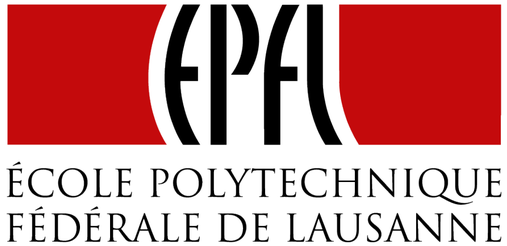


Monads and Effects

Principles of Reactive Programming

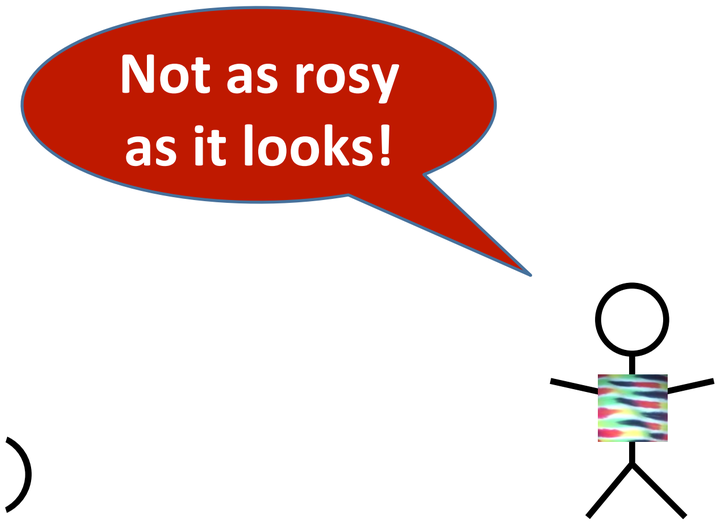
Erik Meijer

The Four Essential Effects In Programming

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

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

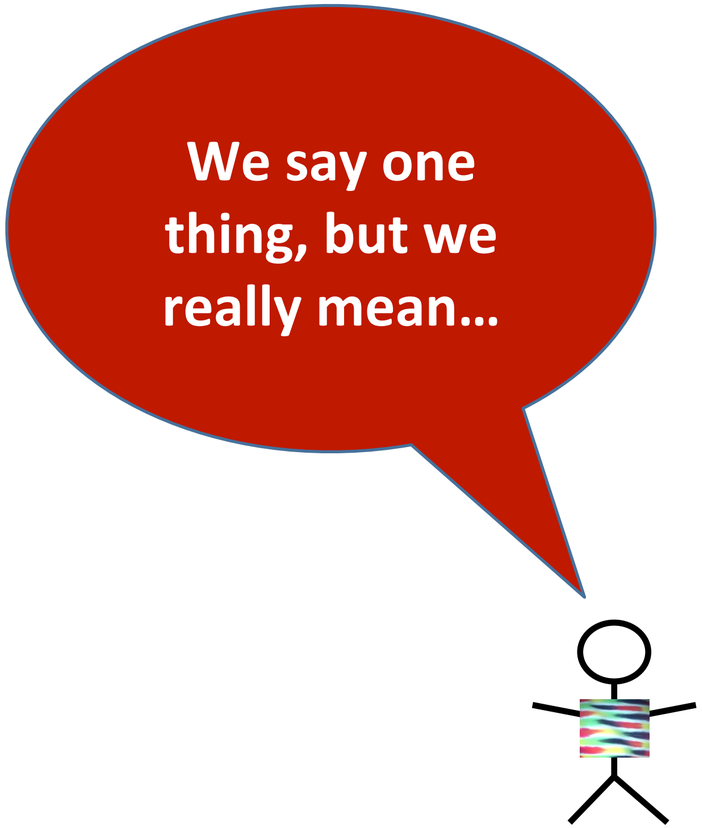
```
val adventure = Adventure()

val coins = adventure.collectCoins()
// 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
```

Expose possibility of failure in the types, honestly

```
T => S
```

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



Making failure evident in types

```
import scala.util.{Try, Success, Failure}

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(t)⇒ 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

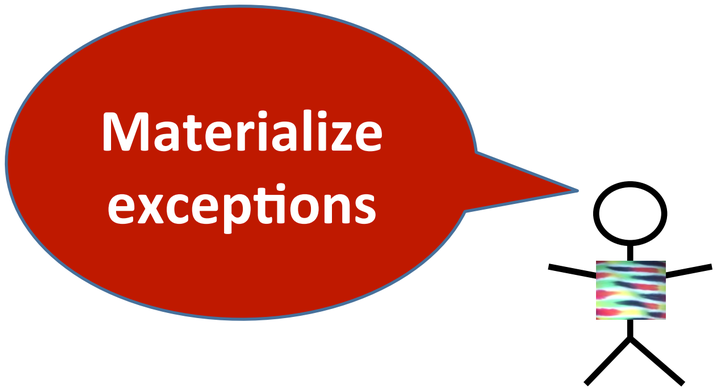
```
val adventure = Adventure()

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) }
}
```



Quiz

a)

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

b)

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

c)

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