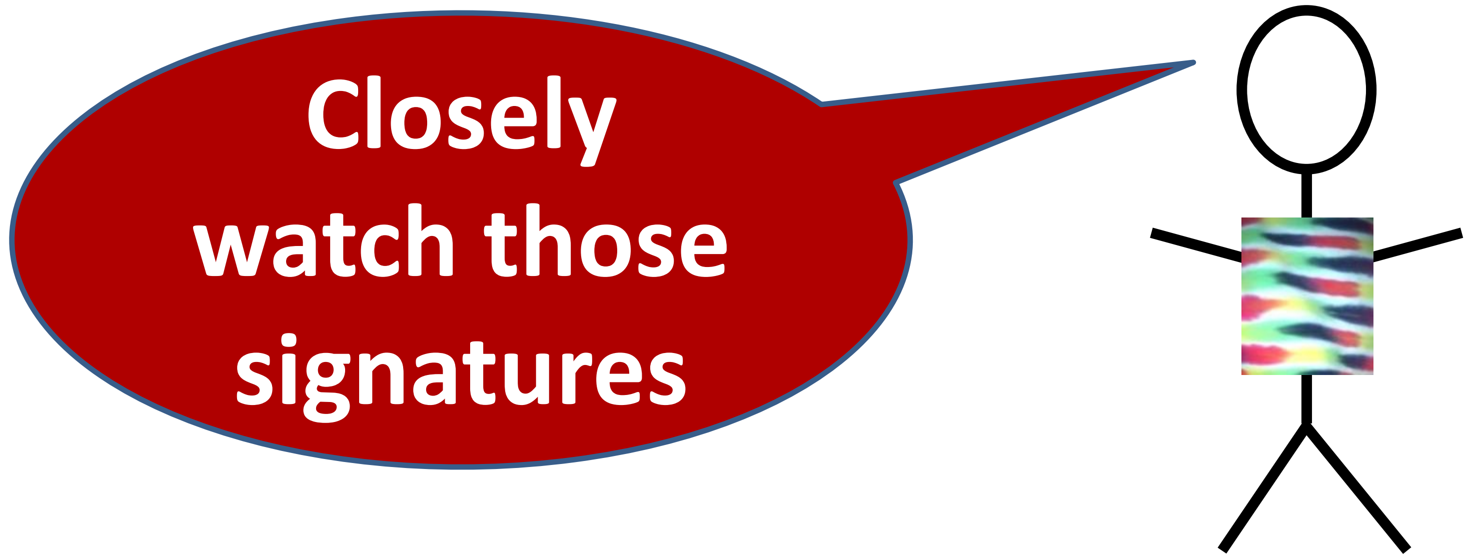


Sending packets using futures robustly (?)

```
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
    }
  }
```

End of Combinators on Futures (1/2)

Principles of Reactive Programming

Erik Meijer

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Combinators on Futures (2/2)

Principles of Reactive Programming

Erik Meijer

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 }
  }
}
```

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

```scala
import scala.language.postfixOps

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


val fiveYears = 1826 minutes
```

End of Combinators on Futures (2/2)

Principles of Reactive Programming

Erik Meijer

Composing Futures (1/2)

Principles of Reactive Programming

Erik Meijer

Flatmap …

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

Or comprehensions?

```
val socket = Socket()
val confirmation: Future[Array[Byte]] = for{
  packet       <- socket.readFromMemory()
  confirmation <- socket.sendToSafe(packet)
} yield confirmation
```

Retrying to send

```
def retry(noTimes: Int)(block: =>Future[T]):
Future[T] = {
    … retry successfully completing block
      at most noTimes
    … and give up after that
}
```

Retrying to send

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
  if (noTimes == 0) {
      Future.failed(new Exception(“Sorry”))
  } else {
      block fallbackTo {
          retry(noTimes-1){ block }
      }
  }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Composing Futures (1/2)

Principles of Reactive Programming

Erik Meijer

End of Composing Futures (1/2)

Principles of Reactive Programming

Erik Meijer

Composing Futures (2/2)

Principles of Reactive Programming

Erik Meijer

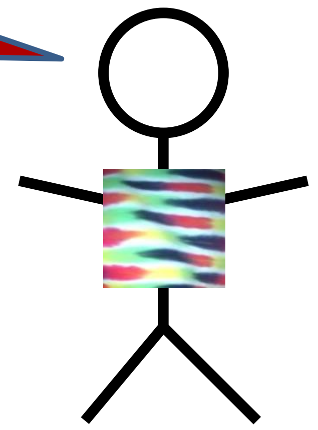
Avoid Recursion


