Introduction to Rx Observables

first: let's define some convenience methods and values (will use it quite often...):

```
1  import rx.{Observer => JObserver, Subscription => JSubscription}
2
3  def startOnThread(body: => Unit): Thread = {
4    val t = new Thread {
5      override def run = body
6    }
7    t.start
8    t
9  }
10
11 val zeroTime = System.currentTimeMillis
12
13 def threadTimeString = {
14   val name = Thread.currentThread.getName
15   val time = System.currentTimeMillis - zeroTime
16   s"current time on $name is: $time."
17 }
```

a “simple” example implementation for an Observable serving as a tick source:

```
1  val ticks: Observable[String] = Observable(observer => {
2    var cancelled = false
3    val t = startOnThread {
4      try {
5        while (!cancelled) {
6          try {
7            observer.onNext(threadTimeString)
8            Thread.sleep(100)
9          } catch {
10           case e: java.lang.InterruptedException => //DO NOTHING
11         }
12       }
13       observer.onCompleted()
14     } catch {
15       case e: Throwable => observer.onError(e)
16     }
17   }
18   Subscription(new JSubscription {
19     override def unsubscribe() = {
20       cancelled = true
21       t.interrupt
22       t.join
23     }
24   })
25 })
```

well, how would we use it? simple:

```
1  val subscription = ticks.subscribe(println(_))
2  Thread.sleep(1000)
3  subscription.unsubscribe
```

which would print something like:

```
1  current time on Thread-0 is: 222.
2  current time on Thread-0 is: 337.
3  current time on Thread-0 is: 437.
4  current time on Thread-0 is: 538.
5  current time on Thread-0 is: 638.
6  current time on Thread-0 is: 739.
7  current time on Thread-0 is: 839.
8  current time on Thread-0 is: 940.
9  current time on Thread-0 is: 1040.
10 current time on Thread-0 is: 1141.
```

but if want better control, in case we want to treat errors or execute something when the Ovservable announces completion, we will need to define an Observer

```
1  val ticksConsumer: Observer[String] = Observer(
2    new JObserver[String]{
3      def onNext(s: String) = println(s)
4      def onError(e: Throwable) = logger.error(e.getMessage)
5      def onCompleted() = println("DONE!")
6    }
7  )
```

usage is similar:

```
1  val subscription = ticks.subscribe(ticksConsumer)
2  Thread.sleep(1000)
3  subscription.unsubscribe
```

and now, the output contains a “DONE!” print:

```
current time on Thread-0 is: 201.
current time on Thread-0 is: 314.
current time on Thread-0 is: 414.
current time on Thread-0 is: 515.
current time on Thread-0 is: 615.
current time on Thread-0 is: 716.
current time on Thread-0 is: 816.
current time on Thread-0 is: 917.
current time on Thread-0 is: 1017.
current time on Thread-0 is: 1118.
DONE!
```

but the shown code has some problems:

- the `Observable`'s code is somewhat boilerplate
- the whole chain of execution is done on a single thread
- `Scheduler`s...?
- what would happen if more than one `Observer` subscribes to this `Observable`?

let's try to improve our solution.
first, consider a very naive implementation of a "ticks source":

```
1  object TickSource {
2    private var callbacks: Map[Int, () => Unit] = Map.empty
3    private var counter = 0
4    private var cancelled = false
5    private val t = startOnThread {
6      while (!cancelled) {
7        Thread.sleep(100)
8        callbacks.foreach(_._2())
9      }
10   }
11
12   def onTick(callback: => Unit): Int = {
13     counter = counter + 1
14     callbacks = callbacks +(counter, () => callback)
15     counter
16   }
17
18   def remove(key: Int): Unit = {callbacks = callbacks - key}
19
20   def shutdown = {cancelled = true; t.join}
21 }
```

NOTE: the above code is very naive. I kept it short for this toy example. a real implementation though must take care of synchronizing the `onTick` and `remove` methods (among other things...).

also, we'll need to define the `Observable` that will use `TickSource` as it's source.

```
1  val ticks: Observable[String] = Observable(observer => {
2    val key = TickSource.onTick {
3      observer.onNext(threadTimeString)
4    }
5    Subscription(new JSubscription {
6      override def unsubscribe() = TickSource.remove(key)
7    })
8  })
```

so, now we can use it exactly as before:

```
1  val subscription = ticks.subscribe(ticksConsumer)
2  Thread.sleep(1000)
3  subscription.unsubscribe
4  TickSource.shutdown //just to ensure the proccess won't hang...
```

