A Playful Introduction To Rx

Hacking should be fun

Erik Meijer
www.applied-duality.com
Programming, Motherfucker
Do you speak it?

We are a community of motherfucking programmers who have been humiliated by software development methodologies for years.

We are tired of XP, Scrum, Kanban, Waterfall, Software Craftsmanship (aka XP-Lite) and anything else getting in the way of... Programming, Motherfucker.

http://programming-motherfucker.com/
Two Games

Classic Platform Game

Started the Mobile Gaming Craze
A Bug’s Life

Snake

Source Code
Programming with Streams

Interface Collection<E>

Type Parameters:
- E - the type of elements in this collection

All Superinterfaces:
- Iterable<E>

All Known Subinterfaces:
- BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:
- AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashset, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector
In Java 8

java.util.stream

**Interface Stream<T>**

Type Parameters:

T - the type of the stream elements

All Superinterfaces:

AutoCloseable, BaseStream<T, Stream<T>>
Down the Rabbit Hole

interface BaseStream<T, S extends BaseStream<T,S>>
interface Stream<T> extends BaseStream<T, Stream<T>>

Stream<T>
BaseStream<T, Stream<T>>
BaseStream<T, BaseStream<T, Stream<T>>>
BaseStream<T, BaseStream<T, BaseStream<T, Stream<T>>>>
...
BaseStream<T, BaseStream<T, BaseStream<T, BaseStream<T, ...>>>>

Compared to this, everything else is simple
BaseStream Is An Iterable

```java
interface BaseStream<T, S extends BaseStream<T,S>> extends AutoClosable {
    Iterator<T> iterator()
}
```

Denial of fact

In this form of denial, someone avoids a fact by utilizing deception. This lying can take the form of an outright falsehood (commission), leaving out certain details to tailor a story (omission), or by falsely agreeing to something (assent, also referred to as "yessing" behavior). Someone who is in denial of fact is typically using lies to avoid facts they think may be painful to themselves or others.

Stream Pipeline

```java
int sum = widgets.stream()
    .filter(w -> w.getColor() == RED)
    .mapToInt(w -> w.getWeight())
    .sum();
```

“Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source.”
Filter

filter { }
Map

map \{ \textcircled{\textbullet} \rightarrow \textdiamond \}
Sum

\[ \sum_{i=1}^{n} X_i \]
Bulk Operations Over Streams

Source Stream

Target Stream

https://github.com/Netflix/RxJava/wiki
Streams do not provide a means to directly access or manipulate their elements

interface Iterable<T> {
    Iterator<T> iterator()
}

interface Iterator<T> {
    boolean hasNext()
    T next()
}
class MainJava {

    public static void main(String[] args) {
        RandomStream integers = new RandomStream();
        integers.stream().limit(5).forEach(System.out::println);
    }
}

class RandomStream implements Iterable<Integer> {
    public Stream<Integer> stream() {
        return StreamSupport.stream(this.spliterator(), false);
    }

    @Override
    public Iterator<Integer> iterator() {
        final Random random = new Random();
        return new Iterator<Integer>() {
            @Override public boolean hasNext() { return true; }
            @Override public Integer next() { return random.nextInt(); }
        };
    }
}
Dalai Lama About Programming With Streams

“Live in the moment. Do not forget nor dwell on the past, but do forgive it. Be aware of the future but do no fear or worry about it. **Focus on the present moment, and that moment alone.**”
See Streams Everywhere

Current mouse position

(10,50)

(20,20)

(40,10)

(60,40)
See Streams Everywhere

http://finance.yahoo.com/q?s=MSFT

Current stock price
Screaming Babies

You cannot pressure your baby not to cry

http://en.wikipedia.org/wiki/Baby_colic
These Streams Don’t Wait For You

Only when you ask for it.

Whether you want it or not. Whether you can handle it or not.
Observable Streams

```java
public class Observable<T> {
    Subscription subscribe(Observer<T> observer)
}
public interface Observer<T> {
    void onCompleted();
    void onError(Throwable error);
    void onNext(T value);
}
public interface Subscription {
    void unsubscribe();
    boolean isUnsubscribed();
}
```
class MainJava {

    public static void main(String[] args) {
        RandomObservable integers = new RandomObservable();
        integers.take(5).subscribe(System.out::println);
    }
}

class RandomObservable extends Observable<Integer> {
    public RandomObservable() {
        super(subscriber -> {
            Random random = new Random();
            while(!subscriber.isUnsubscribed()) {
                subscriber.onNext(random.nextInt());
            }
            subscriber.onNext(random.nextInt());
        });
    }
}

Elements of the stream do not exist before (and after) they are accessed.
Bulk Operations Over Streams

Source Stream

Target Stream
Filter
Map

map \{ \text{shape} \rightarrow \text{diamond} \}
Observable Collections

class ObservableCollection<T> : Collection<T>,
    INotifyCollectionChanged, INotifyPropertyChanged

Represents a dynamic data collection that provides notifications when items get added, removed, or when the whole list is refreshed.

public interface ObservableList<E>
extends java.util.List<E>, Observable

A list that allows listeners to track changes when they occur.
Cross The Streams

Observable Collection

Iterable<T>/Stream<T>

Observable<Change<T>>

Mutate
Log Concept

- Log is a history of all changes to the state.
- Log + old state gives new state
- Log + new state gives old state (not in this picture)
- Log is a sequential file.
- Complete log is the complete history
- Current state is just a "cache" of the log records.
Why use a database when you have a log?

