

# Introduction to Functional Reactive Programming

Shekhar Gulati  
@shekhargulati

# Assumptions

- You understand basic functional programming concepts
- You know how to read Java code

# WTF is FRP?

- Wikipedia says
  - Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter).

# Simpler Definition

- FRP is a functional way to work with asynchronous stream (could be infinite) of events.
- Here events could be anything from mouse clicks, tweets, etc.
- Here functional means that you can use functional concepts like higher order functions, lazy evaluation, and rich API like map, filter, etc.

# Before we embark on FRP journey

- Lets write a simple program that prints first N Natural numbers greater than 0.

`naturalNumbers :: Int -> [Int]`

Ex. `naturalNumbers(5) = [1,2,3,4,5]`

# Imperative way

```
public class Example1 {  
    public static void main(String[] args) {  
        List<Long> naturalNumbers = naturalNumbers(100);  
        for (Long naturalNumber : naturalNumbers) {  
            System.out.println(naturalNumber);  
        }  
    }  
  
    public static List<Long> naturalNumbers(Long n) {  
        List<Long> naturalNumbers = new ArrayList<>();  
        for (long i = 1; i <= n; i++) {  
            naturalNumbers.add(i);  
        }  
        return naturalNumbers;  
    }  
}
```

# How can we make this code functional?

```
public class Example1 {  
    public static void main(String[] args) {  
        List<Long> naturalNumbers = naturalNumbers(100);  
        for (Long naturalNumber : naturalNumbers) {  
            System.out.println(naturalNumber);  
        }  
    }  
  
    public static List<Long> naturalNumbers(long n) {  
        List<Long> naturalNumbers = new ArrayList<>();  
        for (long i = 1; i <= n; i++) {  
            naturalNumbers.add(i);  
        }  
        return naturalNumbers;  
    }  
}
```

Recursion  
(show Java, Scala,  
and Haskell code)



# FP in Action

```
public static List<Long> naturalNumbersR(long n){  
    if (n == 0){  
        return new ArrayList<>();  
    }  
    List<Long> xs = naturalNumbersR(n - 1);  
    xs.add(n);  
    return xs;  
}
```

Java

Scala

```
def naturalNumbers(n: Int): List[Int] = {  
    n match {  
        case 0 => List()  
        case n => naturalNumbers(n-1) :+ n  
    }  
}
```

example.hs

```
1 naturalNumbers :: Int -> [Int]  
2 naturalNumbers 0 = []  
3 naturalNumbers n = naturalNumbers (n - 1) ++ [n]  
4
```

Haskell

# What will happen with Java and Scala version?

- $n = 100$
- $n = 1000$
- $n = 10000$
- $n = 100000$

# Possible issues with imperative code

- We are leaking implementation detail i.e. List
- Imperative code
- Not Lazy (Generate one million and take only first 5 elements)
- Not well encapsulated

# Can we use our own class with for..each ?

```
public class Example2 {  
    NaturalNumbers naturalNumbers =  
        NaturalNumbers.naturalNumbers(100);  
  
    for (Long naturalNumber : naturalNumbers) {  
        System.out.println(naturalNumber);  
    }  
}
```

@Sameer, @Aditya, and @Ankur please don't answer

Write the complete  
Example

# Iterables are very powerful type

- They were introduced in Java 5 to allow you to allow an object to be target of for..each statement.
- The full collection API is based on Iterable
- Iterable is a general concept found in most programming languages. .Net, Scala, Java, Ruby, etc.



































# Iterable implementations in JDK 8

```
*
Implementing this interface allows an object to be the target of
the "for-each loop" statement. See
<strong>
<a href="{@docRoot}/../technotes/guides/language/foreach.htm">
</strong>
@param <T> the type of elements returned by the iterator

@since 1.5
@jls 14.14.2 The enhanced for statement
/
public interface Iterable<T> {
/**
 * Returns an iterator over elements of type {@code T}.
 *
 * @return an Iterator.
 */
Iterator<T> iterator();

/**
 * Performs the given action for each element of the {@code Iterable};
 * until all elements have been processed or the action throws an
 * exception. Unless otherwise specified by the implementing class,
 * actions are performed in the order of iteration (if an iteration order
 * is specified). Exceptions thrown by the action are relayed to the
 * caller.
 *
 * @implSpec
 * <p>The default implementation behaves as if:
 * <pre>{@code
 *     for (T t : this)
 *         action.accept(t);
 * }</pre>
 *
 * @param action The action to be performed for each element
 * @throws NullPointerException if the specified action is null
 * @since 1.8
 */
}
```

