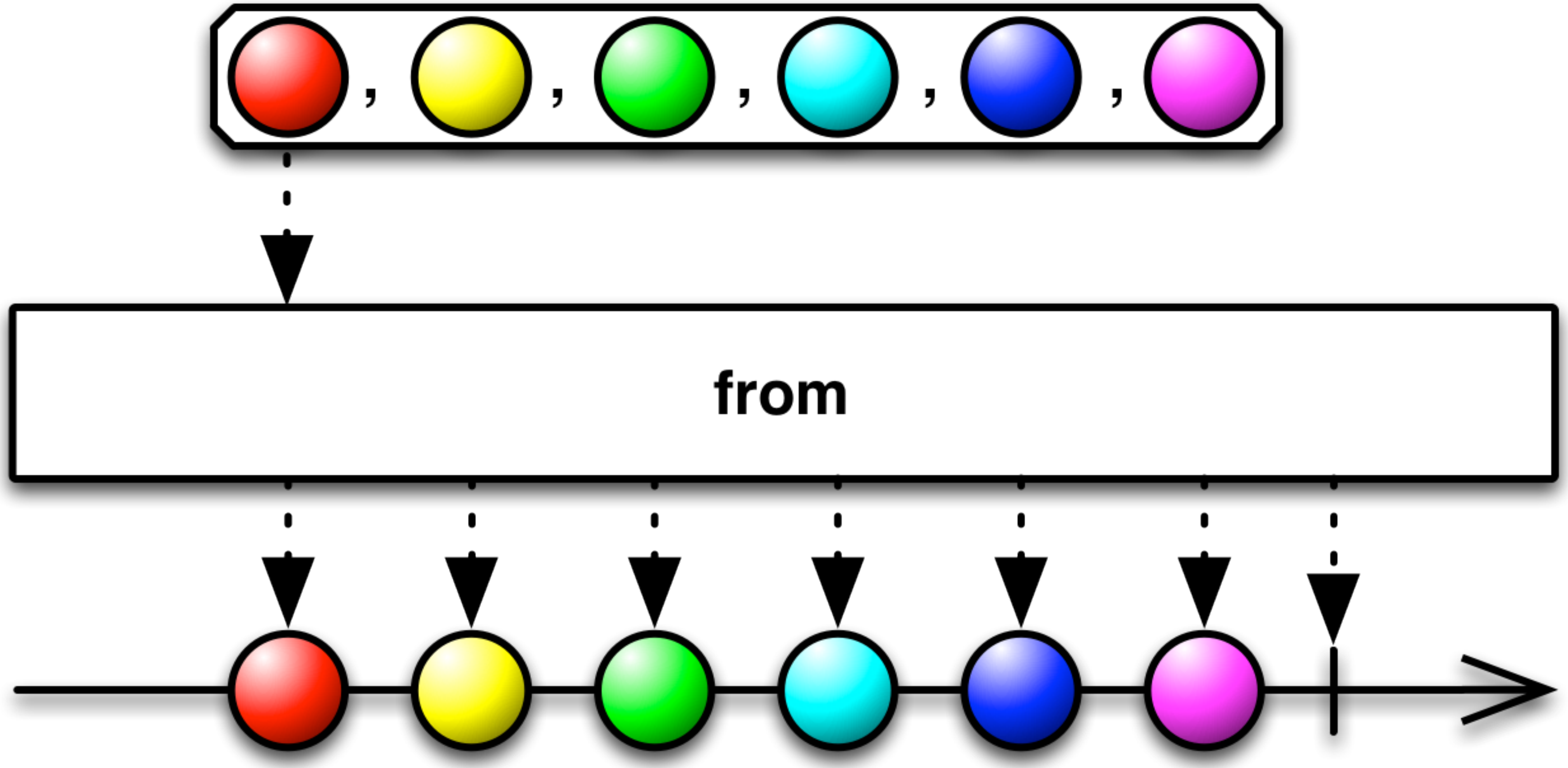
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Schedulers I

