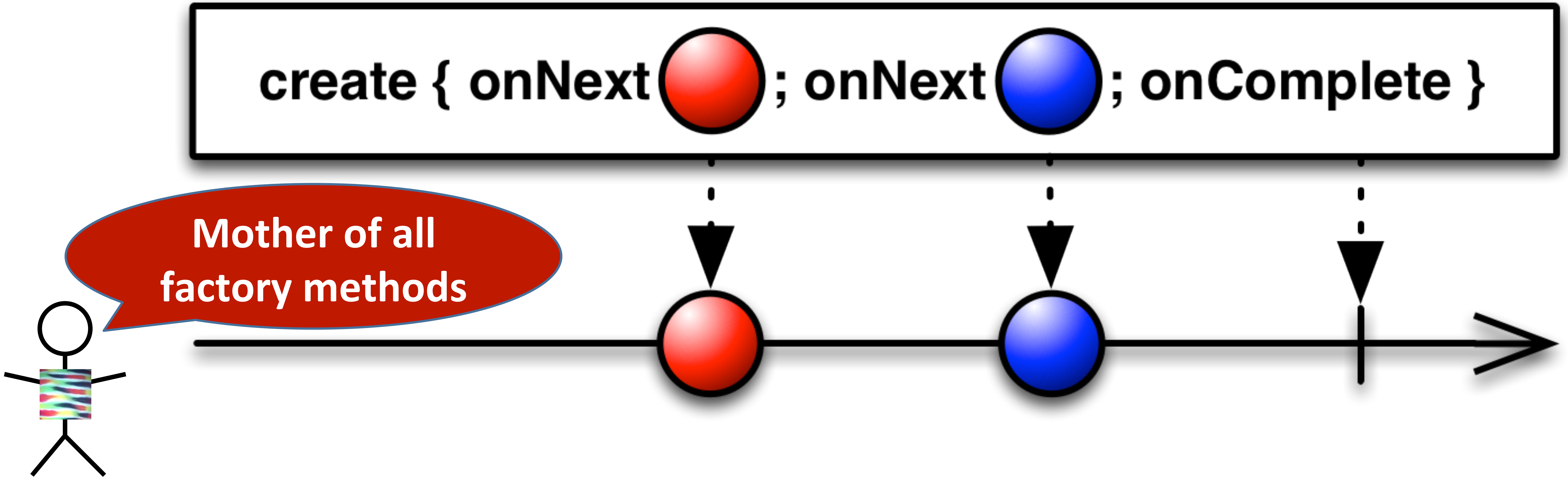
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Creating Rx Streams

Principles of Reactive Programming

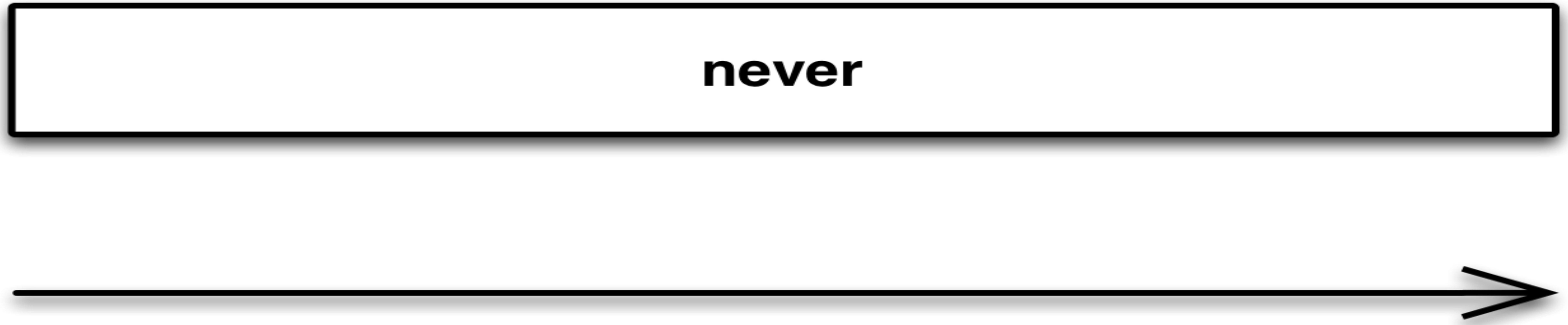
Erik Meijer

Creating Observables : “Observable.create”


create { onNext ; onNext ; onComplete }
Mother of all factory methods

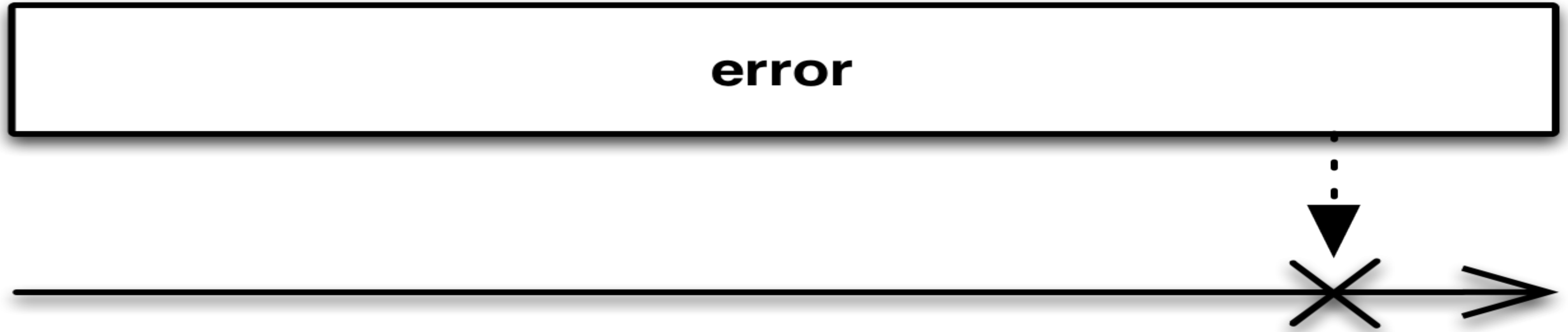
```
object Observable {
  def apply[T](s: Observer[T] ⇒ Subscription): Observable[T]
}
```

Creating Observables – never: Observable[Nothing]



never

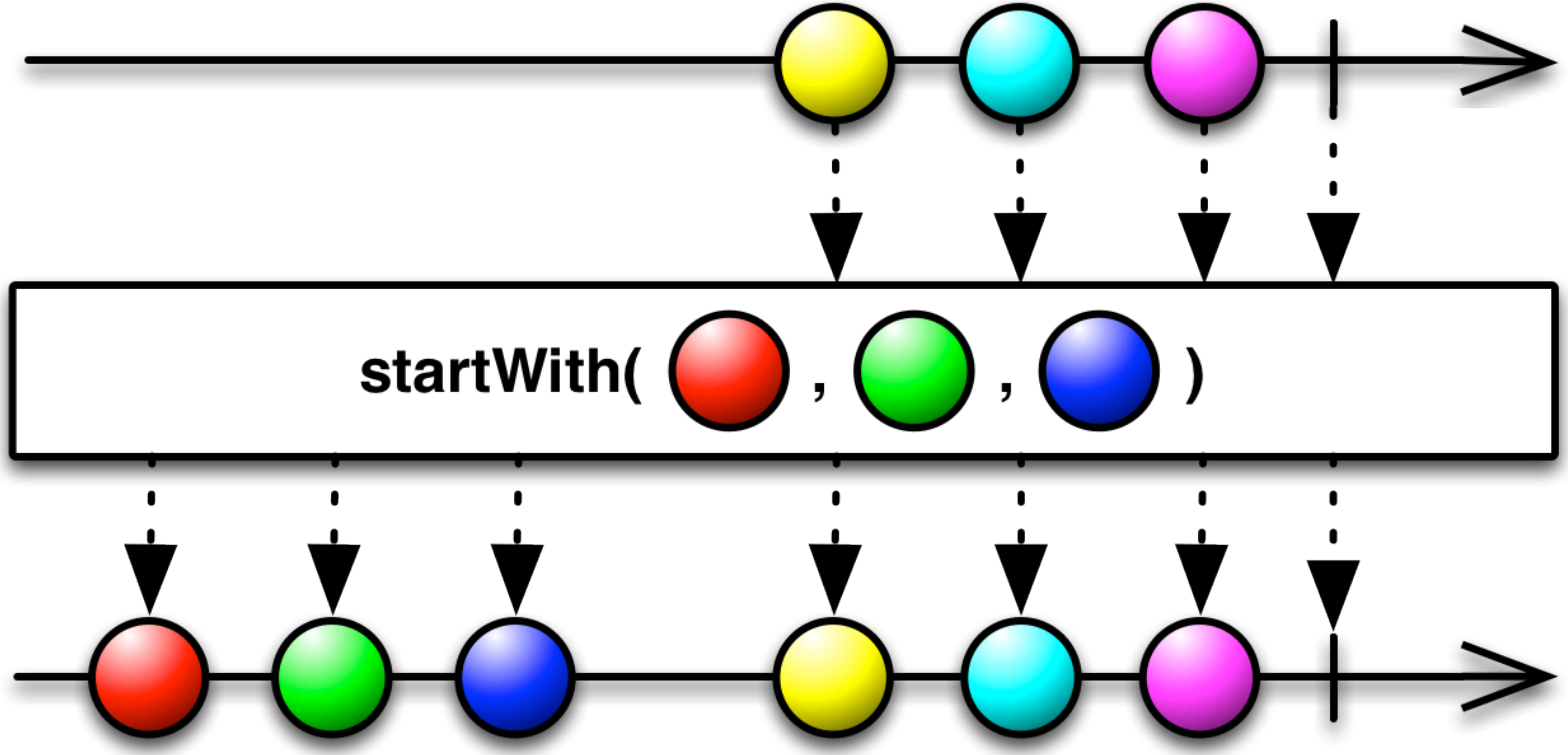
Creating Observables – error: Observable[T]



