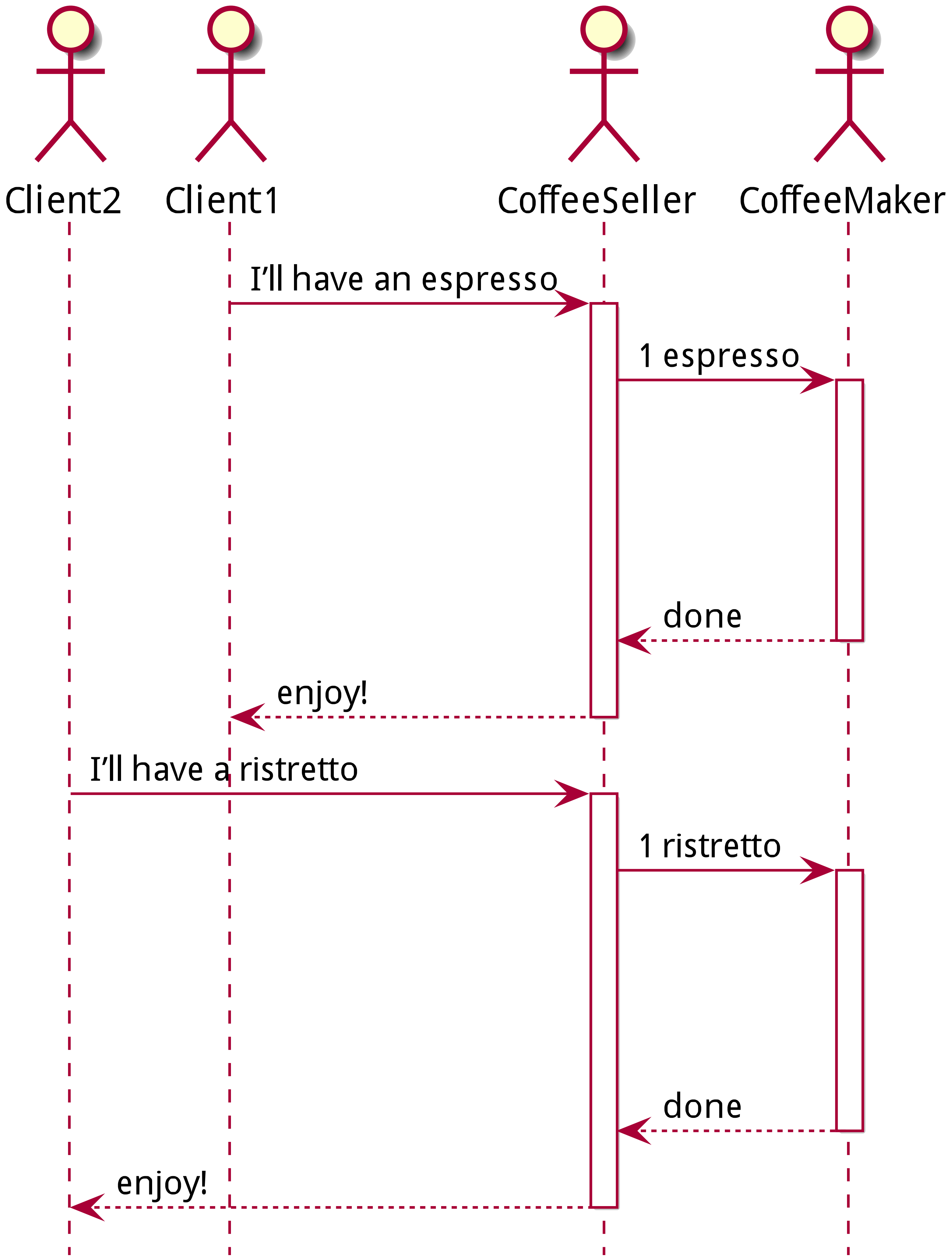
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Asynchronous Programming

