GOING REACTIVE WITH RXJAVA

JAVACRO Rovinj – May 2016

Hrvoje Crnjak
Java Developer @ Five
@HrvojeCrnjak

Going Reactive with RxJava

So what does it mean for the App to be Reactive?

His Majesty

Reactive Manifesto

Reactive App should be …

Message-Driven

- Components communicate via asynchronous messages (errors are messages also)

Resilient

- System stays responsive in the face of failure

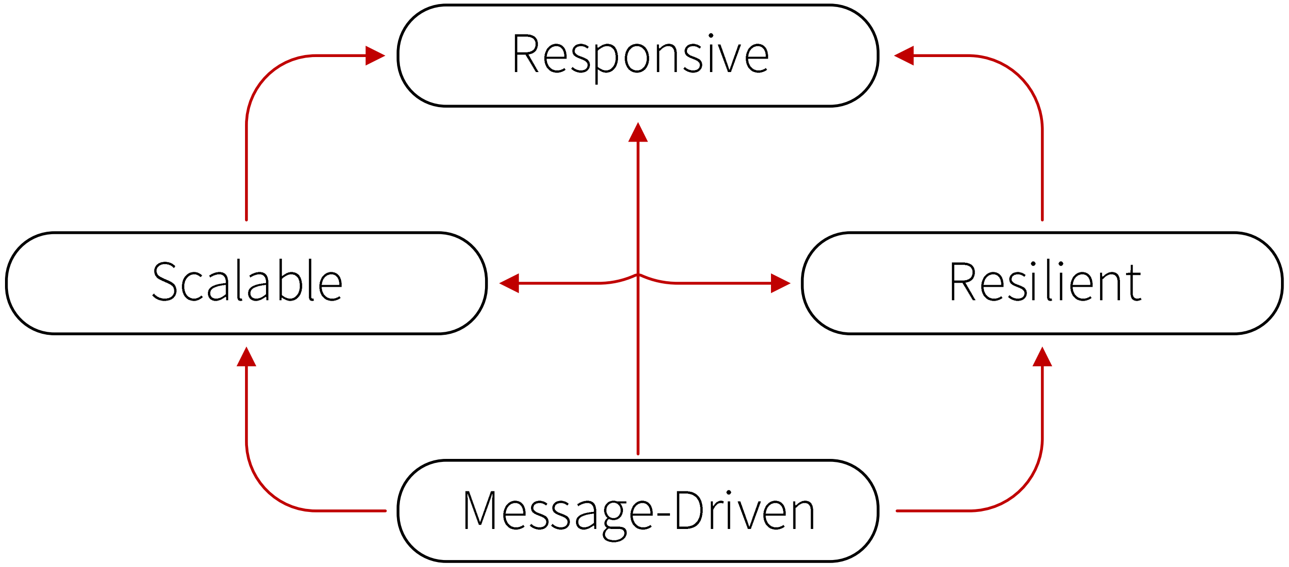
Elastic (Scalable)

- System stays responsive under varying workload

Responsive

- System responds in **timely** manner if at all possible

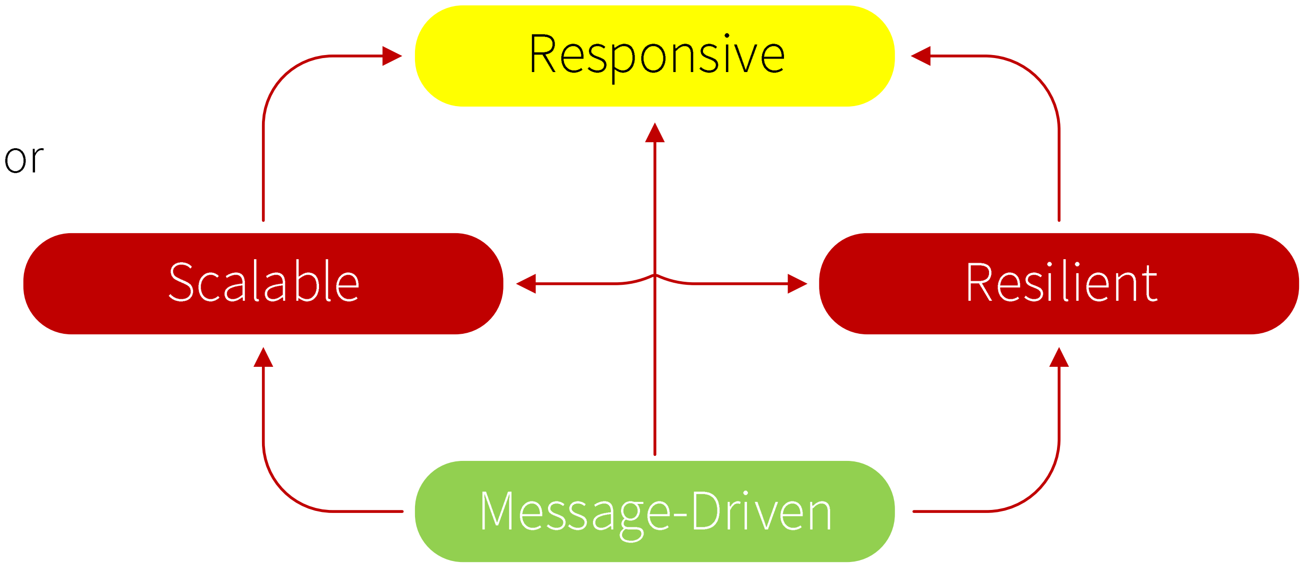
Or if we put it in a diagram . . .