```
foldRight
foldLeft
```

Folding lists

```
List(a,b,c).foldRight(e)(f)
=
f(a, f(b, f(c, e)))
```

Northern wind comes from the North (Richard Bird)



```
List(a,b,c).foldLeft(e)(f)
=
f(f(f(e, a), b), c)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: =>Future[T]):
Future[T] = {
  val ns = (1 to noTimes).toList
  val attempts = ns.map(_ => ()=>block)
  val failed = Future.failed(new Exception("boom"))
  val result = attempts.foldLeft(failed)
     ((a,block) => a recoverWith { block() })
  result
}                retry(3) { block }
                 = unfolds to
                 ((failed recoverWith {block1()})
                    recoverWith {block2()})
                      recoverWith { block3 ()}
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
   …
   val attempts = ns.map(_=> ()=>block)
   …
}

 ns =       List(1,         2,          …, noTimes)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
   …
   val attempts = ns.map(_=> ()=>block)
   …
}

 ns =       List(1,         2,         …, noTimes)
 attemps = List(()=>block, ()=>block, …, ()=>block)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
  …
  val result = attempts.foldLeft(failed)
     ((a,block) => a recoverWith { block() })
  result
}
```

ns = List(1, 2, …, noTimes)

attemps = List($()$=>block$_1$, $()$=>block$_2$, …, $()$=>block$_{noTimes}$)

result = (…((failed recoverWith { block$_1$() })

Retrying to send using foldRight

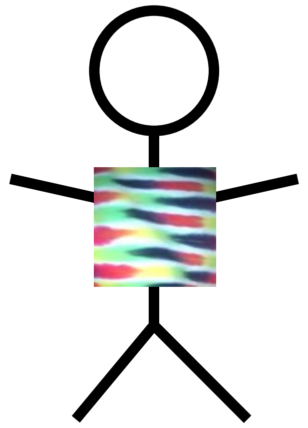
```
def retry(noTimes: Int)(block: =>Future[T])= {
  val ns = (1 to noTimes).toList
  val attempts: = ns.map(_ => () => block)
  val failed = Future.failed(new Exception)
  val result = attempts.foldRight(() =>failed)
    ((block, a) => () => { block() fallbackTo { a()
  result ()
}
```

```
retry(3) { block } ()
= unfolds to
```

$\texttt{block}_1$ `fallbackTo {` $\texttt{block}_2$ `fallbackTo {` $\texttt{block}_3$ `fallbackTo { failed }}}`

Use Recursion

Often, straight recursion is the way to go



And just leave the HO functions to the FP hipsters!

`foldRight`

`foldLeft`

End of Composing Futures (2/2)

Principles of Reactive Programming

Erik Meijer

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Async await

Principles of Reactive Programming

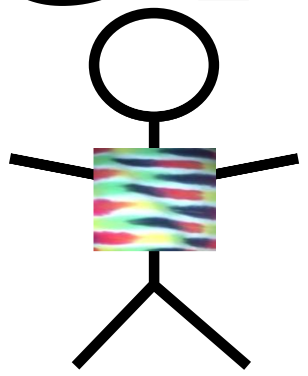
Erik Meijer

Making effects implicit

```
T => Future[S]
```

We say one thing, but we really want…

```
T => Try[S] or even
T => S
```



Async await magic

```
import scala.async.Async._

def async[T](body: =>T)
(implicit context: ExecutionContext)
:Future[T]
def await[T](future: Future[T]): T
```