Programming Reactive Systems

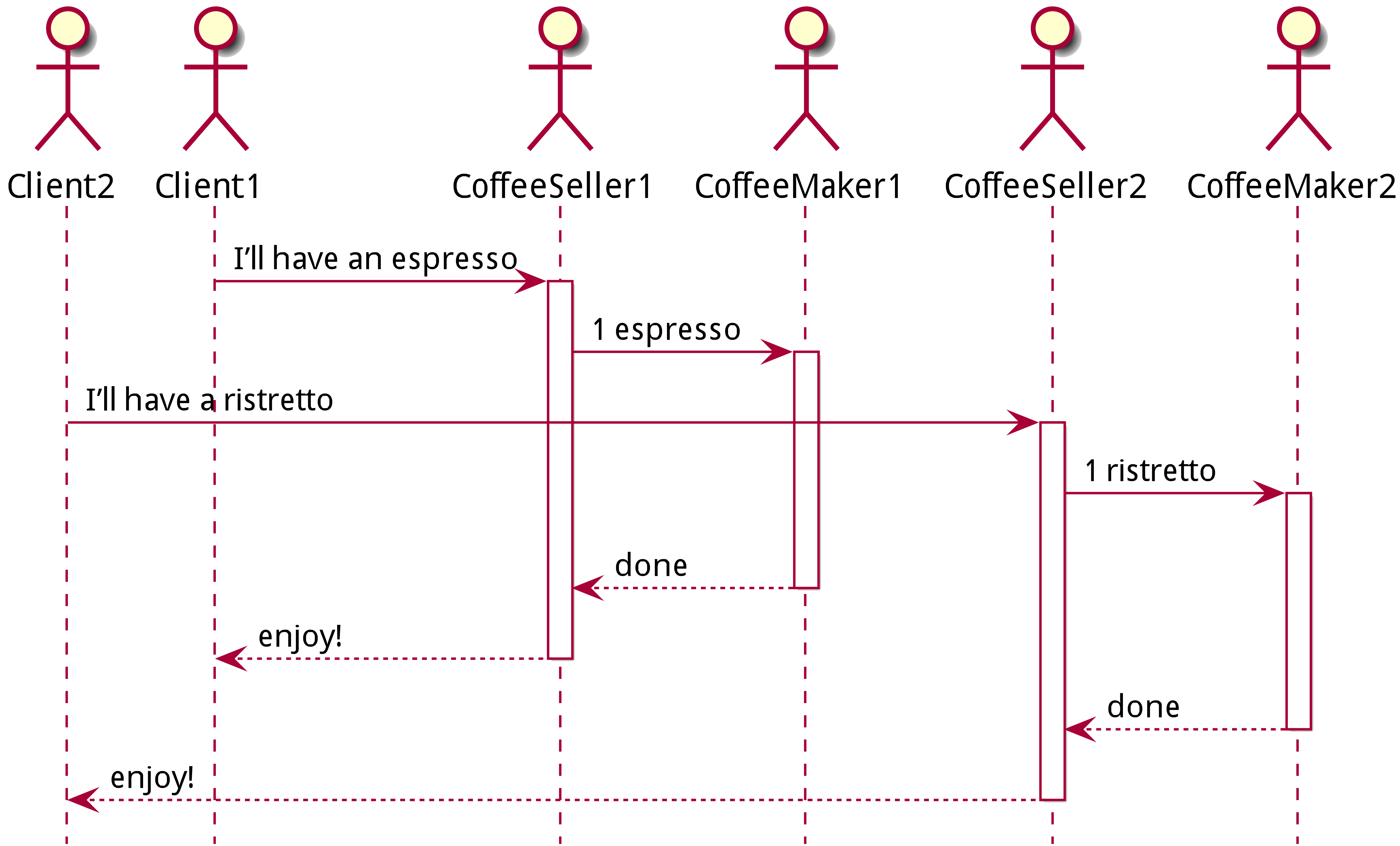
Julien Richard-Foy

StarBlocks

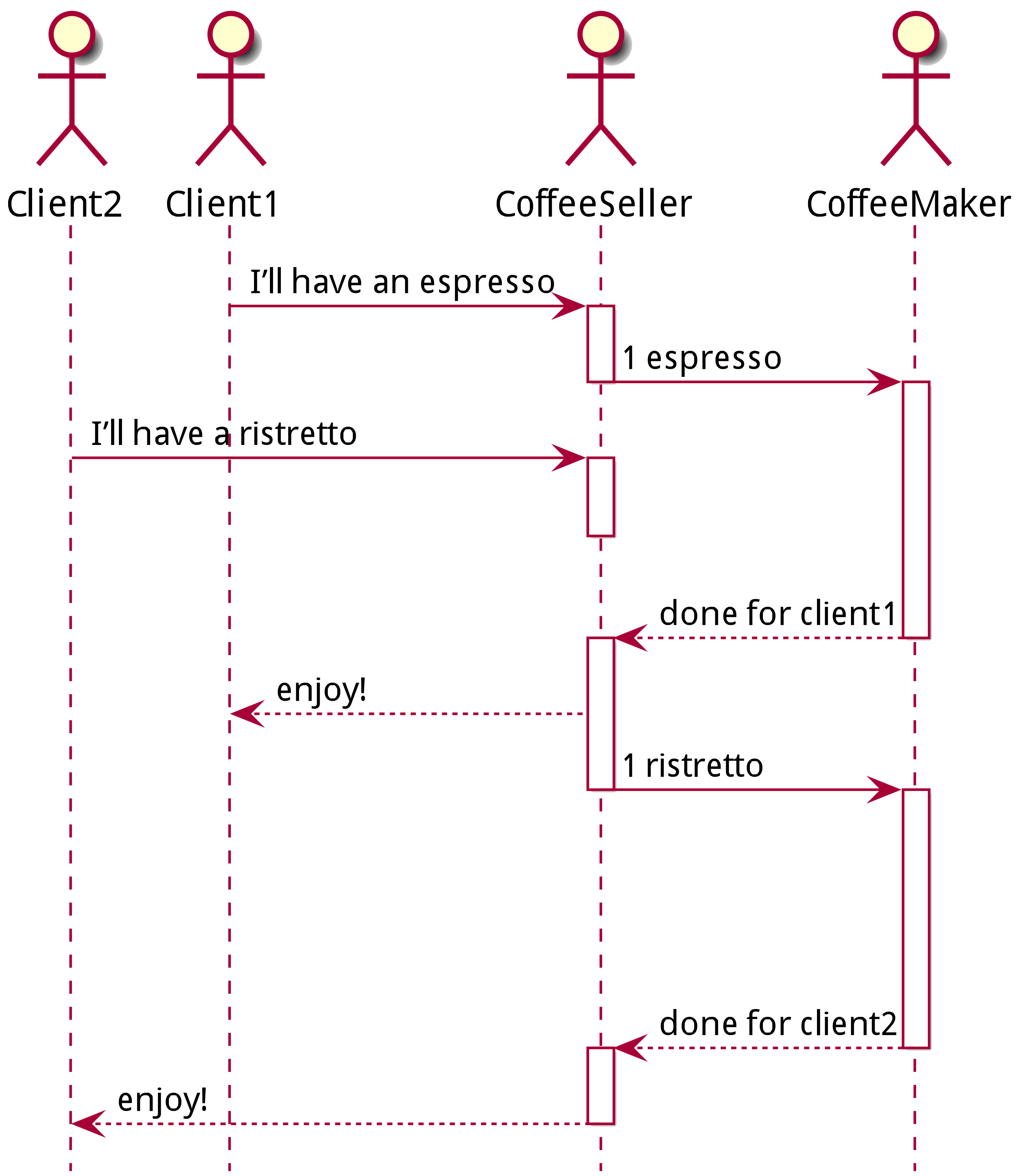


Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
done
enjoy!
I'll have a ristretto
1 ristretto
done
enjoy!

StarBlocks Scaled

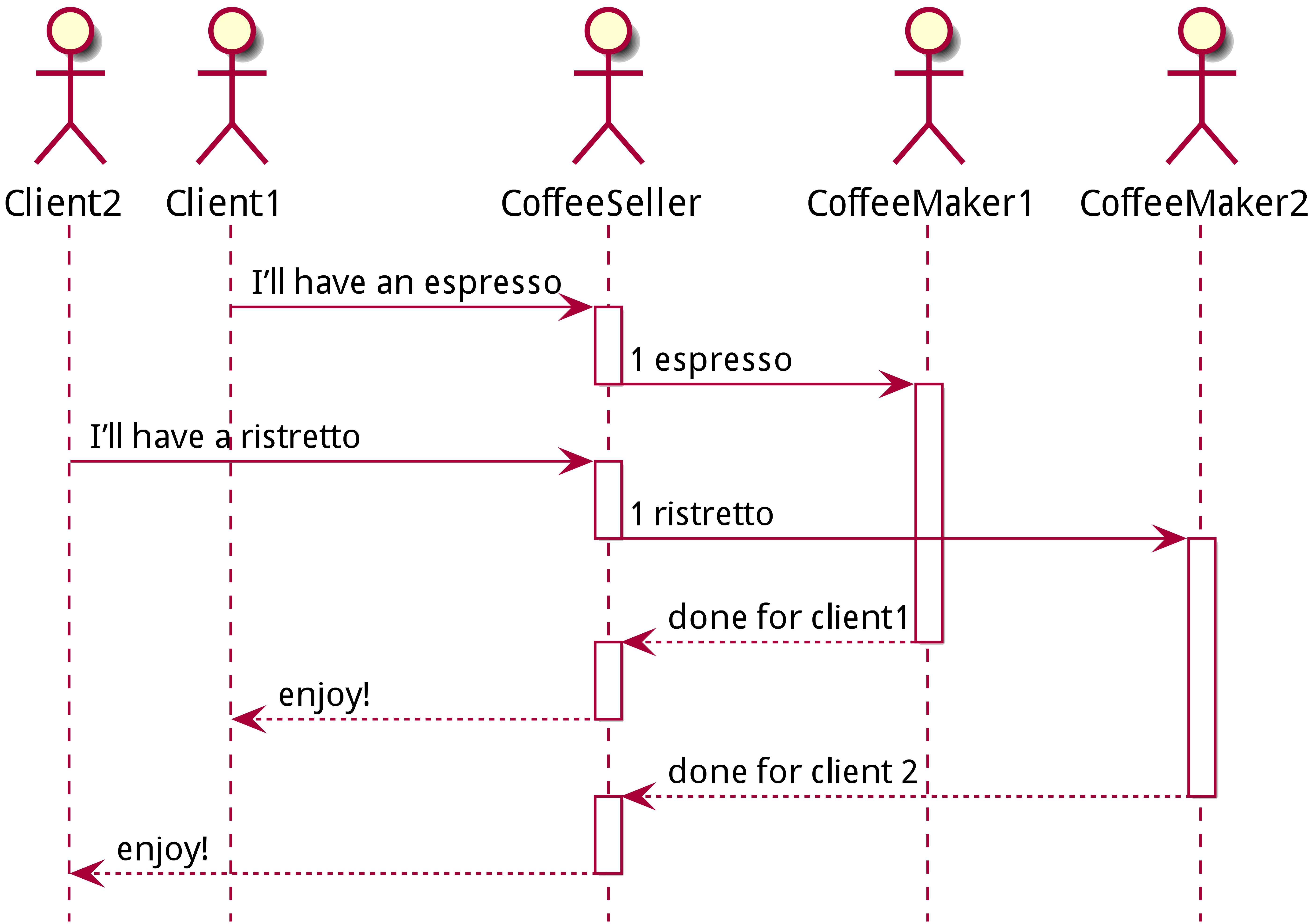


ScalaBucks



Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
I'll have a ristretto
done for client1
enjoy!
1 ristretto
done for client2
enjoy!

ScalaBucks Scaled



Asynchronous Execution

- Execution of a computation on *another* computing unit, without *waiting* for its termination ;
- Better resource efficiency.

Concurrency Control of Asynchronous Programs

What if a program `A` *depends on* the result of an asynchronously executed program `B`?

```
def coffeeBreak(): Unit = {
  val coffee = makeCoffee()
  drink(coffee)
  chatWithColleagues()
}
```

Callback

```scala
def makeCoffee(coffeeDone: Coffee => Unit): Unit = {
  // work hard ...
  // ... and eventually
  val coffee = ...
  coffeeDone(coffee)
}

def coffeeBreak(): Unit = {
  makeCoffee { coffee =>
   drink(coffee)
  }
  chatWithColleagues()
}
```

From Synchronous to Asynchronous Type Signatures

A synchronous type signature can be turned into an asynchronous type signature by:

- returning `Unit`
- and taking as parameter a **continuation** defining what to do after the return value has been computed

```
def program(a: A): B

def program(a: A, k: B => Unit): Unit
```

Combining Asynchronous Programs (1)

```scala
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit = ???
```

Combining Asynchronous Programs (2)

```scala
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit = {
  var firstCoffee: Option[Coffee] = None
  val k = { coffee: Coffee =>
    firstCoffee match {
      case None          => firstCoffee = Some(coffee)
      case Some(coffee2) => coffeesDone(coffee, coffee2)
    }
  }
  makeCoffee(k)
  makeCoffee(k)
}
```

Callbacks All the Way Down (1)

What if another program *depends on* the coffee break to be done?

```scala
def coffeeBreak(): Unit = ...
```

- We need to make `coffeeBreak` take a callback too!

Callbacks all the Way Down (2)

```
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit = {
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
}
```

Callbacks all the Way Down (2)

```scala
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit = {
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
}
```

- Order of execution follows the indentation level!