OOP, State of the Union

State and Behavior
are Joined

State is Mutated



Error Handling up to
the Client

We'll pack Events
into Messages

🟢 The Good
🟡 The Bad
🔴 The Ugly

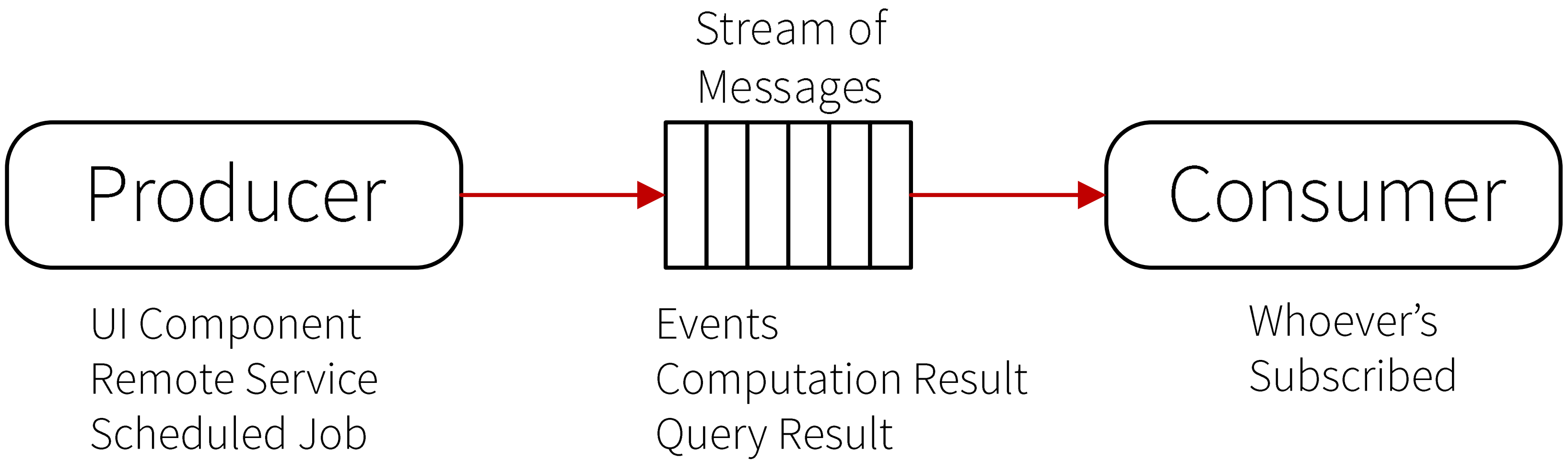
So how does RxJava fit into all of this?

