# Practical Concurrent and Parallel Programming IX

## RxJava

**IT University of Copenhagen**

**Jørgen Staunstrup**

Friday 2020-10-23

Starts at 8:00

# Plan for today

- RxJava

  timer.filter(value -> myUI.running()).subscribe(display);

looks similar, but is not the same as Java streams

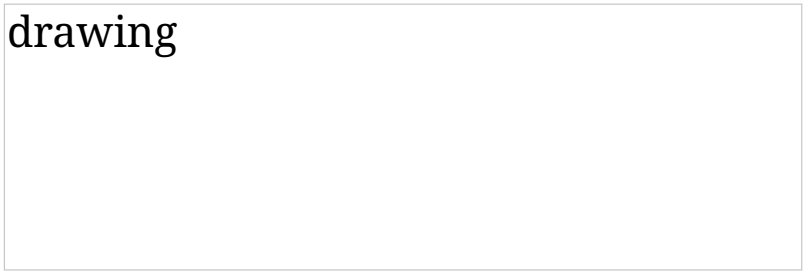
- Helicopter perspective on concurrency
- - Shared variables (monitors)
- - Message passing
- - RxJava (Reactive programming)

# Hand-out material

see readme file for this week:

- Concurrency concepts by Jørgen Staunstrup
- Introduction to RxJava
- Reactive Programming Operators
- RxJava Tutorial
- A short intro to message passing

# Helicopter perspective on concurrency (1)




Let us forget about thread overhead, memory models, locking costs, etc.

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

**Does the programming concepts support nice and readable abstractions?**

# Helicopter perspective on concurrency (2)

Fundamental motivations for concurrency:

- User interfaces and other kinds of input/output (Nygaard Intrinsic).
- Hardware capable of simultaneously executing multiple streams of statements (Nygaard Exploitation),
  a special (but important) case is communication and coordination of independent computers on the internet.
- Enabling several programs to share some resources in a manner where each can act if they had sole ownership (Nygaard Concealed).

All three motivations for concurrency may be handled with the simple abstraction of an *independent* stream of statements.

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

Not to be confused with the Java data type `Stream`.

# Examples of exploitation of concurrency

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 of intrinsic concurrency

Handling a user interfaces

```
//Code for a user interface
stream searchField= new stream(() -> {
  newText= await(searchField);
      search(newText);
  });

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

# Example of concealed concurrency

Apps on a smartphone.

```
//e-mail app

stream Email= new stream(() -> {
  do {
    await(email);
    notify(user);
    store(email_in_inbox);
  } forever
})

// alarm app
stream alarm= new stream(() -> {
  do {
    alarmAt= await(start button);
    waitUntil(alarmAt);
    notify(user);
  } forever
})
```

# Stream coordination

Every now and then streams need to coordinate (their execution), e.g. to

- exchange data
- use a common ressource (screen, network, ...)
- ...

Examples of coordination mechanism are: monitors (locks), semaphores, messages, wait/signal, ...

These can be divied into two classes: *object sharing* and *message passing*

Many programming languages/protocols focus on one of the two

Concurrent Pascal was an early object oriented language based on monitors

The Http protocol is based om message passing (with the extra feature that only the client can initate communication). Other examples are e-mail/SMS (see https://cs.lmu.edu/~ray/notes/messagepassing).

Java is like a stew, it has both, in a large number of variations.

Message passing and object sharing are equally powerful.

# Example: the Stopwatch


drawing

```
send(UIstream, "tick");    message= receive(Clockstream);
```

# Stopwatch code (message passing)

```
stream clockstream= new stream(() -> {
  while  true  {
    TimeUnit.SECONDS.sleep(1);
    if (running) {
      seconds++;
      send(UIstream, "tick"+seconds);
    }
  }
}}

//UI stream
messages.addListener( (message) -> {
    updateUI(message.seconds);
});
```

# Stopwatch code 1 (shared objects)

```
object SecCounter {
  private int t;  private boolean running;

  setRunning(boolean running) { this.running= running;  }

  boolean running(){ return this.running;  }

  int incr(){
    if running seconds++;
    return running;
  }

  reset(){
    t= 0;  running= false;
  }
}
```

# Stopwatch code 2(shared objects)

Using this shared object, the five streams become:

```
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 textField= new stream(() -> {
  await
    tick: write(SecCounter.incr());
  }
});
stream clock = new stream(() -> {
    await{1 second};
    notify(textField);
});
```

# Stopwatch code 3(shared objects)

```
object SecCounter {
  private int t;  private boolean running;
  ...
}

stream startButton= new stream(() -> { ... });

stream stopButton= new stream(() -> { ... });

stream resetButton= new stream(() -> { ... });

stream textField= new stream(() -> { ... });

stream clock = new stream(() -> { ... });
```

The message passing and shared variables solutions look very similar,

but now consider a stopwatch with two timers.

# Stopwatch2 (message passing)

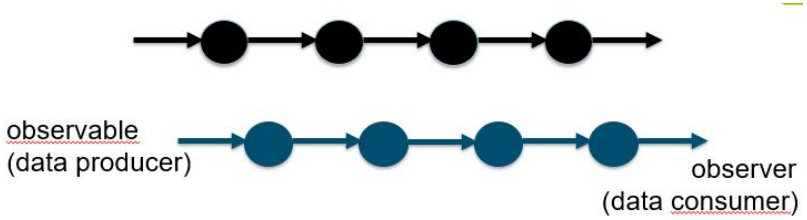
```
stream clockstream= new stream(() -> {
  while true {
    TimeUnit.SECONDS.sleep(1);
      if running1 {
        seconds1++;
        send(UIstream, "tick1"+seconds1);
      }
      if running2 {
        seconds2++;
        send(UIstream, "tick2"+seconds2);
      }
  }
}}