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
  - We need a way to propagate the failure to the call site

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
  - We need a way to propagate the failure to the call site

```
def makeCoffee(coffeeDone: Try[Coffee] => Unit): Unit = ...
```

Summary

In this video, we have seen:

- How to *sequence* asynchronous computations using **callbacks**
- Callbacks introduce complex type signatures
- The continuation passing style is tedious to use

Asynchronous Programming with `Future`

Programming Reactive Systems

Julien Richard-Foy

From Synchronous to Asynchronous Type Signatures (using Future)

Remember the transformation we applied to a synchronous type signature to make it asynchronous:

```
def program(a: A): B
```

```
def program(a: A, k: B => Unit): Unit
```

From Synchronous to Asynchronous Type Signatures (using `Future`)

Remember the transformation we applied to a synchronous type signature to make it asynchronous:

```
def program(a: A): B
```

```
def program(a: A, k: B => Unit): Unit
```

What if we could model an asynchronous result of type `T` as a return type `Future[T]`?

```
def program(a: A): Future[B]
```

From Continuation Passing Style to Future

```
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

From Continuation Passing Style to Future

```scala
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```scala
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit
```

From Continuation Passing Style to Future

```scala
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```scala
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit

// by introducing a type alias
type Future[+T] = (T => Unit) => Unit
def program(a: A): Future[B]
```

From Continuation Passing Style to Future

```scala
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```scala
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit

// by introducing a type alias
type Future[+T] = (T => Unit) => Unit
def program(a: A): Future[B]

// bonus: adding failure handling
type Future[+T] = (Try[T] => Unit) => Unit
```

Towards a Brighter `Future`

```
type Future[+T] = (Try[T] => Unit) => Unit
```

Towards a Brighter Future

```scala
type Future[+T] = (Try[T] => Unit) => Unit

// by reifying the alias into a proper trait
trait Future[+T] extends ((Try[T] => Unit) => Unit) {
  def apply(k: Try[T] => Unit): Unit
}
```

Towards a Brighter Future

```
type Future[+T] = (Try[T] => Unit) => Unit

// by reifying the alias into a proper trait
trait Future[+T] extends ((Try[T] => Unit) => Unit) {
  def apply(k: Try[T] => Unit): Unit
}

// by renaming ‘apply‘ to ‘onComplete‘
trait Future[+T] {
  def onComplete(k: Try[T] => Unit): Unit

}
```

coffeeBreak Revisited With Future

```scala
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit = {
  val eventuallyCoffee = makeCoffee()
  eventuallyCoffee.onComplete { tryCoffee =>
    tryCoffee.foreach(drink)
  }
  chatWithColleagues()
}
```

Handling Failures

```scala
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit = {
  makeCoffee().onComplete {
    case Success(coffee) => drink(coffee)
    case Failure(reason) => ...
  }
  chatWithColleagues()
}
```

Handling Failures

```
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit = {
  makeCoffee().onComplete {
    case Success(coffee) => drink(coffee)
    case Failure(reason) => ...
  }
  chatWithColleagues()
}
```

- However, most of the time you want to transform a successful result and delay failure handling to a later point in the program

Transformation Operations

- `onComplete` suffers from the same composability issues as callbacks
- `Future` provides convenient high-level transformation operations

(Simplified) API of `Future`:

```scala
trait Future[+A] {
  def onComplete(k: Try[A] => Unit): Unit
  // transform successful results
  def map[B](f: A => B): Future[B]
  def flatMap[B](f: A => Future[B]): Future[B]
  def zip[B](fb: Future[B]): Future[(A, B)]
  // transform failures
  def recover(f: Exception => A): Future[A]
  def recoverWith(f: Exception => Future[A]): Future[A]
}
```

map Operation on Future

```scala
trait Future[+A] {
  def map[B](f: A => B): Future[B]
}
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => B` after the `Future[A]` has completed
- Automatically propagates the failure of the former `Future[A]` (if any), to the resulting `Future[B]`

```scala
def grindBeans(): Future[GroundCoffee]
def brew(groundCoffee: GroundCoffee): Coffee

def makeCoffee(): Future[Coffee] =
  grindBeans().map(groundCoffee => brew(groundCoffee))
```

flatMap Operation on Future

```scala
trait Future[+A] {
  def flatMap[B](f: A => Future[B]): Future[B]
}
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => Future[B]` after the `Future[A]` has completed
- Returns a failed `Future[B]` if the former `Future[A]` failed or if the `Future[B]` resulting from the application of the function `f` failed.

```scala
def grindBeans(): Future[GroundCoffee]
def brew(groundCoffee: GroundCoffee): Future[Coffee]

def makeCoffee(): Future[Coffee] =
  grindBeans().flatMap(groundCoffee => brew(groundCoffee))
```

`zip` Operation on `Future`

```scala
trait Future[+A] {
  def zip[B](other: Future[B]): Future[(A, B)]
}
```

- Joins two successful `Future[A]` and `Future[B]` values into a single successful `Future[(A, B)]` value
- Returns a failure if any of the two `Future` values failed
- Does *not* create any dependency between the two `Future` values!

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee() zip makeCoffee()
```

zip vs flatMap

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee() zip makeCoffee()
```

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().flatMap { coffee1 =>
    makeCoffee().map(coffee2 => (coffee1, coffee2))
  }
```

zip vs flatMap (2)

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee() zip makeCoffee()
```

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] = {
  val eventuallyCoffee1 = makeCoffee()
  val eventuallyCoffee2 = makeCoffee()
  eventuallyCoffee1.flatMap { coffee1 =>
    eventuallyCoffee2.map(coffee2 => (coffee1, coffee2))
  }
}
```

Sequencing Futures (1)

```
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] = ...

def workRoutine(): Future[Work] = {
  work().flatMap { work1 =>
    coffeeBreak().flatMap { _ =>
      work().map { work2 =>
        work1 + work2
      }
    }
  }
}
```

Sequencing Futures (2)

```scala
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] = ...

def workRoutine(): Future[Work] =
  for {
    work1 <- work()
    _     <- coffeeBreak()
    work2 <- work()
  } yield work1 + work2
```

- Back to a familiar layout to sequence computations!

coffeeBreak, Again

```scala
def coffeeBreak(): Future[Unit] = {
  val eventuallyCoffeeDrunk = makeCoffee().flatMap(drink)
  val eventuallyChatted     = chatWithColleagues()

  eventuallyCoffeeDrunk.zip(eventuallyChatted)
    .map(_ => ())
}
```

`recover` and `recoverWith` Operations on `Future`

Turn a failed `Future` into a successful one

```scala
trait Future[+A] {
  def recover[B >: A](pf: PartialFunction[Throwable, B]): Future[B]
  def recoverWith[B >: A](pf: PartialFunction[Throwable, Future[B]]): Future[B]
}
```

```scala
grindBeans()
  .recoverWith { case BeansBucketEmpty =>
    refillBeans().flatMap(_ => grindBeans())
  }
  .flatMap(coffeePowder => brew(coffeePowder))
```

Execution Context

- So far, we haven't said anything about where continuations are executed, *physically*
- How do we control that?
  - Single thread? Fixed size thread pool?

Execution Context

- So far, we haven't said anything about where continuations are executed, *physically*
- How do we control that?
  - Single thread? Fixed size thread pool?

```scala
trait Future[+A] {
  def onComplete(k: Try[A] => Unit)(implicit ec: ExecutionContext): Unit
}

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

Lift a Callback-Based API to Future (1)

```
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] = ...
```

Lift a Callback-Based API to Future (2)