```
async{ … await{…}…}
```

Async, the small print

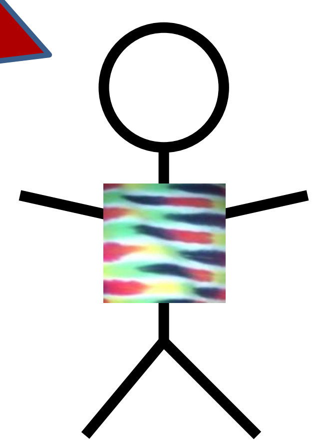
Illegal Uses

The following uses of await are illegal and are reported as errors:

- await requires a directly-enclosing async; this means await must not be used inside a closure nested within an async block, or inside a nested object, trait, or class.
- await must not be used inside an expression passed as an argument to a by-name parameter.
- await must not be used inside a Boolean short-circuit argument.
- return expressions are illegal inside an async block.
- **await should not be used under a try/catch**.

Warning

Getting async await to work,
Dealing with the compiler error messages,
Navigating the limitations,
…
can be frustrating.
But ultimately, it will pay off!



Retrying to send using await (an no recursion)

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] =
async {
  var i = 0
  var result: Try[T] = Failure(new Exception("…"))
  while (result.isFailure && i < noTimes) {
    result = await { Try(block) }
    i += 1
  }
  result.get
}
```

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

Reimplementing filter using await

```
def filter(p: T ⇒ Boolean): Future[T] = async {
  val x = await { this }
  if (!p(x)) {
    throw new NoSuchElementException()
  } else {
    x
  }
}
```

Reimplementing flatMap using await

```
def flatMap[S](f: T ⇒ Future[S]): Future[S] = async {
  val x: T = await { this }
  await { f(x) }
}
```

Reimplementing filter without await

```
def filter(pred: T ⇒ Boolean): Future[T] = {
  val p = Promise[T]()

  this onComplete {
    case Failure(e) ⇒
      p.failure(e)
    case Success(x) ⇒
      if (!pred(x)) p.failure(new NoSuchElementException)
      else p.success(x)
  }

  p.future
}
```

