



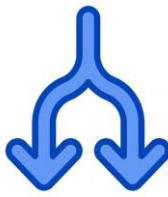
Practical Concurrent and Parallel Programming IX

RxJava

Jørgen Staunstrup

Motivations for Concurrency

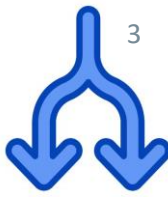
From Week01



Inherent: User interfaces and other kinds of input/output

Exploitation: Hardware capable of simultaneously executing multiple streams of statements

Hidden: Enabling several programs to share some resources in a manner where each can act as if they had sole ownership



- Motivation (Inherent concurrency)
- **Handling interaction in Java (Android, Swing, ...)**
- Reactive programming (RxJava)



```
St(a)rt, St(o)p or (R)eset:
```

```
Scanner myObj= new Scanner(System.in);  
System.out.println(" St(a)rt, St(o)p or (R)eset: ");  
String name= myObj.nextLine(); // Read user input
```

File: `week09/code-lecture/SimpleRead.java`

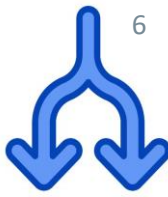
Problem: This will wait (and do nothing else) for user input



```
St(a)rt, St(o)p or (R)eset:
```

```
public class Button extends Thread {  
    public void run() {  
        Scanner myObj= new Scanner(System.in);  
        System.out.println(" St(a)rt, St(o)p or (R)eset: ");  
        String name= myObj.nextLine(); // Read user input  
        ...  
    }  
  
    new Button().start();  
    //The main thread continues ...  
}
```

File: week09/code-lecture/SimpleTRead.java



```
Button.setOnClickListener(  
    v -> // Code handling button  
);
```

This implicitly creates a thread



```
startButton.setOnClickListener( ... );
```

```
stopButton.setOnClickListener( ... );
```

```
resetButton.setOnClickListener( ... );
```



Not part of Java => external library

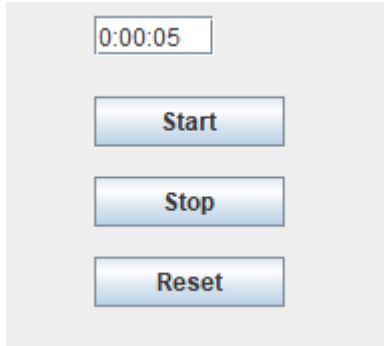
For the exercises we will use Java Swing

```
JButton startButton
```

```
startButton.addActionListener(e -> ...));
```