Choose Implementation of Iterable (404 found)		
	AbstractCollection (java.util)	< 1.8 > (rt.jar) 
	AbstractList (java.util)	< 1.8 > (rt.jar) 
	AbstractObjectList in XSSimpleTypeDecl (com.sun.org.apache.xerces.internal.impl.dv.xs)	< 1.8 > (rt.jar) 
	AbstractPath (sun.nio.fs)	< 1.8 > (rt.jar) 
	AbstractQueue (java.util)	< 1.8 > (rt.jar) 
	AbstractSequentialList (java.util)	< 1.8 > (rt.jar) 
	AbstractSet (java.util)	< 1.8 > (rt.jar) 
	ArrayBlockingQueue (java.util.concurrent)	< 1.8 > (rt.jar) 
	ArrayDeque (java.util)	< 1.8 > (rt.jar) 
	ArrayLinkedList in VirtualFlow (com.sun.javaafx.scene.control.skin)	< 1.8 > (jfxrt.jar) 
	ArrayList (java.util)	< 1.8 > (rt.jar) 
	ArrayList in Arrays (java.util)	< 1.8 > (rt.jar) 
	ArrayListWrapper in ProxyBuilder (com.sun.javaafx.fxml.builder)	< 1.8 > (jfxrt.jar) 
	ArrayQueue (com.sun.jmx.remote.internal)	< 1.8 > (rt.jar) 
	AsLIFOQueue in Collections (java.util)	< 1.8 > (rt.jar) 
	AscendingEntrySetView in AscendingSubMap in TreeMap (java.util)	< 1.8 > (rt.jar) 

# How does Iterable<T> work?

- Check if the collection has a value by calling hasNext() method on iterator
- Call next() method to get the value
- Wait for result
- Store the return value from that method in a variable
- Use that variable to do something useful



# Limitations of Iterable<T>

- Synchronous
- Pull based
- Low level – lacking functional constructs

# Java 8 made Java Little Functional

- Streams
- Higher order functions using Lambda
- New immutable DateTime API
- Method references

# Java 8 made data flow programming easy

```
import java.util.stream.IntStream;
```

```
public class Example3 {
```

```
    public static void main(String[] args) {
```

```
        IntStream.rangeClosed(1,100).forEach(System.out::println);
```

```
    }
```

```
}
```

# Java 8 Streams are very powerful abstraction

- Allows you to work with collections in a functional manner using lambdas
- They are lazy by default
- Readonly and immutable

# Infinite Stream of Natural Numbers

```
import java.util.stream.LongStream;

public class Example4 {
    public static void main(String[] args) {
        LongStream.iterate(1, val -> val + 1).forEach(System.out::println);
    }
}
```

# Some functional methods

Map →

```
IntStream.rangeClosed(1, 10).map(num -> num + 10).forEach(System.out::println);
```

Filter →

```
IntStream.rangeClosed(1, 10).filter(num -> num % 2 == 0).forEach(System.out::println);
```

Combine Map and Filter →

```
IntStream.rangeClosed(1, 10).map(num -> num + 10).filter(num -> num % 2 == 0).forEach(System.out::println);
```

Sum, skip, limit()

```
IntStream.rangeClosed(1, 10).sum()
```

```
IntStream.rangeClosed(1, 100).skip(50).limit(10).forEach(System.out::println);
```

# Streams does not solve all the problems

- Streams are pull based
- Streams are synchronous in nature

# Lets change the requirement

Think of infinite sequence of natural numbers as an event stream.

We want two consumers one that would add 1 to the number and second consumer that would square the number.

Teacher == [1,2,3,4,5]

Student 1 (add 1 to  $n$ ) == [2,3,4,5,6]

Student 2 (square  $n$ ) == [1,4,9,16,25]



# Reactive Programming could help

- Programming model is based on push rather than pull
- Values are emitted asynchronously when ready without any blocking
- Allows multiple consumers to subscribe to the producer stream.

# Functional Reactive Programming

- FRP is a functional way to work with asynchronous stream (could be infinite) of events. Applying functions to data stream
- It is replacement of Observer pattern, which is usually implemented using callbacks or listeners.

# Reactive Extensions(Rx)

- Collection of helpful functions that let you do reactive programming with ease.
- It was created by .NET team in 2009
- Netflix in 2014 released RxJava
- Netflix uses RxJava to make all their service API asynchronous

# Iterables vs Observables

- Iterables allow you to query data at rest where as Observables allows you to query data in motion

Event	single item	multiple items
sync	T getData()	Iterable<T> getData
async	Future<T> getData	Observable<T> getData

# Iterable vs Observable

- Observable is push/async dual of Iterable

event	Iterable (pull)	Observable (push)
retrieve data	<code>T next()</code>	<code>onNext(T)</code>
discover error	throws <code>Exception</code>	<code>onError(Exception)</code>
complete	<code>!hasNext()</code>	<code>onCompleted()</code>

# What makes RxJava

- Observable – event stream source(producer)
- Observer – it subscribes to the Observable and listen for events
- Observer can react to events emitted by Observable
- More than one observers can subscribe to a single Observable



# Observer Interface

- `onNext` – This method is called by Observable zero or many times whenever Observable emits a value
- `onError` – To indicate failure scenarios. This stops the Observable and it won't make further calls
- `onCompleted` – To mark successful completion

# Say Hello to Observable

```
import rx.Observable;

public class Examples {

    public static void main(String[] args) {

        Observable<String> observable =
Observable.create(subscriber -> subscriber.onNext("Hello world"));

        observable.subscribe(System.out::println);

    }

}
```



# Observable are very powerful..

- They are composable in nature. They can be chained together or combined
- They can emit – single event, multiple events, or infinite events
- Free from callback hell: you can transform one Observable into another

# Code

Infinite natural number sequence with multiple subscribers.

One subscriber calculate factorial

Second factorial add 10 to the number

# Real world example

Sentiment analysis of  
twitter stream

Thanks

<https://github.com/sheekhargulati/frp-xke>