

# Practical Concurrent and Parallel Programming X

## Streams, Parallel Streams and RxJava

Jørgen Staunstrup

- Java Streams

- RxJava

Previously on PCPP:

**Distribution** (similar tasks to many threads)



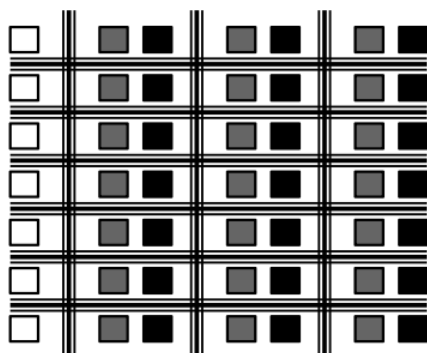
Streams:

**Pipeline** (different tasks to each thread F, G, H, ...)



# Motivation

## Supercomputer



□ IO  
■ Logic  
■ BRAM

DATALOGFORENINGEN | BRIAN VINTER

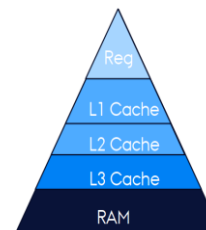
...

- Data is often communicated as a stream
- from a file,
- from the internet
- from the user interface
- Reducing memory overhead

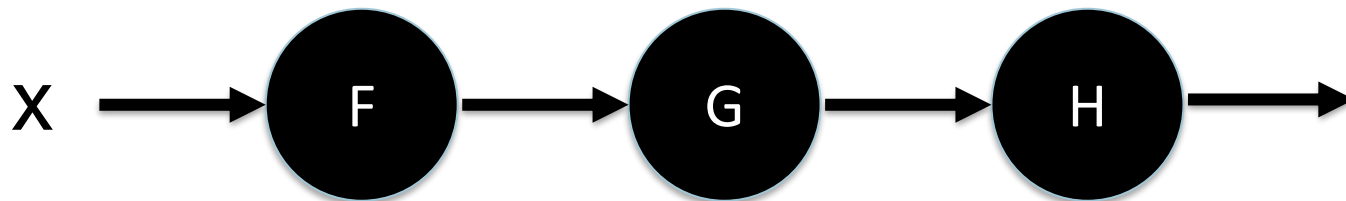
Processor @3.3GHz app 0.1 ns pr instruction  
L1 Data Cache Latency = 4 cycles  
L2 Cache Latency = 12 cycles  
L3 Cache Latency = 36 cycles (3.4 GHz i7-4770)  
RAM Latency = 36 cycles + 57 ns (3.4 GHz i7-4770)



· 3



Wasted inst
0
12
36
108
678



Java Stream

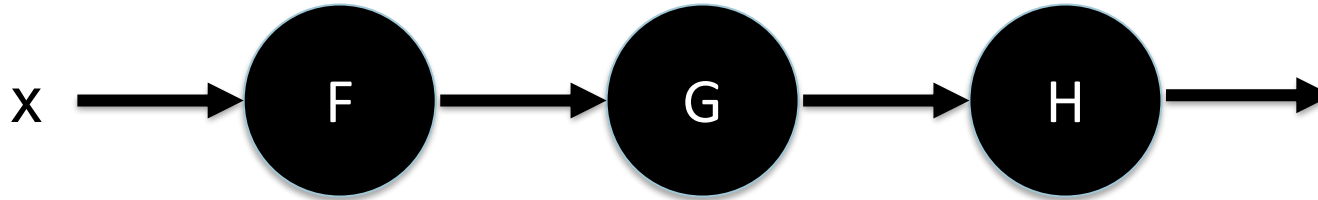
**$x().F().G().H()$**



Closely related to:  $H(G(F(x)))$

functional programming

# Stream example



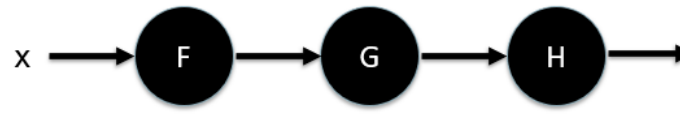
X is a stream of words: ..... ablaut able ableeze ... zyme