Principles of Reactive Programming

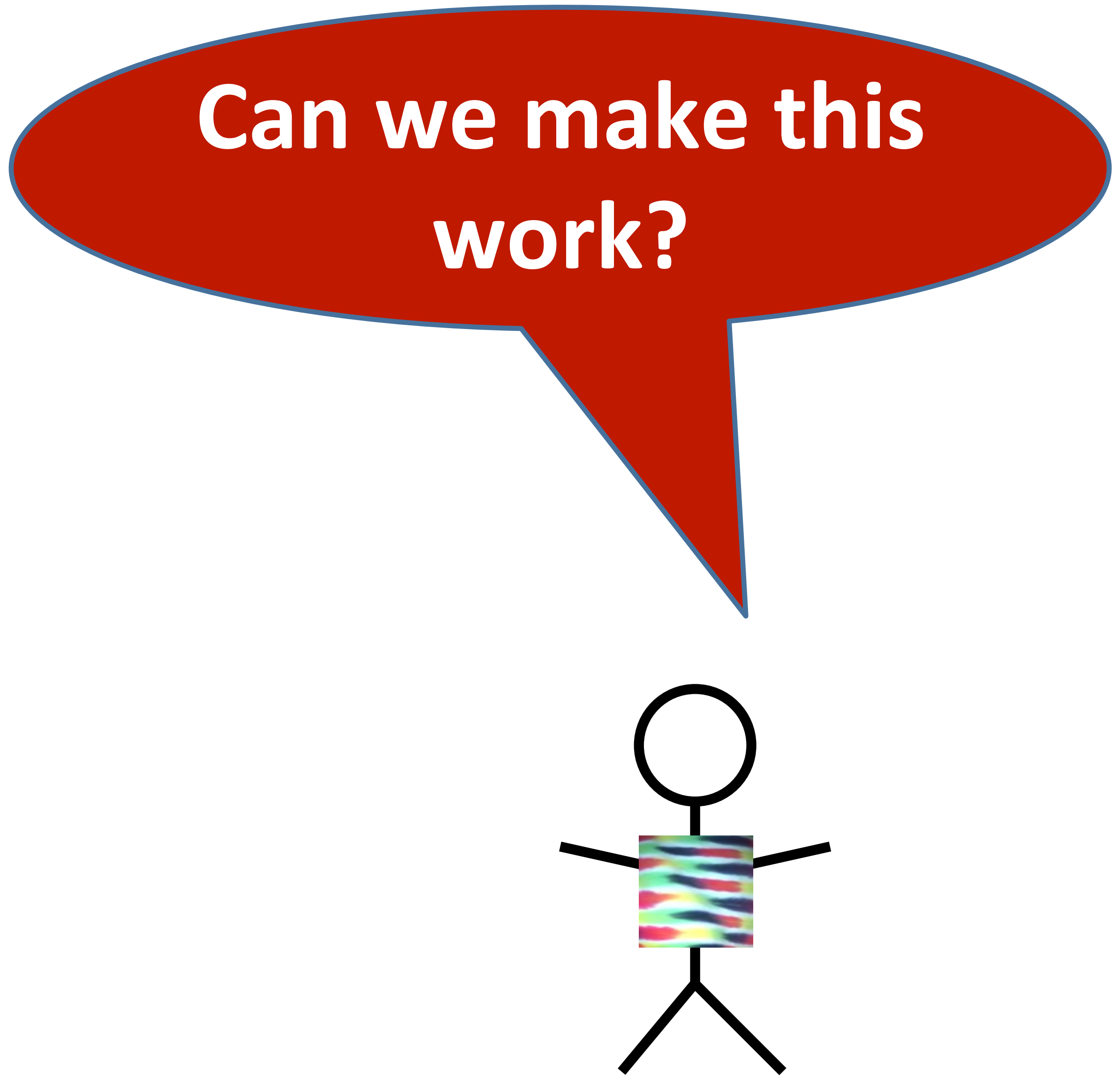
Erik Meijer

Convert Iterable to Observable



Unsubscribing from an infinite sequence

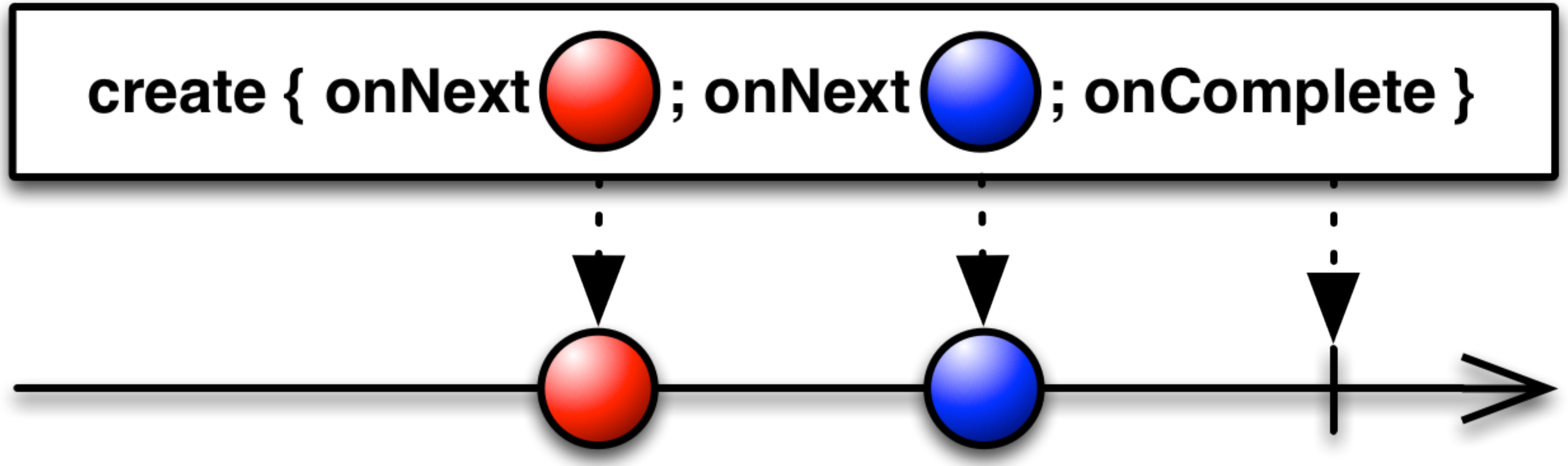
```
def from[T](seq: Iterable[T]) : Observable[T] = {…}

val infinite: Iterable[Int] = nats()

val subscription = from(infinite).
      subscribe(x⇒println(x))

subscription.unsubscribe()
```



