ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

From Futures to Observables

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]`

Future[T] and Try[T] are dual

```
trait Future[T] {
    def OnComplete[U](func: Try[T]⇒U)
    (implicit ex: ExecutionContext): Unit
}
```

simplify → (Try[T]⇒Unit)⇒Unit ← simplify

Future[T] and Try[T] are dual

```
(Try[T]⇒Unit)⇒Unit
```

reverse

```
Unit⇒(Unit⇒Try[T])
```

reverse

simplify

```
()⇒(()⇒Try[T])≈Try[T]
```

simplify

Future[T] and Try[T] are dual

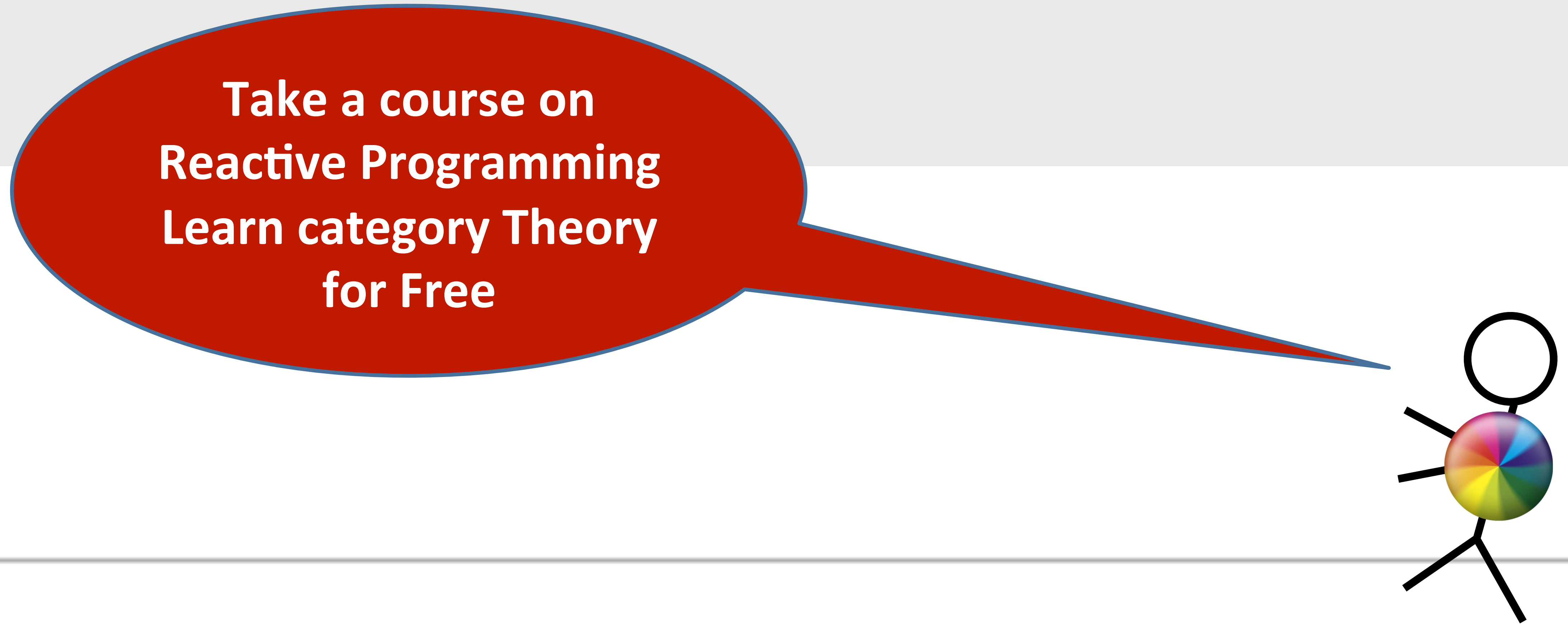
Receive result of type `Try[T]`
by passing callback `(Try[T]⇒Unit)` *to method*

```
def asynchronous(): Future[T] = { … }
```

Receive result of type `Try[T]`
by blocking until method returns

```
def synchronous(): Try[T] = { … }
```

Buy One Get One Free



Dual (category theory)

From Wikipedia, the free encyclopedia

In category theory, a branch of mathematics, **duality** is a correspondence between properties of a category C and so-called **dual properties** of the opposite category C^{op}. Given a statement regarding the category C, by interchanging the source and target of each morphism as well as interchanging the order of composing two morphisms, a corresponding dual statement is obtained regarding the opposite category C^{op}. **Duality**, as such, is the assertion that truth is invariant under this operation on statements. In other words, if a statement is true about C, then its dual statement is true about C^{op}. Also, if a statement is false about C, then its dual has to be false about C^{op}.

Given a concrete category C, it is often the case that the opposite category C^{op} per se is abstract. C^{op} need not be a category that arises from mathematical practice. In this case, another category D is also termed to be in **duality** with C if D and C^{op} are equivalent as categories.

In the case when C and its opposite C^{op} are equivalent, such a category is **self-dual**.