F discards all words with just one character

G discards all words that are capitalized

H counts number of words

# Java (imperative)



```
public static String F(String s) {  
    return s.length() > 1 ? s : null;  
}
```

```
public static String G(String s) {  
    if (s==null) return null;  
    return Character.isLowerCase(s.charAt(0)) ? s : null;  
}
```

```
public static int H(String s) {  
    return s==null ? 0 : 1;  
}
```

```
...  
while ((line= reader.readLine()) != null) {  
    word= F(line);  
    smallLetters= G(word);  
    count+= H(smallLetters);  
}
```



**x().F().G().H()**

```
count= readWords(filename) // makes a stream of words
      .filter( w -> w.length()>1 ) // F
      .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
      .count(); // H
```

Easy to parallelize:

```
count= readWords(filename)
      .parallel()
      .filter( w -> w.length()>1 )
      .filter( w -> Character.isLowerCase(w.charAt(0)) )
      .count();
```



**x().F().G().H()**

```
count= readWords(filename) // makes a stream of words
      .filter( w -> w.length()>1 ) // F
      .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
      .count(); // H
```

A sequence of stream  
method calls is commonly  
known as *stream pipeline*

Lambda expressions



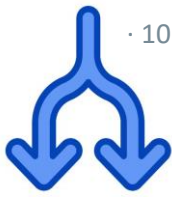


## The streams are lazy (driven by the terminal operation)

```
count= readWords(filename)  // source
      .filter( w -> w.length()>1 ) // intermediate
      .filter( w -> Character.isLowerCase(w.charAt(0)) )
      .count(); // terminal
```

There are three different types of stream elements:

- *sources* (arrays, collections, IO, generators)
- *intermediate operations* (transforming one stream into another (e.g. filter))
- *terminal operations* ( count, sum, forEach, ...)



## Provides the data for the stream

Examples of stream sources are:

- Input (files or network)
- The collection classes have a number of utilities, for example:  
**`Arrays.stream(arr)`**

Other examples of collections?

<https://howtodoinjava.com/java/stream/java-streams-by-examples/>



## Provides the data for the stream

Examples of stream sources are:

- Input (files or network)
- The collection classes have a number of utilities, for example:

**`Arrays.stream(arr)`**

List, Set, Map, ...

- All java collections have a **`stream()`** method
- **`Stream.of("Huey", "Dewey", "Louie")`**  
returns a stream of the three strings

<https://howtodoinjava.com/java/stream/java-streams-by-examples/>

# Intermediate operators

· 12



## A computation/transformation on *each* element of the string

Examples of intermediate operations are:

- filter – takes a lambda expression lambda returning a boolean, if the boolean is true the element is included in the output stream
- map – transforms each element
- limit( $n$ ) – returns a stream of the first  $n$  elements
- skip( $n$ ) – returns a stream without the first  $n$  elements
- distinct – returns a stream without duplicated elements
- sorted - returns a stream with the elements sorted

See Sestoft's Java precisely and the java documentation for a complete list

Section 24



```
count= readWords(filename) // makes a stream of words  
    .filter( w -> w.length()>1 ) // F  
    .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G  
    .count(); // H
```

Which operation(s) is/are  
intermediate?



```
count= readWords(filename) // makes a stream of words
      .filter( w -> w.length()>1 ) // F
      .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
      .count(); // H
```

filter

# BufferedReader (file)

· 15



## Exercise: 10.2

```
public static void main(String[] args) {  
    String filename = "src/main/resources/english-words.txt";  
    System.out.println( readWords(filename).count());  
}  
  
public static Stream<String> readWords(String filename) {  
    try {  
        BufferedReader reader= new BufferedReader(new FileReader(filename));  
        return ... // TO DO: Implement properly  
    } catch (IOException exn) { return Stream.<String>empty(); }  
}
```