Event-Driven → Message-Driven

Everything is a message

- Including errors

Everyone (each component) can be Producer and Consumer of messages

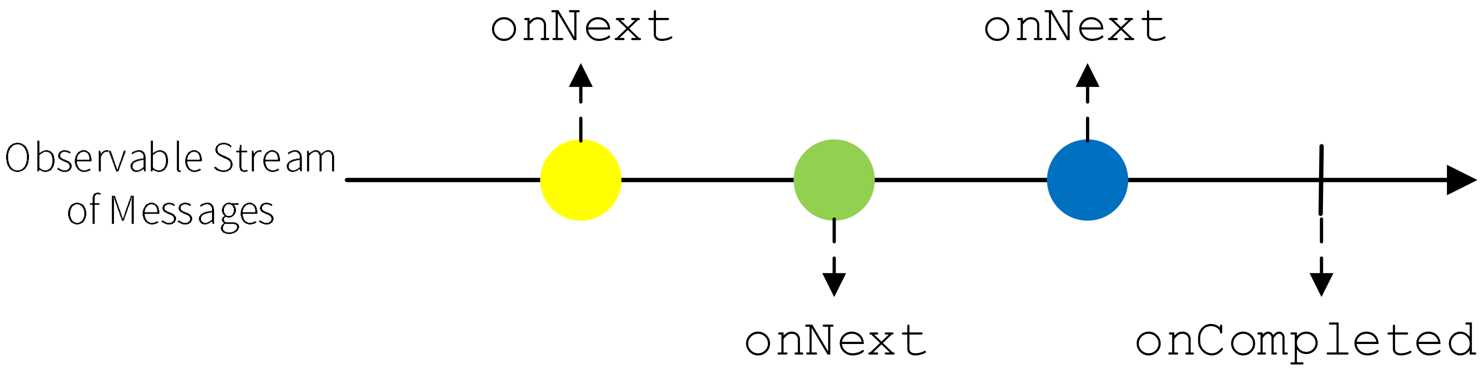


Making Streams of Messages with RxJava

Observable – Representation of the Message **Producer**

Observer – Representation of the Message **Consumer**

- `onNext`
- `onCompleted`
- `onError`

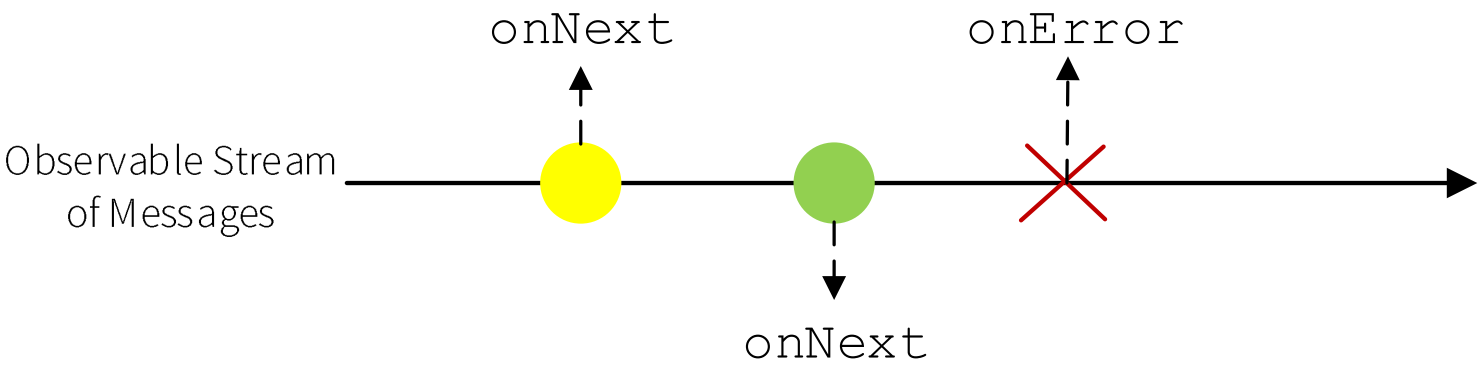


Making Streams of Messages with RxJava

Observable – Representation of the Message **Producer**

Observer – Representation of the Message **Consumer**

- `onNext`
- `onCompleted`
- `onError`



Making an Observable

Predefined Observable templates

- `Observable.from`
- `Observable.just`
- Factory Methods
  - `Observable.interval` , `Observable.range` , `Observable.empty`

```
List<String> list = Arrays.asList("blue", "red", "green", "yellow", "orange");
Observable<String> listObservable = Observable.from(list)
```

```
Observable<Character> justObservable = Observable.just('R', 'x', 'J', 'a', 'v', 'a');
```

```
Observable<Integer> rangeObservable = Observable.range(1, 10);
```

```
Observable<Long> intervalObservable = Observable.interval(500L, TimeUnit.MILLISECODS);
```

Making an Observable

The real Power lies in

- `Observable.create`

```java
public static Observable<SomeDataType> getData(String someParameter) {

    return Observable.create(subscriber -> {
        try {
            SomeDataType result = SomeService.getData(someParameter);
            subscriber.onNext(result);
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

Consuming an Observable

At it’s Core very simple

- `observableInstance.subscribe`