<https://docs.oracle.com/javase/tutorial/uiswing/index.html>

Example: the Stopwatch



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

⇒ 4 streams

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

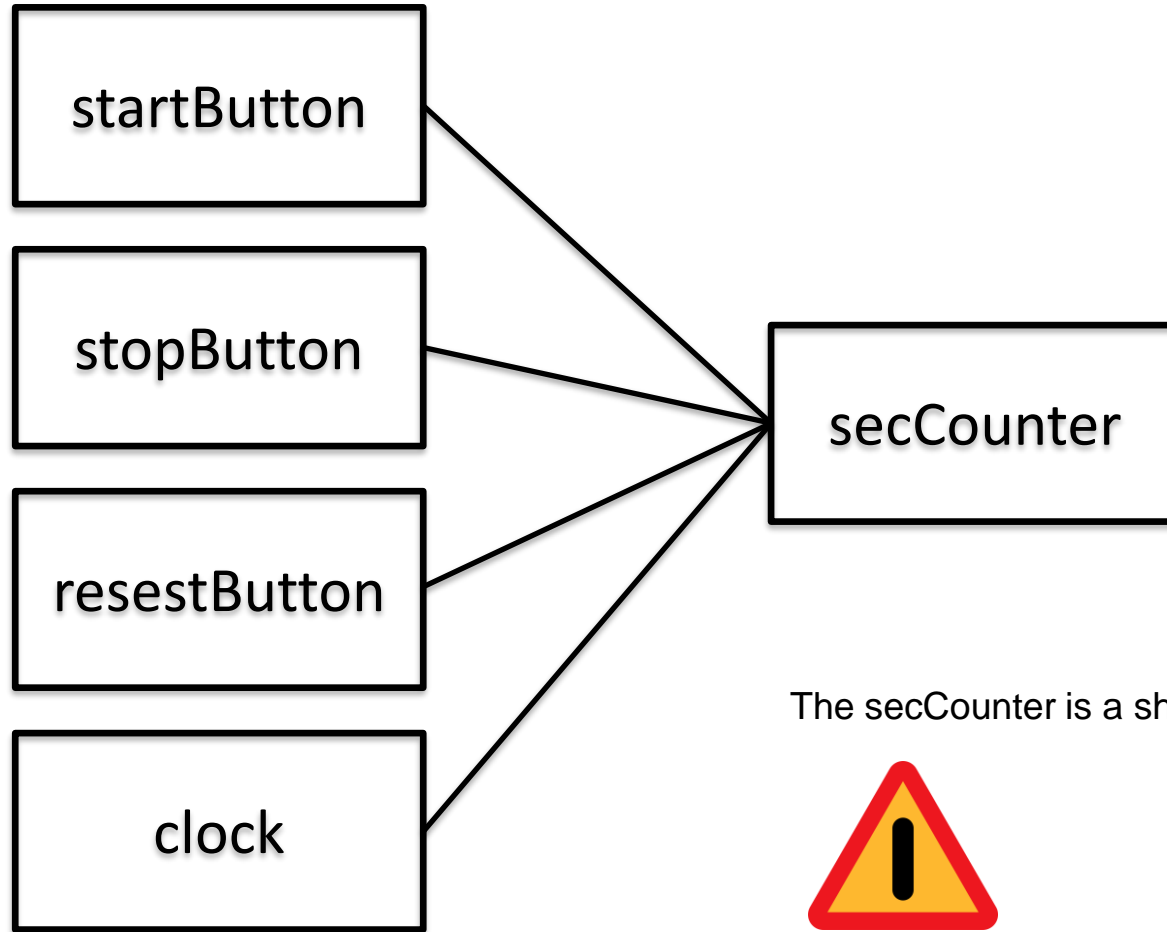
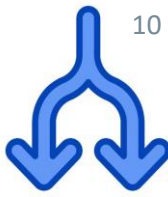
User interfaces and other kinds of input/output (*Inherent concurrency*)

Stopwatch (pseudocode)

9



```
stream startButton= new stream() -> {  
    await(startButton); SecCounter.setRunning(true);  
});  
  
stream stopButton= new stream() -> {  
    await(startButton); SecCounter.setRunning(false);  
});  
  
stream resetButton= new stream() -> {  
    await(startButton); SecCounter.reset();  
});  
  
stream clock = new stream() -> {  
    sleep(1 second); write(SecCounter.incr());  
});
```



The secCounter is a shared object



Swing example

11

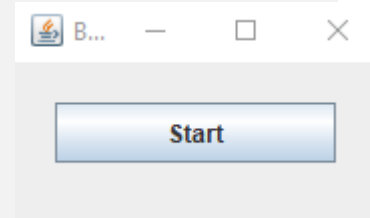
For the exercises we will use Java Swing

<https://docs.oracle.com/javase/tutorial/uiswing/index.html>

```
import java.awt.event.*;
import javax.swing.*;

class swingButton {
    public static void main(String[] args) { new swingButton(); }
    final private static JFrame f= new JFrame("Button Demo");
    final private JButton startButton= new JButton("Start");

    public swingButton(){
        f.setBounds(0, 0, 200, 120);
        f.setLayout(null);
        f.setVisible(true);
        startButton.setBounds(20, 20, 140, 30);
        startButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                System.out.println("Start pushed");
            }
        });
        // set up user interface
        f.add(startButton);
    }
}
```



File: week09/code-lecture/app.../swingButton.java

Swing example: Stopwatch

12



0:00:05

Start

Stop

Reset

```
private static JFrame lf;  
  
final private JButton startButton= new JButton("Start");  
final private JButton stopButton= new JButton("Stop");  
final private JButton resetButton= new JButton("Reset");  
  
final private JTextField tf= new JTextField();
```