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>

`Stream<String>`

`lines()`

Returns a `Stream`, the elements of which are lines read from this `BufferedReader`

# Defining a Java stream



- Using the `Arrays` class
  - `Arrays.stream(array)`
- Most Java collections have a method `stream()` that turns the collection into a stream
- `Stream.of(1,2,3,4)` creates a stream with those elements
- Functional iterators for infinite streams
  - `IntStream nats = IntStream.iterate(0, x->x+1)`
- **`BufferedReader`** (important for exercises)

```
Stream<String>
```

```
lines()
```

Returns a `Stream`, the elements of which are lines read from this `BufferedReader`





## Provides the *result* of the stream computation

Examples of terminal operations are:

- min, max, sum, average, count (number streams)
- forEach e.g., `forEach(System.out::println)`
- reduce / collect (introduced shortly)

<https://howtodoinjava.com/java/stream/java-streams-by-examples/>



```
count= readWords(filename) // makes a stream of words
      .filter( w -> w.length()>1 ) // F
      .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
      .count(); // H
```

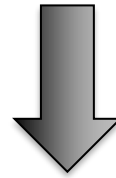
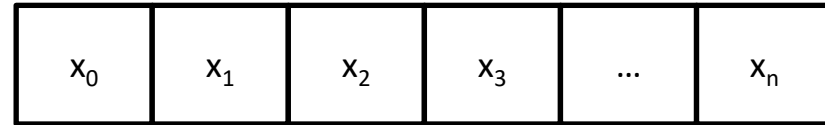
Which operation(s) is/are terminal?



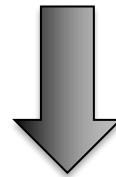
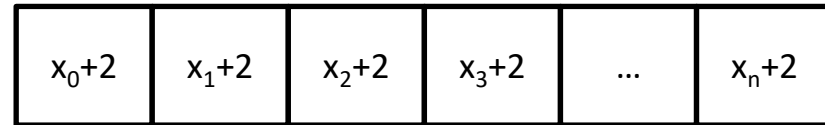
```
count= readWords(filename)  // makes a stream of words
    .filter( w -> w.length()>1 ) // F
    .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
    .count(); // H
```

count

# Intermediate operations



`map ( x -> x+2 )`



`limit (2)`



# Terminal operation reduce



- Reduce all elements of the stream to a single value by applying a function
- **reduce(identity, accumulator)**
  - **identity**: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream.
  - **accumulator**: The accumulator function takes two parameters: a partial result of the reduction and the next element of the stream.
- Example
  - Sum of squares of first 100 natural numbers
    - `IntStream.range(0,100).reduce(0, (a,b) -> a+b*b)`

An `Optional` value is:

- A container object which may or may not contain a non-null value.
- Needed in case the reduction is performed on an empty stream.