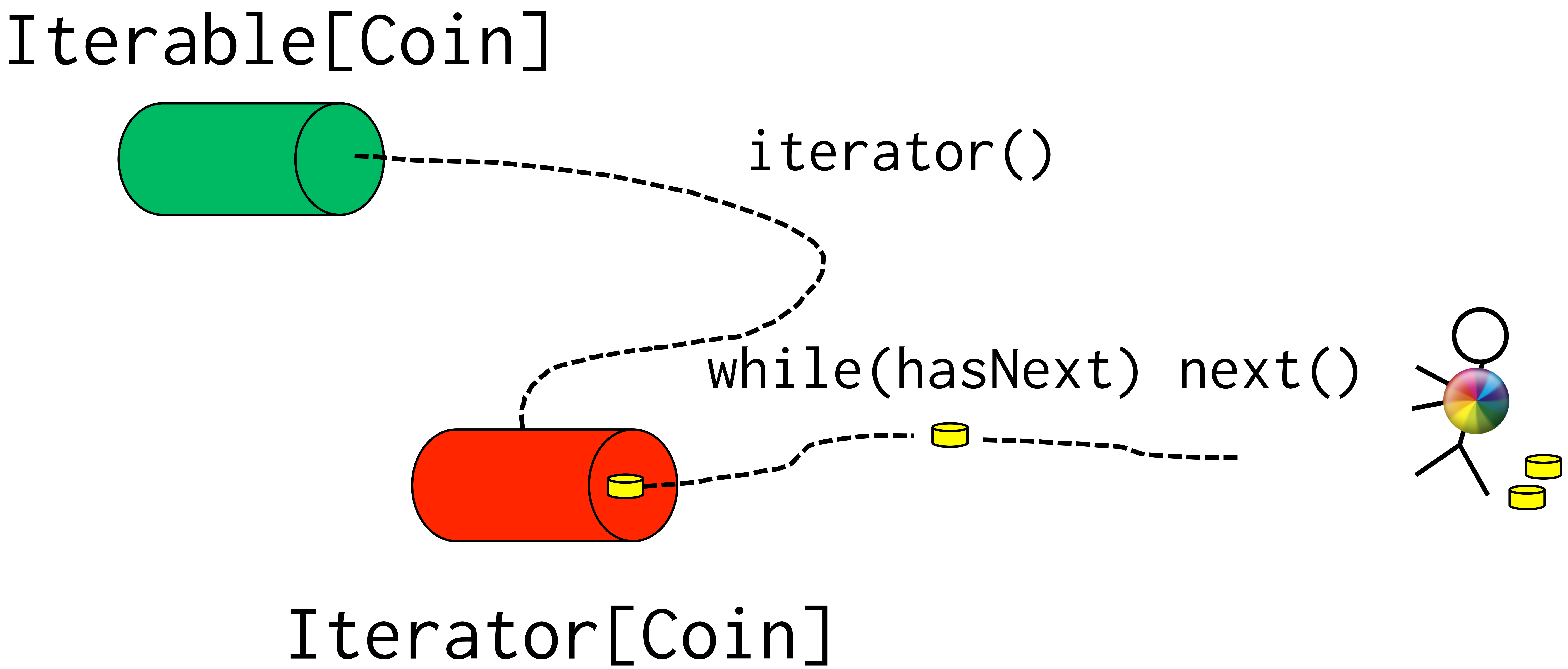
The Four Essential Effects In Programming

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

Synchronous Data Streams: Iterable[T]

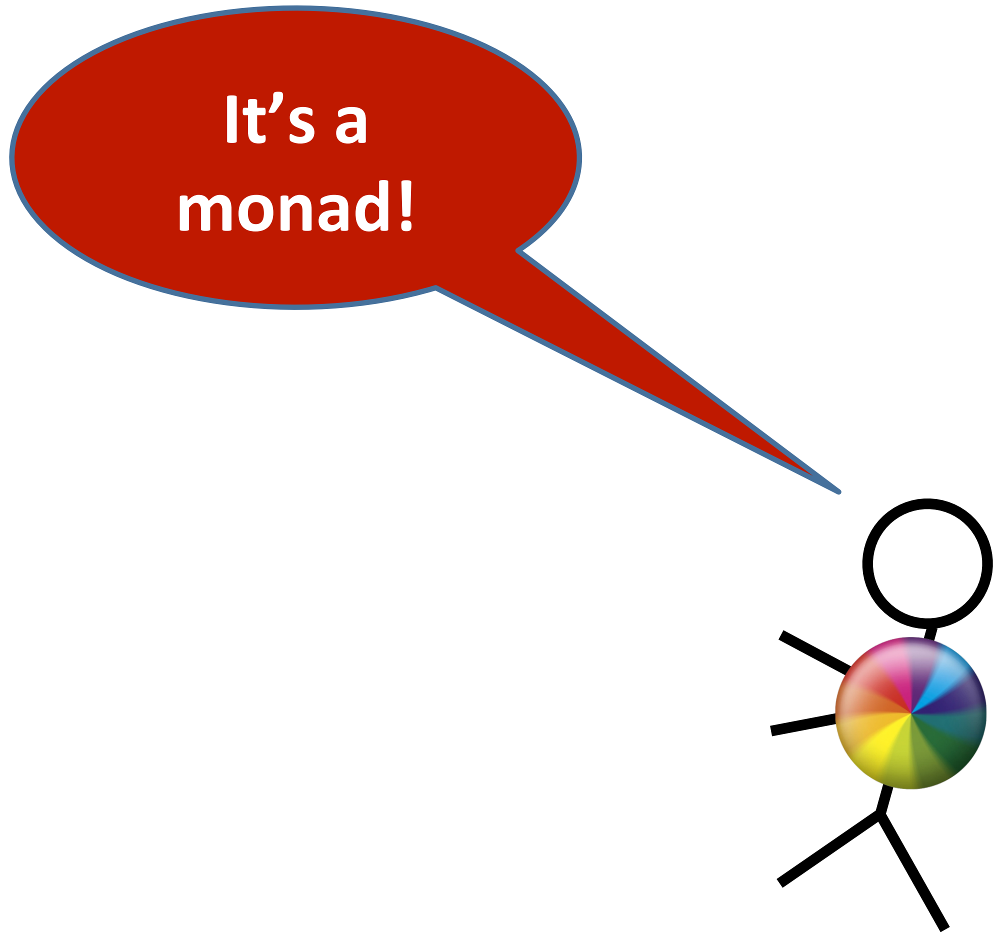
```scala
// This is a base trait for all Scala collections
// that define an iterator method to step through
// one-by-one the collection's elements.

trait Iterable[T] {  def iterator(): Iterator[T] }

// Iterators are data structures that allow to iterate over a sequence of
// elements. They have a hasNext method for checking if there is a next
// element available, and a next method which returns the next element.

trait Iterator[T] { def hasNext: Boolean; def next(): T }
```