End of Async await

Principles of Reactive Programming

Erik Meijer

Promises, promises, promises

Principles of Reactive Programming

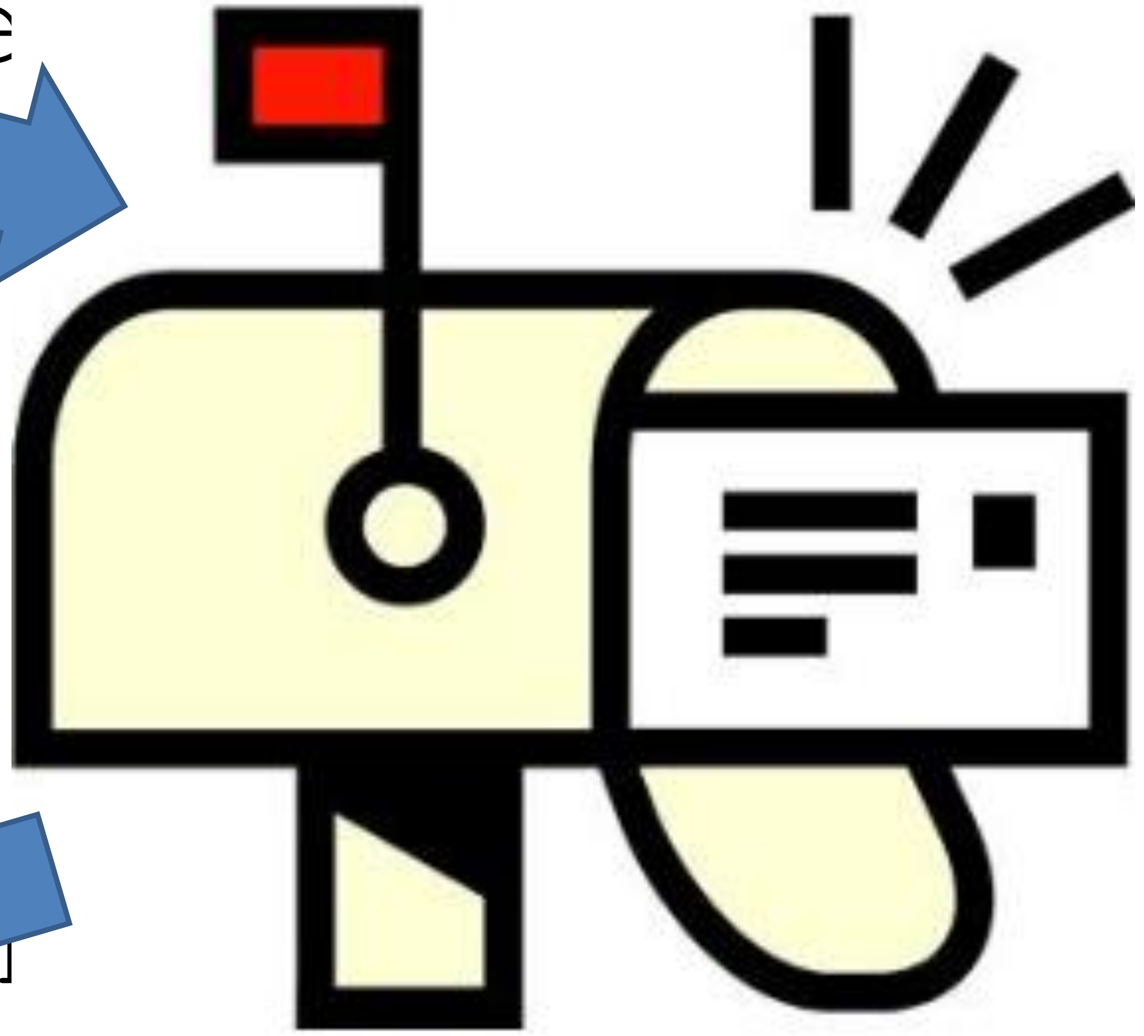
Erik Meijer

Reimplementing filter without await

```
def filter(pred: T ⇒ Boolean): Future[T] = {
  val p = Promise[T]()

  this onComplete {
    case Failure(e) ⇒
      p.failure(e)
    case Success(x) ⇒
      if (!pred(x)) p.failure(new NoSuchElementException)
      else p.success(x)
  }

  p.future
}
```