```
observableInstance.subscribe(new Observer<SomeDataType>() {
    @Override
    public void onNext(SomeDataType message) {
        // Do something on each Message received
    }

    @Override
    public void onError(Throwable error) {
        // Do something on Error
    }

    @Override
    public void onCompleted() {
        // Do something when Observable completes
    }
});
```

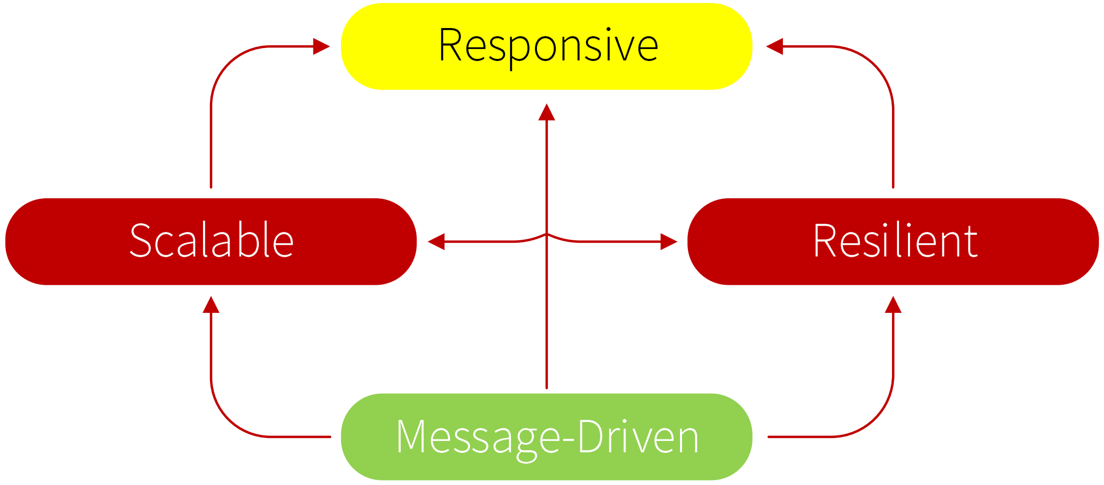
Consuming an Observable

At it’s Core very simple

- `observableInstance.subscribe`

```
observableInstance.subscribe(
        (SomeDataType message) -> {/*onNext*/},
        (Throwable error) -> {/*onError*/},
        () -> {/*onCompleted*/});
```

OOP + RxJava, State of the Union



Observer + Observable

Making the System Scalable

How to approach the problem

- Scale up – **I don’t think so**
- Scale out – **That’s more like it**
  - A lot of Cores and Memory!

Desired Characteristics of our System

- Program logic should execute in Parallel
- Data immutability is Allowed/Encouraged

The answer

- Functional programming

Making the System Scalable

Why FP Approach

- State Handled Transparently
- Highly composable

When we apply this to Rx world …

- Data manipulation
  - Composable FP style Observable methods
- State change
  - Each change of state will be a new message in the Stream

Composable methods with RxJava

There are methods for

- Content filtering
- Time filtering
- Data transformation
- Stream composition

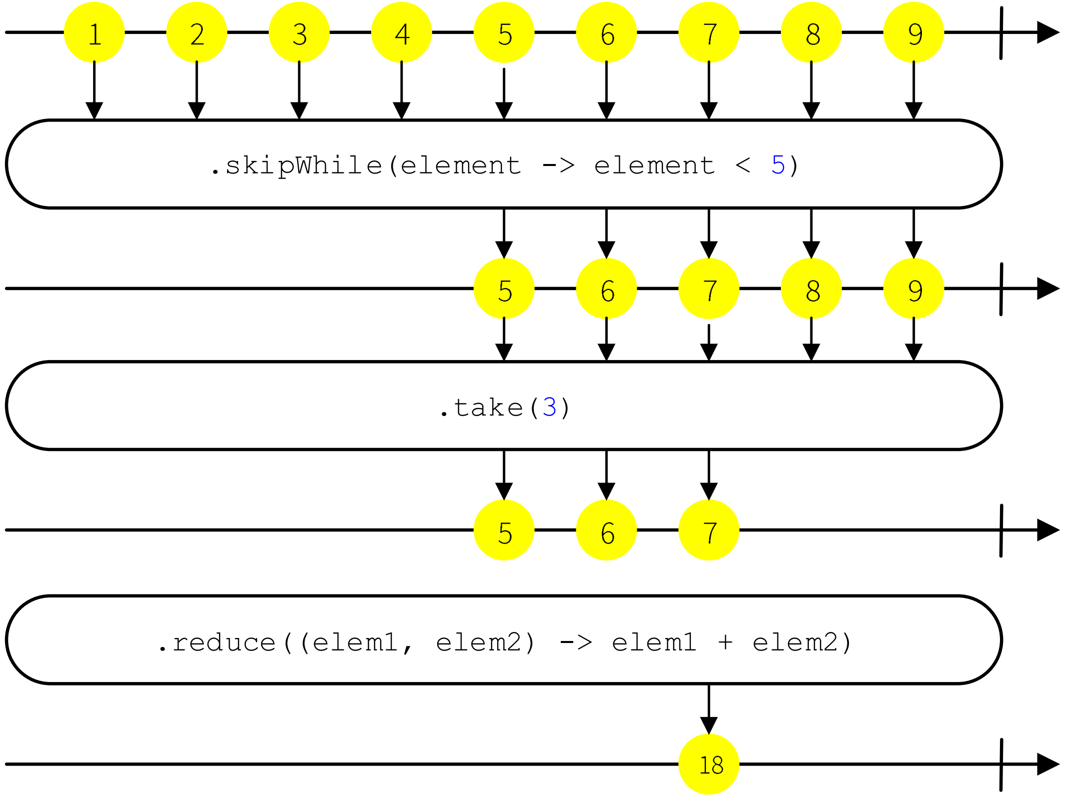
```
observableInstance.filter(element -> element < 10)
```