```scala
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] = {
  val p = Promise[Coffee]()
  makeCoffee(
    coffee => p.trySuccess(coffee),
    reason => p.tryFailure(reason)
  )
  p.future
}
```

Summary

In this video, we have seen:

- The `Future[T]` type is an equivalent alternative to continuation passing
- Offers convenient *transformation* and *failure recovering* operations
- `map` and `flatMap` operations introduce *sequentiality*

Backup Slides

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Monads and Effects (1/2)

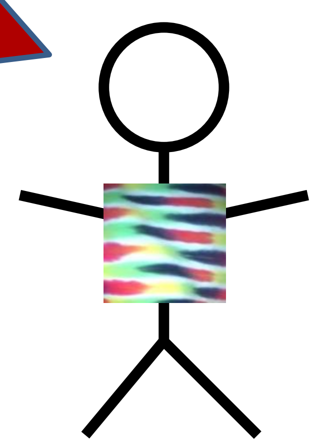
Principles of Reactive Programming

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.

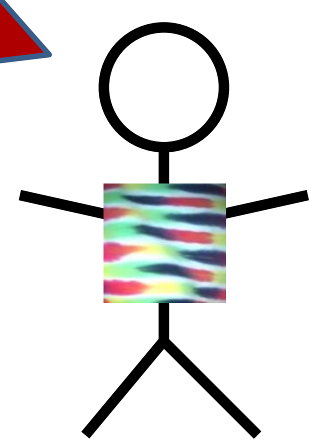
The only artifact you can trust is actual source code.



Warning

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

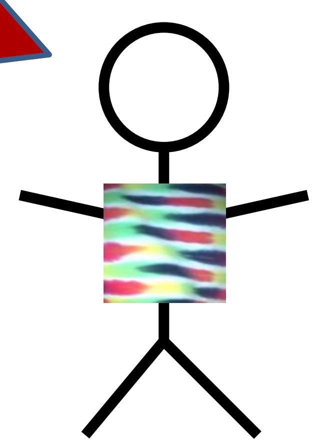
The RxScala method names do not necessarily correspond 1:1 with the underlying RxJava method names.



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

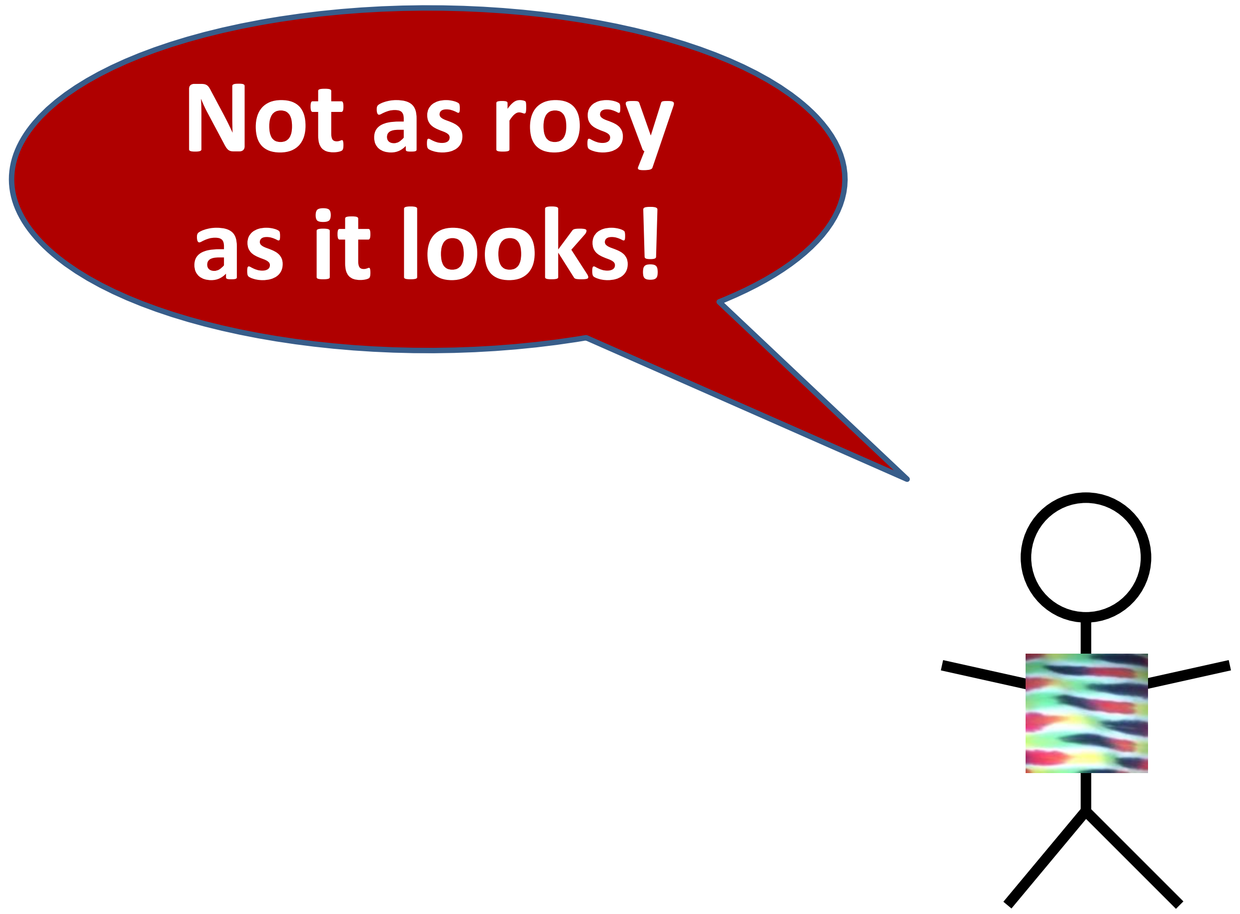
	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

The Four Essential Effects In Programming

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

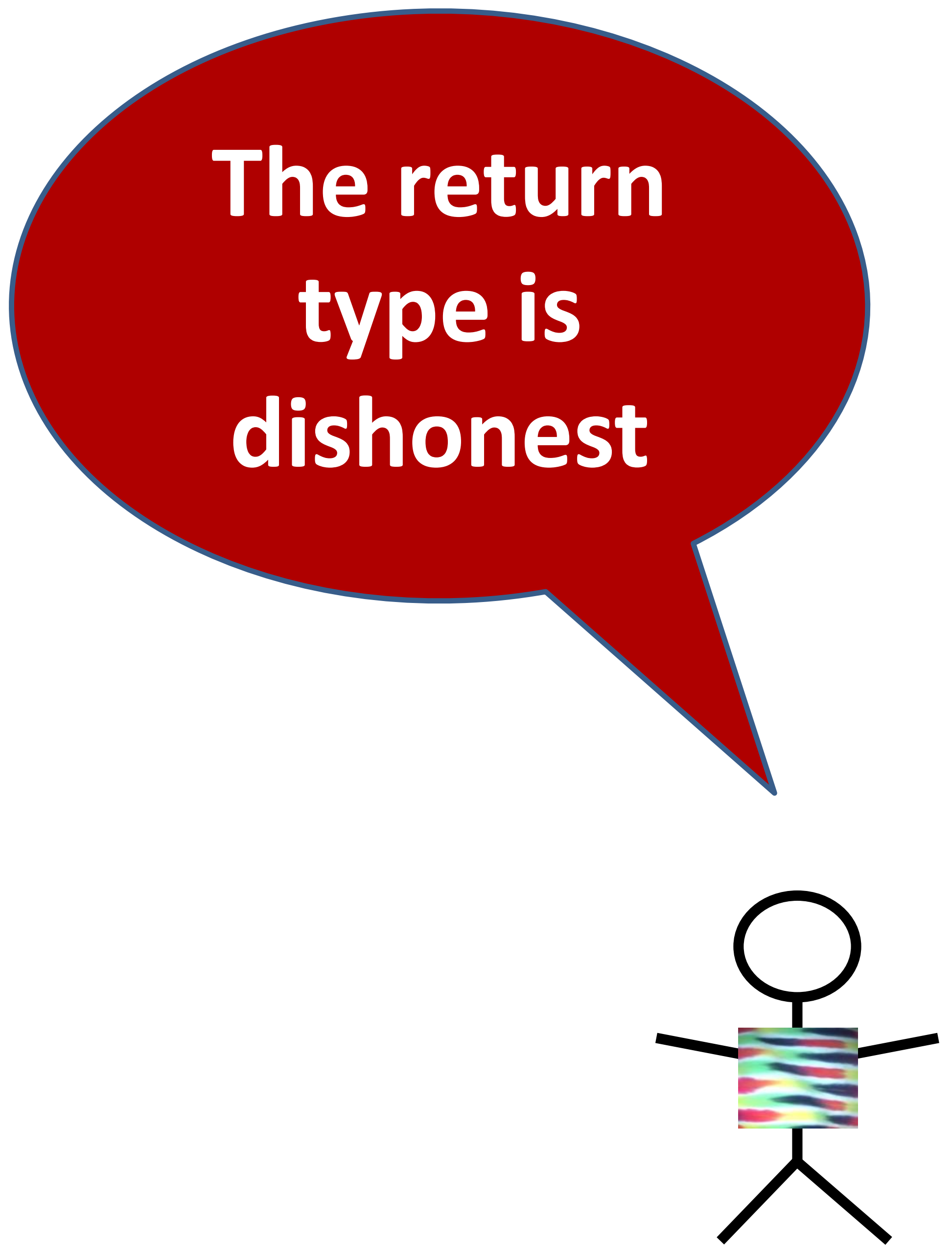
A simple adventure game

```
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)
```



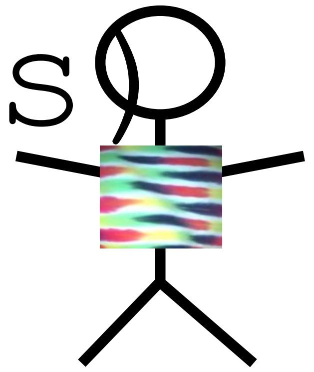
Actions may fail

```
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

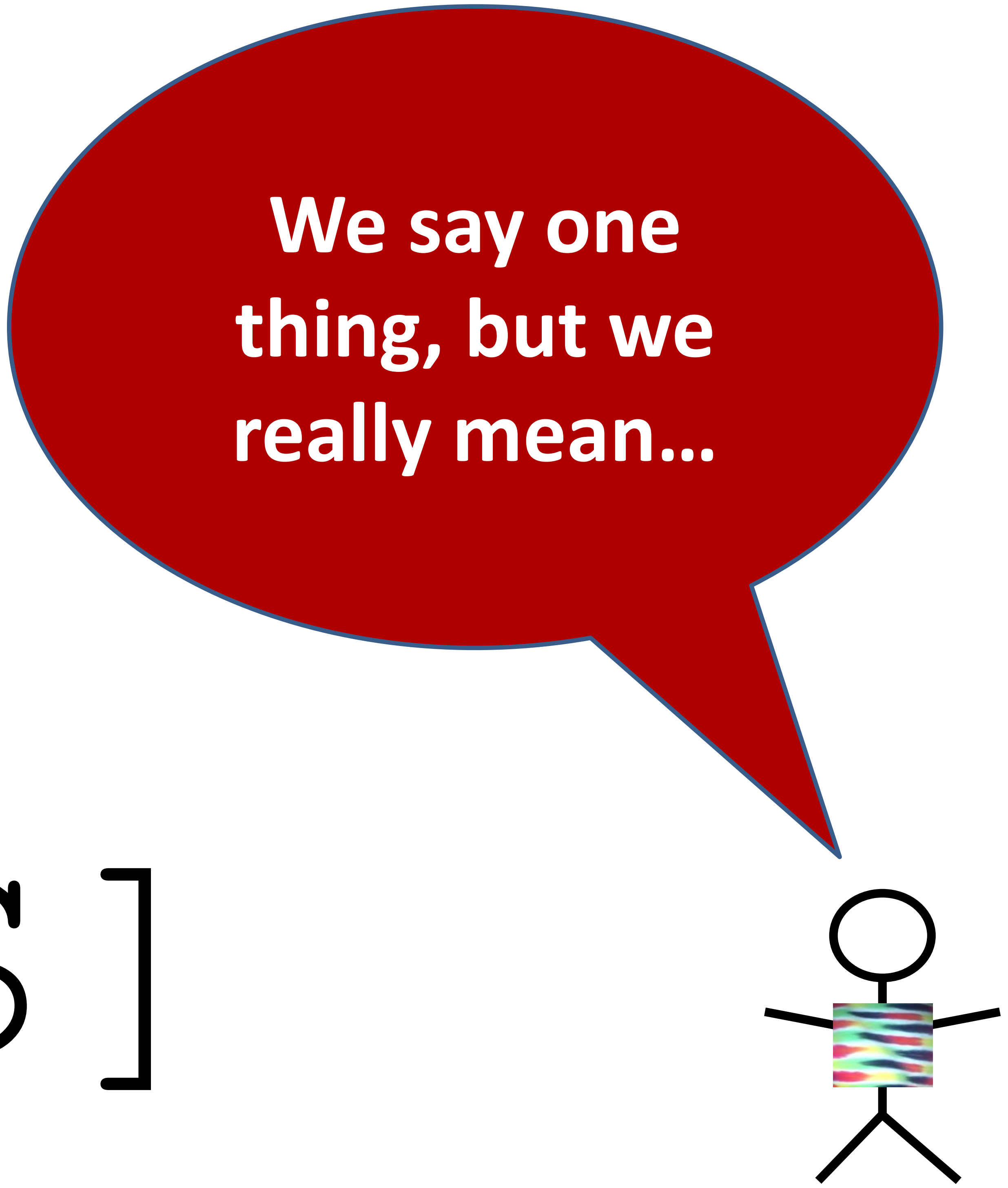
```
val adventure = Adventure()

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

