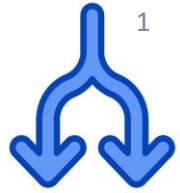
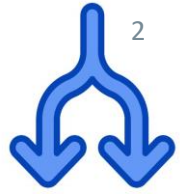


Practical Concurrent and Parallel Programming IX
RxJava

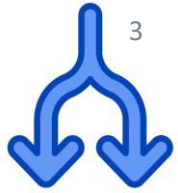
Jørgen Staunstrup

Draft version 6/10 -2021

Agenda



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



Asynchronous

Distributed

Parallel

Concurrent

…

Helicopter view on concurrency

This week, we will again focus on more qualitative aspects of concurrency

Programming concepts supporting nice and readable abstractions?

multiple streams of operations

[concurrency.pdf](#)
(week 01)

Helicopter perspective on concurrency (2)

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

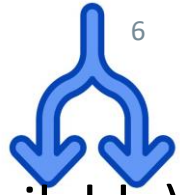
Hardware capable of simultaneously executing multiple streams of statements (*Exploitation*)

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

Our simple abstraction (for all three):

```
stream t = new stream(() -> {
  s1;s2;s3; ...
});
```

Example: exploitation



Speeding up a computation (when several processors/cores are available)

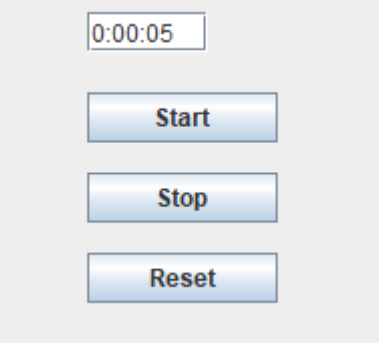
```
stream t1= new stream(() -> {
  for (int i=0; i<999999; i++)
    if (isPrime(i)) counter.increment();
});

stream t2= new stream(() -> {
  for (int i=1000000; i<1999999; i++)
    if (isPrime(i)) counter.increment();
});

stream t3= new stream(() -> {
  for (int i=2000000; i<2999999; i++)
    if (isPrime(i)) counter.increment();
});
```

Example: inherent

Handling user interfaces

```
stream stopApp= new stream(() -> {
   await(stopButton);
   exitApp;
});


stream searchField= new stream(() -> {
   newText= await(searchField);
   search(newText);
});
```

Example: the Stopwatch

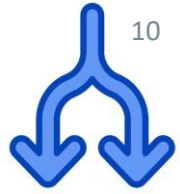


All three buttons must respond to clicking
When started the display must update every second

$\Rightarrow$ 4 streams
- one for each button
- one for the display

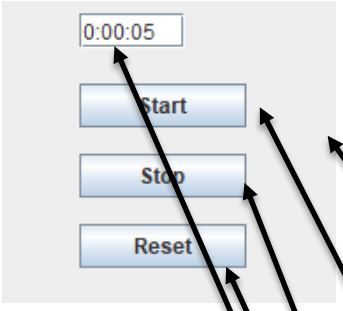
User interfaces and other kinds of input/output (*Inherent concurrency*)