Creating Observables

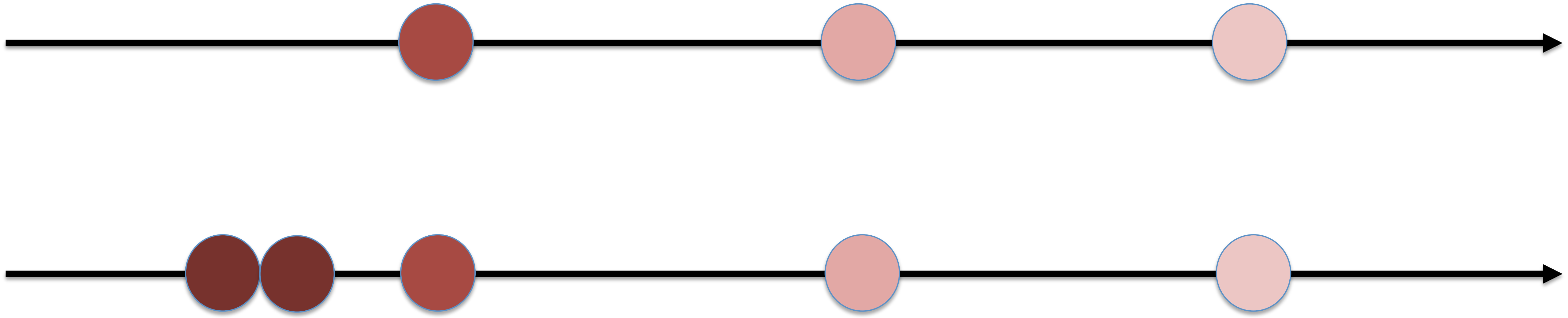
```
object Observable {
  def apply[T](subscribe: Observer[T] ⇒ Subscription): Observable[T]
}


def never(): Observable[Nothing] = Observable[Nothing](observer ⇒ {
  Subscription {}
})


def apply[T](error: Throwable): Observable[T] =
   Observable[T](observer ⇒ {
     observer.onError(error)
     Subscription {}
})
```

Creating Observables

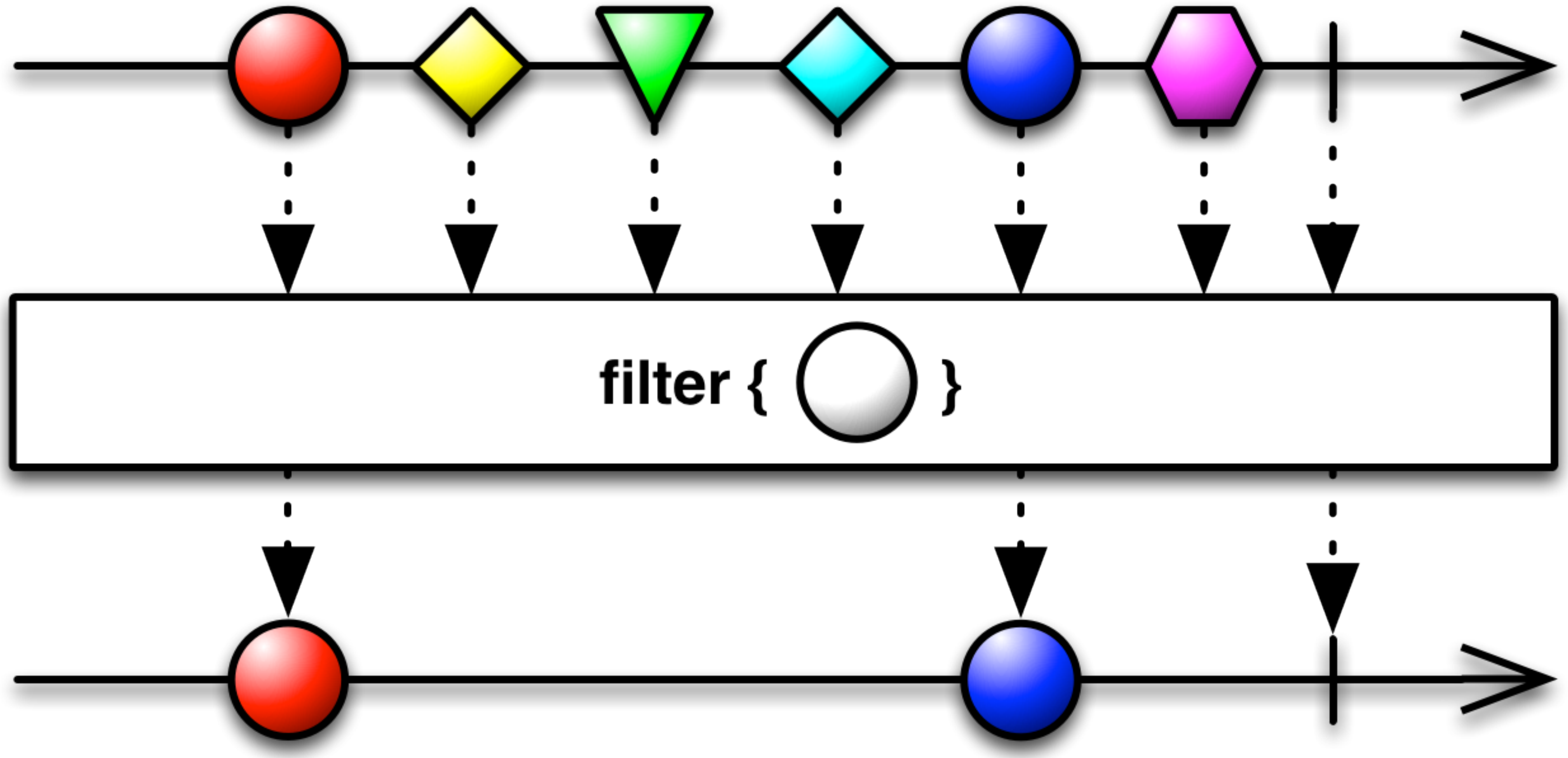


Creating Observables

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


def startWith(ss: T*): Observable[T] = {
  Observable[T](observer ⇒ {
     for(s <- ss) observer.onNext(s)
     subscribe(observer)
  })
})
```



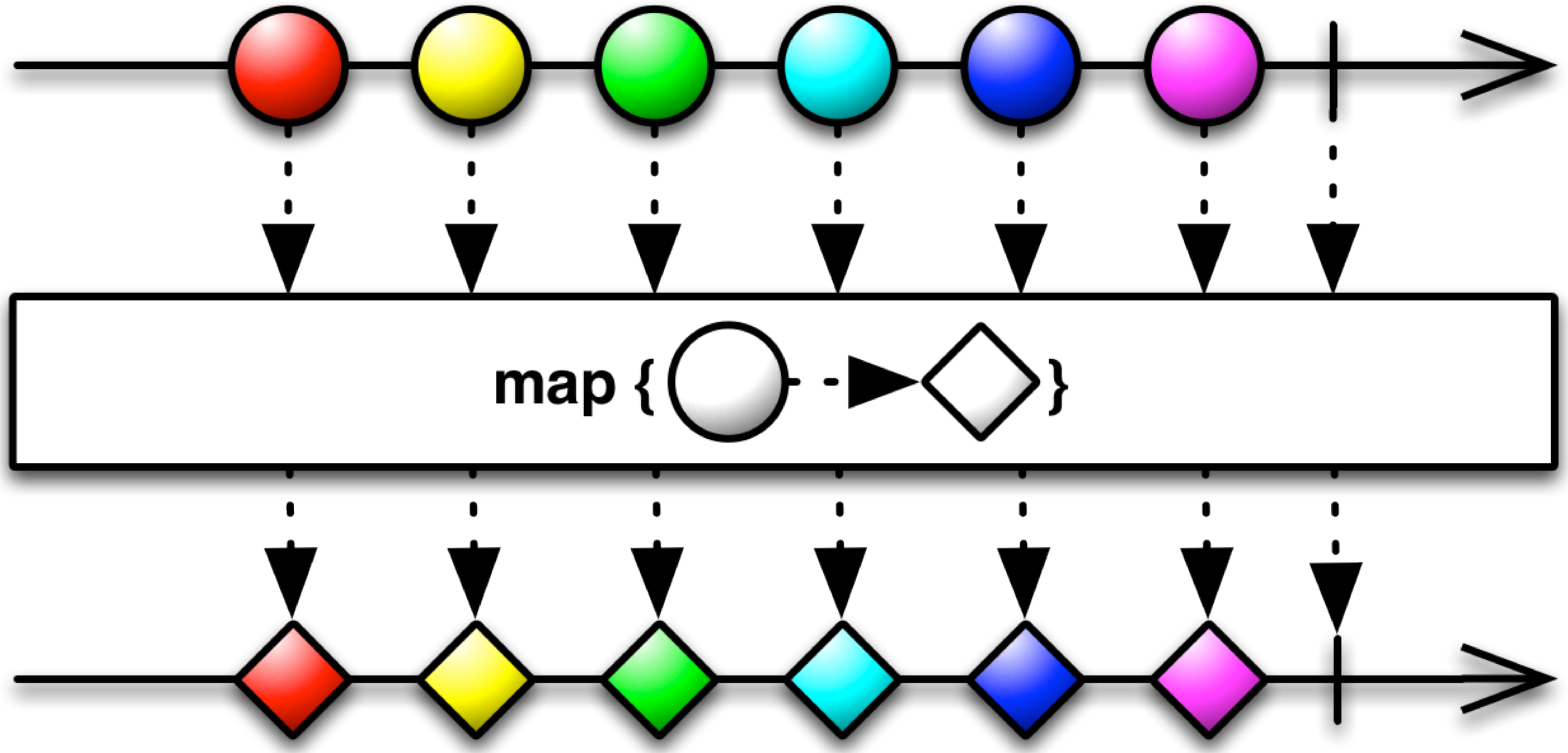
Creating Observables: filter



Creating Observables

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


def filter(p: T ⇒ Boolean): Observable[T] = {
   Observable[T](observer ⇒ {
     subscribe (
         (t: T) ⇒ { if(p(t)) observer.onNext(t) },
         (e: Throwable) ⇒ { observer.onError(e) },
         () ⇒ { observer.onCompleted() }
      )
   })
}
```