```scala
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]
```

Monads guide you through the happy path

`Try[T]`

A monad that handles **exceptions**.

Noise reduction

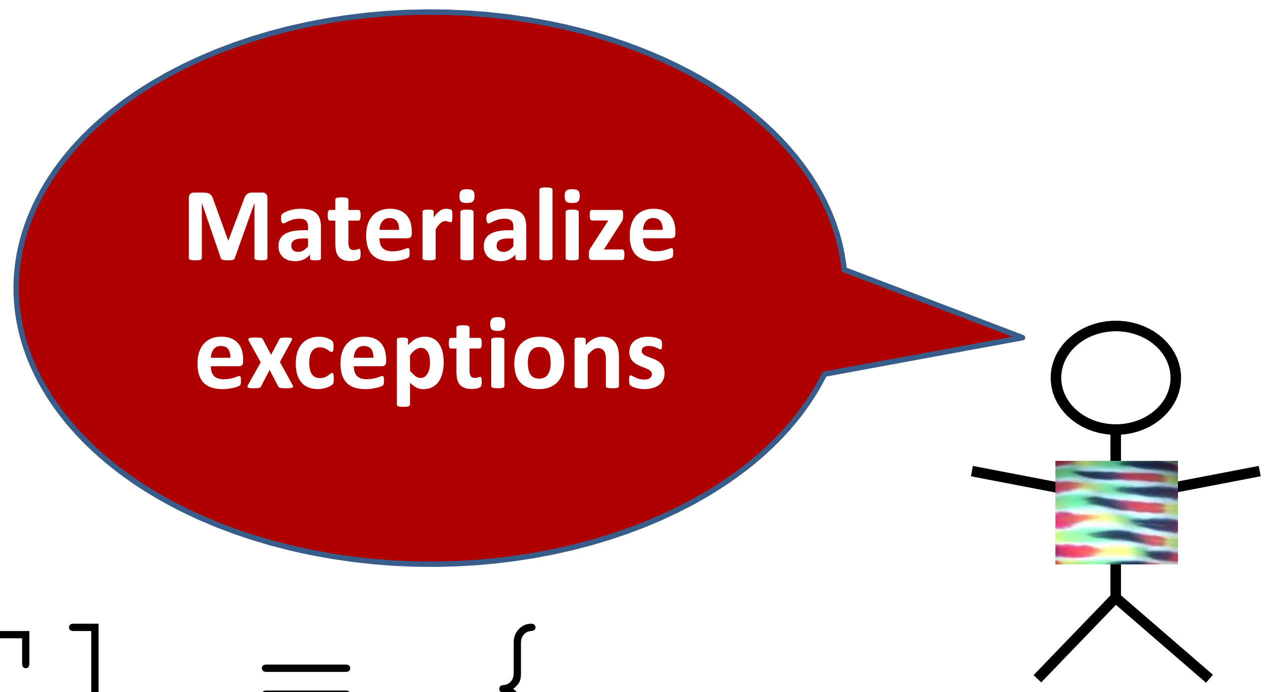
```
val adventure = Adventure()

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)
```

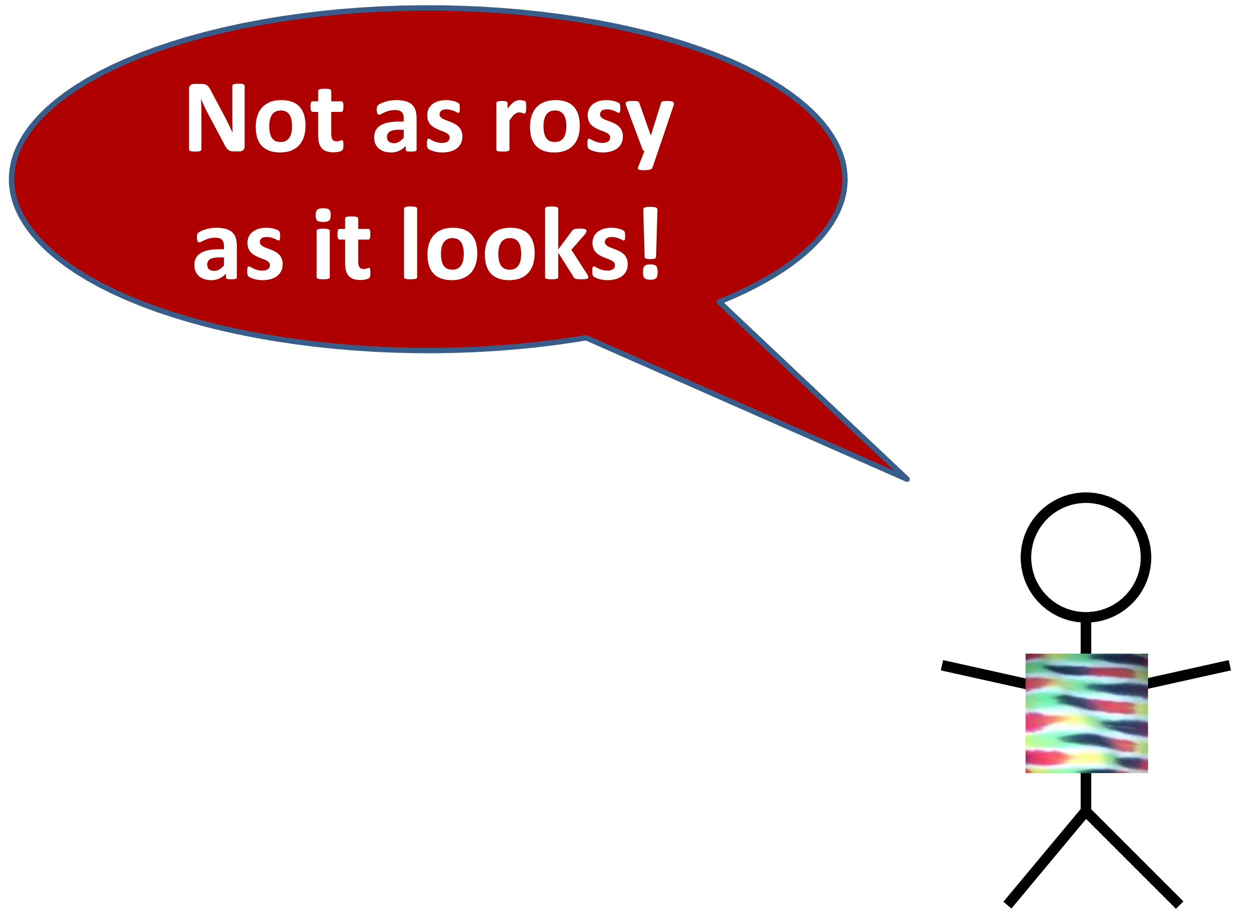
Recall our simple adventure game ….