`findAny`

<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#findAny-->

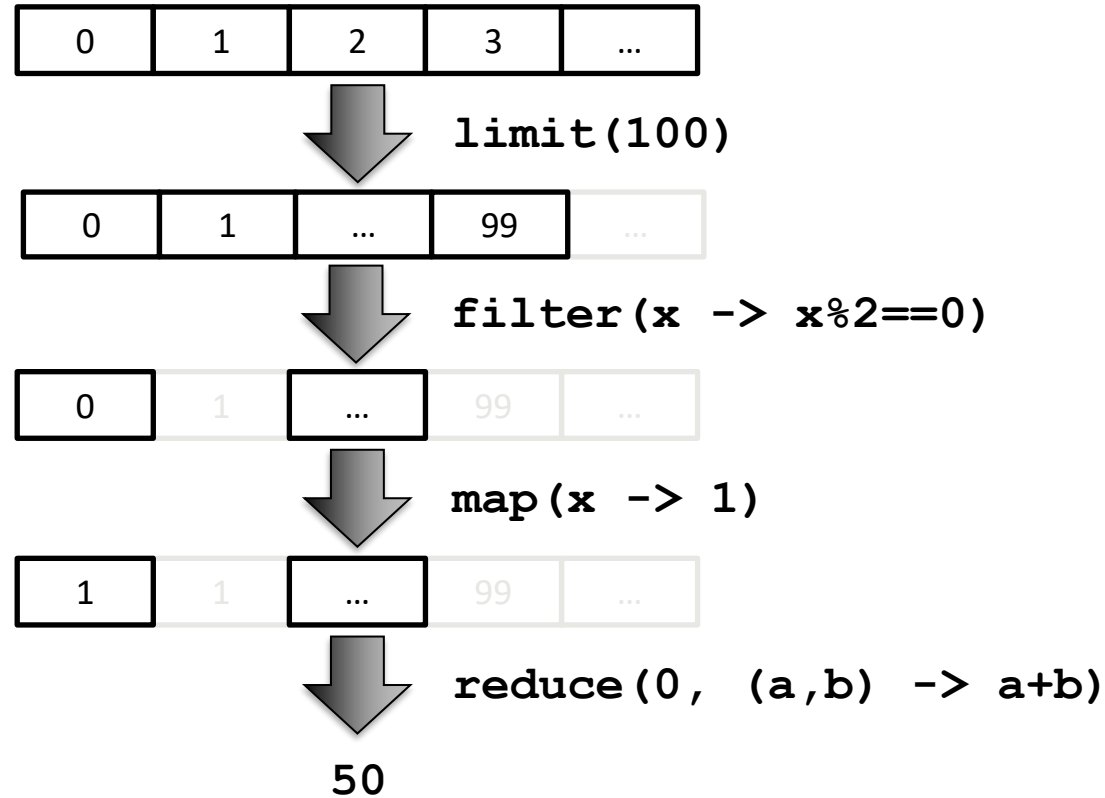


- Reduce can also be called without identity parameter
- Then it returns an `Optional` value
  - A container object which may or may not contain a non-null value.
  - Needed in case the reduction is performed on an empty stream.
- Example
  - Sum of squares of first 100 natural numbers
    - `IntStream.range(0,100).reduce((a,b) -> a+b*b).orElse(0)`
- There are other built-in reductions: sum, max, min, average, etc...

# Example with everything so far



· 25





# Ordering

· 27



Streams may or may not have a defined *encounter order*

- **List** and **Arrays** are intrinsically ordered
- **HashSet** is not ordered

An intermediate operation e.g., **sorted** transforms an unordered **Stream** into an ordered **Stream**

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Ordering>

# Parallel (intermediate)

· 28



```
IntStream.range(0, range)
    .parallel()
    .filter(i -> isPrime(i))
    .count()
```

← ordered

`IntStream.range` produces an *ordered* stream

`parallel` is an operation in `BaseStream`

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3  
java.util.stream

**Interface BaseStream**<T,S extends BaseStream<T,S>>

**parallel()**

Returns an equivalent stream that is parallel.

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html>



- If you try to modify a stream you are operating you will get a `ConcurrentModificationException` at runtime

- So don't do it 😊

- Cannot be detected at compile time. It depends on the programmer.

- From the Java documentation

*• Streams enable you to execute possibly-parallel aggregate operations over a variety of data sources, including even non-thread-safe collections such as `ArrayList`. This is possible only if we can prevent interference with the data source during the execution of a stream pipeline. [...] For most data sources, preventing interference means ensuring that the data source is not modified at all during the execution of the stream pipeline.*

# Example: stream of objects



```
class Employee {
    int id;
    String dept;
    int salary;

    public Employee(int id, String dept, int salary) {
        this.id      = id;
        this.dept    = dept;
        this.salary   = salary;
    }
    public int getId() { return this.id; }
    public String getDept() { return this.dept; }
    public int getSalary() { return this.salary; }
}

static private Stream<Employee> randomEmployees() {
    ...
}
```



- It is a reduction operation that allows to collect the results of a stream into a Java collection or summarize them using complex criteria
- For instance, converting a stream into a list

```
List<Integer> l = randomEmployees()  
    .limit(50)  
    .map(Employee::getId)  
    .collect(Collectors.toList());
```