The future mutates the present

the past

the present
Time Is A Stream

val clock: Observable[Long] = 
  Observable.timer(
    initialDelay = 0 seconds,
    period = 1 second
  )
def factors(n: Long): Iterable[Long] = 
    Seq.range(1L, n+1).filter(x => n%x == 0)

def main(args: Array[String]) {
    val clock: Observable[Long] = Observable.timer(0 seconds, 1 second)

    val primes: Observable[Long] = clock.filter(n => factors(n) == Seq(1,n))

    primes.foreach println
    readLine()
}
Your Keyboard is a Stream

Reacts on up, down, left, right.

Reacts on spacebar.
Events Are Streams

```scala
implicit def actionToEventHandler[E <: javafx.event.Event](f: E => Unit): EventHandler[E] = 
  new EventHandler[E] { def handle(e: E): Unit = f(e) }

def keyPresses (scene: Scene): Observable[KeyEvent] = Observable[KeyEvent](observer => {
  val handler: EventHandler[KeyEvent] = (e:KeyEvent) => observer.onNext(e)
  scene.addEventHandler(KeyEvent.KEY_PRESSED, handler)
  observer.add {
    scene.removeEventHandler(KeyEvent.KEY_PRESSED, handler)
  }
})

def spaceBars(scene: Scene): Observable[KeyEvent] = 
  keyPresses(scene).filter(_.getCode == KeyCode.SPACE)
```
Showing Key Events In A Graph

JavaFx line chart of Data[Number, String]
val keys: Series[Number, String] = {
  val ks = new XYChart.Series[Number, String]();
  chart.getData.add(ks);
  ks
}
val keyboard: Observable[String] = keyPresses(scene).map(_.getText)

val withIndex: Observable[(Int, String)] = keyboard.zipWithIndex.map(_.swap)

withIndex.subscribe(s => keys.getData.add(s))
Show me the code

```scala
val keys: ObservableList[Data[Number, String]] = FXCollections.observableArrayList()

lineChart.getData.add(new XYChart.Series[Number, String](keys))

val keyboard: Observable[String] = keyPresses(scene).map(_.getText)
val withIndex: Observable[(Int, String)] = keyboard.zipWithIndex.map(_.swap)

withIndex.subscribe(s => keys.add(s))
```

Model

View

Controller
Rx For Astronaut Architects

observable
observer
observe changes
mutate state

UI
data binding

Observable Collection
Sampling keyboard

val clock: Observable[Long] = timer(initialDelay = 0 seconds, period = 1 second)
val sample = keyboard.map(key => clock.map(i => (i,key))).switch

for each keystroke
Pair keystroke with clock ticks
Now we have a nested stream
Switch whenever a new keystroke happens

sample.subscribe(s => keys.getData.add(s))
Switching Streams

switchOnNext
Sampling Every Tick (With Bug)

Time does not march forward :-(

Diagram showing a graph with points labeled 'a' and 'b' connected by lines, indicating a sequence or pattern in the data.
Cold Observable

val clock: Observable[Long] = timer(initialDelay = 0 seconds, period = 1 second)
val sample = keyboard.map(key => clock.map(i => (i,key))).switch

Every time we switch streams, we resubscribe to the clock, starting at time t=0.
Hot Observable

val clock: Observable[Long] = timer(initialDelay = 0 seconds, period = 1 second)

val sample = clock.publish(_clock =>
    keyboard.map(key => _clock.map(i => (i, key)))).switch

Every time we switch streams, **we share the subscription to the clock**, continuing with time \( t = i \).
Sampling Every Tick

No keystroke for a little bit

cmd-shift-4

time keeps ticking ...
Virtual Time

```scala
testScheduler = TestScheduler()

clock: Observable[Long] = timer(initialDelay = 0 seconds, period = 1 second, testScheduler)

sample = clock.publish(_clock =>
    keyboard.map(key => _clock.map(i => (i, key))).switch
)

keyboard.subscribe(key => {
    testScheduler.advanceTimeBy(0.5 second)
})
```
Say Cheese!