Excursion: iterables are lazy

```
def nats(): Iterable[Int] = new Iterable[Int] {
   var i = -1
   def iterator: Iterator[Int] = new Iterator[Int] {
      def hasNext: Boolean = { true }
      def next(): Int = { i +=1; i }
   }
}
```

Mother of all factory methods



```
object Observable {
  def apply[T](subscribe: Observer[T]⇒Subscription): Observable[T]
}
```

Schedulers

```
def from[T](seq: Iterable[T]) : Observable[T] = {
   Observable(observer ⇒ {
     seq.foreach(s ⇒ observer.onNext(s))
     observer.onCompleted()


     Subscription{}
   })
}


val infinite: Iterable[Integer] = nats()
val subscription = from(infinite).subscribe(x ⇒ println(x))

subscription.unsubscribe()
```

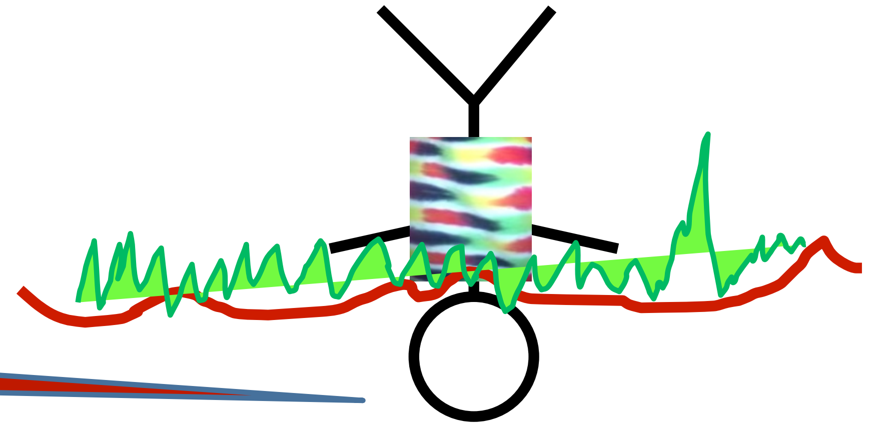
Schedulers

```
def from[T](seq: Iterable[T]) : Observable[T] = {
   Observable(observer ⇒ {
     seq.foreach(s ⇒ observer.onNext(s))
     observer.onCompleted()
     // we never get here
     Subscription{}
   })
}

val infinite: Iterable[Integer] = nats()
val subscription = from(infinite).subscribe(x ⇒ println(x))
// hence we never get here
subscription.unsubscribe()
```