# Terminal operation: collect + groupingBy

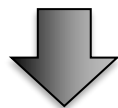
· 32



- Group employees by department

```
Map<String,List<Employee>> m = randomEmployees()  
    .limit(50)  
    .collect(Collectors.groupingBy(Employee::getDept));
```

Id: 0 Dept: CS Salary: 151	Id: 1 Dept: BI Salary: 150	Id: 3 Dept: DD Salary: 149	Id: 0 Dept: DD Salary: 10	...
----------------------------------	----------------------------------	----------------------------------	---------------------------------	-----



`collect(Collectors.groupingBy(Employee::getDept))`

CS	Id: 0 Dept: CS Salary: 151	...	
BI	Id: 1 Dept: BI Salary: 150	...	
DD	Id: 3 Dept: DD Salary: 149	Id: 0 Dept: DD Salary: 10	...

**Map<String,List<Employee>>**



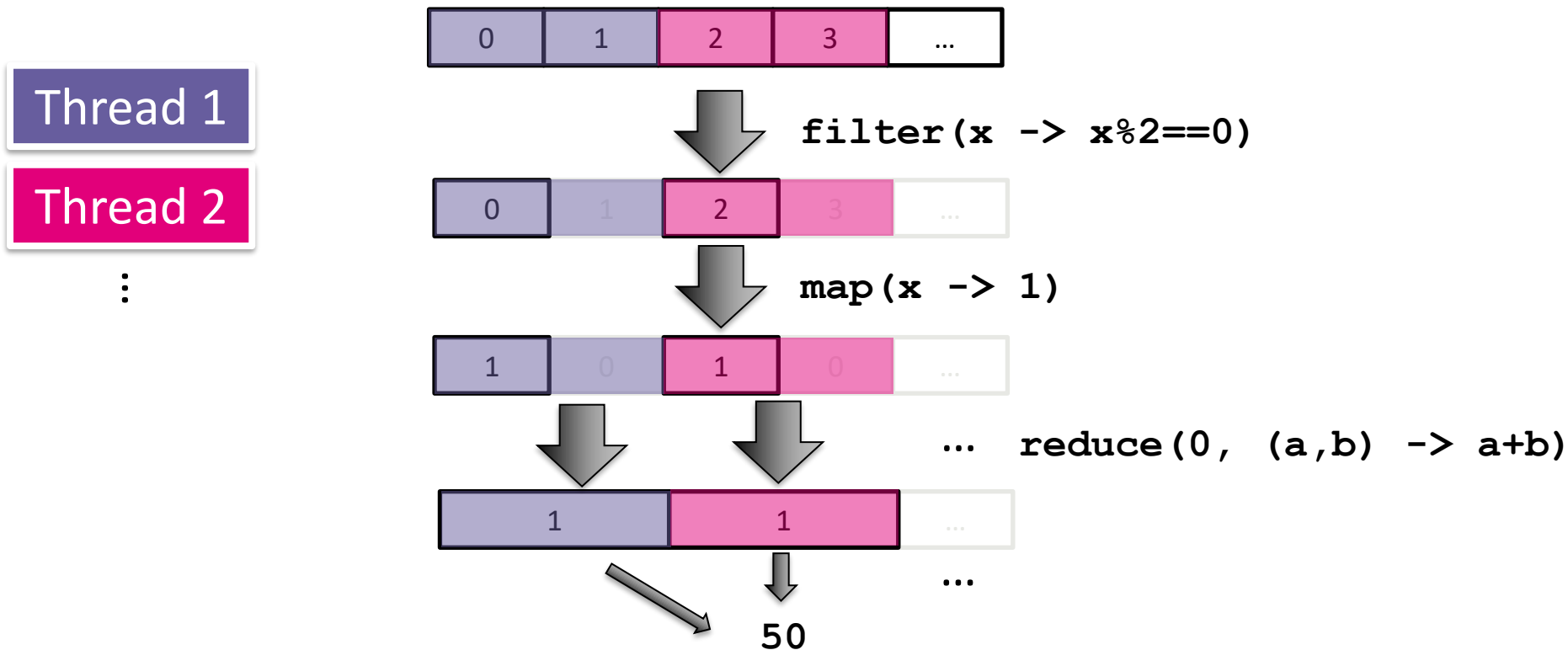
- You can create a parallel stream by calling
  - `parallelStream()` on, e.g., a collection, or
  - `parallel()` on a stream