Creating Observables: map



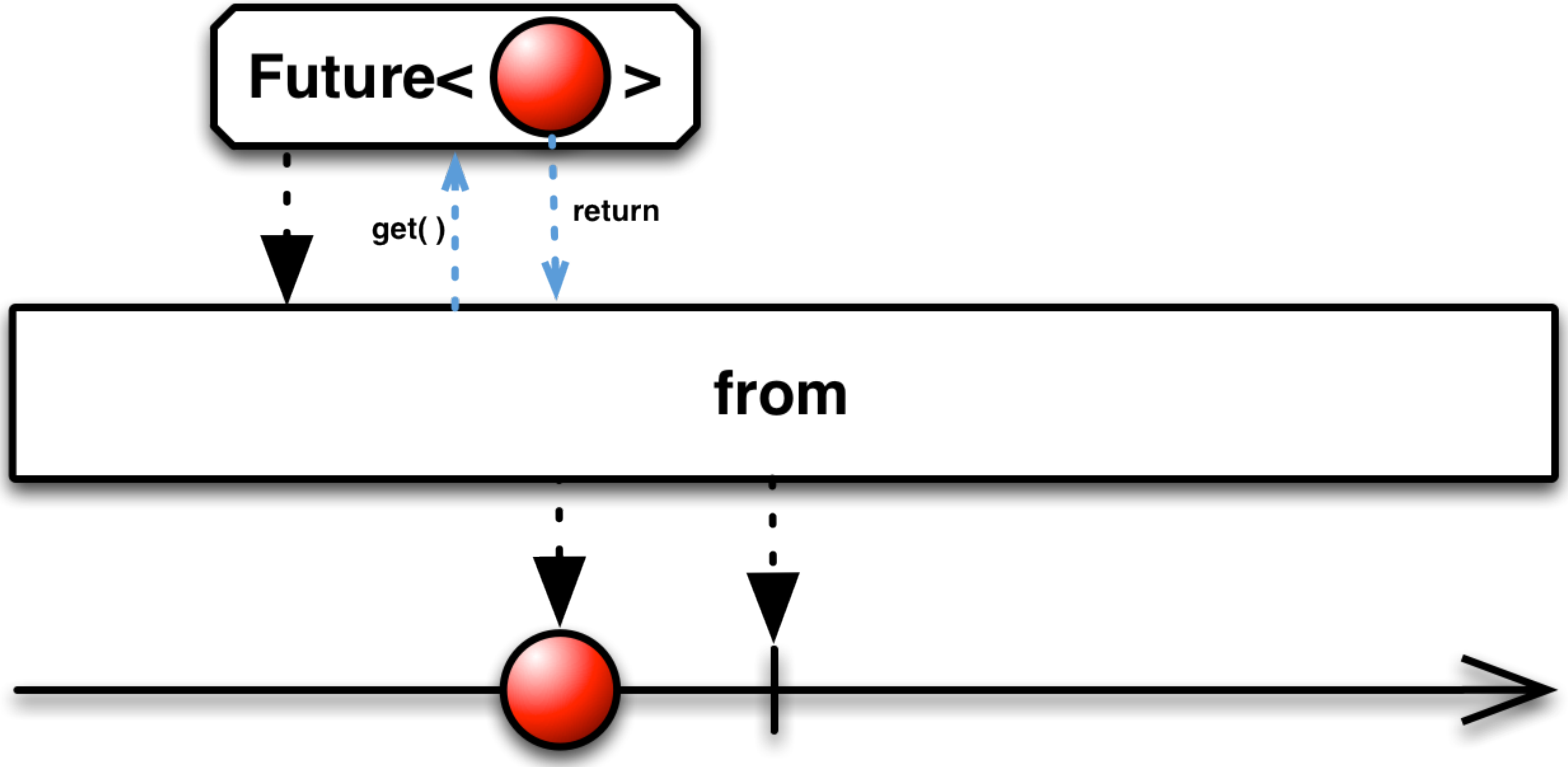
Creating Observables

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


def map[S](f: T ⇒ S): Observable[S] = {
   Observable[S](observer ⇒ {
      subscribe (
         (t: T) ⇒ { observer.onNext(f(t)) },
         (e: Throwable) ⇒ { observer.onError(e) },
         () ⇒ { observer.onCompleted() }
      )
   })
}
```

Creating Iterables: map

```
def map[S](f: T => S): Iterable[S] = {
    new Iterable[S] {
        val it = this.iterator()
        def iterator: Iterator[S] = new Iterator[S] {
            def hasNext: Boolean = { it.hasNext }
            def next(): S = { f(it.next()) }
        }
    }
}
```

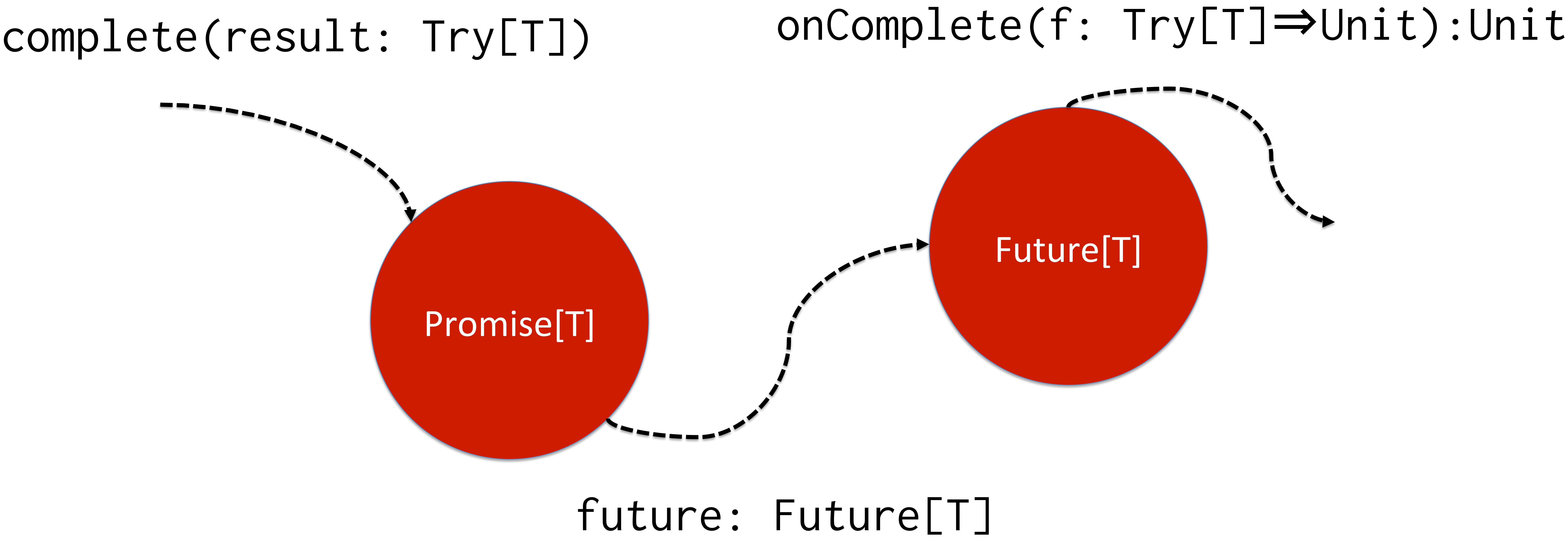
Creating Observables: from a Future[T]



Subjects & Promises

```
def map[S](f: T ⇒ S)
  (implicit executor: ExecutionContext): Future[S] = {
   val p = Promise[S]()

   onComplete {
      case result => {… p.complete(E) …}
   }(executor)

   p.future
}
```

Promise[T] recap



`complete(result: Try[T])`

Promise[T]

`future: Future[T]`

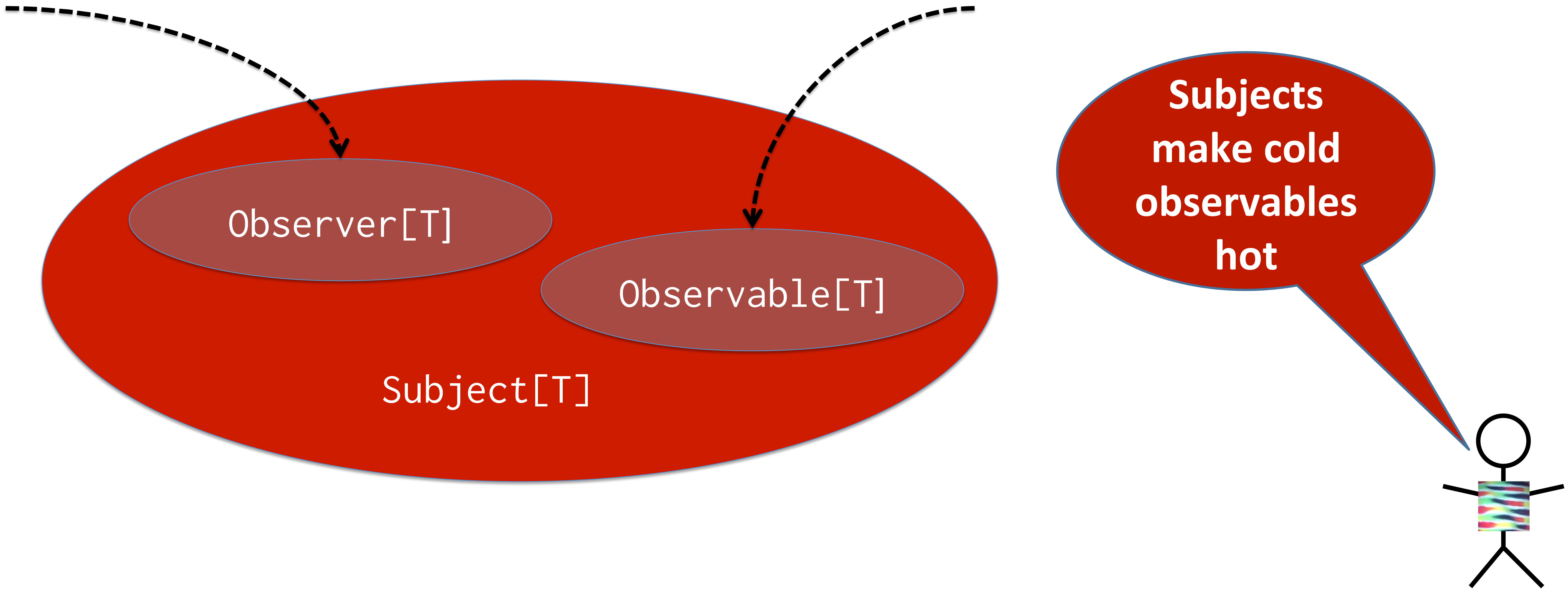
Future[T]

`onComplete(f: Try[T]⇒Unit):Unit`

Subject[T]