We must run the producer on its own thread

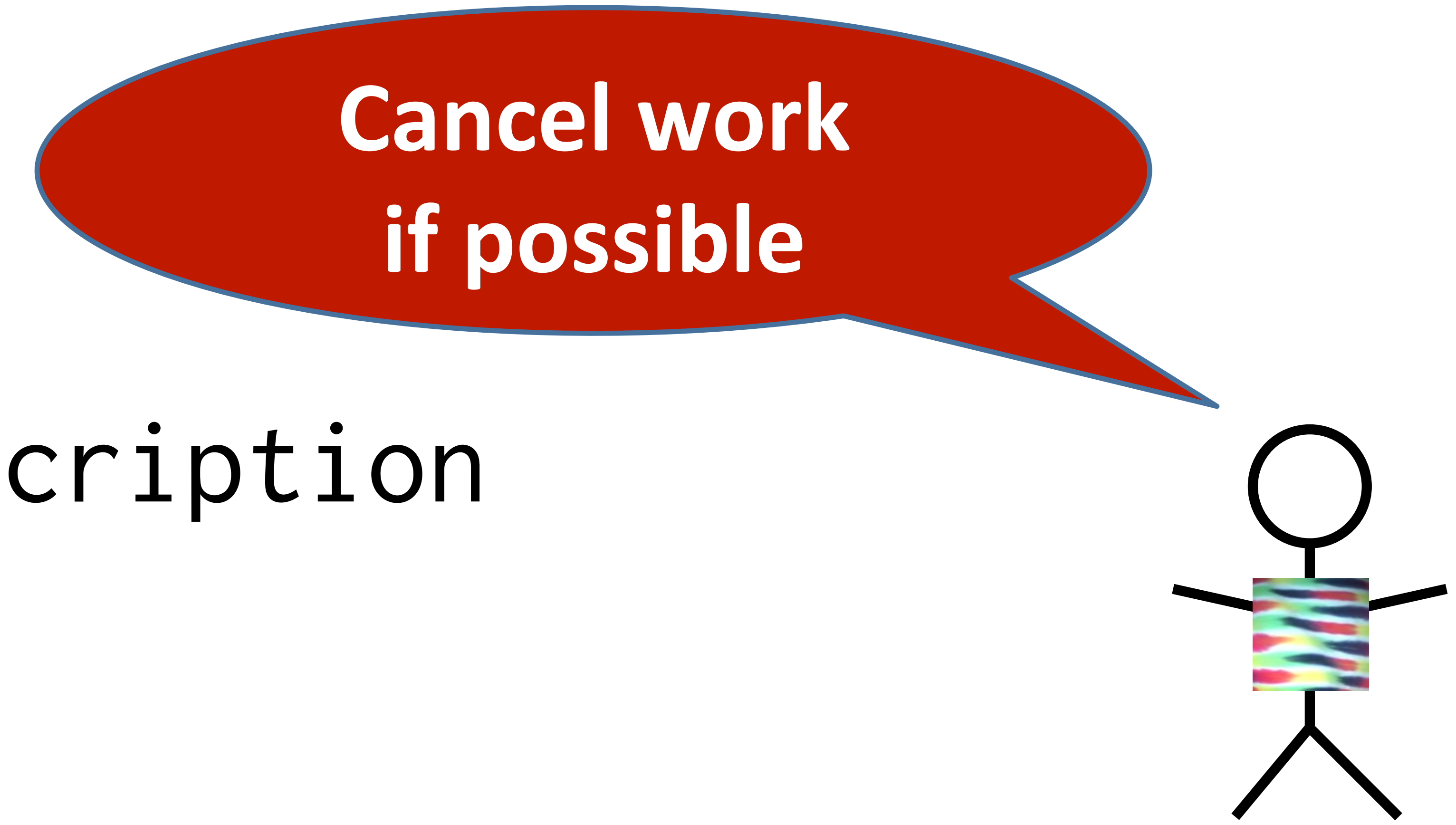
```
object Future {
   def apply[T](body: ⇒T)
      (implicit executor: ExecutionContext): Future[T]
}


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

Bite the bullet



Schedulers

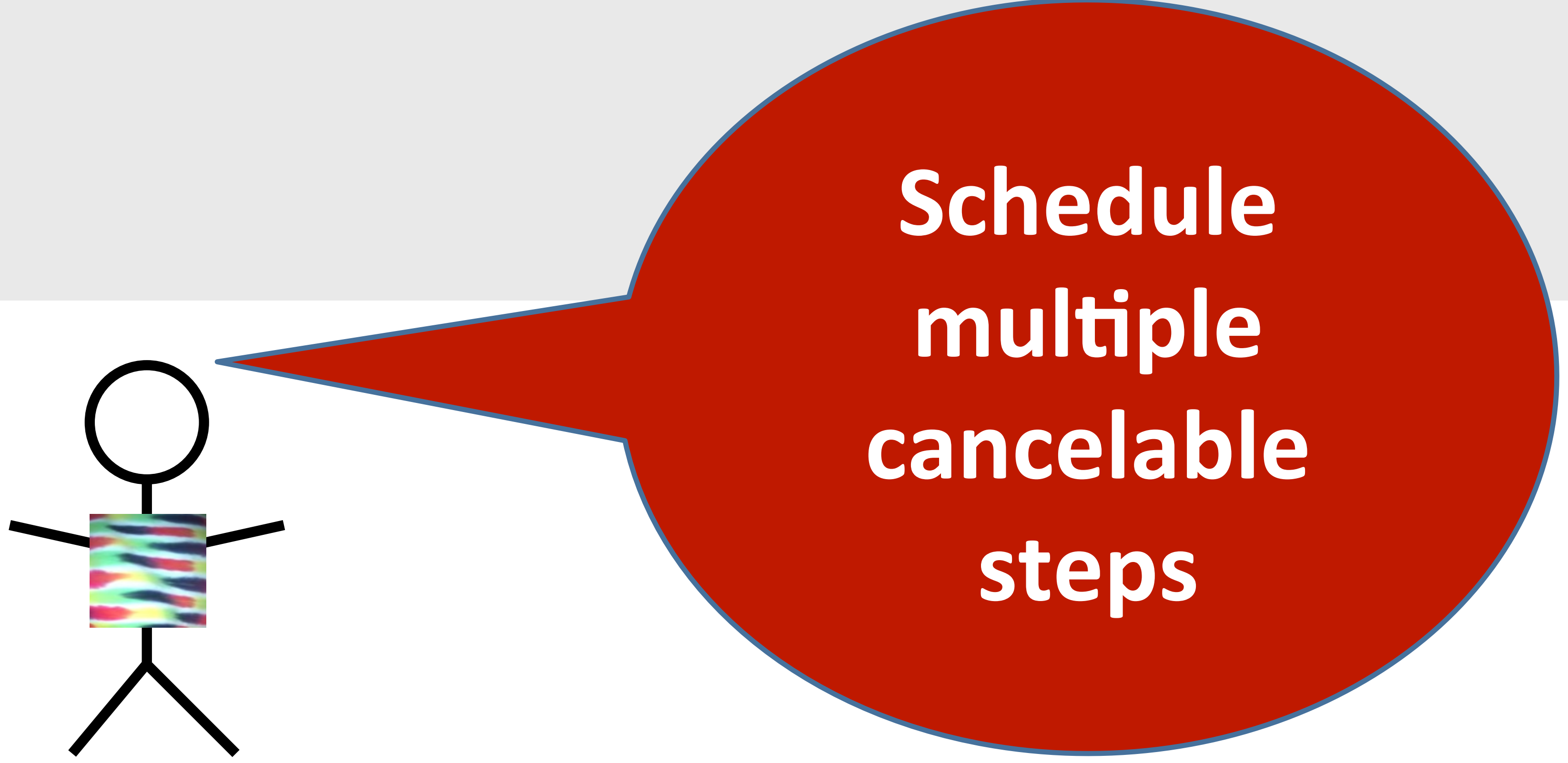
```
trait ExecutionContext {
   def execute(runnable: Runnable): Unit
}