messages.addListener( (message) -> {
    if (message.info=="tick1")  updateUI1(message.seconds);
    else  updateUI2(message.seconds);
});
```

The `if ... then ... else` is called *external selection*.

Imagine the code for a stopwatch (or somthing similar) with a large number of independent counters.

# Reactive programming

streams of messages



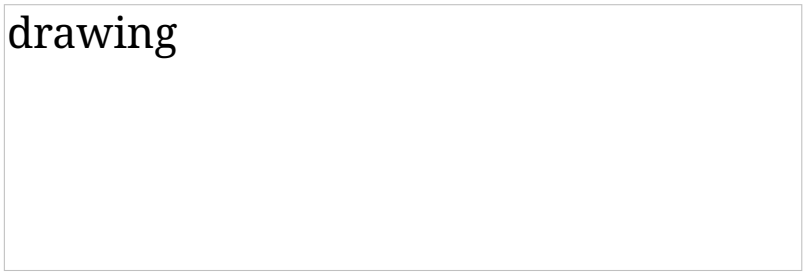

**Stopwatch using RxJava**

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

drawing

# Stopwatch with two timers

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



# RxJava and the UI

(input) UI elements (buttons, textfields, …) can be made observables

(output) UI elements (buttons, textfields, …) can be made observers

Short reminder about `Observable` in Java

# RxJava practicalities

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;
```

This requires that these jar files are accessible: `rxjava-2.2.4.jar, reactive-streams-1.0.2.jar`

These are not the most recent versions of the RxJava libraries, but they are suggested in the tutorial: https://www.tutorialspoint.com/rxjava/index.htm in the readings for this week.

You are welcome to use newer libraries; if you do be prepared to make the necessary modifications in your code.

# RxJava code for the Stopwatch (part 1)

```
public class StopwatchRx {
  private static stopwatchUI myUI;

  //Observable simulating clock ticking every second
    final static Observable<Integer> timer
      = Observable.create(new ObservableOnSubscribe<Integer>() {
        @Override
        public void subscribe(ObservableEmitter<Integer> e) throws Exception {
          Thread t= new Thread() {
            @Override
            public void run() {
              try {
                while ( true ) {
                  TimeUnit.SECONDS.sleep(1);
                  e.onNext(1);
                }
              } ...
        };
        t.start();
    }});
```

# RxJava code for the Stopwatch (part 1)

```
  // Observer updating the display
  final static Observer<Integer> display= new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {  }
    @Override
    public void onNext(Integer value) {  myUI.updateTime(); }
    ...
  };
    public static void main(String[] args) {
    ...
    myUI= new stopwatchUI(0, f);
    timer.subscribe(display);
    }
}
```

The complete code can be found here: Link to StopwatchRx.java

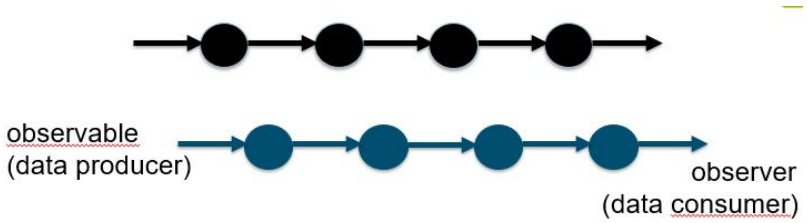
# RxJava operators

```
public static void main(String[] args) {
  ...
  myUI= new stopwatchUI(0, f);
  timer
    .filter(value -> myUI.running())
    .subscribe(display);
}
```

Many other filtering operators, e.g.: `distinct`, `elementAt`, `filter`, `first`, `ignoreElements`, `last`, `...`

Note that `Observable filter` and `Stream filter` are very similar but operate on different types.

# Backpressure




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

Many mechanisms to handle this phenomenon: https://medium.com/@srinuraop/rxjava-backpressure-3376130e76c1.

see also the reading list:

Reactive Programming Operators

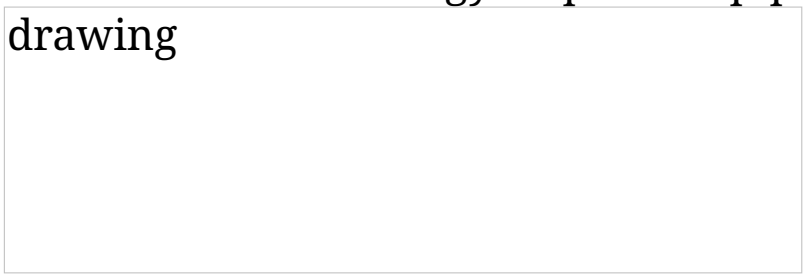
# Schedulers

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

You may "schedule" a subscriber on a particular thread e.g.:

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

From lecture 6: Strategy 1 - parallel pipelining - (not used):




In RxJava parallel pipelining is possible (but limited by no of cores).

# Rx and the user interface

In Java for Android input (UI) elements are handled like this

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //Perform some work//
    }
});
```

UI input elements (Buttons, text entry, ...) can be viewed as ObServables:

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

The Android library for this is called *RxBinding*.
https://code.tutsplus.com/tutorials/rxjava-for-android-apps-introducing-rxbinding-and-rxlifecycle cms-28565?
_ga=2.125428746.1281241990.1512099718-1264555618.1502875086

# 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

# Thank you for today

Any questions?