Synchronous Data Streams: Iterable[T]

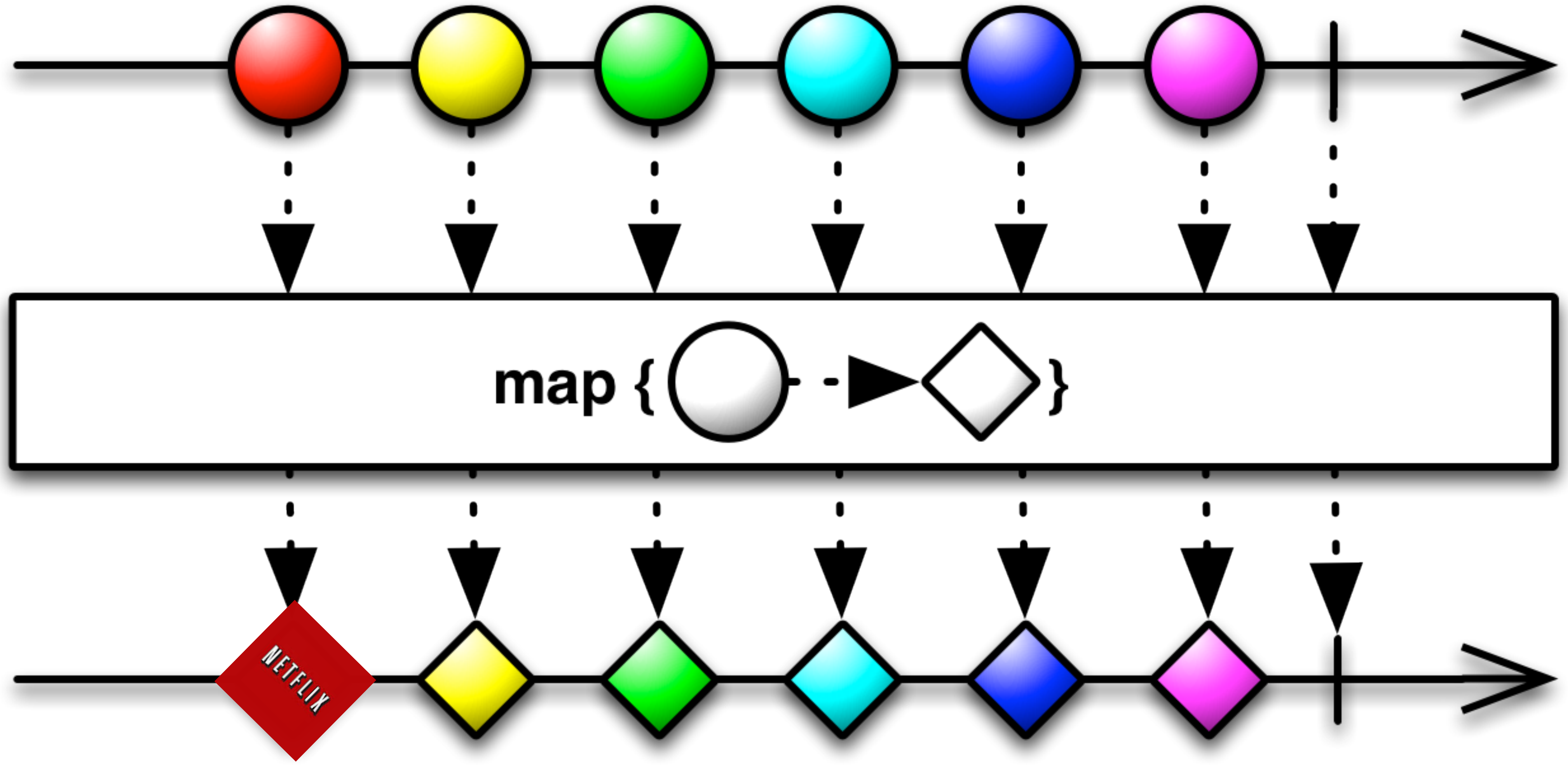


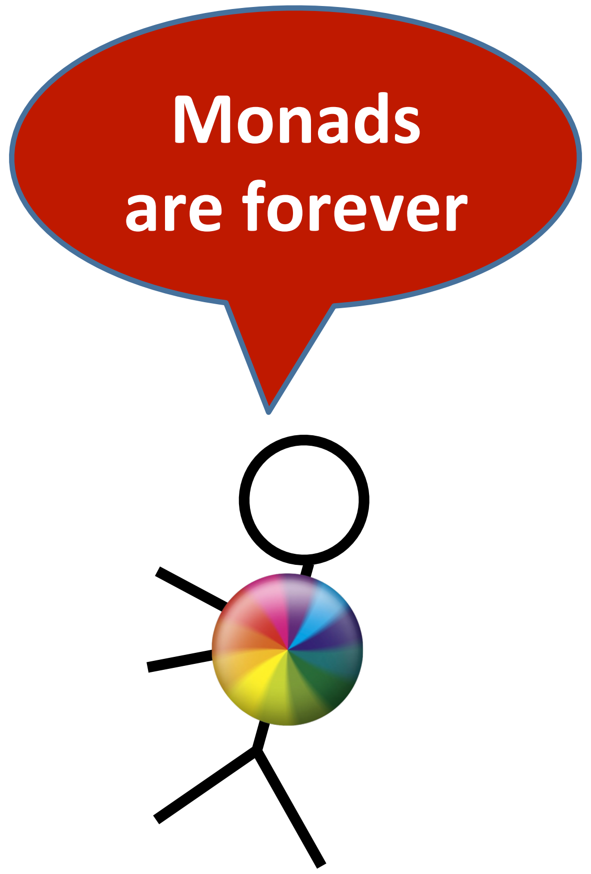
Higher-order Function to manipulate Iterable[T]

```
def flatMap[B](f: A⇒Iterable[B]): Iterable[B]

def map[B](f: A⇒B): Iterable[B]

def filter(p: A⇒Boolean): Iterable[A]

def take(n: Int): Iterable[A]

def takeWhile(p: A⇒Boolean): Iterable[A]

def toList(): List[A]

def zip[B](that: Iterable [B]): Iterable[(A, B)]
```