Promises

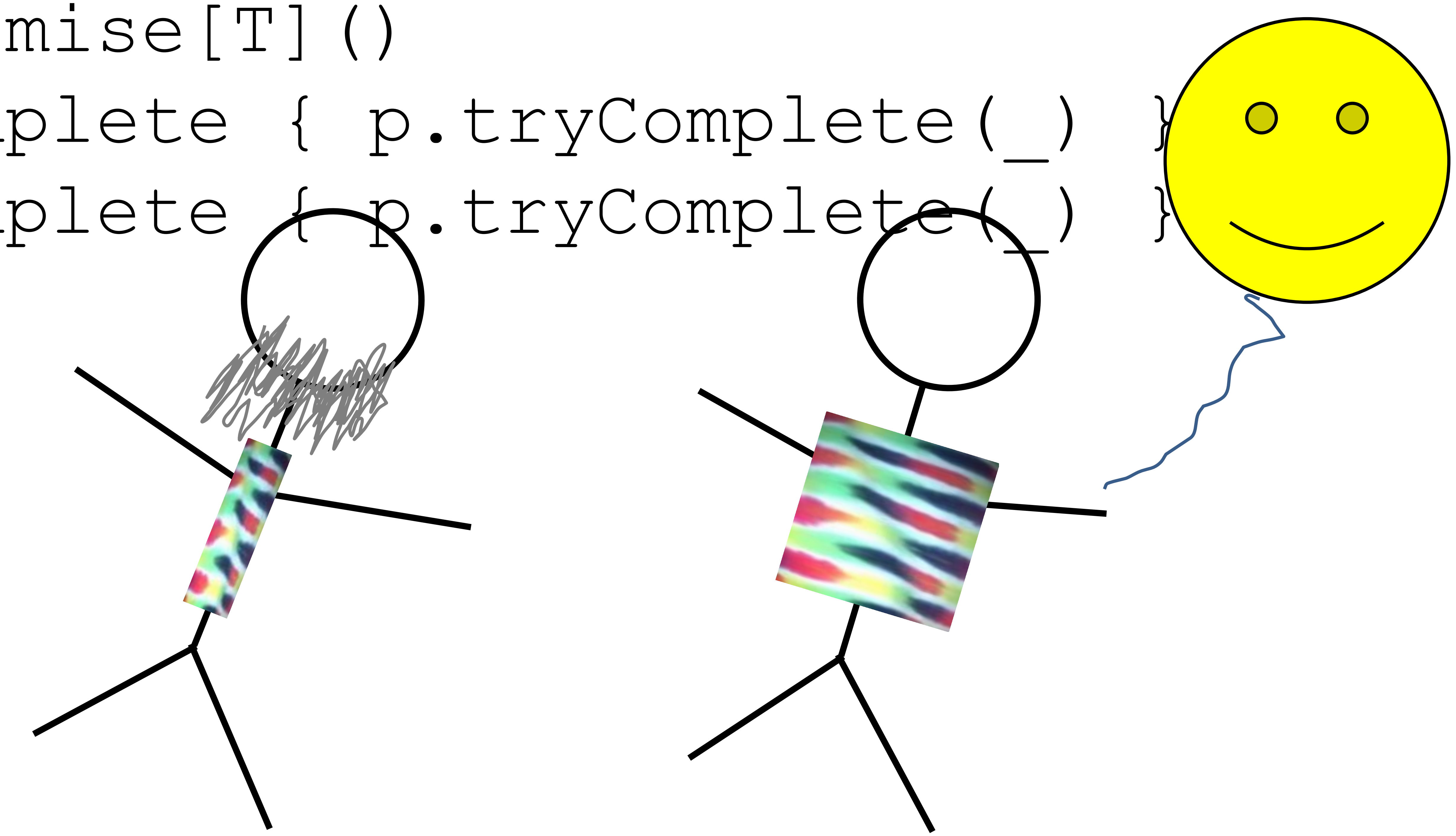
```
trait Promise[T] {
  def future: Future[T]
  def complete(result: Try[T]): Unit
  def tryComplete(result: Try[T]): Boole
}
```



```
trait Future[T] {
  def onCompleted(f: Try[T] => Unit)
}
```

Racing

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

def race[T](left: Future[T], right: Future[T]):
Future[T] = {
  val p = Promise[T]()
  left  onComplete { p.tryComplete(_) }
  right onComplete { p.tryComplete(_) }
  p.future
}
```



Simple helper methods

```
def success(value: T): Unit =
   this.complete(Success(value))

def failure(t: Throwable): Unit =
   this.complete(Failure(t))
```

Reimplementing zip using Promises

```
def zip[S, R](p: Future[S], f: (T, S) => R): Future[R] = {
  val p = Promise[R]()

  this onComplete {
    case Failure(e) ⇒ p.failure(e)
    case Success(x) ⇒ that onComplete {
      case Failure(e) ⇒ p.failure(e)
      case Success(y) ⇒ p.success(f(x, y))s
    }
  }

  p.future
}
```

Reimplementing zip with await

```
def zip[S, R](p: Future[S], f: (T, S) => R): Future[R] =
async {
  f(await { this }, await { that })
}
```

Implementing sequence

```
def sequence[T](fts: List[Future[T]]): Future[List[T]] = {
  fts match {
    case Nil => Future(Nil)
    case (ft::fts)  => ft.flatMap(t => sequence(fts)
                          .flatMap(ts => Future(t::ts)))
  }
}
```

Implementing sequence with await

```
def sequence[T](fs: List[Future[T]]): Future[List[T]] =
async {
  var _fs = fs
  val r = ListBuffer[T]()
  while (_fs != Nil) {
    r += await { _fs.head }
    _fs = _fs.tail
  }
  r.toList
}
```

Implement sequence with Promise

```
def sequence[T](fs: List[Future[T]]): Future[List[T]] = {
  val p = Promise[List[T]]()
  ???
  p.future
}
```

The Four Essential Effects In Programming

	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

End of Promises, promises, promises

Principles of Reactive Programming

Erik Meijer