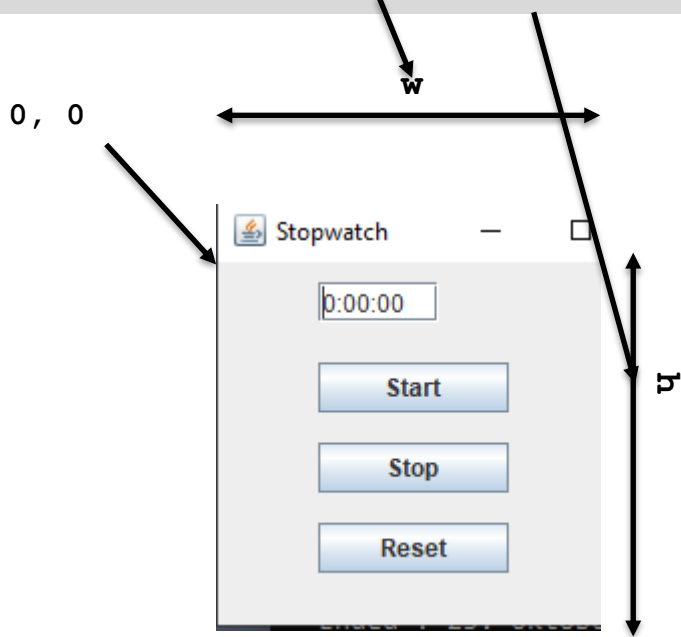
No need to learn Swing details for PCPP exercises !!!!

Swing canvas

13



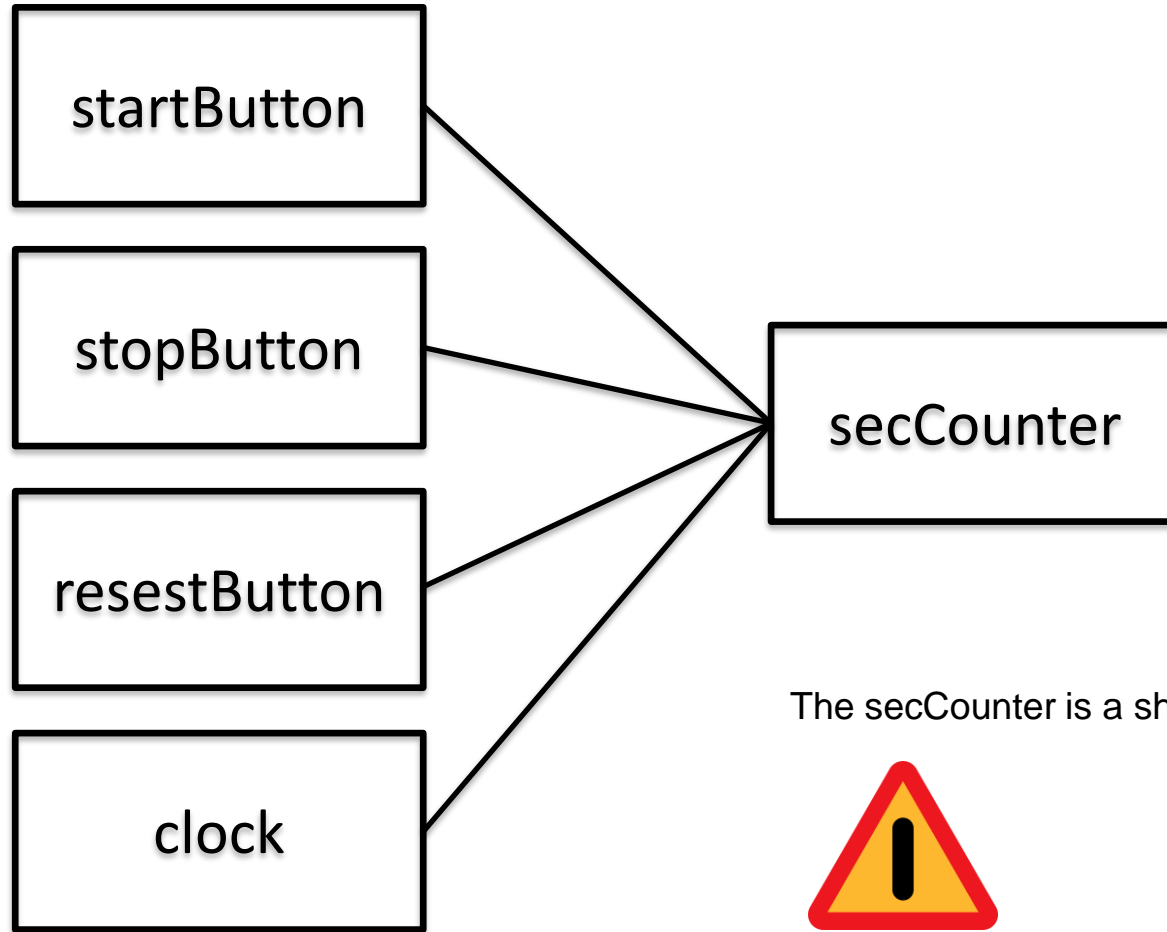
```
final JFrame f= new JFrame("Stopwatch");  
f.setBounds(0, 0, 220, 220);
```



```
startButton.setBounds(50, 50, 95, 25);  
stopButton.setBounds(50, 90, 95, 25);  
resetButton.setBounds(50, 130, 95, 25);
```

```
public void setBounds(int x, int y, int width, int height)
```