Marble Diagrams





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

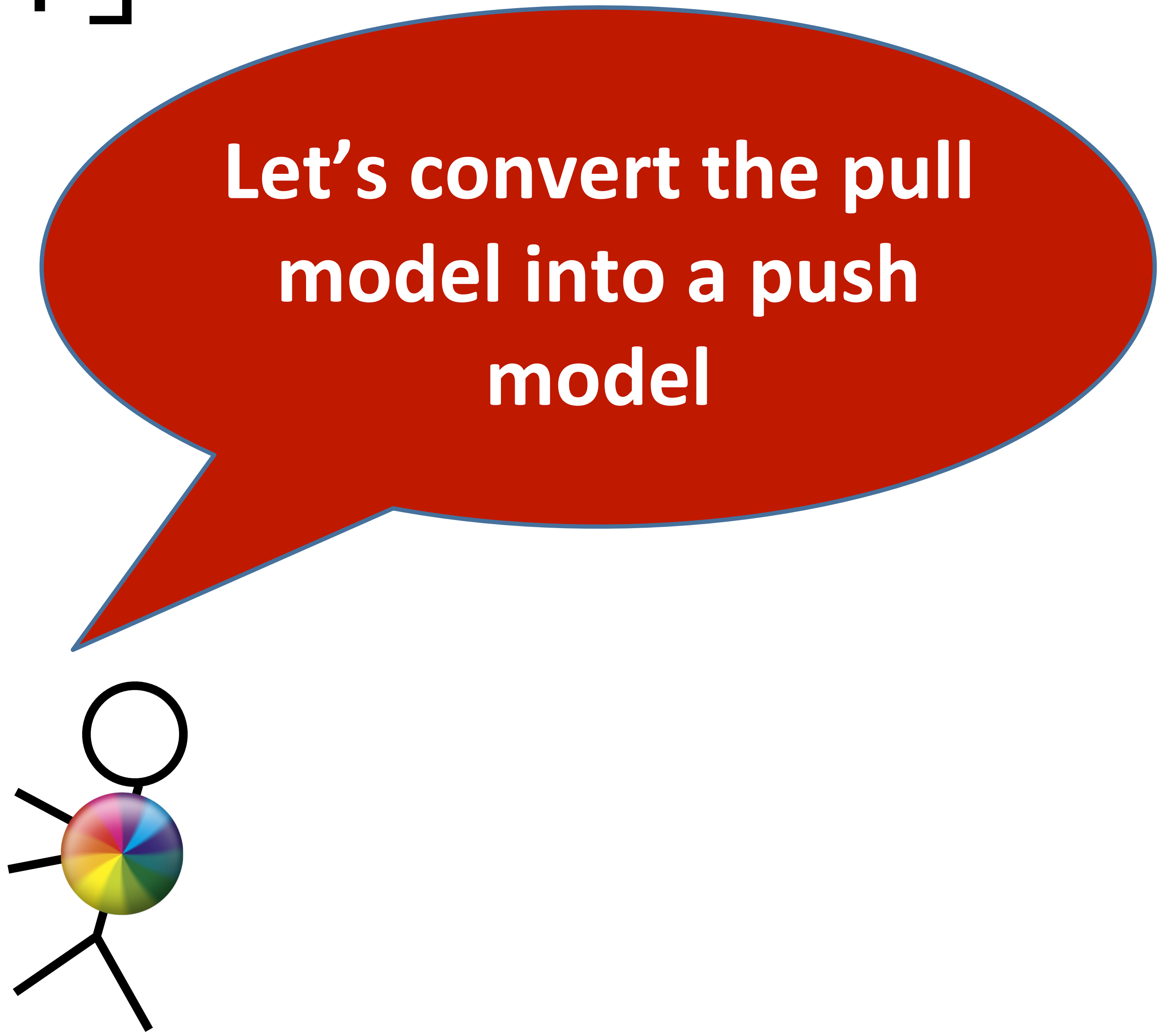
http://norvig.com/21-days.html#answers

Reading Files From Disk Using Iterators

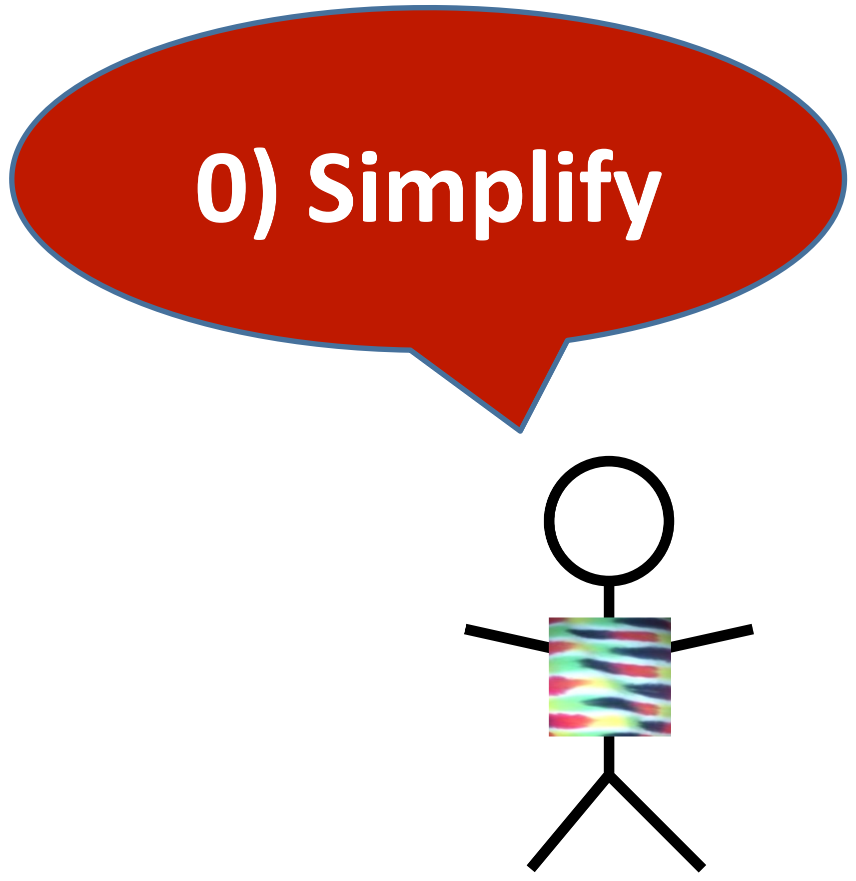
```
def ReadLinesFromDisk(path: String): Iterator[String] = {
   Source.fromFile(path).getLines()
}

val lines = ReadLinesFromDisk(“\c:\tmp.txt”)

for (line <- lines) {
  … DoWork(line) …
}
```

The return of the magic dualization trick

```
trait Iterable[T] {
    def iterator(): Iterator[T]
}
trait Iterator[T] {
    def hasNext: Boolean
    def next(): T
}
```