trait Scheduler {
   def schedule(work: ⇒Unit): Subscription
}

val scheduler = Scheduler.NewThreadScheduler
val subscription = scheduler.schedule {
   println(“hello world”);
}
```



First attempt, like a Future

```
def from[T](seq: Iterable[T])
   (implicit scheduler: Scheduler): Observable[T] = {
   Observable[T](observer ⇒ {
      scheduler.schedule {
         seq.foreach(s ⇒ observer.onNext(s))
         observer.onCompleted()
      }
   })
}
```

Schedulers



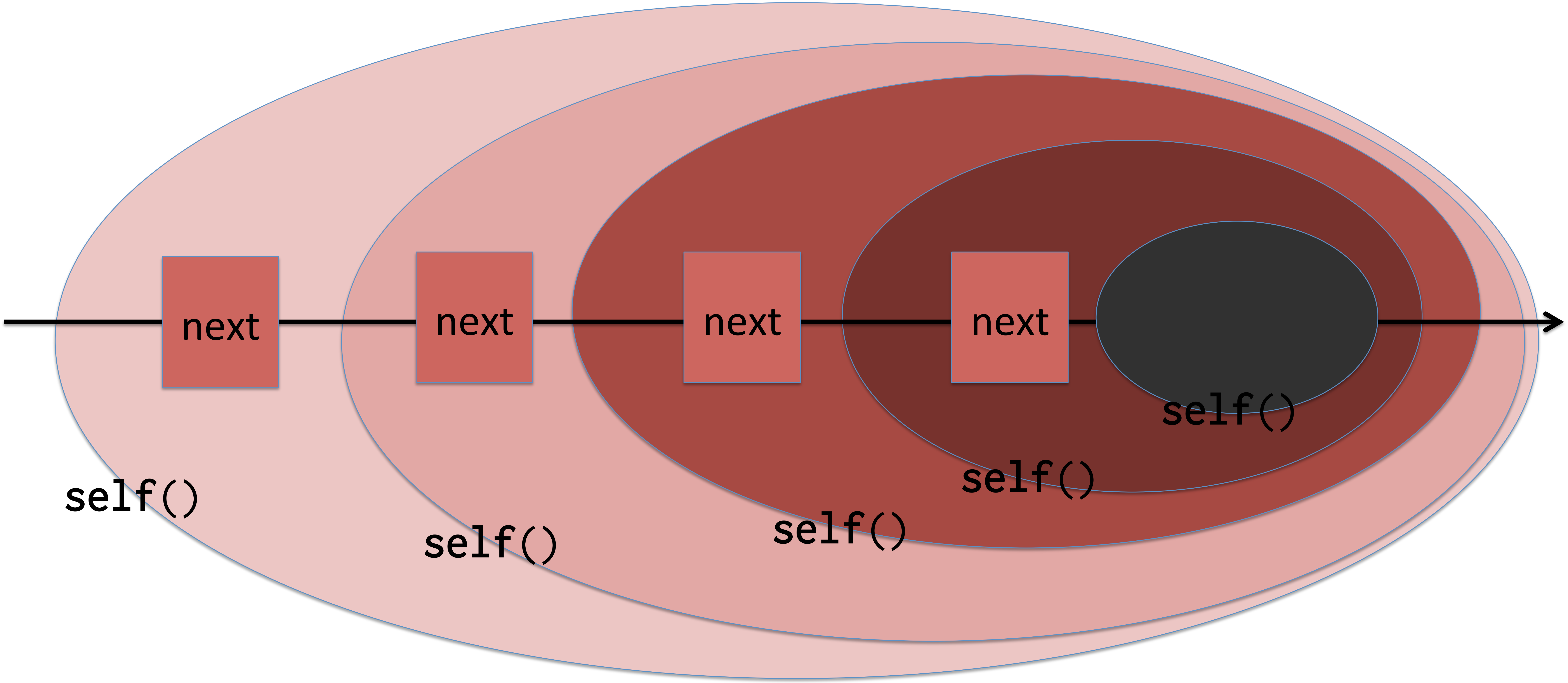
```
trait Scheduler {

   def schedule(work: ⇒Unit): Subscription

   def schedule(work: Scheduler⇒Subscription):
                                          Subscription


   def schedule(work: (⇒Unit)⇒Unit): Subscription
}
```

Second attempt

```
def from[T](seq: Iterable[T])
                  (implicit scheduler: Scheduler): Observable[T] = {
   Observable[T](observer ⇒ {

     val it = seq.iterator()

      scheduler.schedule(self ⇒ {

         if (it.hasnext) { observer.onNext(it.next()); self() }
         else { observer.onCompleted() }
      })
   })
}
```

Convert Scheduler to Observable[Unit]