# Java Parallel Streams

· 35



- Parallelization of streams is very easy (remember the beginning of the lecture). Disjoint streams (from the original stream) are assigned to distinct threads from a thread pool





- Since execution is parallel the processing of the stream is not guaranteed to be in order
- For instance, run this program
  - `IntStream.range(0,10).parallel().forEach(System.out::println);`
- In this case, it may be mitigated with `forEachOrdered`
  - `IntStream.range(0,10).parallel().forEachOrdered(System.out::println);`

# Counting primes on Java 8 streams

· 42



Our old standard Java for loop:

Classical efficient  
imperative loop

```
int count = 0;
for (int i=0; i<range; i++)
    if (isPrime(i))
        count++;
```

Sequential Java 8 stream:

```
IntStream.range(0, range)
    .filter(i -> isPrime(i))
    .count()
```

Parallel Java 8 stream:

```
IntStream.range(0, range)
    .parallel()
    .filter(i -> isPrime(i))
    .count()
```

PCPP 2019

# Counting primes on Java 8 streams

· 43



Our old standard Java for loop:

Classical efficient  
imperative loop

```
int count = 0;
for (int i=0; i<range; i++)
    if (isPrime(i))
        count++;
```

Sequential Java 8 stream:

Pure functional  
programming ...

```
IntStream.range(0, range)
    .filter(i -> isPrime(i))
    .count()
```

Parallel Java 8 stream:

```
IntStream.range(0, range)
    .parallel()
    .filter(i -> isPrime(i))
    .count()
```

PCPP 2019

# Counting primes on Java 8 streams

· 44



Our old standard Java for loop:

Classical efficient  
imperative loop

```
int count = 0;
for (int i=0; i<range; i++)
    if (isPrime(i))
        count++;
```

Sequential Java 8 stream:

Pure functional  
programming ...

```
IntStream.range(0, range)
    .filter(i -> isPrime(i))
    .count()
```

Parallel Java 8 stream:

... and thus  
parallelizable and  
thread-safe

```
IntStream.range(0, range)
    .parallel()
    .filter(i -> isPrime(i))
    .count()
```

PCPP 2019

Counting the primes in 2 ... 100.000

Using Mark7

Sequential	4891635.9	ns	21879.73	64
Stream	4953867.6	ns	63873.82	64
ParallelStream	1363886.8	ns	10621.99	256

Intel i7 (4 cores) speed-up: 3.6 x

# Agenda



- Java Streams
- **RxJava**

# RxJava and the UI



48

(input) UI elements (buttons, textfields, ...): **observables**

(output) UI elements (textfields, ...): **observers**

Example: RxJava for Android:

```
Button button; ...
RxView.clicks(button)
    .subscribe(i -> {
        //Perform some work here//
    });
```

[https://code.tutsplus.com/tutorials/rxjava-for-android-apps-introducing-rxbinding-and-rxlifecycle--cms-28565?\\_ga=2.125428746.1281241990.1512099718-1264555618.1502875086](https://code.tutsplus.com/tutorials/rxjava-for-android-apps-introducing-rxbinding-and-rxlifecycle--cms-28565?_ga=2.125428746.1281241990.1512099718-1264555618.1502875086)



```
Observable.from(letters)
    .map(String::toUpperCase)
    .subscribe(letter -> result += letter);
assertTrue(result.equals("ABCDEFGG"));
```