Agenda



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

Digression: making a user interface in Java



For the exercises we will use Java Swing

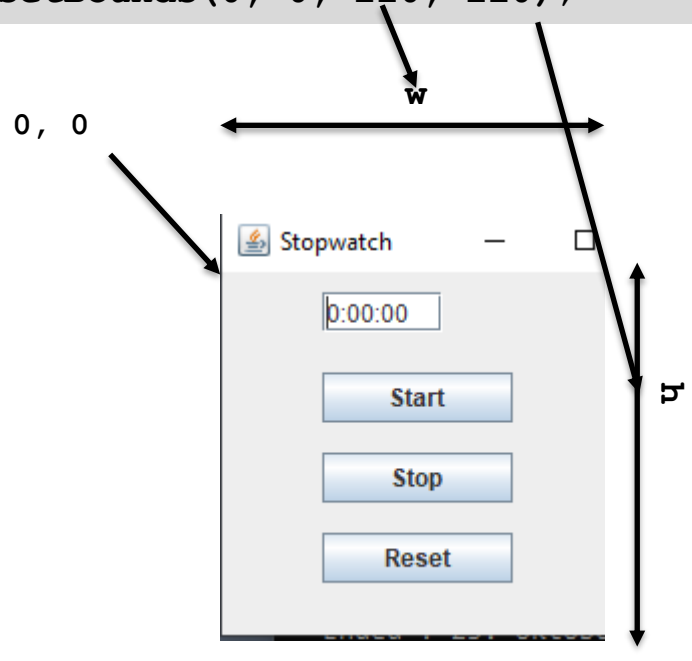
https://docs.oracle.com/javase/tutorial/uiswing/index.html

```
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

```
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)
```

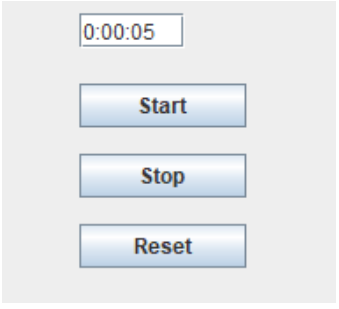
The unit of x, y, width and height is pixels

Code for stopwatch

```
public Stopwatch() {

   new stopwatchUI( ... );


 new Thread() {
   @Override
   public void run() {
     while ( true ) {
       sleep(1);
       myUI.updateTime();
     }
   }}.start();
```

```
class stopwatchUI {
  private Jbutton startButton
     = new JButton("Start");
  ...
  public void updateTime(){ ... }

  ...
  startButton.addActionListener(
    ... );

  // similar for the other buttons

}
```

Digression: making a user interface in Java



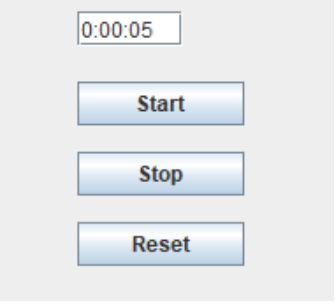
For the exercises we will use Java Swing

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

```
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"

Digression: making a user interface in Java



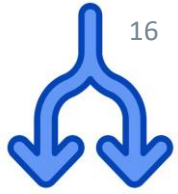
For the exercises we will use Java Swing

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

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

Inherent concurrency

Stopwatch: clock



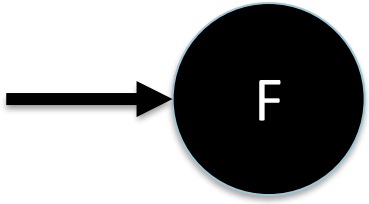
```
// Background Thread simulating a clock ticking every 1 seconde
new Thread() {
  private int seconds= 0;

  @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: **`ExercisesCode/…/Stopwatch.java`**

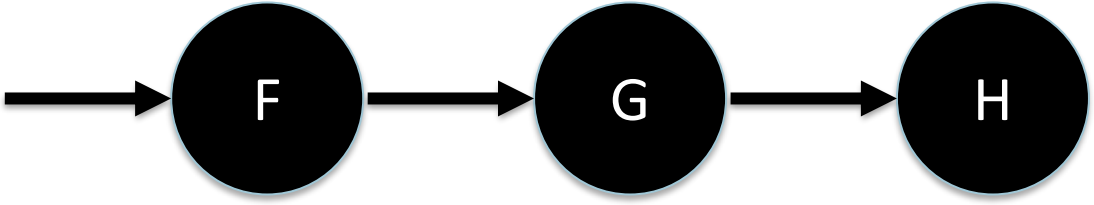
Agenda

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

Reactive programming (RxJava)

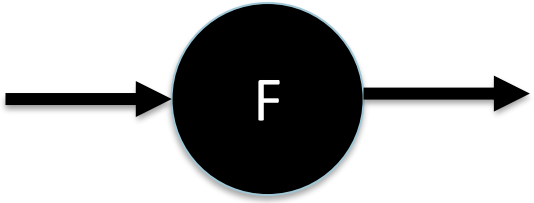


Compute a function F whenever
a new input is provided



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

Observer/Observable