```
onNext(value: T)
onCompleted()
onError(e: Throwable)
```

```
subscribe(o: Observer[T])
```



Observer[T]

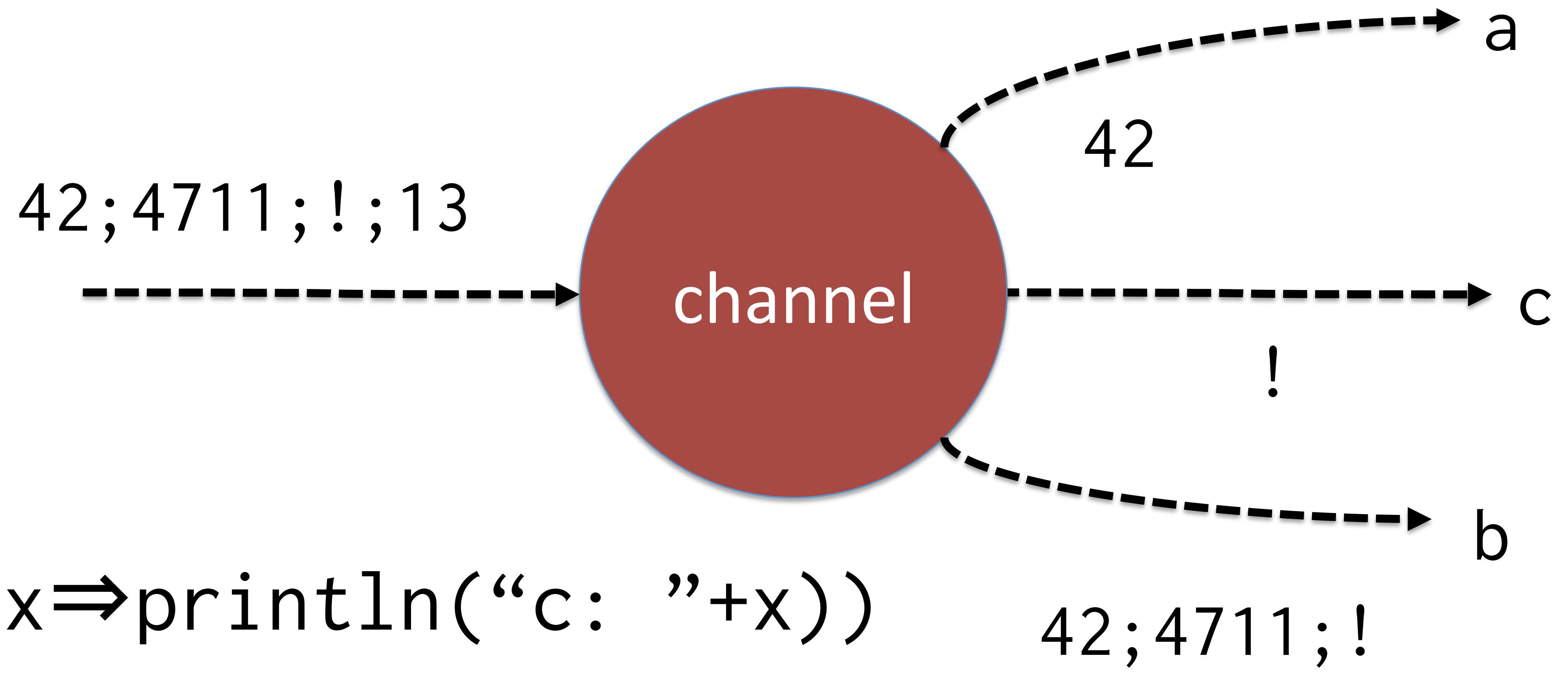
Observable[T]

Subject[T]

Subjects make cold observables hot

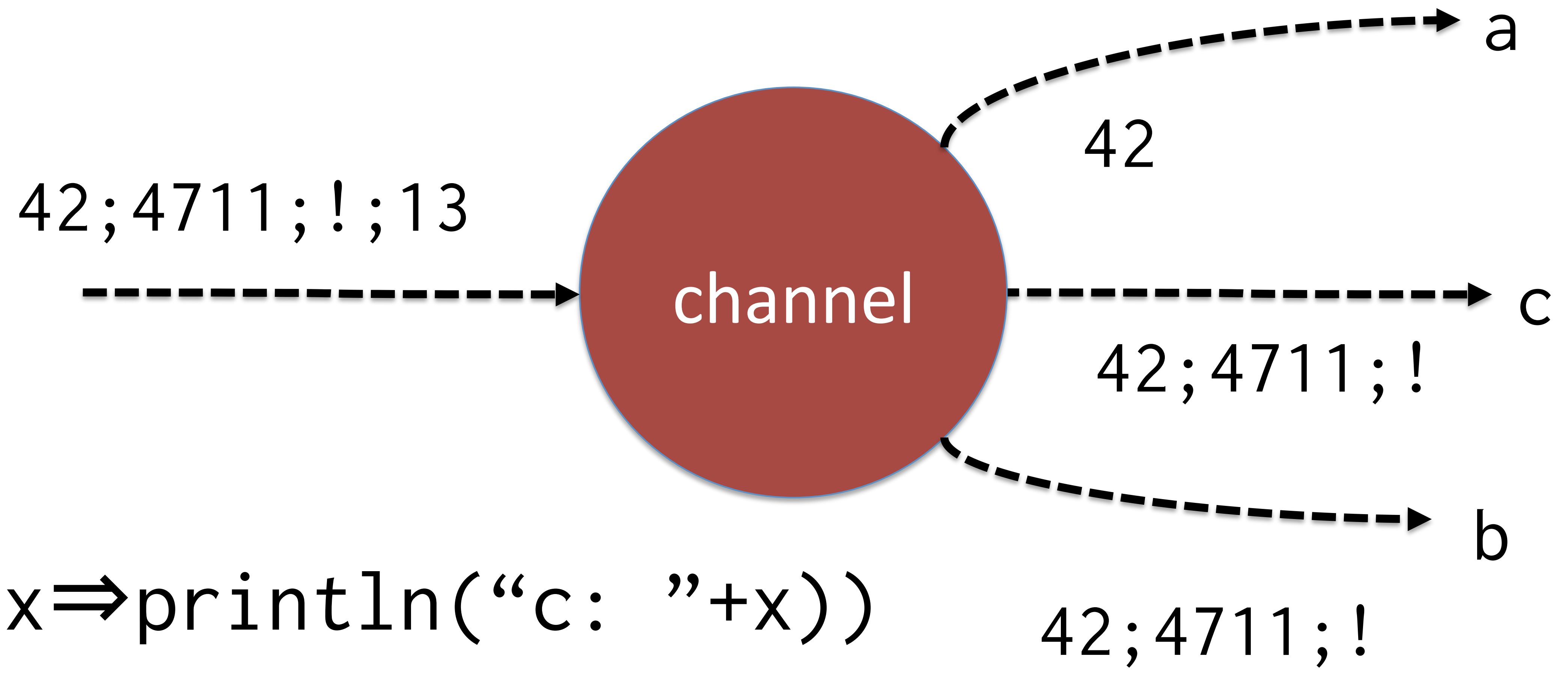
Example: Subjects are like channels

```
val channel = PublishSubject[Int]()

val a = channel.subscribe(x⇒println(“a: ”+x))
val b = channel.subscribe(x⇒println(“b: ”+x))

channel.onNext(42)
a.unsubscribe()
channel.onNext(4711)
channel.onCompleted()
val c = channel.subscribe(x⇒println(“c: ”+x))
channel.onNext(13)
```

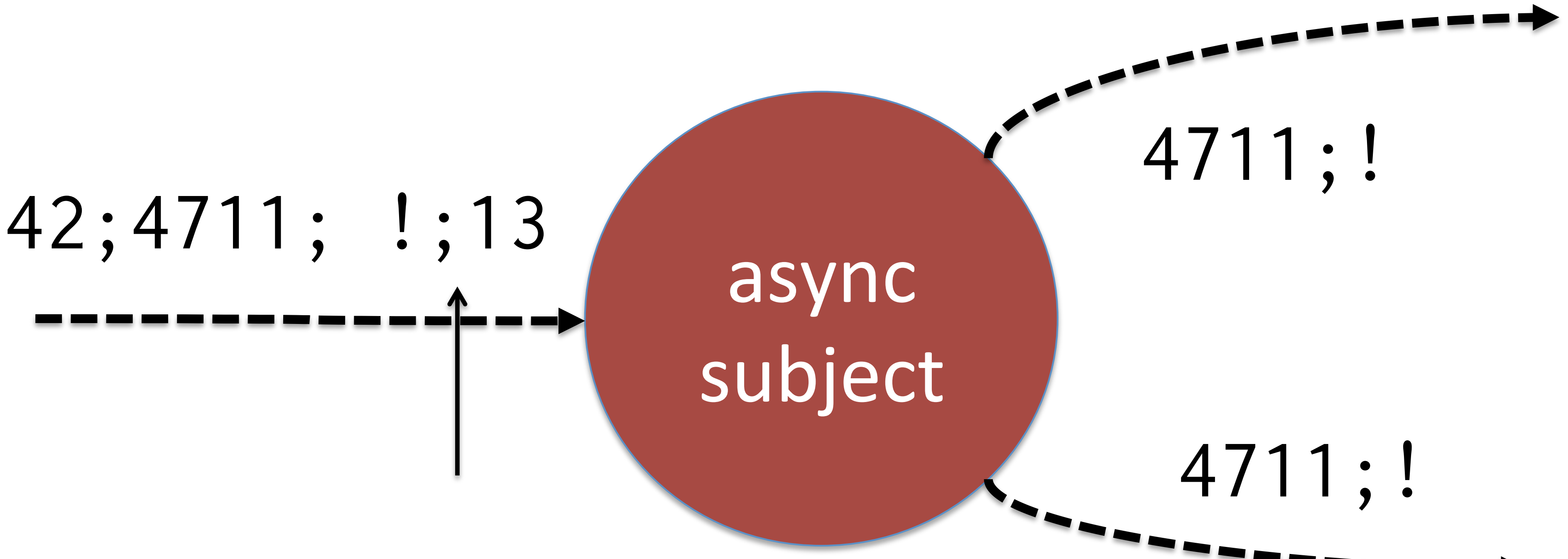


Example: Subjects are like channels

