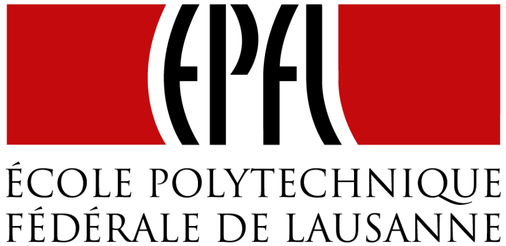


Monads and Effects (2/2)

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]
```

Monads guide you through the happy path

```
Try[T]
```

A monad that handles **exceptions**.

Noise reduction

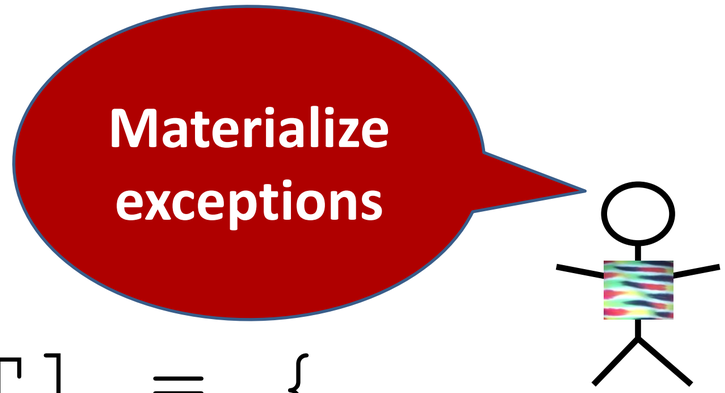
```
val adventure = Adventure()

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

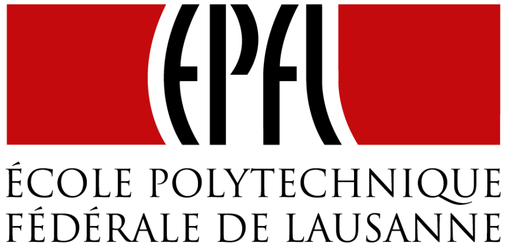
Using comprehension syntax

```
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) }
}
```





End of Monads and Effects (2/2)

Principles of Reactive Programming

Erik Meijer