The unit of x, y, width and height is pixels



The secCounter is a shared object



startButton

16



0:00:05

Start

Stop

Reset

```
JButton startButton= new JButton("Start");
JTextField tf= new JTextField();
..
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        lC.setRunning(true);
    }
});
...
tf.setText(time);
```

"runnable"



Stopwatch: clock



17

```
// Background Thread simulating a clock ticking every 1 seconde
new Thread() {
    private int seconds= 0;

    @Override
    public void run() {
        try {
            while ( true ) {
                TimeUnit.SECONDS.sleep(1);
                myUI.updateTime();
            }
        } catch (java.lang.InterruptedException e) {System.out.println(e.toString());}
    }
}.start();
```

Complete code in: `code-exercises/.../Stopwatch.java`



```
public class SecCounter {  
    private int seconds= -1;  
    private boolean running= false;  
    ...  
    public SecCounter(int s, boolean r, JTextField tf){    ...    }  
  
    public synchronized void reset() {  
        running= false;  
        seconds= 0;  
        ...  
    }  
    public synchronized void setRunning(boolean running) { this.running= running; }  
  
    public synchronized int incr(){    ...    }  
}
```

= seconds

???

Stopwatch

19



Complete code in:

`Week09/code-exercises/...`

`Stopwatch.java`

`SecCounter.java`

`stopWatchUI.java`

and

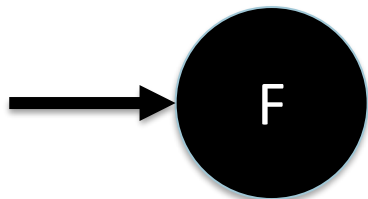
`Week09/code-lecture/...`

`Stopwatch.java`

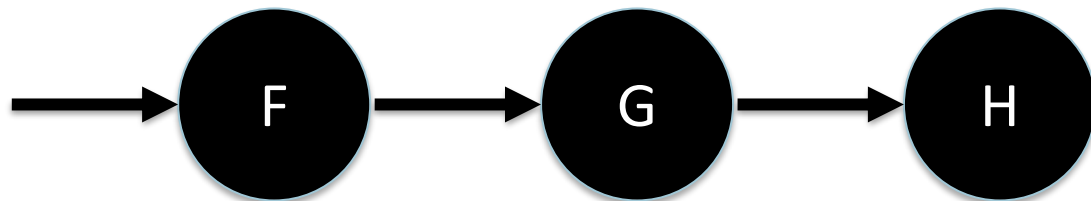
`SecCounter.java`

`stopWatchUI.java`

- Helicopter view on concurrency
- Handling interaction in Java (Android, Swing, ...)
- **Reactive programming (RxJava)**



Compute a function F whenever
a new input is provided



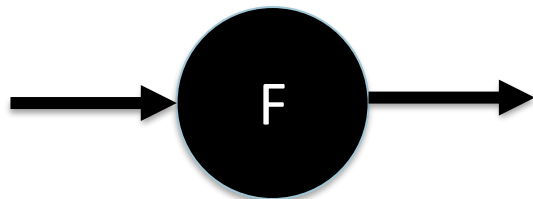
Similarity with Java streams, but some important differences
(more in a few minutes)