```
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

$\rightarrow$

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

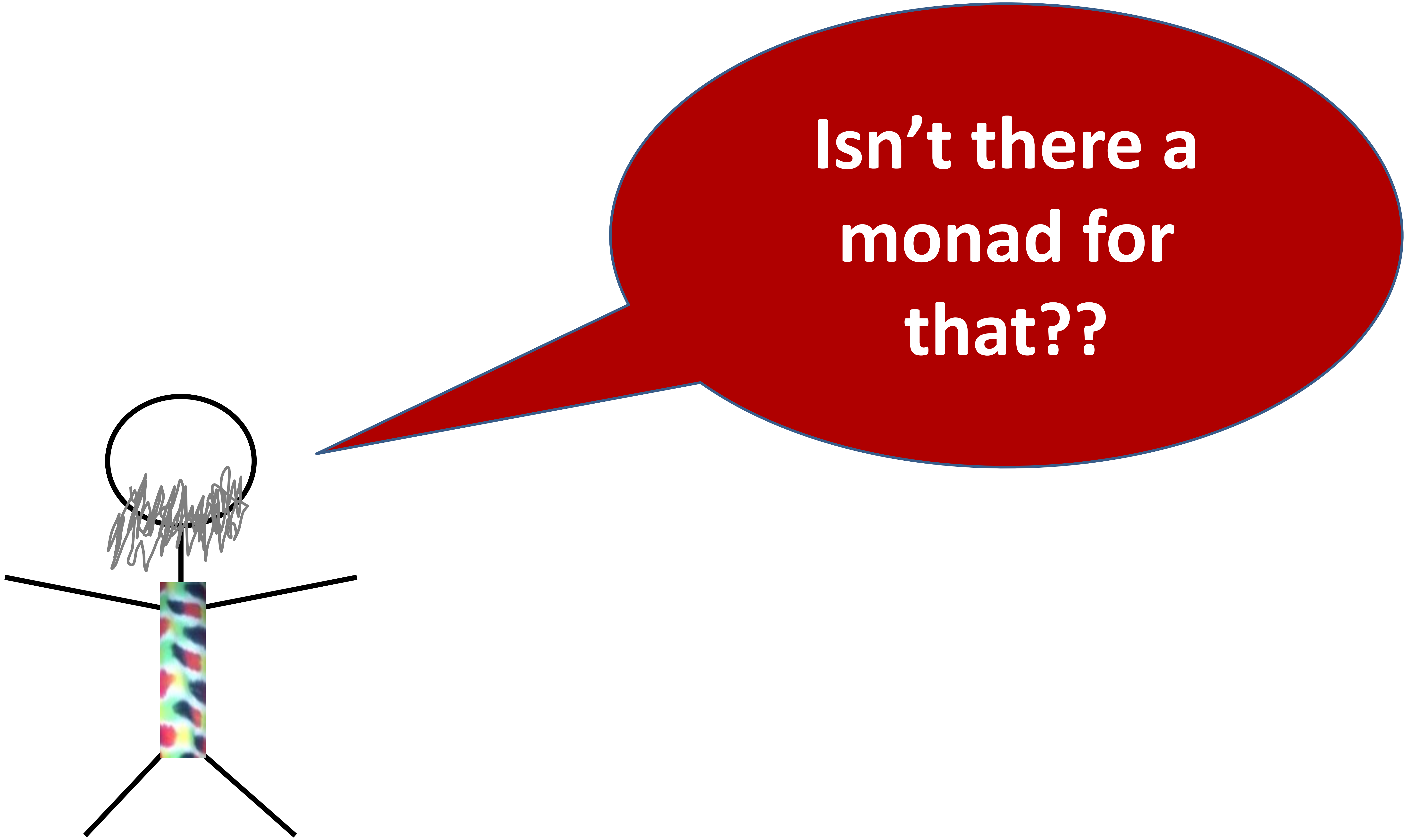
```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 3 days
// only continue if there is no exception
val confirmation = socket.sendToEurope(packet)
// block for 5 years
// only continue if there is no exception
```

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

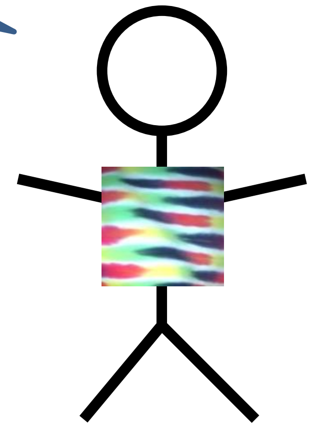
```
Future[T]
```

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

Futures asynchronously notify consumers

```scala
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

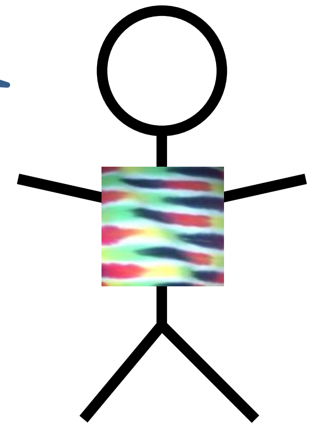


Futures asynchronously notify consumers

```
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)
```

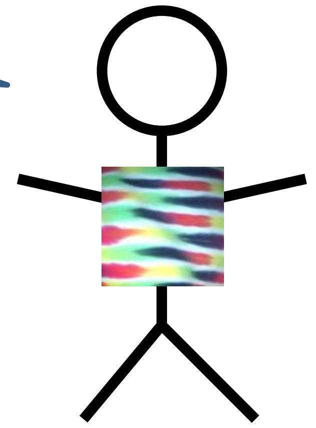


Futures asynchronously notify consumers

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

boilerplate code

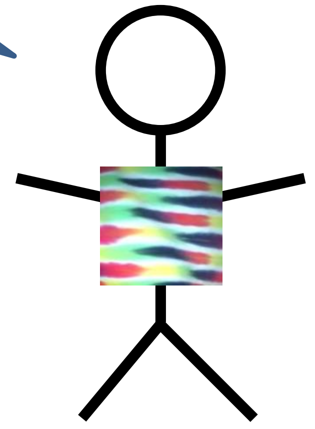
```
ts match {
    case Success(t) =>
onNext(t)
    case Failure(e) =>
onError(e)
}
```



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.

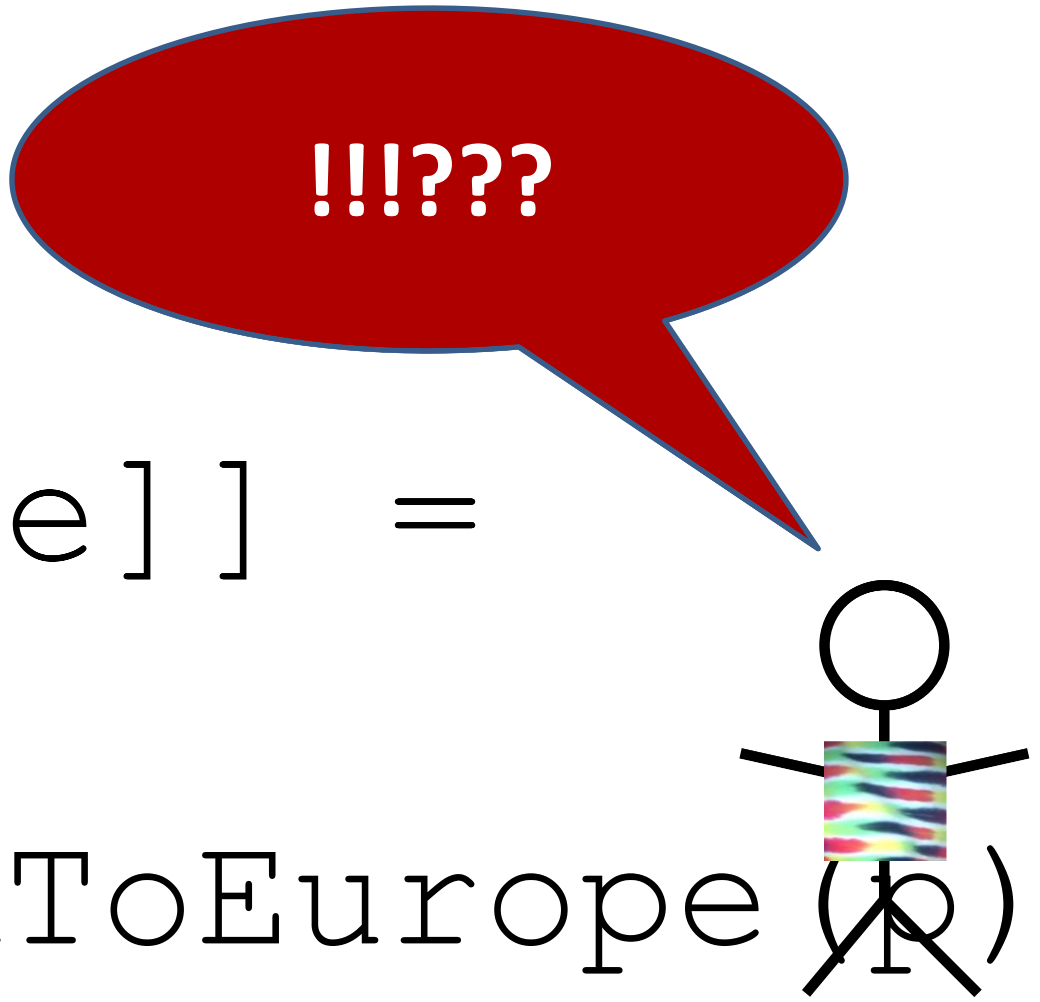


Futures asynchronously notify consumers