```
val channel = ReplaySubject[Int]()

val a = channel.subscribe(x⇒println(“a: ”+x))
val b = channel.subscribe(x⇒println(“b: ”+x))

channel.onNext(42)
a.unsubscribe()
channel.onNext(4711)
channel.onCompleted()
val c = channel.subscribe(x⇒println(“c: ”+x))
channel.onNext(13)
```



Subjects: meet the family



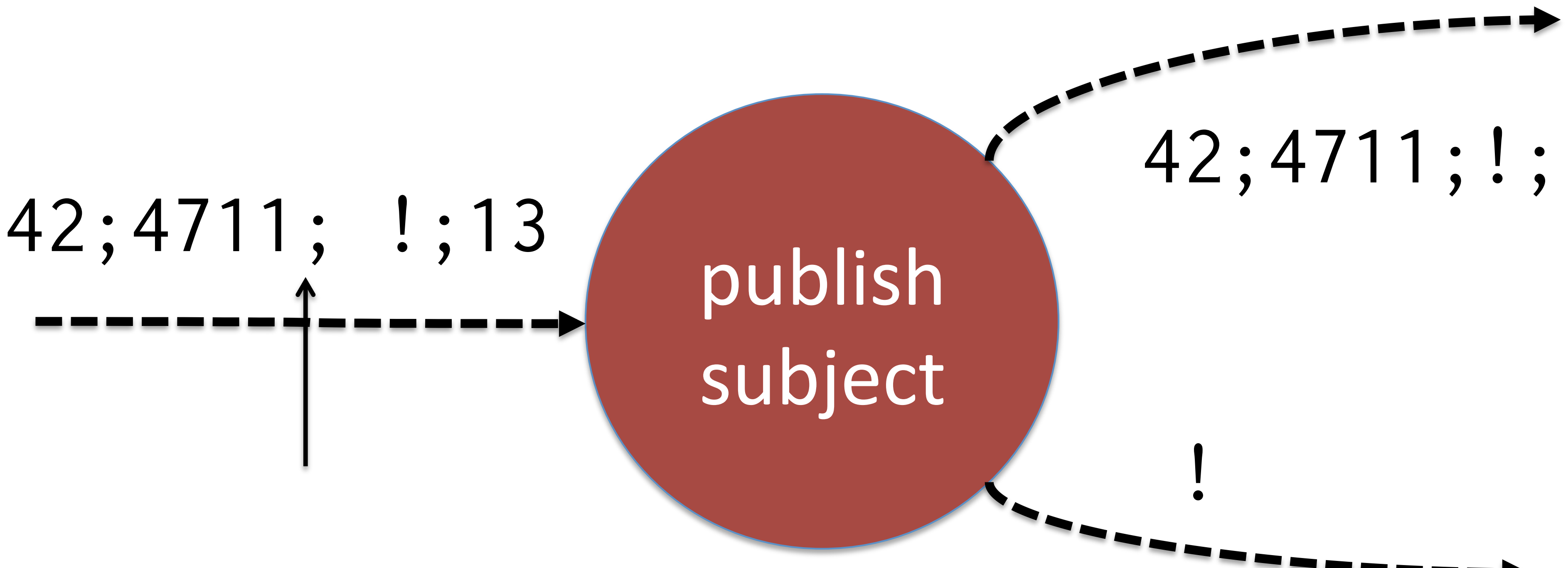
42;4711; !;13

async subject

4711;!

4711;!

caches final value



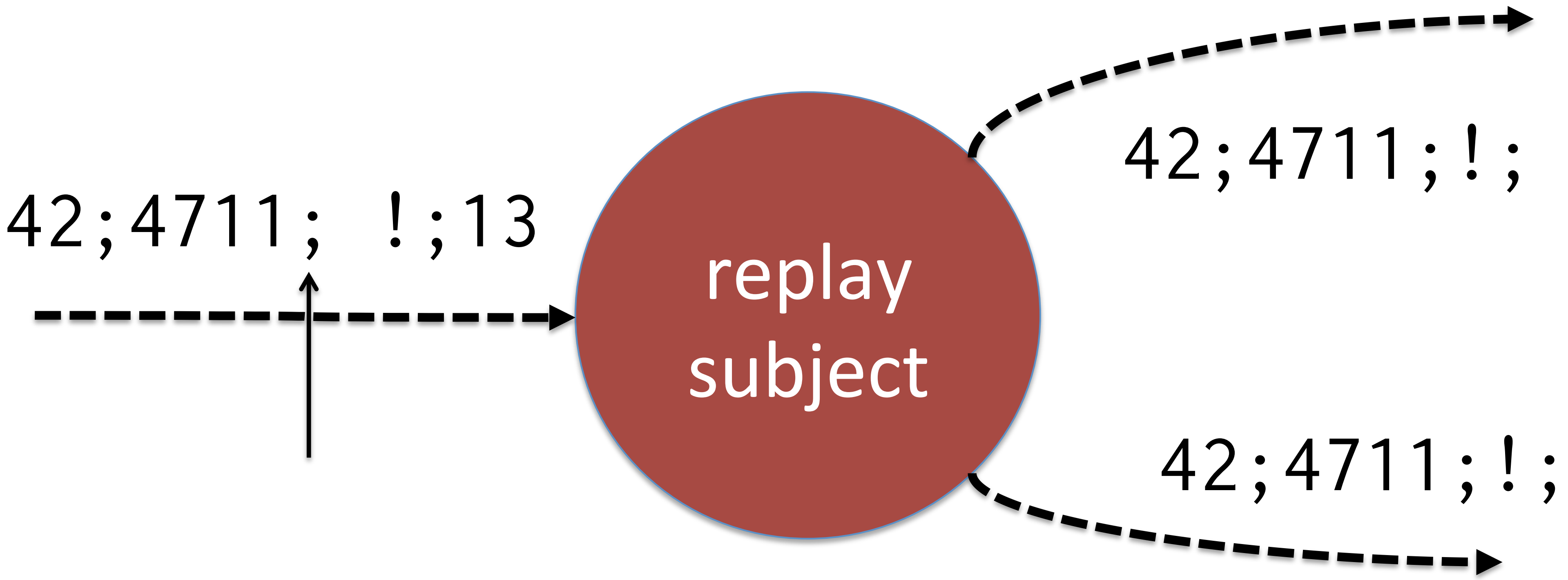
42;4711; !;13

publish subject

42;4711;!;

!

current value



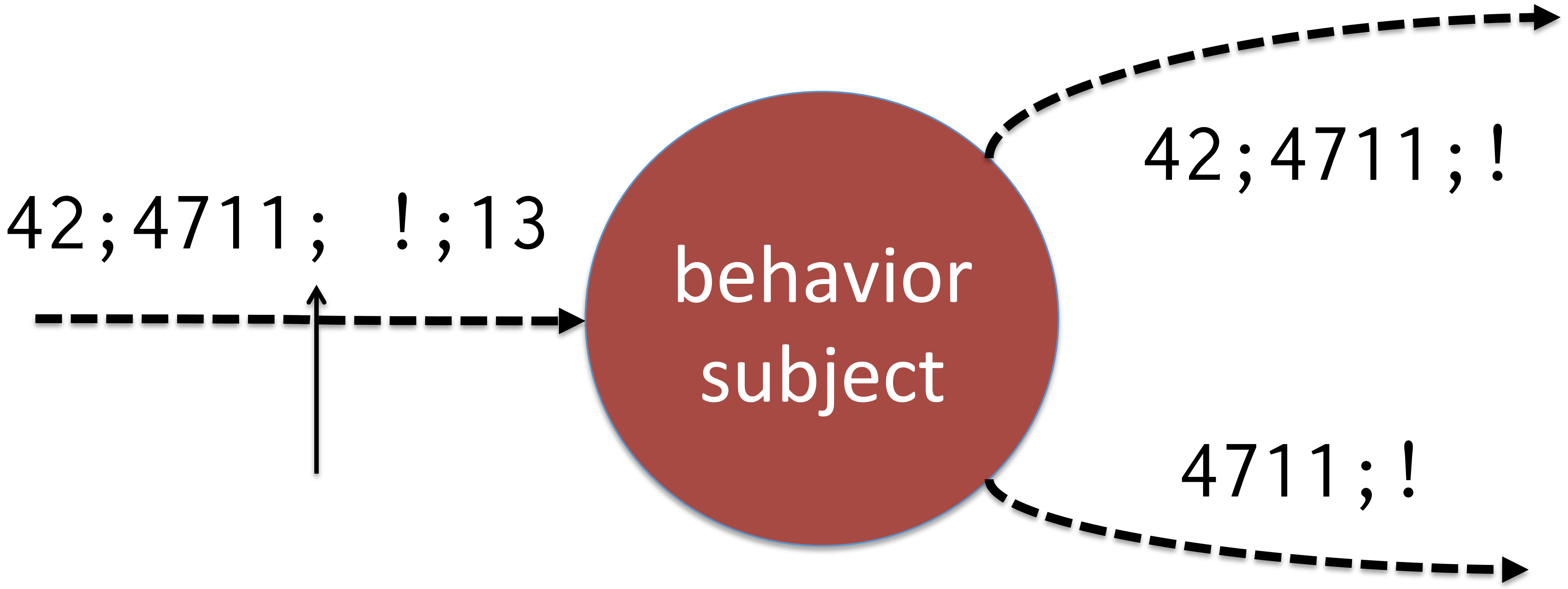
42;4711; !;13

replay subject

42;4711;!;

42;4711;!;

caches all values



42;4711; !;13

behavior subject

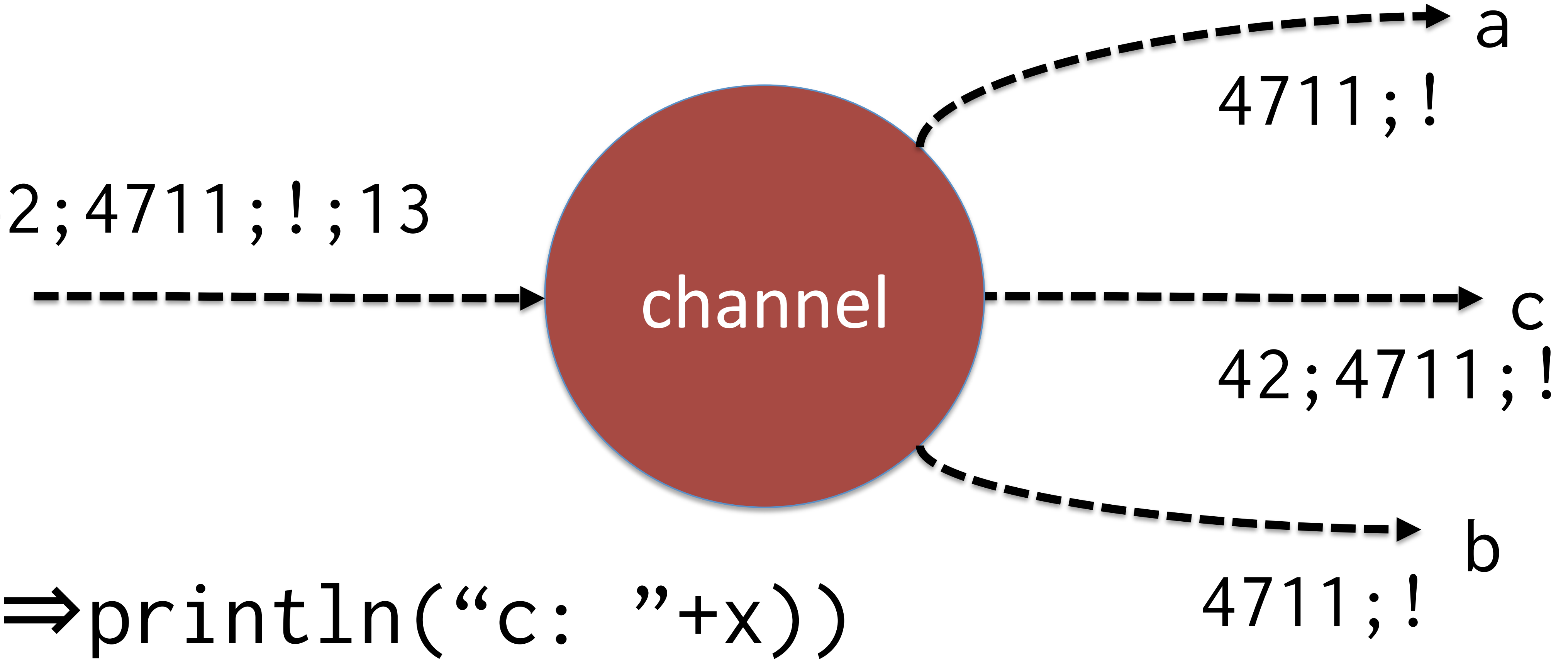
42;4711;!

4711;!

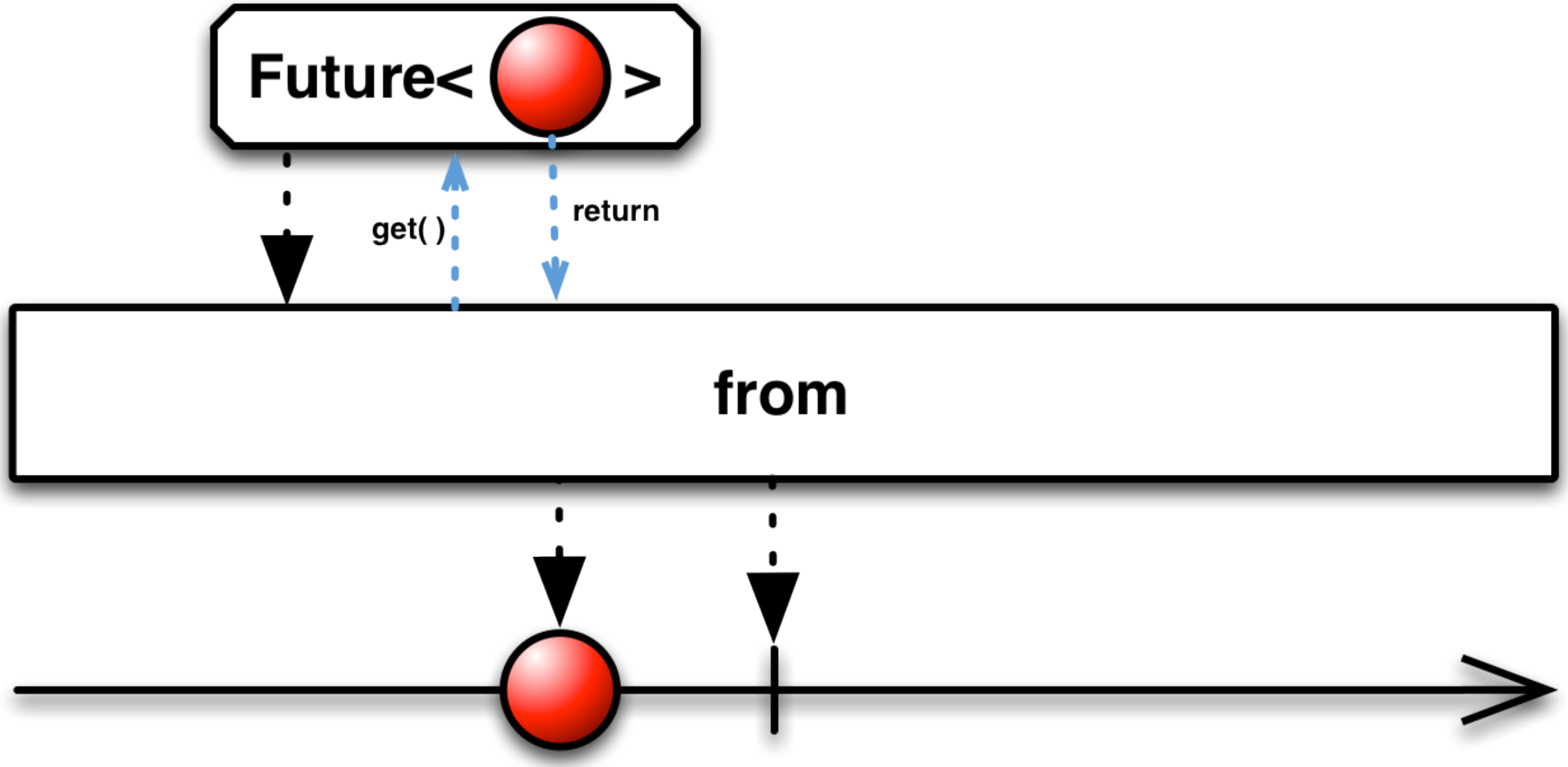
caches latest value

Quiz

