ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Promises, promises, promises

Principles of Reactive Programming

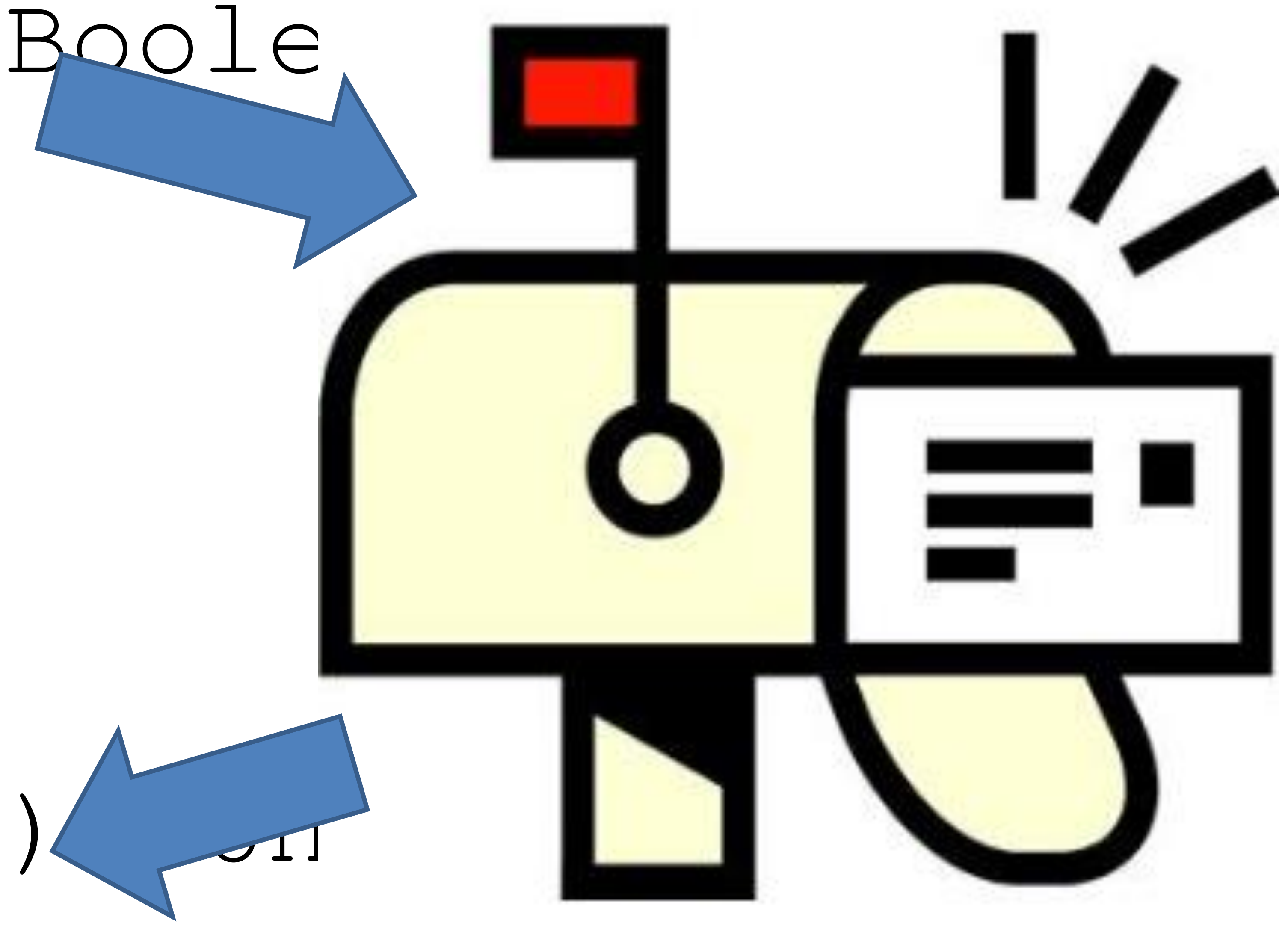
Erik Meijer

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
}
```