```
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]]
}
```

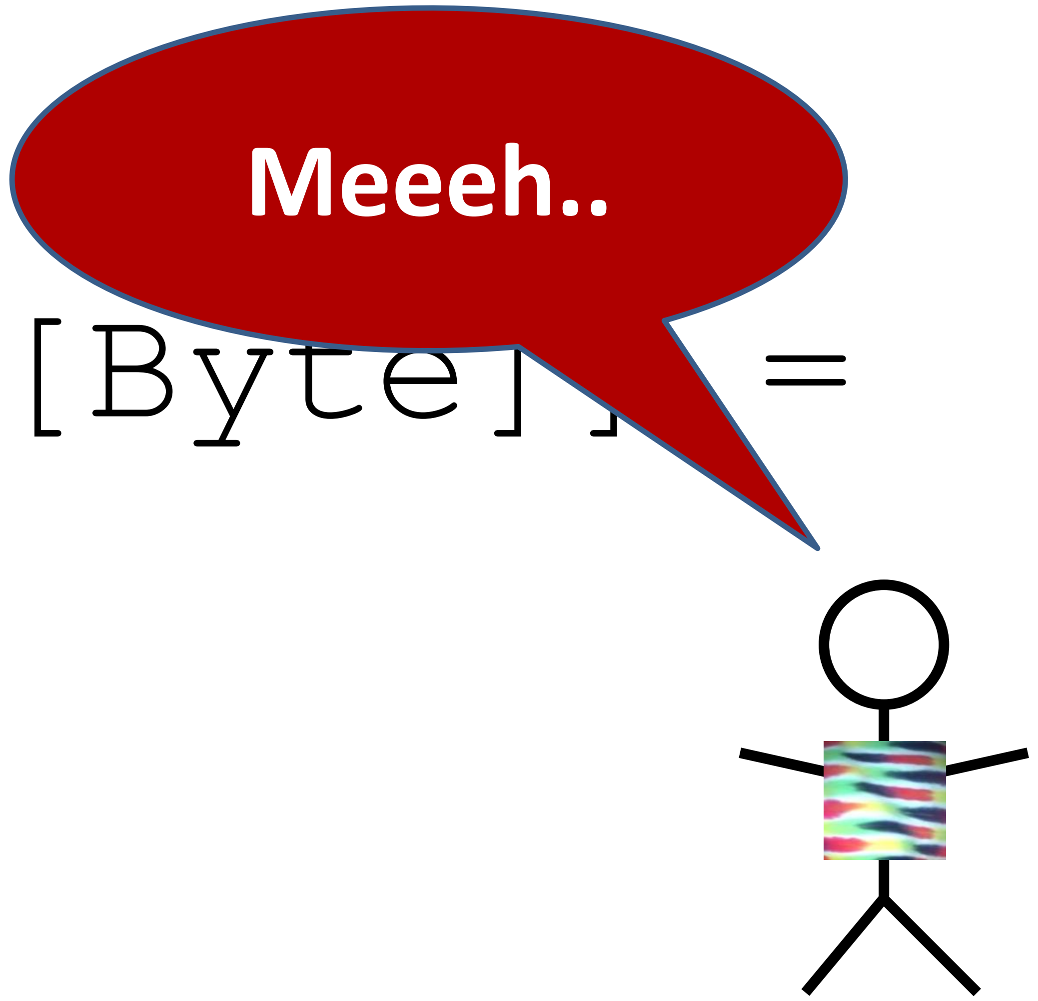
Send packets using futures I

```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)
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

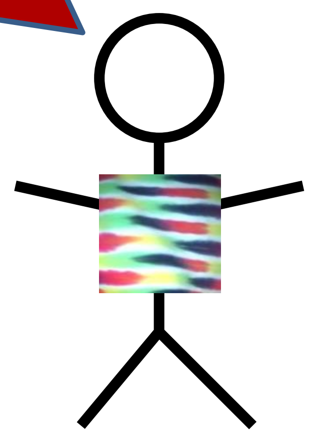
Combinators on Futures (1/2)

Principles of Reactive Programming

Erik Meijer

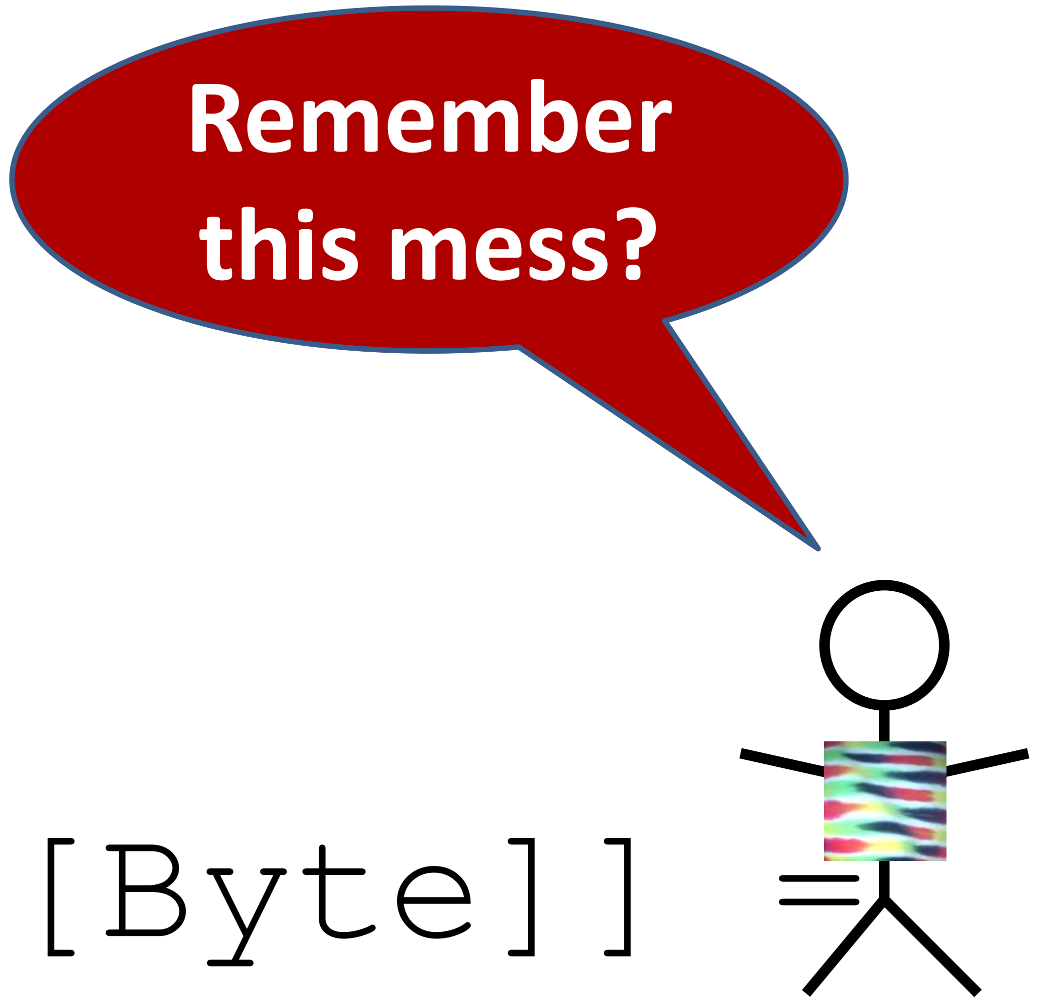
Futures recap

```
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]
}
```



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) ⇒ …
}
```



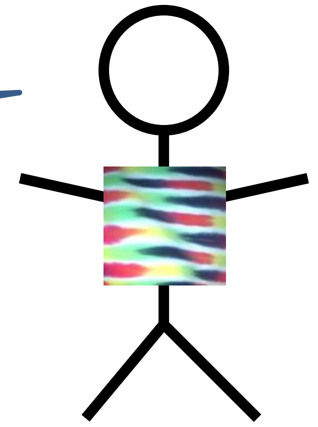
Flatmap to the rescue

```
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.toByte
```

But, this can still fail!

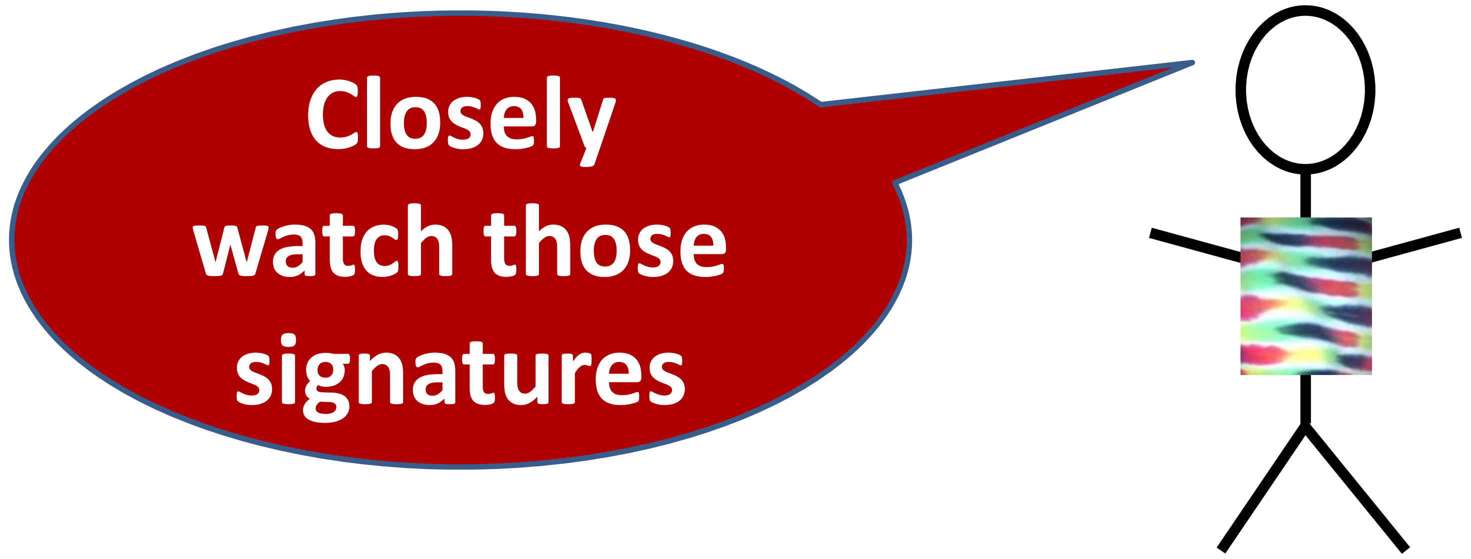


Sending packets using futures robustly (?)