```
val channel = AsyncSubject[Int]()

val a = channel.subscribe(x⇒println(“a: ”+x))
val b = channel.subscribe(x⇒println(“b: ”+x))

channel.onNext(42)
a.unsubscribe()
channel.onNext(4711)
channel.onCompleted()
val c = channel.subscribe(x⇒println(“c: ”+x))
channel.onNext(13)
```



Creating Observables



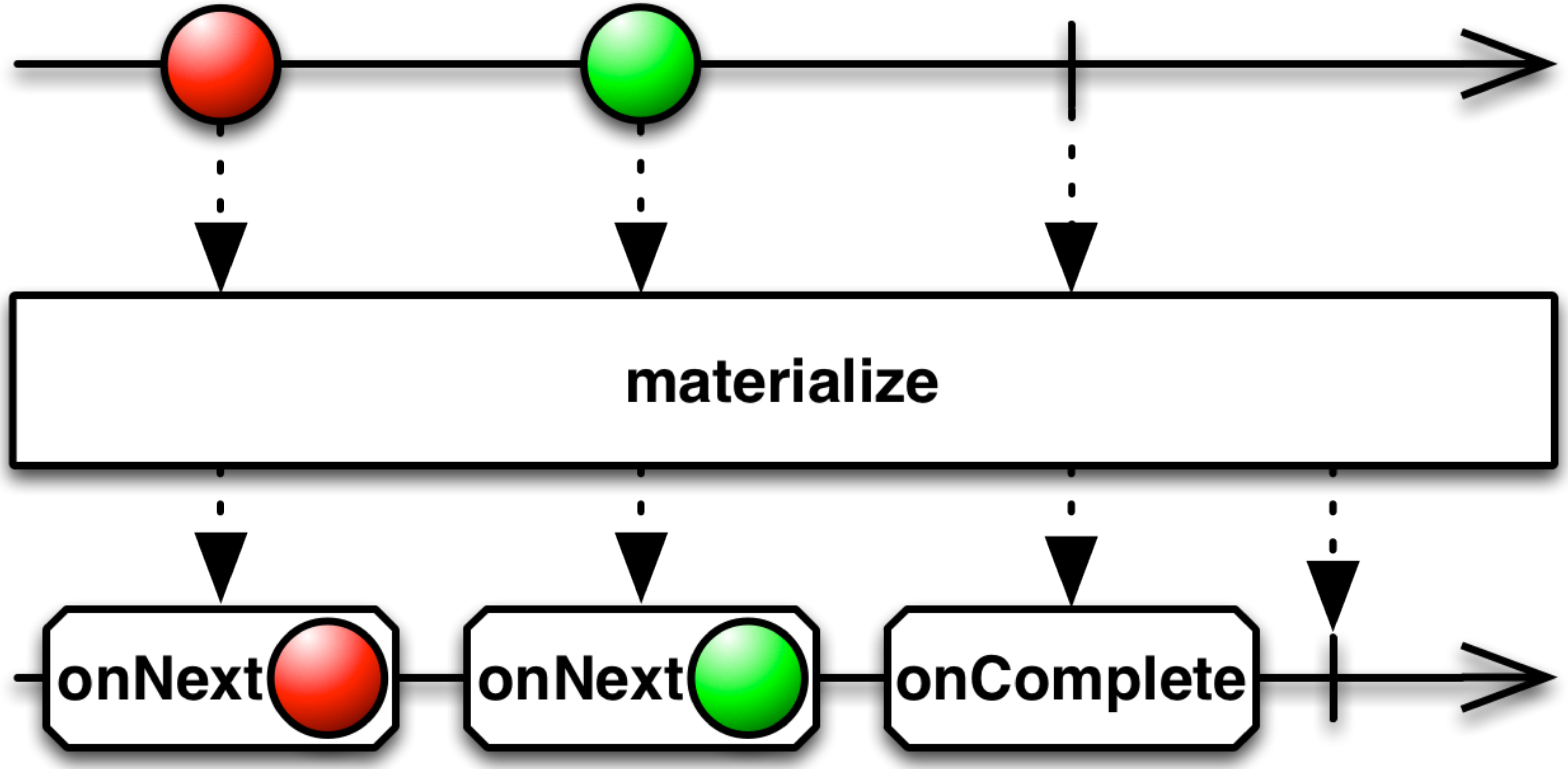
Converting Future[T] to Observable[T]