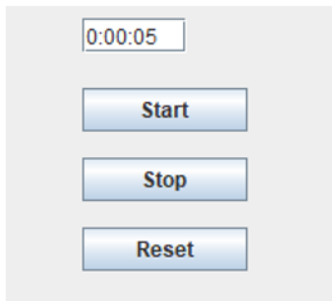
- *Observable* propagates data from a data source
- An *observer* receives data (via its subscribe method)

from <https://www.baeldung.com/rx-java>



# RxJava version of a Stopwatch

50



All three buttons must respond to clicking  
When started the display must update every second

⇒ 4 observers

- one for each button
- one for handling the clock ticking

```
timer.subscribe(display) ;  
rxPushStart.subscribe(displaysetRunningTrue) ;  
rxPushStop.subscribe(displaysetRunningFalse) ;  
rxPushStart.subscribe(displaysetAllzero) ;
```



```
const button= document.querySelector("button");
const observer = {
  next: function(value) {
    ... // handle click
  },
  error: function(err) { ... },
  complete: function() { ... }
};

// Create an Observable from event
const observable= Rx.Observable.fromEvent(button, "click");
// Subscribe to begin listening for async result
observable.subscribe(observer);
```

<https://rxjs.dev/guide/overview>

# Different types of Observables (2)

52



Observables can be created in many different ways, e.g.

```
String[] letters= {"a", "b", "c", "d", "e", "f", "g"};  
Observable<String> observable= Observable.fromArray(letters);
```

```
List<Integer> list= new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6));  
Observable<Integer> observable= Observable.fromIterable(list);
```

```
Observable<Integer> observable= Observable.range(11, 111);
```

```
Observable<Integer> observable= Observable.just(1, 4, 9, 221);
```

<https://betterprogramming.pub/rxjava-different-ways-of-creating-observables-7ec3204f1e23>



```
Observable<Integer> observable= Observable.range(11, 111).take(10);
```

```
Observable.range(11, 111)  
    .filter(i -> (i%2)==0)  
    .subscribe(System.out::println);
```

<https://github.com/ReactiveX/RxJava/wiki/Alphabetical-List-of-Observable-Operators>

# Many subscribers



54

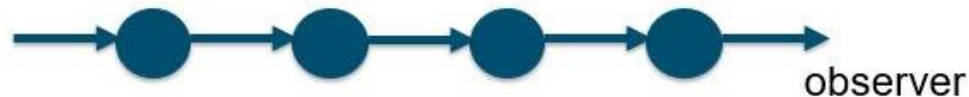
An observable can have several observes

important difference to Java stream !!!

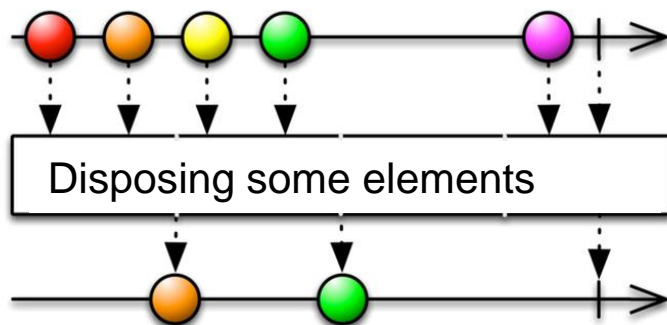
```
rxPush.subscribe(display1) ;  
rxPush.subscribe(display2) ;
```

# Backpressure

· 55



An observable may emit items so fast that the consumer can not keep up, this is called *backpressure*



Advice on handling backpressure

<https://medium.com/@srinuraop/rxjava-backpressure-3376130e76c1>

# RxJava vs Java stream

56



## RxJava

push-based  
many subscribers  
has rich API  
must be added as  
dependency

## Java Stream

pull-based (terminal operator)  
one subscriber  
few methods  
built into Java

<https://www.reactiveworld.net/2018/04/29/RxJava-vs-Java-Stream.html>

Libraries for many languages: Java, .net, JavaScript, ...

[ReactiveX website](#)

Nice introduction to RxJava: <https://github.com/ReactiveX/RxJava>