The return of the magic dualization trick

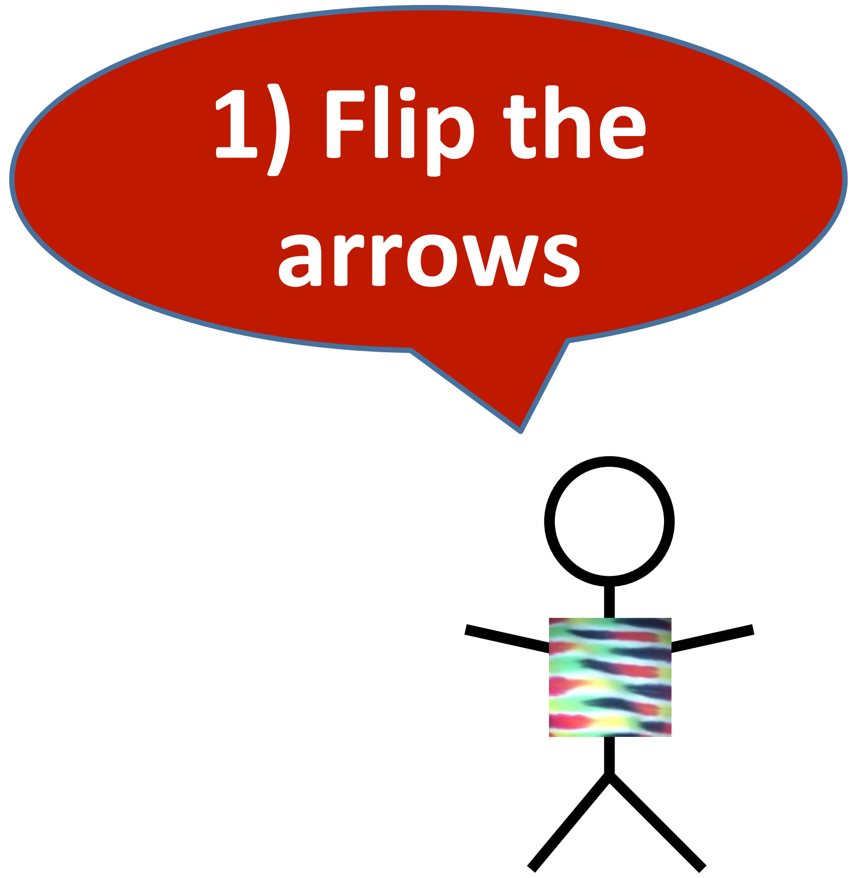



```
trait Iterable[T] {
    def iterator(): Iterator[T]
}

trait Iterator[T] {
    def hasNext: Boolean
    def next(): T
}
```

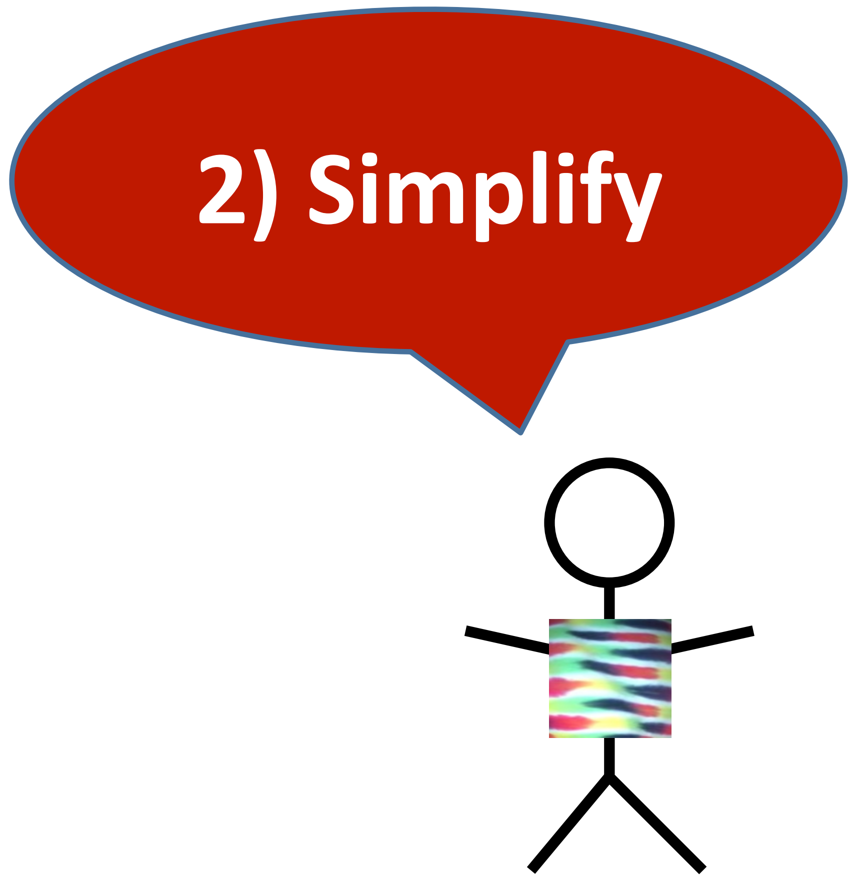
```
()⇒
(()⇒Try[Option[T]])
```

The return of the magic dualization trick

```
()⇒(()⇒Try[Option[T]])

(Try[Option[T]]⇒Unit)⇒Unit
```

