ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Schedulers II

Principles of Reactive Programming

Erik Meijer

Rx contract Quiz

```
val xs: Observable[Int] = Observable(observer => {
  observer.onNext(42)
  observer.onCompleted()
  observer.onNext(4711)
  Subscription {}
})

xs.subscribe(println(_))
```

The output of this code is:

(a) 4711, 42
(b) 4711
(c) 42, 4711
(d) 42
(e) Prints nothing

Blowing our mental stack

```
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()
```

Unfolding ticks

```
ticks
    .subscribe(observer)
= unfold ticks
Observable( {o⇒scheduler.schedule {
            self⇒o.OnNext(()); self()
}}).subscribe(observer)
```

Unfolding ticks

```
ticks
   .subscribe(observer)
= unfold ticks
Observable( {o⇒scheduler.schedule {
          self⇒o.OnNext(()); self()
}}).subscribe(observer)
= unfold create (apply)
               scheduler.schedule {
          self ⇒observer.OnNext(()); self()
}
```

Unfolding ticks

```
Observable( {o⇒scheduler.schedule {
          self⇒o.OnNext(()); self()
}}).subscribe(observer)
= unfold create (apply)
                scheduler.schedule {
          self ⇒observer.OnNext(()); self()
}
= re-arrange
scheduler.schedule { self ⇒observer.OnNext(()); self() }
```

Remember loop

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

Substitute actual work in scheduler call

```
scheduler.schedule { self ⇒observer.OnNext(()); self() }
= unfold schedule
val subscription = new MultipleAssignmentSubscription()
schedule(scheduler⇒{
  def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      { self ⇒observer.OnNext(()); self() }({ loop() })
     }
  }
  loop()
  subscription
})
```

Apply closure

```
scheduler.schedule { self ⇒observer.OnNext(()); self() }
= unfold schedule
val subscription = new MultipleAssignmentSubscription()
schedule(scheduler⇒{
  def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
  }
  loop()
  subscription
})
```

Ready to schedule work

```
val subscription = new MultipleAssignmentSubscription()
schedule(scheduler⇒{
  def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
  }
  loop()
  subscription
})
```

Ready to schedule work

```
val subscription = new MultipleAssignmentSubscription()

  def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
  }

schedule(scheduler⇒{
  loop()
  subscription
})
```

Move loop out

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}

schedule(scheduler⇒{ loop(); subscription })
```

Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}

loop(); subscription
```

Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
loop(); subscription
```

= *unfold loop()*

```
subscription.Subscription = scheduler.schedule {
   observer.OnNext(()); loop()
}; subscription
```

Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
subscription.Subscription = scheduler.schedule {
   observer.OnNext(()); loop()
}; subscription
```

= *schedule work and assign subscription*

```
subscription ---> { observer.OnNext(()); loop() }
```

Unfolding ticks

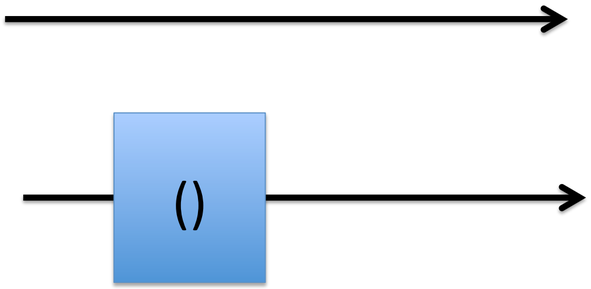
```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
subscription ---> { observer.OnNext(()); loop() }
```

Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
subscription ---> { observer.OnNext(()); loop() }
```

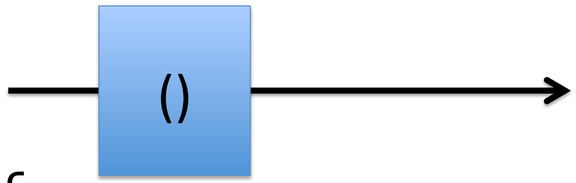
= *send tick to observer*

```
subscription ---> { loop() }
```



Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
subscription ---> { loop() }
= unfold loop()
subscription ---> {
      subscription.Subscription = scheduler.schedule {
          observer.OnNext(()); loop()
}}
```

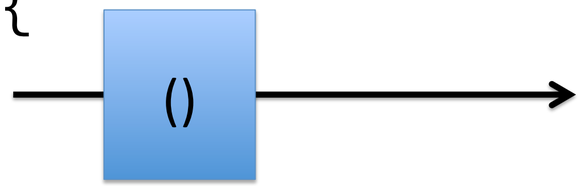


Unfolding ticks

```
val subscription = new MultipleAssignmentSubscription()
def loop(): Unit = {
    subscription.Subscription = scheduler.schedule {
      observer.OnNext(()); loop()
    }
}
subscription ---> {
      subscription.Subscription = scheduler.schedule {
          observer.OnNext(()); loop()
}}
```

= *unfold schedule and re-assign*

```
subscription ---> { observer.OnNext(()); loop() }
```



Range

```
implicit val scheduler: Scheduler = Scheduler.NewThreadScheduler

def range(start: Int, count: Int): (implicit s: Scheduler)
                                                  Observable[Int] = {
   Observable(observer ⇒ {
      var i = 0
      Observable().subscribe(u ⇒ {
         if(i < count) { observer.onNext(start+i); i += 1 }
         else { observer.onCompleted() }
      })
   })
}
```

Using range

```
implicit val scheduler: Scheduler = Scheduler.NewThreadScheduler

val xs = range(1, 10)
xs.subscribe(x ⇒ println(x))
```

```
1
2
3
…
10
```