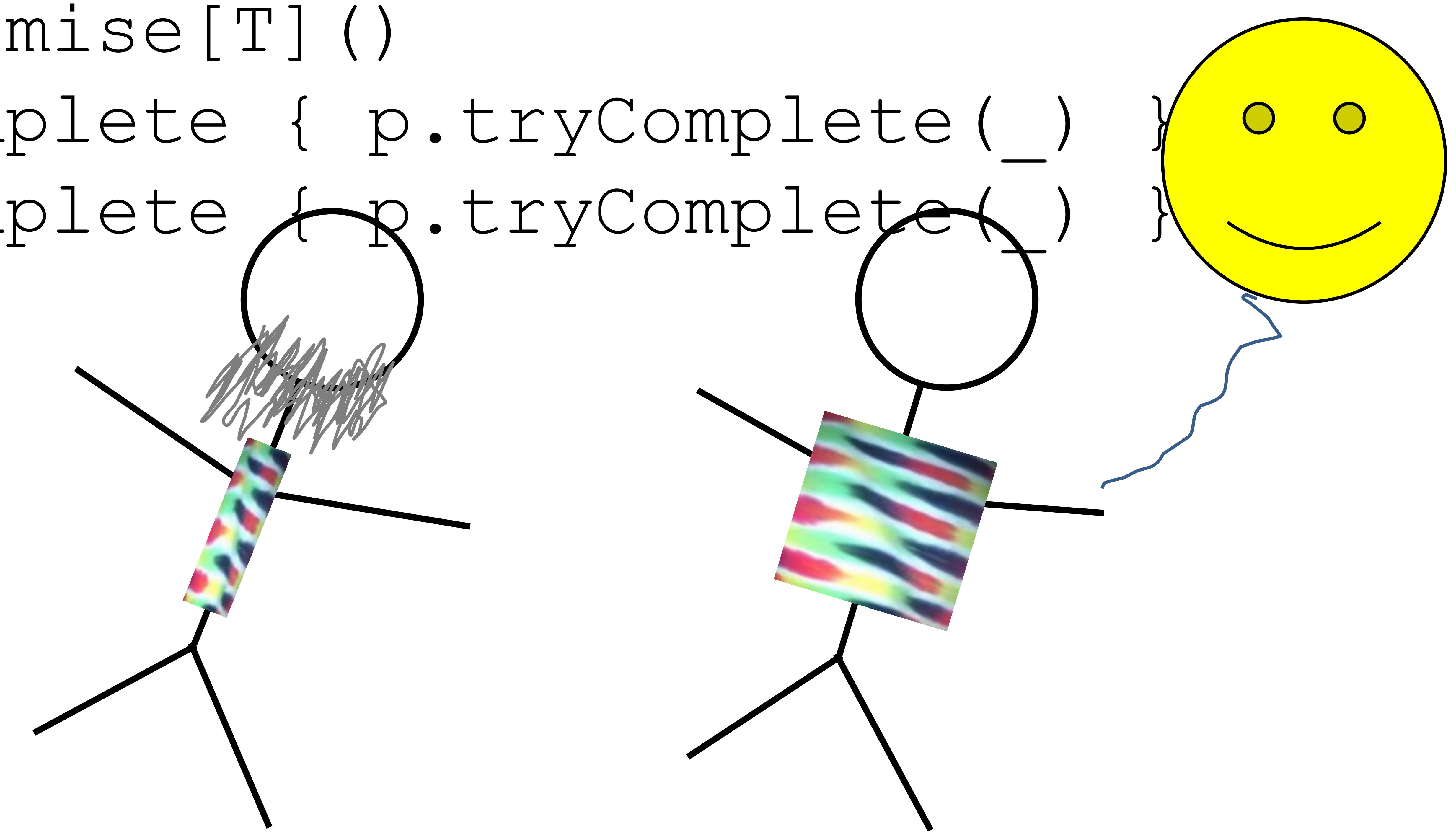
Promises

```
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]`

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

End of Promises, promises, promises

Principles of Reactive Programming

Erik Meijer