Observer/Observable

22



Data producer
observable



Data consumer
observer

```
public class Data extends Observable {
```

```
... this.setChanged(); notifyObservers();
```

```
}
```

```
public class Compute implements Observer {
```

```
public void update(Observable observable,  
                  Object data) {
```

```
...
```

```
}
```

```
}
```

Stopwatch in RxJava

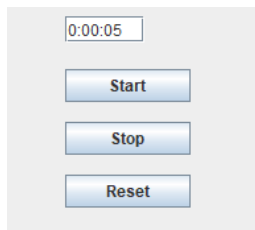
23



```
timer.subscribe(display);
```

observable

observer



```
timer1.subscribe(display1);  
timer2.subscribe(display2);
```



```
Observer<T> o= new Observer<T>() {

    @Override
    public void onSubscribe(Disposable d) {    }

    @Override
    public void onNext(<T> value) {
        ... // consume value
    }
    ...
}
```



```
Observable<T> ov
= Observable.create(new ObservableOnSubscribe<T>() {
    @Override
    public void subscribe(ObservableEmitter<T> e) {
        ...
        e.onNext( );
    }
})
```

More on other kinds of Observables later



To use RxJava (in your exercises) import (at least):

```
import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;
```

Your `build.gradle` must contain

```
implementation
'io.reactivex.rxjava2:rxjava:2.2.21'
```

See example: `code-exercises/app/build.gradle`

RxJava code for the Stopwatch (part 1)

27



The clock emitting ticks is an observable

```
Observable<Integer> timer
= Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(Observer<Integer> e) throws Exception {
        new Thread() {
            @Override
            public void run() {
                try {
                    while ( true ) {
                        TimeUnit.SECONDS.sleep(1);
                        e.onNext(1);
                    }
                } catch (java.lang.InterruptedException e) { }
            }
        }.start();
    }
});
```

RxJava code for the Stopwatch (part 2)

28



The buttons are also Observables

```
Observable<Integer> rxPush
    = Observable.create(new ObservableOnSubscribe<Integer>() {
        @Override
        public void subscribe(Observer<Integer> e) throws Exception {
            nameButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ee) {
                    e.onNext(1);
                }
            });
        }
    });
```

Complete code in: `code-exercises/... /rxButton.java`



The display is an Observer

```
Observer<Integer> display= new Observer<Integer>() {  
    @Override  
    public void onNext(Integer value) {  
        tf.setText(time); // tf id the swing object for the text field  
    }  
};
```

Different types of Observables (1)

30



A Java stream can be made into an Rx observable

```
public static Stream<String> readWords(String filename) {  
    ...    // from week 05  
}
```

Different types of Observables (1)

31



A Java stream can be made into an Rx observable

```
public static Stream<String> readWords(String filename) {  
    ...    // from week 05  
}
```

```
public static Observable<String> readWords  
= Observable.create(new ObservableOnSubscribe<String>() {  
    @Override  
    public void subscribe(ObservableEmitter<String> s) throws Exception {  
        try {  
            BufferedReader reader= new BufferedReader(new FileReader(filename));  
            String next= reader.readLine();  
            while (next != null) {  
                s.onNext(next); next= reader.readLine();  
            }  
        } catch (IOException exn) { System.out.println(exn); } // filename err  
    }  
});
```

Display is an Observer

32



```
readWords.subscribe(display);
```

```
final Observer<String> display= new Observer<String>() {  
    ...  
    public void onNext(String value) {  
        System.out.println(value);  
    }  
    ...  
};
```

Different types of Observables (2)



33

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);
```

Use take instead of limit

```
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



An observable can have several observes

important difference to Java stream !!!

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

Complete code in: `code-lecture/.../TextAndButton.java`



Two observables can be combined with **zip**

also possible to implement for Java streams

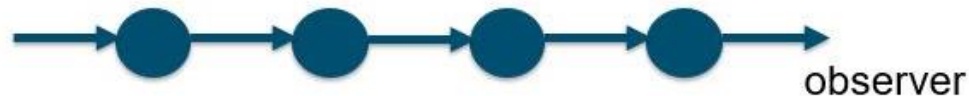
```
Observable<String> firstNames = Observable.just("James", "Jean-Luc", "Benjamin");  
Observable<String> lastNames = Observable.just("Kirk", "Picard", "Sisko");  
  
firstNames.zipWith(lastNames, (first, last) -> first + " " + last)  
    .subscribe(item -> System.out.println(item));
```