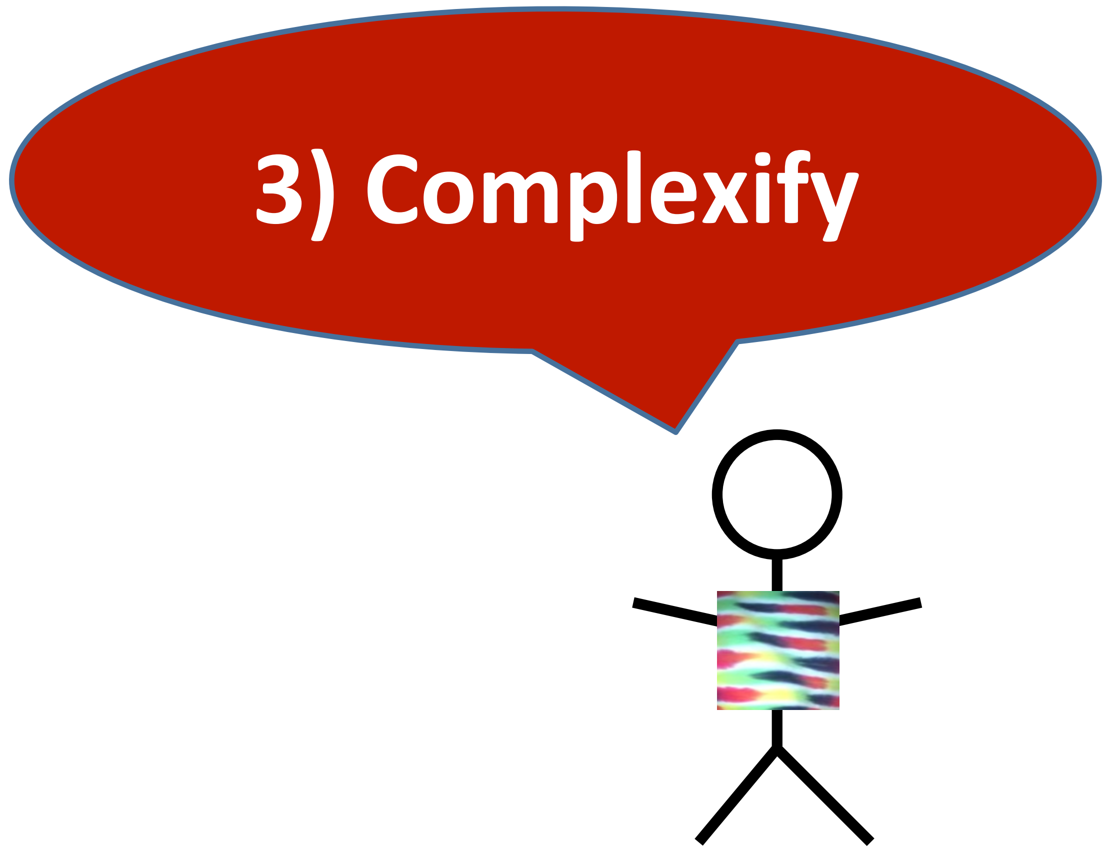



```
( T⇒Unit,
, Throwable⇒Unit
, ()⇒Unit
)⇒Unit
```



The return of the magic dualization trick

```
( T⇒Unit,
, Throwable⇒Unit
, Unit⇒Unit
)⇒Unit
```




```scala
trait Observable[T] {
   def Subscribe(observer: Observer[T]):
          Subscription
}

trait Observer[T] {
   def onNext(value: T): Unit
   def onError(error: Throwable): Unit
   def onCompleted(): Unit
}

trait Subscription {
   def unsubscribe(): Unit
}
```

Iterable[T] and Observable[T] are dual

```
trait Iterable[T] {
   def iterator: Iterator[T]
}

trait Iterator[T] {
   def next(): T

   def hasNext: Boolean
}
```


```
trait Observable[T] {
   def Subscribe(Observer[T] observer):
          Subscription
}

trait Observer[T] {
   def onNext(T value): Unit
   def onError(Throwable error): Unit
   def onCompleted(): Unit
}

trait Subscription {
   def unsubscribe(): Unit
}
```

The Four Essential Effects In Programming

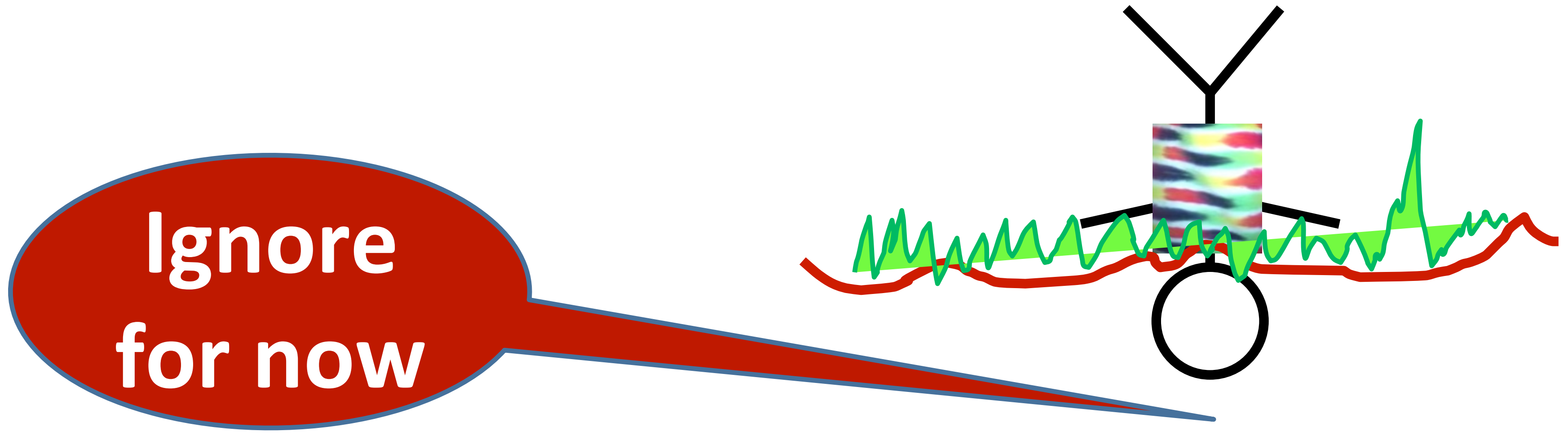
	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