```
observableInstance.timeout(100, TimeUnit.MILLISECONDS)
```

```
observableInstance.map(number -> number * number)
```

```
Observable<String> mergedObservable = Observable
.merge(firstObservable, secondObservable, thirdObservable);
```

Manipulating Streams with RxJava



```
Observable.range(1,9)
```

```
.skipWhile(element -> element < 5)
```

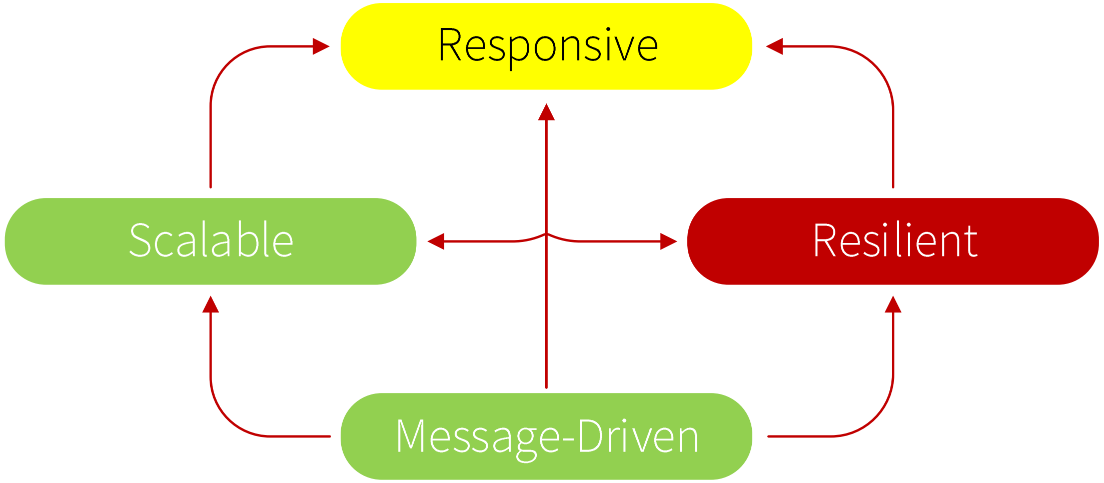
```
.take(3)
```