```
object Observable {
  def apply[T](f: Future[T]): Observable[T] = {
     val subject = AsyncSubject[T]()
     f onComplete {
        case Failure(e) ⇒ { subject.onError(e) }
        case Success(c) ⇒ { subject.onNext(c); subject.onCompleted() }
     }
     subject
  }
}
```

Observable notifications

```
abstract class Try[+T]
case class Success[T](elem: T) extends Try[T]
case class Failure(t: Throwable) extends Try[Nothing]

abstract class Notification[+T]
case class OnNext[T](elem: T) extends Notification[T]
case class OnError(t: Throwable) extends Notification[Nothing]
case object OnCompleted extends Notification[Nothing]

def materialize: Observable[Notification[T]] = { … }
```

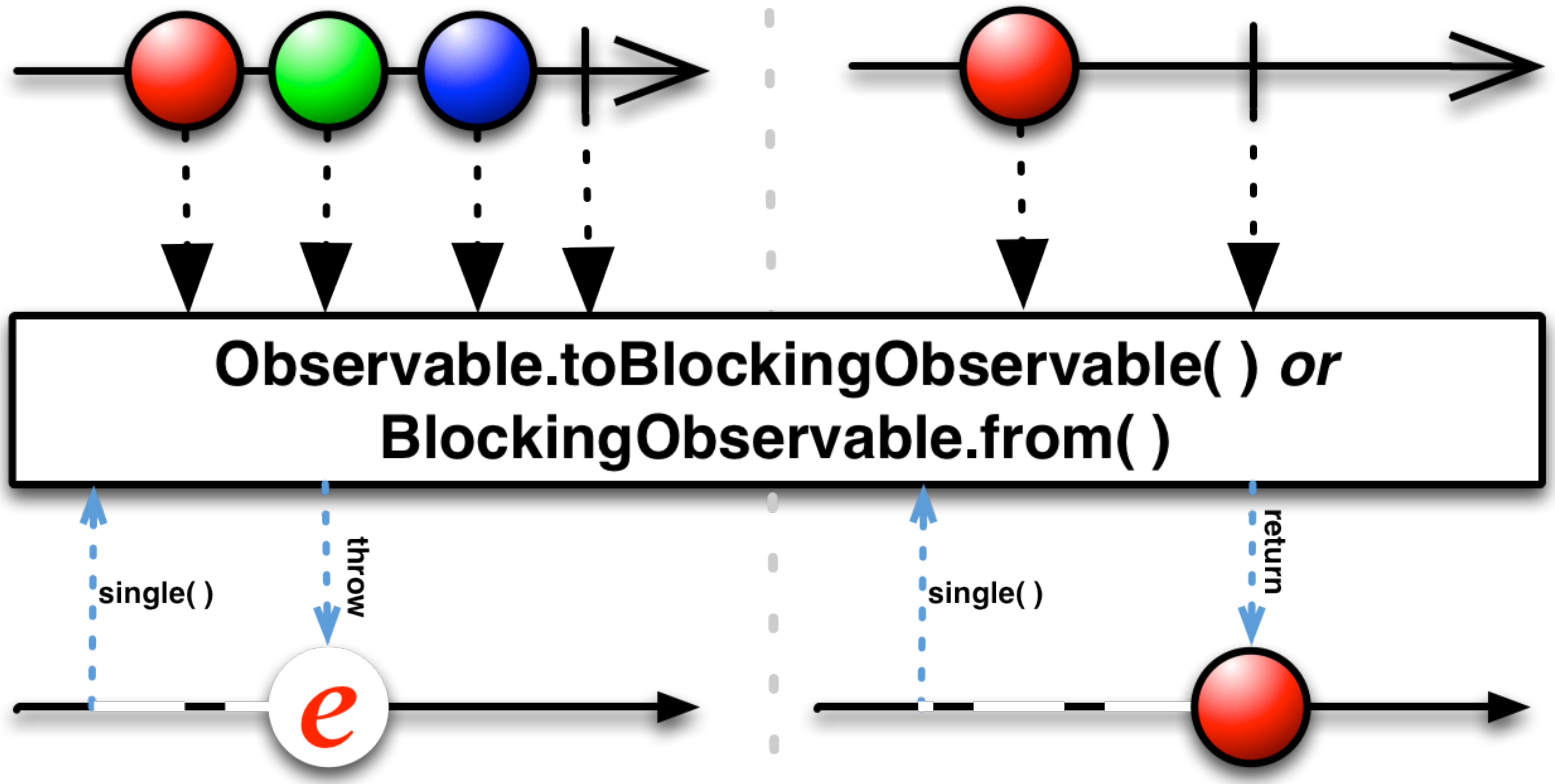
Observable notifications



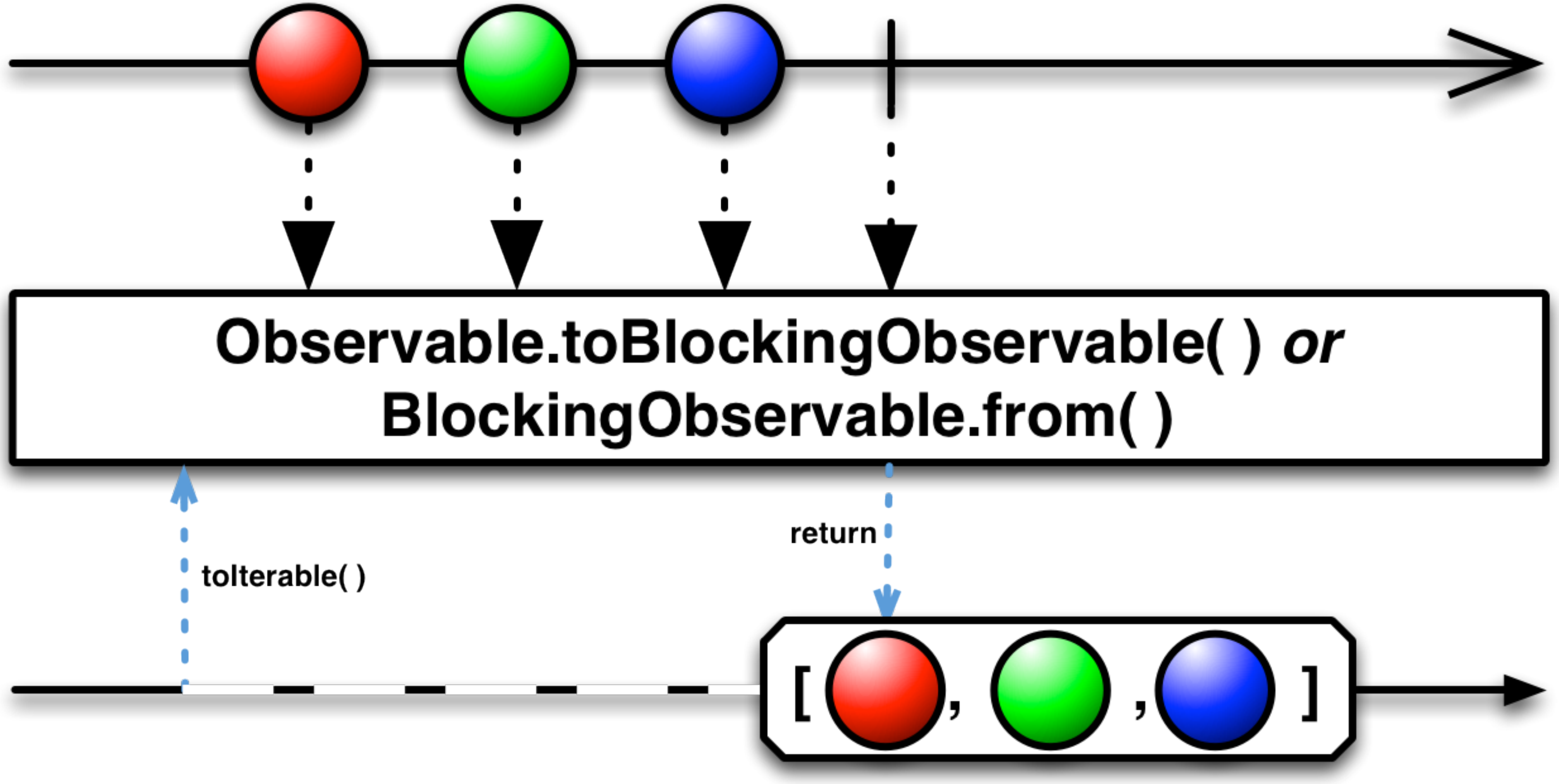
Remember blocking?