Complete code in: `code-lecture/.../zipDemo.java`

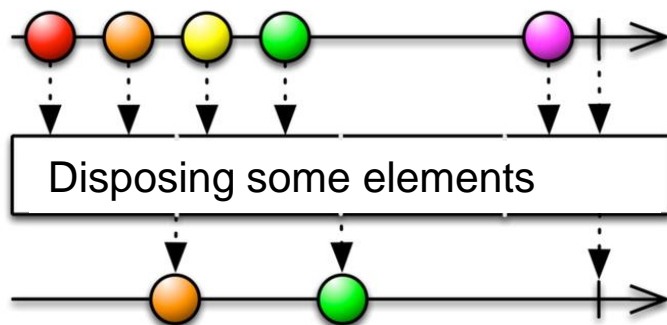
Source: <https://github.com/ReactiveX/RxJava/wiki/Combining-Observables#zip>

Backpressure

37



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>



By default, an Observable emits its data on the thread where you called the subscribe method

However, you may "schedule" a subscriber on a particular thread:

```
timer
    .subscribeOn(Schedulers.newThread())
    .filter(value -> myUI.running())
    .subscribe(display);
```

RxJava vs Java stream

39



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>

RxJava and the UI



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

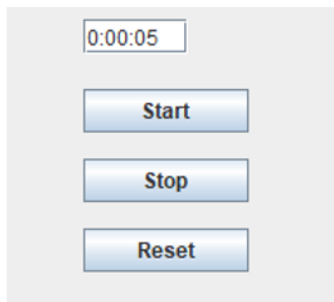
(output) UI elements (textfields, ...): **observers**

Not so easy to exploit with Swing (old)

Much better in Java for Android:

```
Button button; ...
RxView.clicks(button)
    .subscribe(aVoid -> {
        //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



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

⇒ 4 streams

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

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



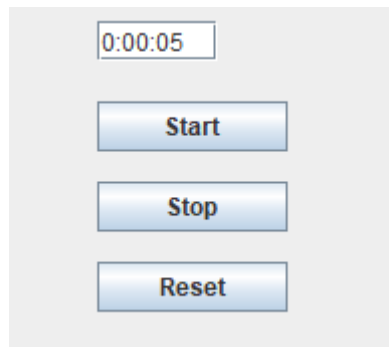
```
const button= document.querySelector("button");
const observer = {
  next: function(value) {
    ... // handle click
  },
  error: function(err) { ... },
  complete: function() { ... }
};

// 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>

Conclusion

44



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



- Helicopter view on concurrency
- Handling interaction in Java (Android, Swing, ...)
- Reactive programming (RxJava)
- **Follow up on Assignment3**

Shutting down an executor pool



```
public static void qsort(.., CyclicBarrier done, AtomicInteger count) {  
    if (a < b) {  
        ...  
  
        if ((j-a)>= threshold) count.incrementAndGet();  
        if ((b-i)>= threshold) count.incrementAndGet();  
  
        if ((j-a)>= threshold) {  
            pool.execute(new QuicksortTask(new Problem(arr, a, j), pool, c) );  
        } else qsort(...) // sequentially  
        if ((b-i)>= threshold) {  
            pool.execute(new solveProblem(new Problem(arr, i, b), pool, c) );  
        } else qsort(...) // sequentially  
  
        ...  
  
        if (count.decrementAndGet() == 0) { done.await(); pool.shutdown(); }  
    }  
}
```

From a solution to Assignment

47



```
ExecutorService pool =  
    Executors.newFixedThreadPool(noOfThreads);  
for( int i = 0; i < N; i++){  
    accounts[i] = new Account(i);  
}  
  
for( int i = 0; i<noOfThreads; i++) {  
    try {  
        pool.execute(() -> doNTransactions(NO_TRANSACTION));  
    } catch(Error ex){... }  
}  
pool.shutdown();
```

This calls for an explanation !!!

How does shutdown work?

48



ExecutorService (Java Platform SE 8) - Oracle

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.. An ExecutorService can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an ExecutorService. The shutdown() method will allow previously submitted tasks to execute before terminating ...

[docs.oracle.com](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#shutdown--)

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#shutdown-->

For the exam: you need to give an explanation either in your own words or by referring to the documentation, textbook or slides