```
.reduce((elem1, elem2) -> elem1 + elem2)
```

```
.subscribe(result -> /*do something*/)
```

OOP + RxJava, State of the Union

Observable
Methods (FP style)
+
Transparent State



Observer + Observable

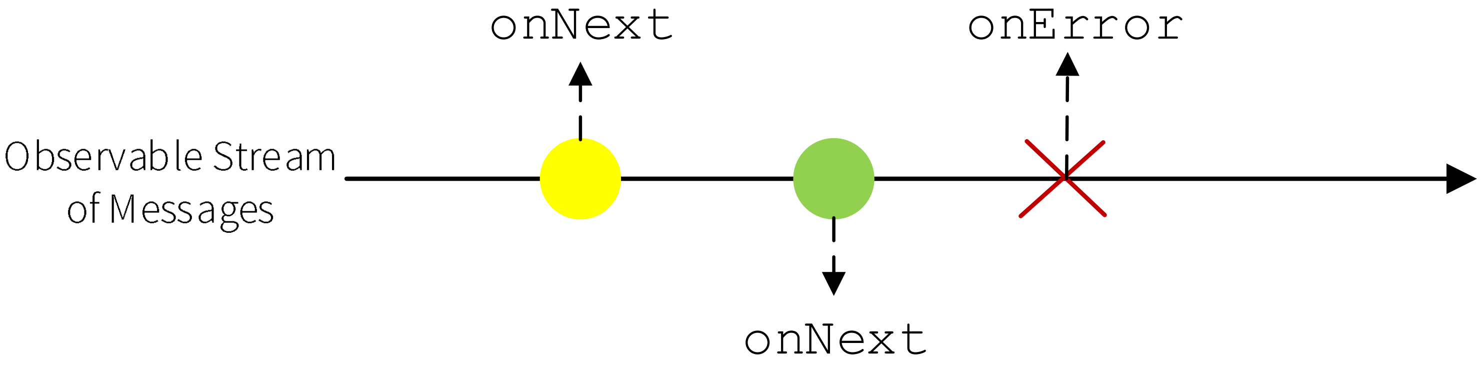
WHEN the System Fails

With classic OOP the Client has to

- `try/catch`
- Resource cleanup

With RxJava the Client has to

- `onError`
- Error is a First-class Citizen



Recovering from Errors

When the Error Occurs

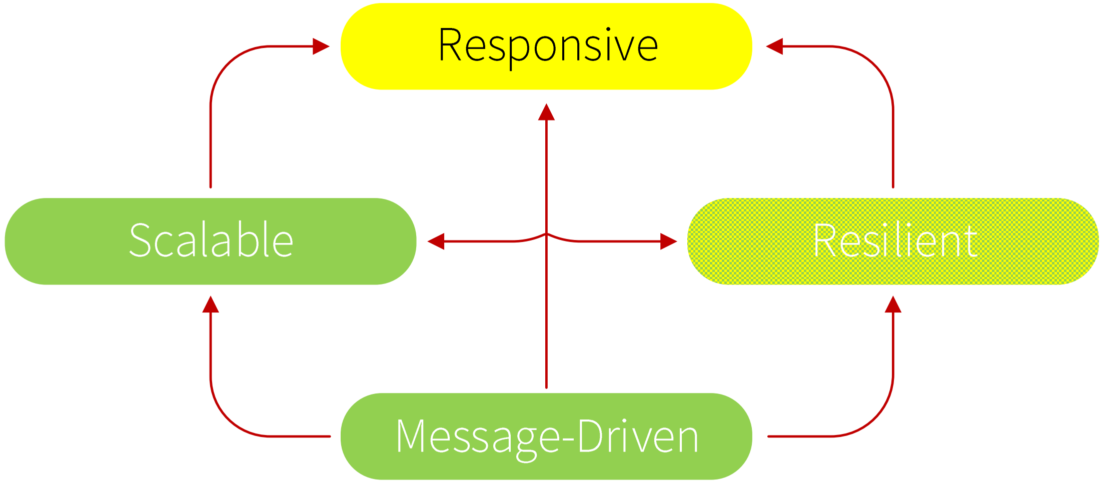
- Observable finishes
- Observer’s recovery options
  - onErrorReturn , onErrorResumeNext, retry

```
mainObservable.onErrorReturn(throwable -> {
    System.out.println("The original feed failed with" + throwable);
    return oneMoreMessage;
}).subscribe(data -> {/* doSomething */});
```