cmd-shift-4
Keyboards Are Naturally Hot

val testScheduler = TestScheduler()

val clock: Observable[Long] = timer(initialDelay = 0 seconds, period = 1 second, testScheduler)

val sample = clock.publish(_clock =>
  keyboard.map(key => _clock.map(i => (i, key))).switch

keyboard.subscribe(key => {
  testScheduler.advanceTimeBy(0.5 second)
})
Ensuring an Observable is Hot


val sample = clock.publish(_clock =>
    keyboard.map(key => _clock.map(i => (i,key))).switch
)

keyboard.subscribe(key => {
    testScheduler.advanceTimeBy(0.5 second))
})

keyboard.connect
Ensuring an Observable is Hot

```scala

val sample = clock.publish(_clock =>
    keyboard.map(key => _clock.map(i => (i,key))).switch
)

keyboard.subscribe(key => {
    testScheduler.advanceTimeBy(0.5 second))
}
```

WARNING: (re)subscribes when refcount goes 0 -> 1
class MainJava {

class HotIterable implements Iterable<T> {

    public Stream<T> stream() {
        return StreamSupport.stream(this.spliterator(), false);
    }

    final State perIterableState = new State();

    @Override
    public Iterator<T> iterator() {
        final State perIteratorState = new State();
        return new Iterator<T>() {
            @Override
            public boolean hasNext() {
                ... perIterableState ...
            }

            @Override
            public Integer next() {
                ... perIterableState ... perIteratorState ...
            }
        };
    }
}

State shared between all iterators ⇒ Hot
Everything Is Relative

The grass moves right to left, the bug stays put.
Show me the code

val grass = new {
    val tile = new Image(s"$resourceDir/GrassBlock.png")
    val height = tile.getHeight
    val nrTiles = Math.ceil(screenWidth/tile.getWidth).asInstanceOf[Int]+1

    val tiles = (0 to nrTiles-1).map(i =>
        new ImageView {
            setImage(tile)
            setTranslateX(i*getImage.getWidth)
            root.getChildren.add(this)
        }).toList
}

Create list of tiles using bulk operations on iterable
Where Is Dr Beckman When You Need Him Most

Show me the code

val grass = new {
  val v0 = 3
  clock.scan(v0)((vi,_)=>vi).subscribe(dX =>
    setTranslateX(
      if(getTranslateX <= -getImage.getWidth) {
        screenWidth-dX
      } else {
        getTranslateX-dX
      })
    )
  )
}

Poor man's physics: Δx = v*Δt where Δt =1, so Δx = v. We'll just take velocity to be pixels/tick and move things accordingly

wrap-around
Scan: Accumulate with Trace

\[
\text{scan} \{ (\diamondsuit, \Box) \rightarrow \Box \}
\]
Computing Velocity

always ignore ticks

$v[i+1] = f(v[i])$
Jumping Bug

v0

v1 = v0 - g

v2 = v1 - g

v2 = v1 - g

v3 = v2 - g = 0

v4 = v3 - g

v5 = v4 - g

v6 = v5 - g = -v0

Every kindergartner recognizes the “scan”
Show Me The Code

```scala
jumps
  .flatMap(v0 =>
    clock
      .scan(v0)((vi,_)=>vi-gravity)
      .takeUntil(jumps)
  )
  .subscribe(dY => ... update position ...)
```

When jump happens

Initial jump velocity

Decrease with “gravity” each tick

Until next jump

Prevent falling through floor by ignoring changes when below sea level
Turn One Stream Off By Another
Show Me The Code

```scala
val jumpSpeed = 8
spaceBars(scene)
  .filter(_ => bug.getTranslateY >= bug.homeY)
  .doOnEach(_ => ... play sound ...)
.subscribe(_ => bug.jumps.onNext(jumpSpeed))
```
SpreadSheet == Mother of All Reactive Programming

Result cell automatically updated whenever any input cell changes
Steal A Trick From Spreadsheets
Hit Detection

```javascript
bugPosition.combineLatest
  (sunPosition, collides)
  .slidingBuffer(2,1)
  .filter(hits => hits(0) != hits(1))
```

Change from sun to heart

Change from heart to sun
Tumbling Buffers

buffer(skip=3, count=2)
That’s Really All There Is

import javafx.geometry._
import javafx.scene.canvas.Canvas
import javafx.scene.paint.Color
import javafx.scene._
import javafx.scene.image._
import javafx.scene.layout.StackPane
import javafx.stage._
import javafx.application._
import games.PlatformScheduler
import rx.lang.scala.schedulers._
import rx.lang.scala._
import scala.language.postfixOps
import scala.concurrent.duration._
import rx.lang.scala.javaFx.utils

In the end we are all just plumbers

Import list nearly long than the code

http://www.mariowiki.com/Image:MarioNSMBWii.PNG
More Info

https://github.com/Netflix/RxJava
https://github.com/Reactive-Extensions/RxJS
https://github.com/Reactive-Extensions/RxCpp
https://github.com/Reactive-Extensions/Rx.NET
https://github.com/ReactiveCocoa/ReactiveCocoa
https://www.dartlang.org/docs/tutorialsstreams/
http://bodil.org/prez/
And Don’t Forget To Sign Up For

Introduction to Functional Programming

The aim of this course is to teach the foundations of functional programming and how to apply them in the real

https://www.edx.org/course/delftx/delftx-fp101x-introduction-functional-2126#.U8jMml2Sylo