```
if (it.hasnext) { observer.onNext(it.next()); self() }
```



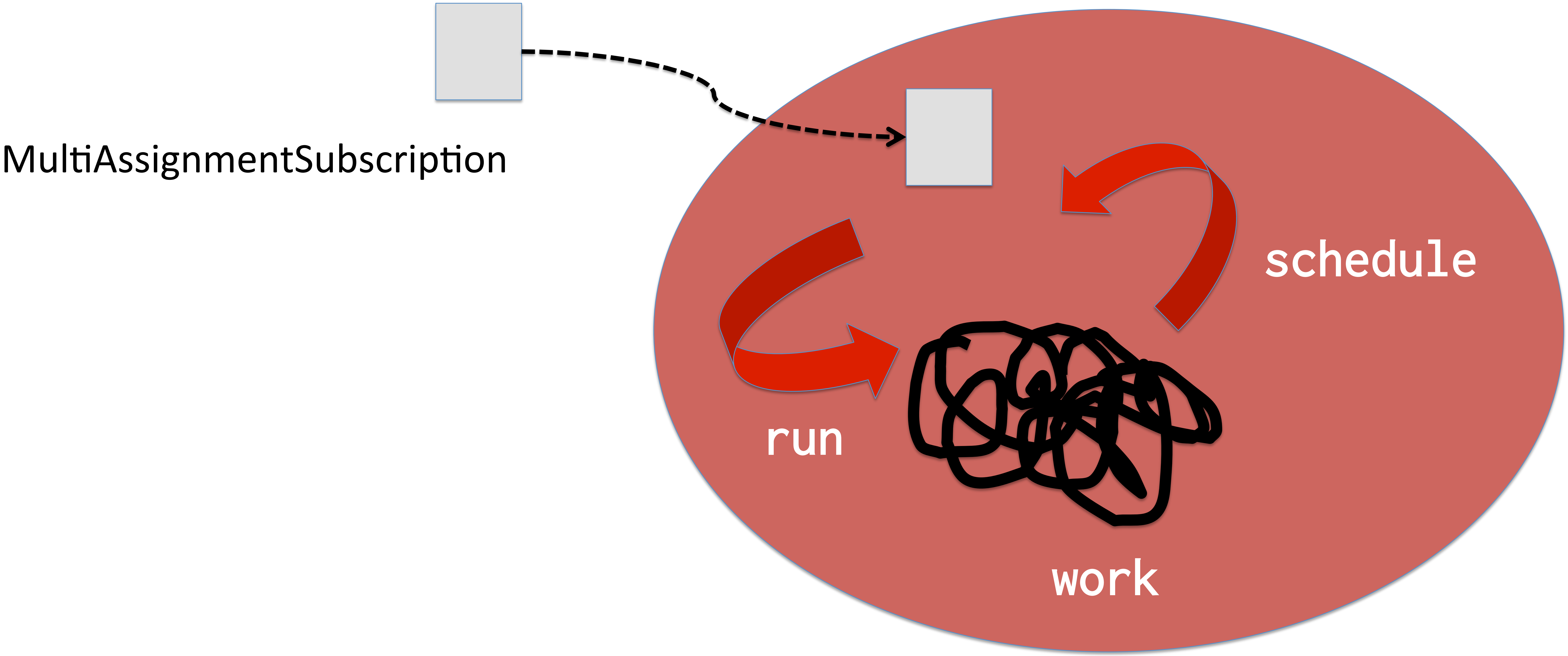
Schedulers

```
val infinite: Iterable[Integer] = nats()

val subscription = from(infinite)
                     .subscribe(x ⇒ println(x))


subscription.unsubscribe()
```

Recursive Scheduling

```
def schedule(work: (⇒Unit)⇒Unit): Subscription = {

    val subscription = new MultipleAssignmentSubscription();

    schedule(scheduler ⇒ {
        loop(scheduler, work, subscription);
        subscription;
    })
}

def loop(s: Scheduler, w: (⇒Unit)⇒Unit), m: MultipleAssignmentSubscription):
                                                                          Unit = {
   m.Subscription = s.schedule { w { loop(s, w, m) } };
}
```

Recursive Scheduling



Recursive Scheduling

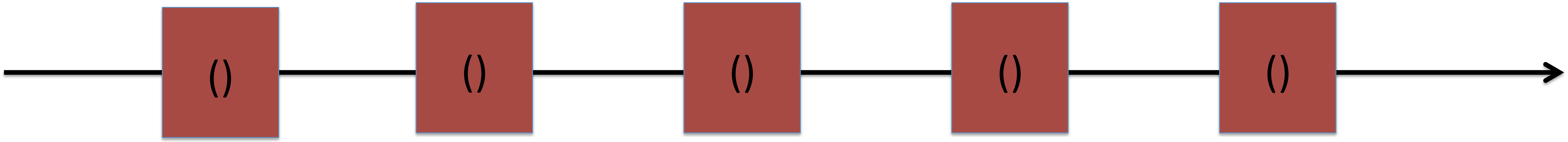
```
def schedule(work: (⇒Unit)⇒Unit): Subscription = {

    val subscription = new MultipleAssignmentSubscription();

    schedule(scheduler ⇒ {
        loop(scheduler, work, subscription);
        subscription;
    })
}

def loop(s: Scheduler, w: (⇒Unit)⇒Unit), m: MultipleAssignmentSubscription):
                                                                        Unit = {
   m.Subscription = s.schedule { w { loop(s, w, m) } };
}
```

Recursive Scheduling Unplugged