```
mainObservable.onErrorResumeNext(throwable -> {
    System.out.println("The original feed failed with" + throwable);
    return backupObservable;
}).subscribe(data -> {/* doSomething */});
```

OOP + RxJava, State of the Union

Observable
Methods (FP style)
+
Transparent State



Error is a First-Class Citizen

Observer + Observable

Let’s Get ResponsiVle

Responsive

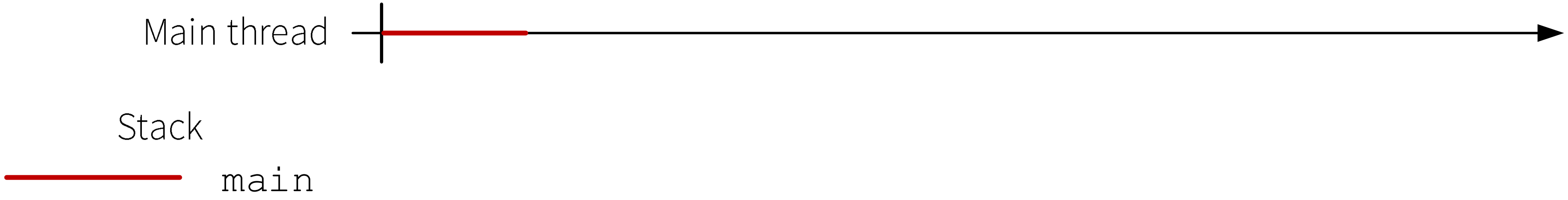
- To Our Client
- Already improved Scalability and Resilience
- Asynchronous execution

Responsible

- To Our System (to our Resources)

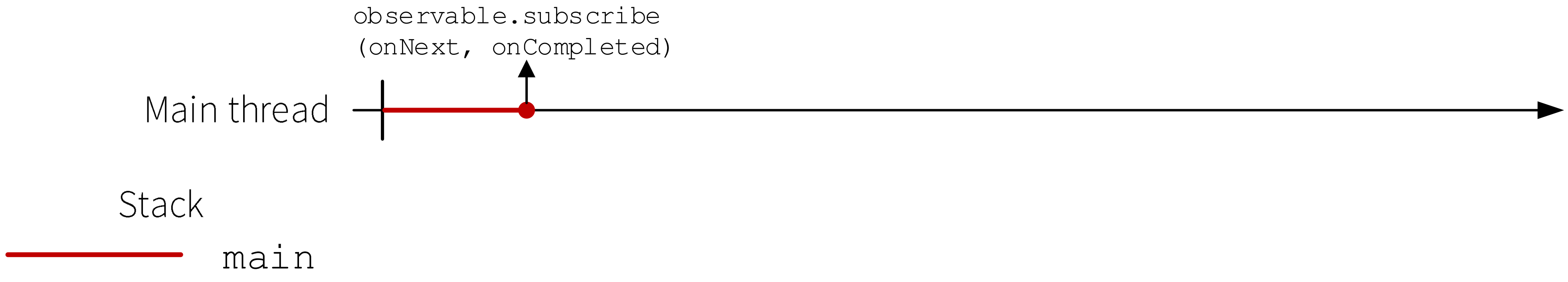
Asynchronous Streams

“Out of the Box”



Asynchronous Streams

“Out of the Box”



```
observable.subscribe
(onNext, onCompleted)
```

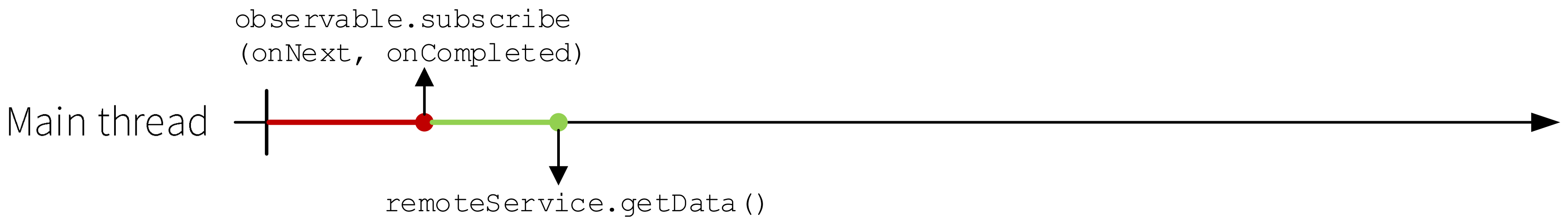
Main thread

Stack