```
val f: Future[String] = future { … }
val text: String = Await.result(f, 10 seconds)
```

Blocking


Observable.toBlockingObservable( ) *or*
BlockingObservable.from( )
single( )
throw
e
single( )
return

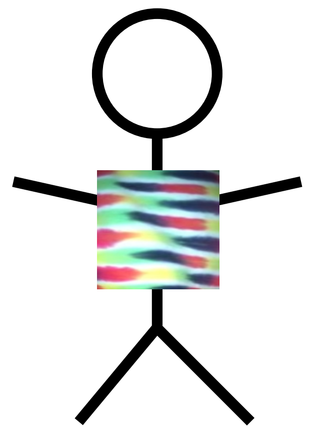
Blocking


Observable.toBlockingObservable( ) *or*
BlockingObservable.from( )
toIterable( )
return
[, ,]

Converting Observables to scalar types

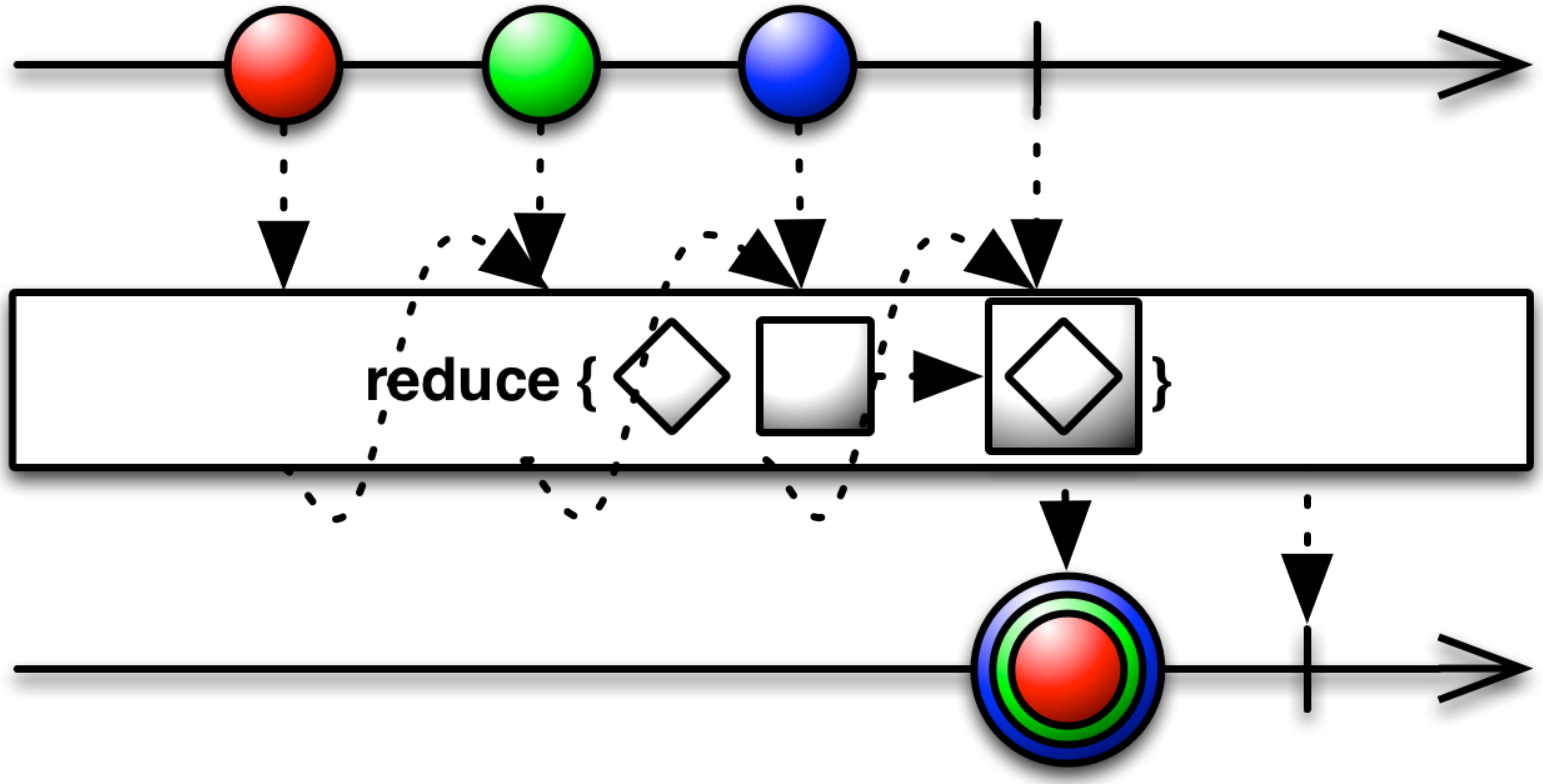
```
val xs: Observable[Long] = Observable.interval(1 second).take(5)
val ys: List[Long] = xs.toBlockingObservable.toList

println(ys)
println(“bye”)

val zs: Observable[Long] = xs.sum
val s: Long = zs.toBlockingObservable.single
```

All Rx operators are non-blocking



“single” throws if not exactly one element

Converting Observables to scalar types



```
def reduce(f: (T, T) => T): Observable[T]
```

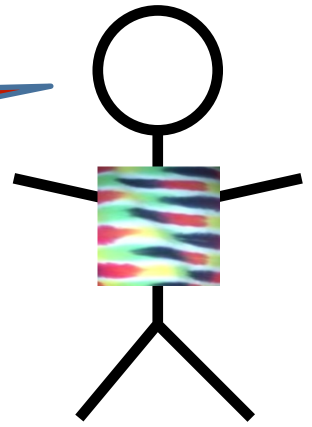
Duality excursion

De Morgan’s duality law

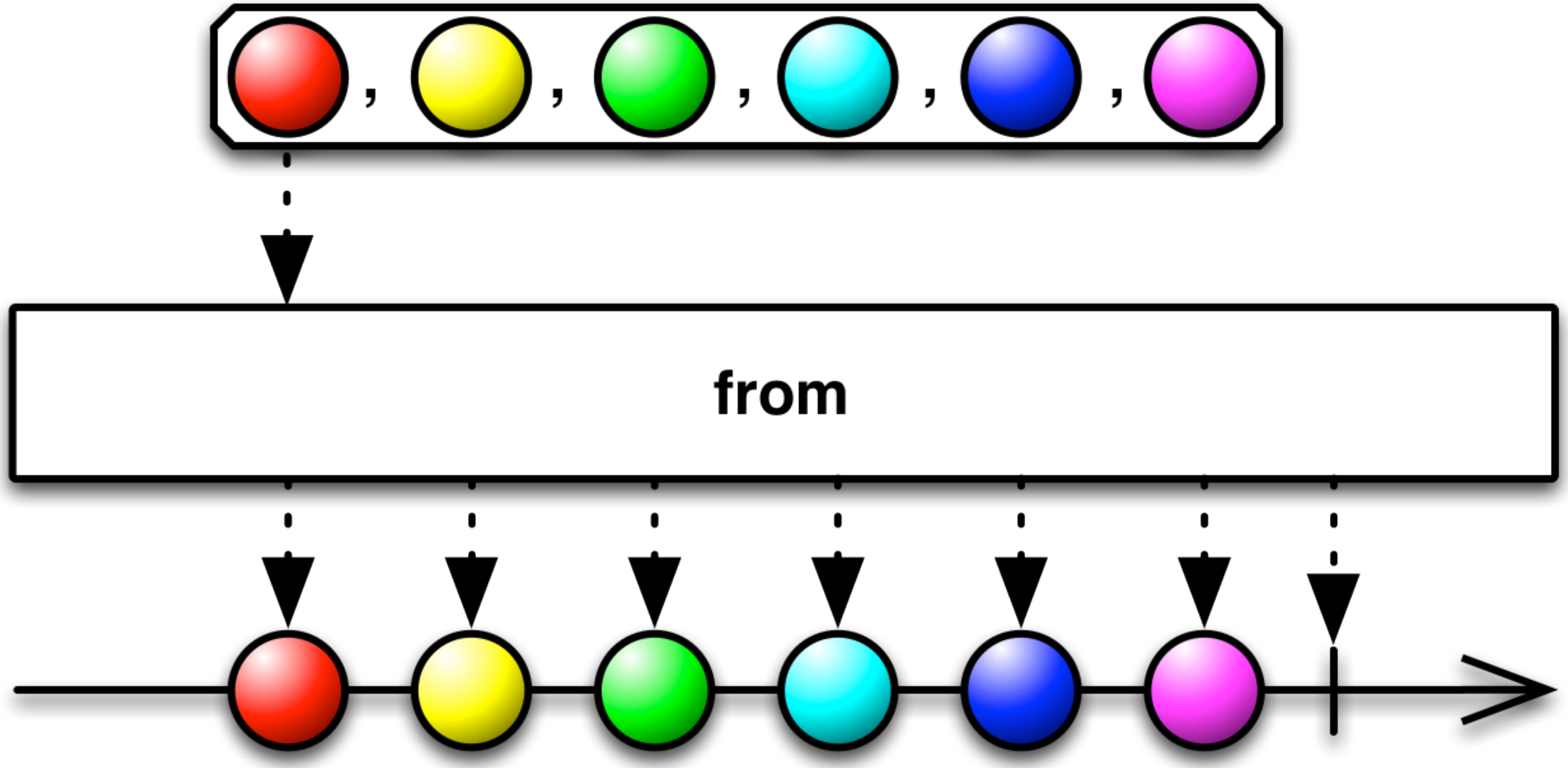
!(a && b) = !a || !b

!(a || b) = !a && !b

AND and OR are dual – NOT is the energy between them



Creating Observables



Creating Observables

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

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