```
def schedule(work: (⇒Unit)⇒Unit): Subscription = {
   val subscription = new MultipleAssignmentSubscription()

   schedule(scheduler⇒{
      def loop(): Unit = {
         subscription.Subscription = scheduler.schedule {
            work { loop() } }
      }
      loop()
      subscription
   })
}
```

Convert Scheduler to Observable[Unit]

```
object Observable {
  def apply() (implicit scheduler: Scheduler): Observable[Unit] = {
   …
  }
}
```



Convert Scheduler to Observable[Unit] (quiz)

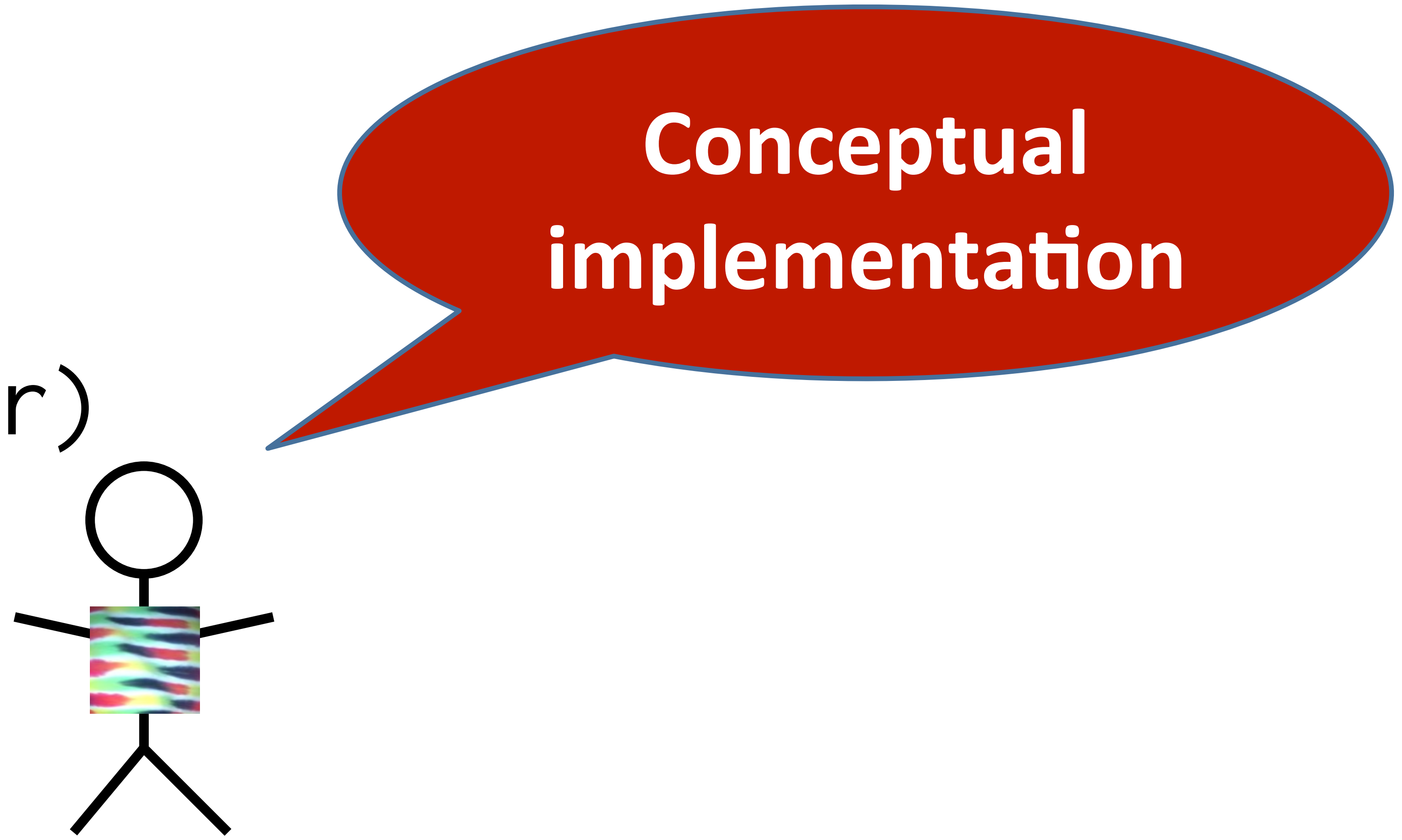
```
object Observable {
  def apply() (implicit scheduler: Scheduler): Observable[Unit] = {
   Observable(observer ⇒ {
      scheduler.schedule(self ⇒ {
         (a) observer.OnNext(()); self()
         (b) self(); observer.OnCompleted()
         (c) self(); onNext(())
         (d) onError(new Throwable(“I have no clue”))
      })
   })
  }
}
```

Convert Scheduler to Observable[Unit]