```
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

```scala
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
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Combinators on Futures (2/2)

Principles of Reactive Programming

Erik Meijer

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Composing Futures (1/2)

Principles of Reactive Programming

Erik Meijer

Composing Futures (2/2)

Principles of Reactive Programming

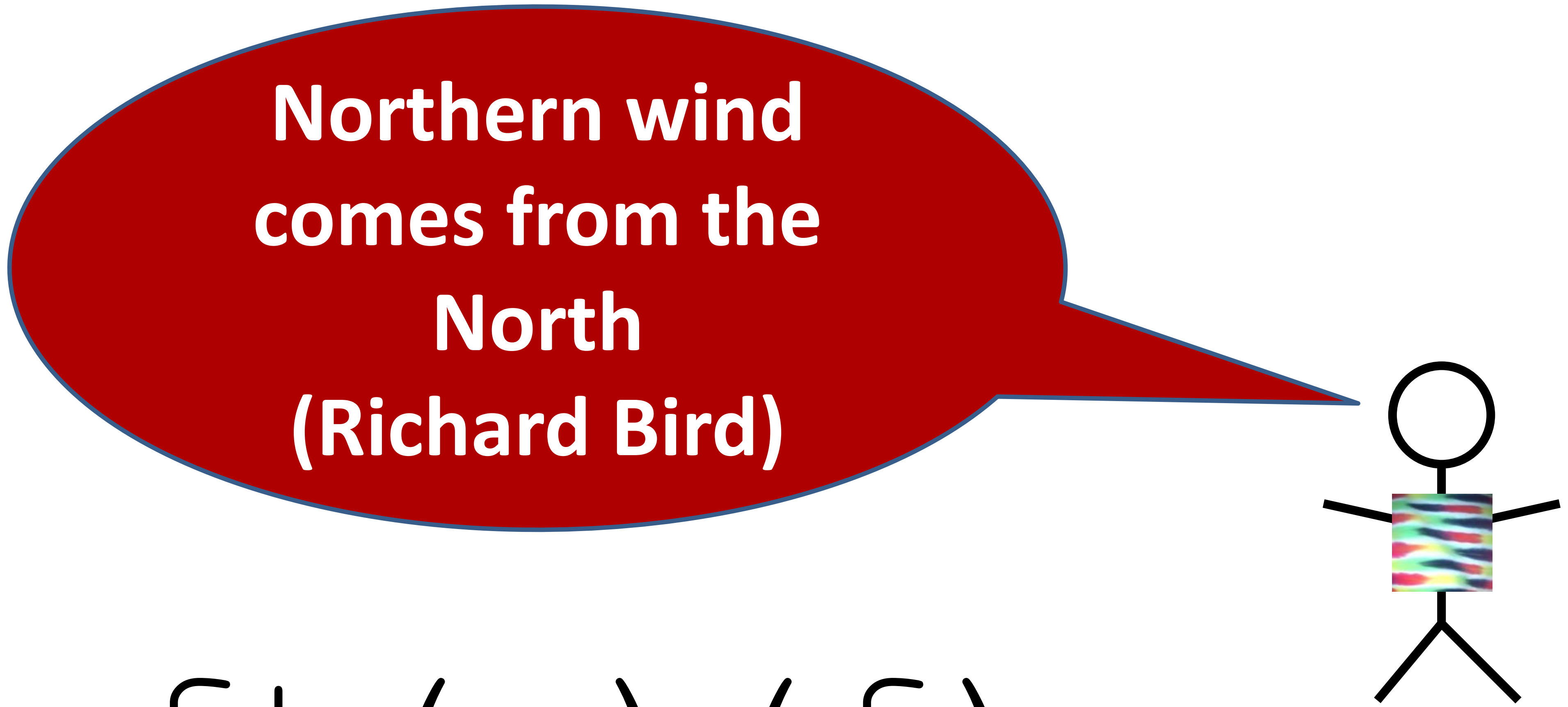
Erik Meijer

Avoid Recursion


```
foldRight
foldLeft
```

Folding lists

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




```
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, …, ()=>$\text{block}_{\text{noTimes}}$)

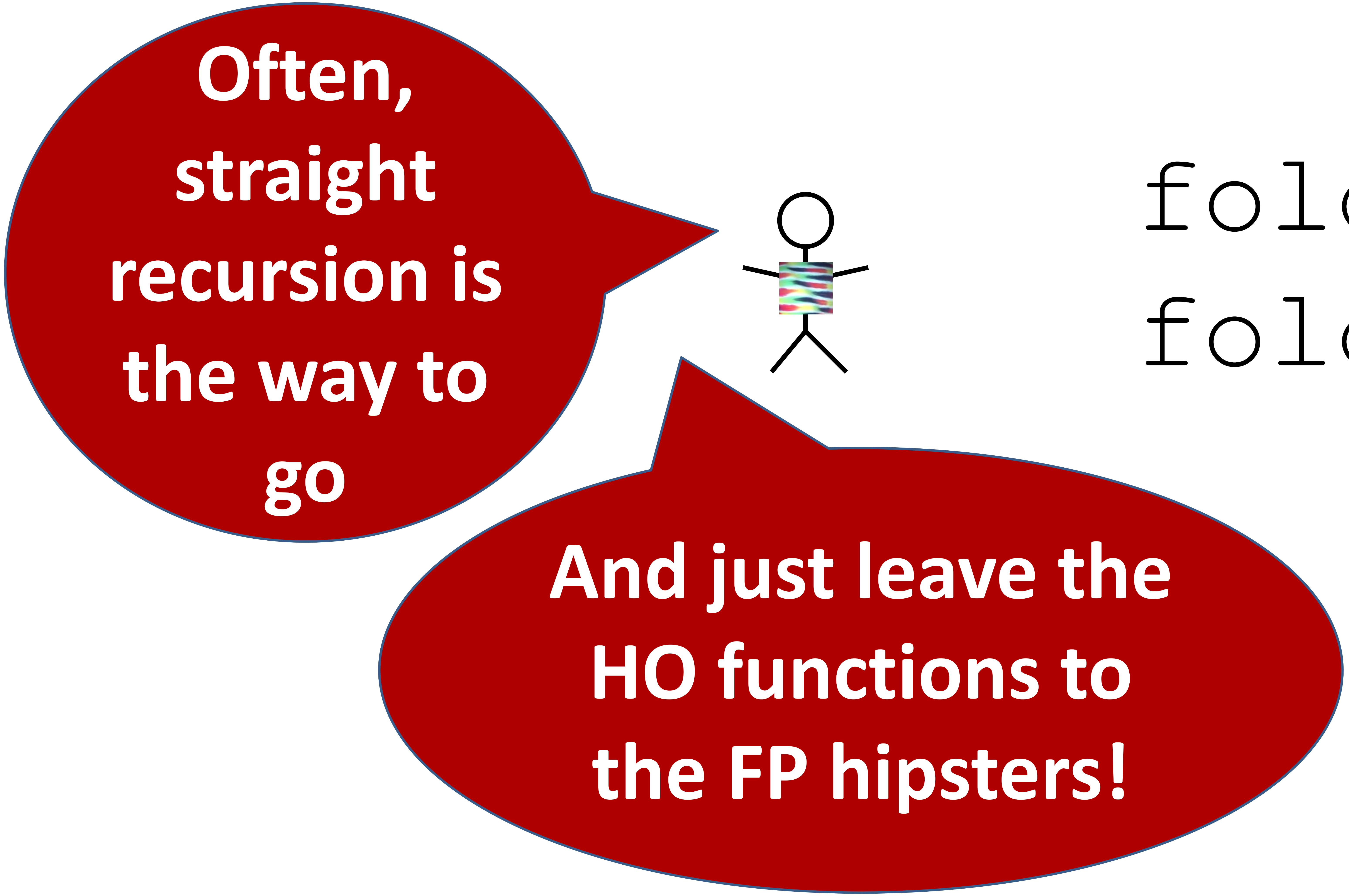
result = (…((failed recoverWith { block_1() })

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
block1 fallbackTo { block2 fallbackTo { block3 fallbackTo
{ failed }}}
```

Use Recursion




```
foldRight
foldLeft
```

End of Composing Futures (2/2)

Principles of Reactive Programming

Erik Meijer