```
public class Data extends Observable {


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


}
```

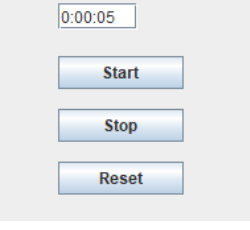
```
public class Compute implements Observer {

  public void update(Observable observable,
                     Object data) {
    ...
  }

}
```

Stopwatch in RxJava


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

observable

observer




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

RxJava Observer

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

  @Override
  public void onSubscribe(Disposable d) {  }

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

RxJava Observable

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

     }
   }
```

More on other kinds of Observables later

RxPracticalities

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: **`ExercisesCode/app/build.gradle`**

RxJava code for the Stopwatch (part 1)

The clock emitting ticks is an observable

```
Observable<Integer> timer
  = Observable.create(new ObservableOnSubscribe<Integer>() {
      @Override
      public void subscribe(ObservableEmitter<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)

The buttons are also Observables

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

See example: **`ExercisesCode/… /RxButton.java`**

RxJava code for the Stopwatch (part 3)

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)

A Java stream can be made into an Rx observable

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

```
readWords.subscribe(displayLines);
```

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

Different types of Observables (2)

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

RxJava Operators

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

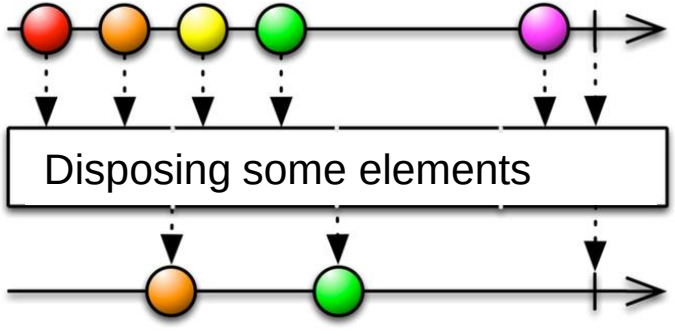
Use take instead of limit

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

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

Backpressure

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

Schedulers

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 e.g.:

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

RxJava vs Java stream

RxJava	Java Stream
push-based	pull-based (terminal operator)
many subscribers	one subscriber
has rich API	few methods
must be added as dependency	built into Java

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

Reactive programming

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 do this with Swing (old)

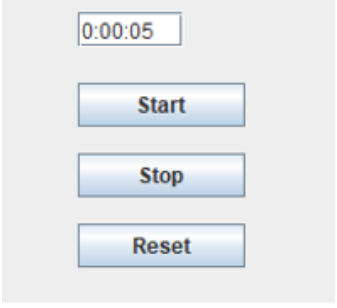
Much better in Java for Android:

```
Button button= (Button) findViewById(R.id.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

RxJava version of Stopwatch

Slide 8



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

⇒ 4 streams
- one for each button
- one for the display

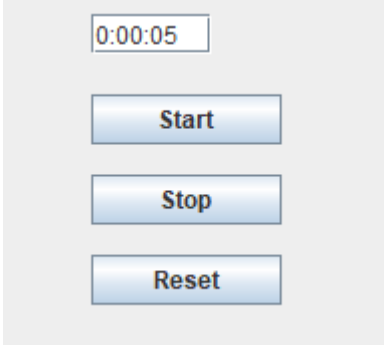
```
timer.subscribe(display);
rxPushStart.subscribe(displaysetRunningTrue);
rxPushStop.subscribe(displaysetRunningFalse);
rxPushStart.subscribe(displaysetAllzero);
```

RxJS (Javascript)

```
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




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