Future versus Observable

```
Observable[T] = (Try[Option[T]]⇒Unit)⇒Unit

Future[T]     = (Try[       T] ⇒Unit)⇒Unit
```

What about concurrency?

```
object Future {
   def apply[T](body: ⇒T)
      (implicit executor: ExecutionContext): Future[T]
}

trait Observable[T] {
   def observeOn(scheduler: Scheduler): Observable[T]
}
```

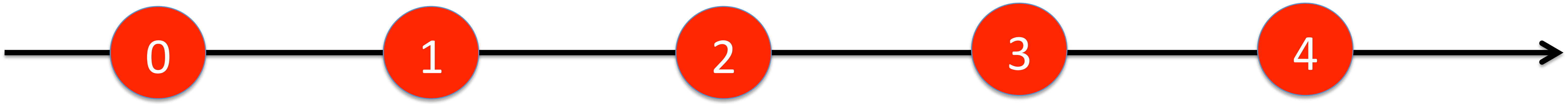


Hello Observables

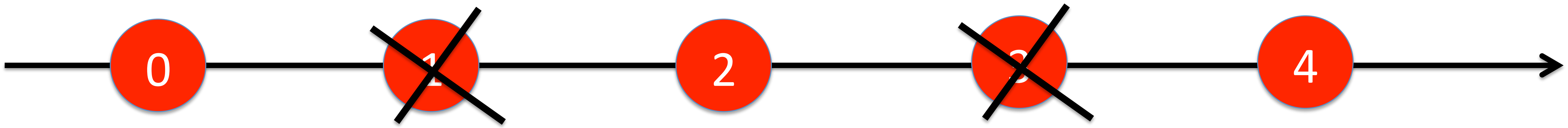
```
val ticks: Observable[Long] = Observable.interval(1 seconds)

val evens: Observable[Long] = ticks.filter(s⇒s%2==0)

val bufs: Observable[Seq[Long]] = ticks.buffer(2,1)

val s = bufs.subscribe(b⇒printLn(b))

readLine()

s.unscubscribe()
```

Hello Observables

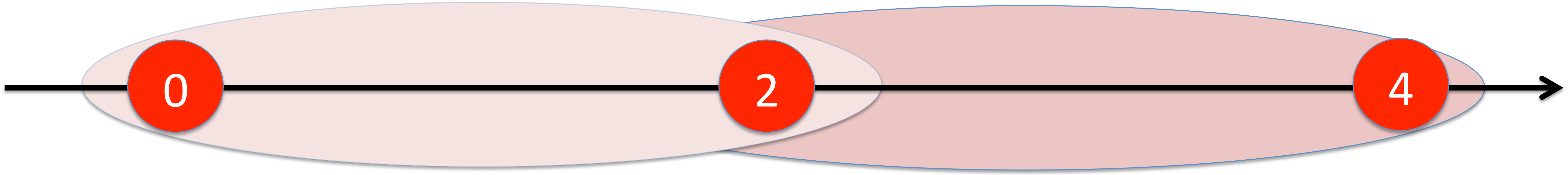
```scala
val ticks: Observable[Long] = Observable.interval(1 seconds)
```




```scala
val evens: Observable[Long] = ticks.filter(s⇒s%2==0)
```



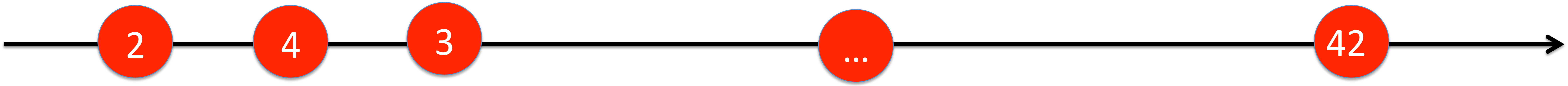

```scala
val bufs: Observable[Seq[Long]] = ticks.buffer(2,1)
```



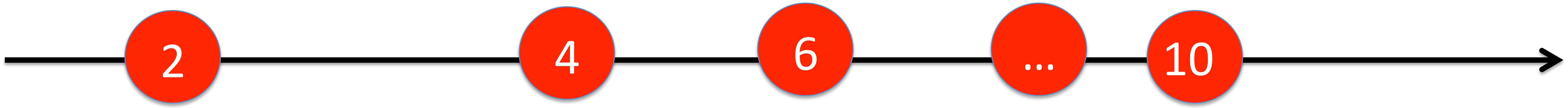
Quiz

```
val xs = Observable.range(1,10)
```


```
val ys = xs.map(x=>x+1)
```

A) 

B) 

C) 

D)