```
object Observable {
  def apply() (implicit scheduler: Scheduler): Observable[Unit] = {
   Observable(observer ⇒ {
      scheduler.schedule(self ⇒ {
          observer.OnNext(())
          self()
      })
   })
  }
}
implicit val scheduler = Scheduler.NewThreadScheduler
val ticks: Observable[Unit] = Observable()
```

Remember create

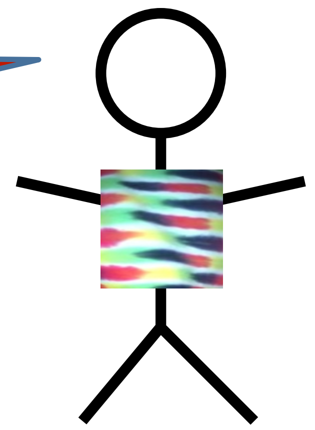
```
object Observable {
  def apply(s: Observer[T]⇒Subscription) = new Observable[T] {
    def subscribe(o: Observer[T]): Subscription = { Magic(s(o)) }
  }
}

val s = Observable(o⇒F(o)).subscribe(observer)
= conceptually
val s = Magic(F(observer))
```



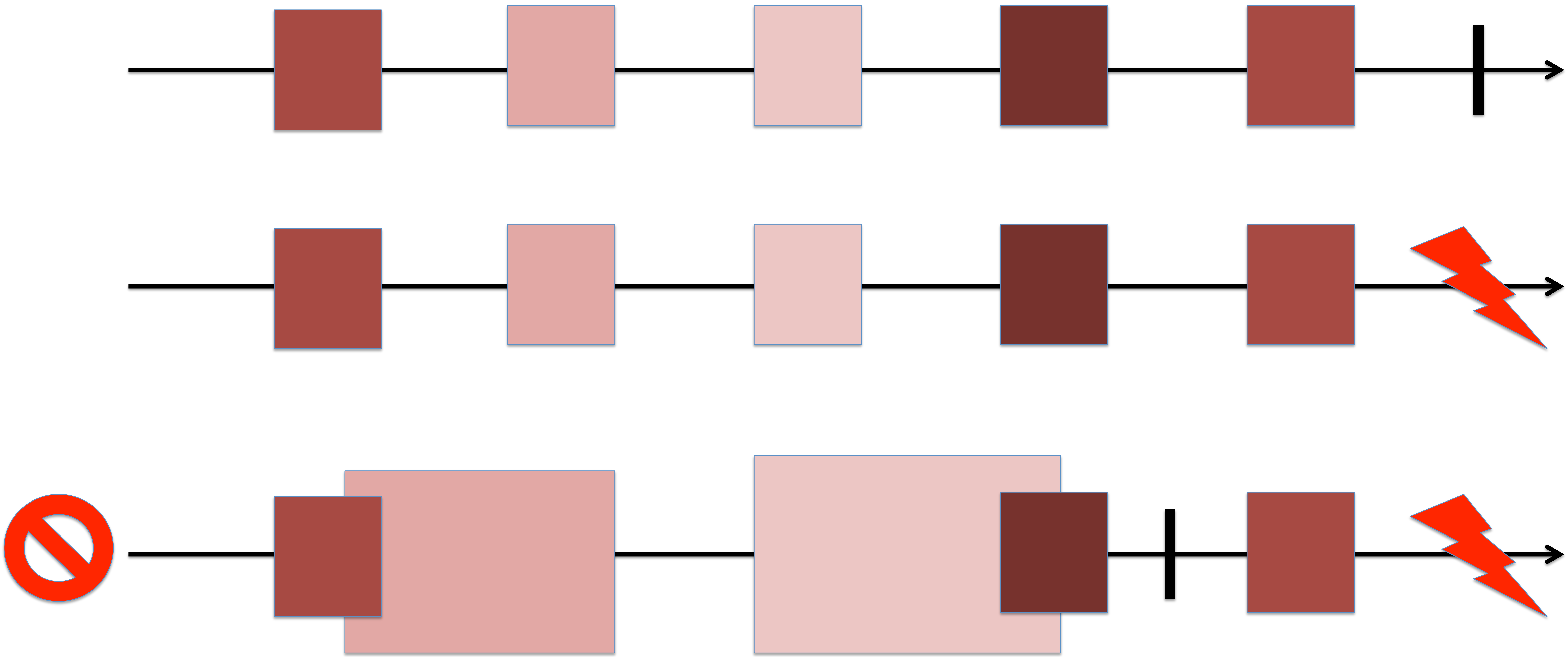
Auto-unsubscribe

```
val s = Observable(o⇒F(o)).subscribe(observer)
= conceptually
val s = Magic(F(observer))
```

When F calls observer.onCompleted or onError, s is automatically unsubscribed



Rx Contract: (onNext)*(onCompleted+onError)?



Warning

Never, ever, implement `Observable[T]` or `Observer[T]` yourself.

Always use the factory methods `Observable(…)` and `Observer(…)`