NOTE: there is no `onCompleted` method in this scenario. the source is infinite, and does not "completes"

and we'll get:

```
current time on Thread-0 is: 271.
current time on Thread-0 is: 385.
current time on Thread-0 is: 485.
current time on Thread-0 is: 586.
current time on Thread-0 is: 686.
current time on Thread-0 is: 787.
current time on Thread-0 is: 887.
current time on Thread-0 is: 988.
current time on Thread-0 is: 1088.
current time on Thread-0 is: 1189.
```

well, that took care of some problems. but it's still not perfect. we would want to sepparate the source work and consume work to be done on different threads. luckily, there's a simple to use API for that: `observeOn`.

```
val scheduler = rx.lang.scala.concurrency.Schedulers.newThread
val subscription = ticks.observeOn(scheduler).subscribe{s =>
  println(s"$s\n${threadTimeString}")
}
Thread.sleep(1000)
subscription.unsubscribe
```

and the output now looks like:

```
current time on Thread-0 is: 305.
current time on RxNewThreadScheduler-1 is: 328.
current time on Thread-0 is: 426.
current time on RxNewThreadScheduler-1 is: 427.
current time on Thread-0 is: 527.
current time on RxNewThreadScheduler-1 is: 527.
current time on Thread-0 is: 627.
current time on RxNewThreadScheduler-1 is: 628.
current time on Thread-0 is: 728.
current time on RxNewThreadScheduler-1 is: 729.
current time on Thread-0 is: 829.
current time on RxNewThreadScheduler-1 is: 829.
current time on Thread-0 is: 929.
current time on RxNewThreadScheduler-1 is: 930.
current time on Thread-0 is: 1030.
current time on RxNewThreadScheduler-1 is: 1030.
current time on Thread-0 is: 1130.
current time on RxNewThreadScheduler-1 is: 1131.
```

since a "NewThread" `Schduler` is usually not what we want,
when in doubt, you can use this `Schduler` instead:

```
import rx.lang.scala.concurrency.Schedulers.executor
import scala.concurrent.ExecutionContext.Implicits.global
val scheduler = executor(global)
```

cool stuff:

```
1  def actOn(ss: Seq[String]) = {
2    val t = threadTimeString
3    val s = ss.mkString("\n")
4    println(s"$t\n${t.map(_ => '*')}\n$s\n")
5  }
6
7  val multipleTicksConsumer = Observer(new JObserver[Seq[String]]{
8    def onNext(ss: Seq[String]) = actOn(ss)
9    def onError(e: Throwable) = logger.error(e.getMessage)
10   def onCompleted() = println("DONE!")
11 })
12
13 val scheduler2 = rx.lang.scala.concurrency.Schedulers.newThread
14
15 //previous ticks was a toy example. there's a method call for that:
16 val ticks1 = Observable.interval(100 millis, scheduler)
17 val ticks2 = ticks1.map(_ => threadTimeString)
18 val ticks3 = ticks2.observeOn(scheduler2)
19 val ticks4 = ticks3.buffer(500 milliseconds, 4)
20
21 val subscription = ticks4.subscribe(multipleTicksConsumer)
22 Thread.sleep(1000)
23 subscription.unsubscribe
```

and the output:

```
current time on RxNewThreadScheduler-1 is: 611.
*********************************************
current time on ForkJoinPool-1-worker-5 is: 274.
current time on ForkJoinPool-1-worker-5 is: 386.
current time on ForkJoinPool-1-worker-3 is: 487.
current time on ForkJoinPool-1-worker-5 is: 587.

current time on RxNewThreadScheduler-1 is: 991.
*********************************************
current time on ForkJoinPool-1-worker-3 is: 688.
current time on ForkJoinPool-1-worker-5 is: 789.
current time on ForkJoinPool-1-worker-3 is: 889.
current time on ForkJoinPool-1-worker-5 is: 990.
```

name	type	runs on
ticks1	Observable[Long]	ForkJoinPool-1-worker-N
ticks2	Observable[String]	ForkJoinPool-1-worker-N
ticks3	Observable[String]	RxNewThreadScheduler-1
ticks4	Observable[Seq[String]]	RxNewThreadScheduler-1

Appendix

How to write an `Observable` the "scala way" with no `TickSource`, and a "built in" `Scheduler`:

```
1  val ticks: Observable[String] = Observable(observer => {
2    scheduler.scheduleRec(self => {
3      observer.onNext(threadTimeString)
4      Thread.sleep(100)
5      self
6    })
7  })
```