```
main
```

Asynchronous Streams

“Out of the Box”



observable.subscribe
(onNext, onCompleted)

Main thread

remoteService.getData()

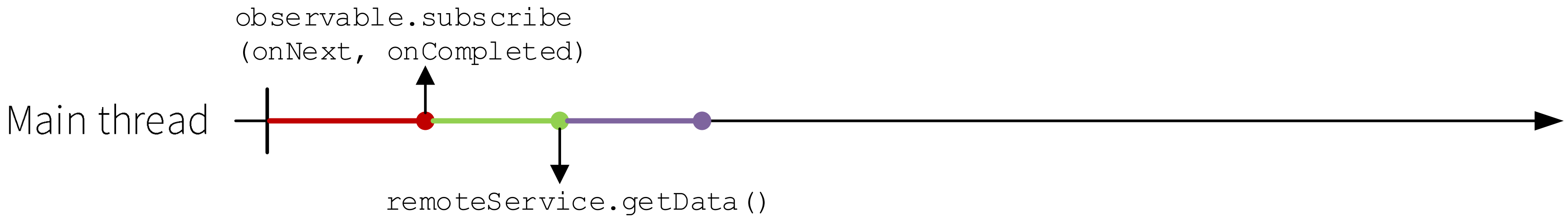
Stack

🟩 observable.subscribe

🟥 main

Asynchronous Streams

"Out of the Box"



Stack

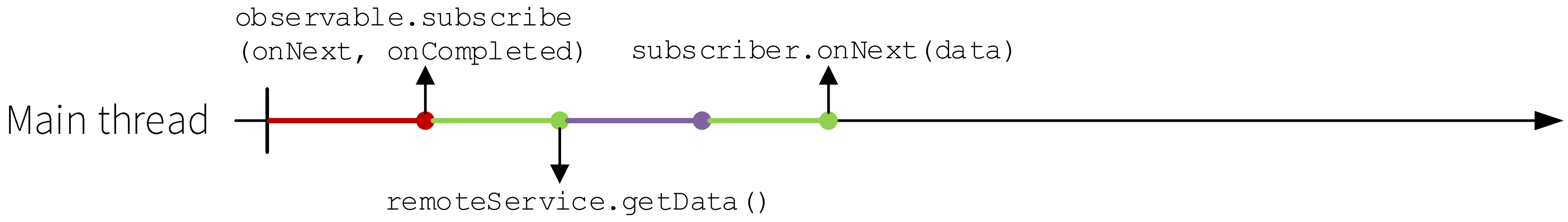
🟪 `remoteService.getData`

🟩 `observable.subscribe`

🟥 `main`

Asynchronous Streams

“Out of the Box”



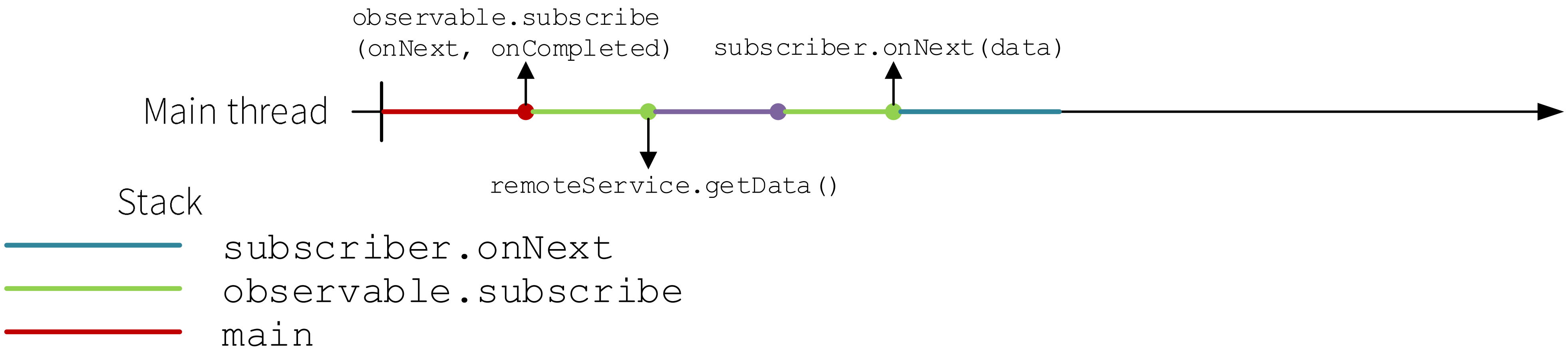
Stack

🟩 `observable.subscribe`

🟥 `main`

Asynchronous Streams

“Out of the Box”



Main thread

observable.subscribe
(onNext, onCompleted)

remoteService.getData()

subscriber.onNext(data)

Stack

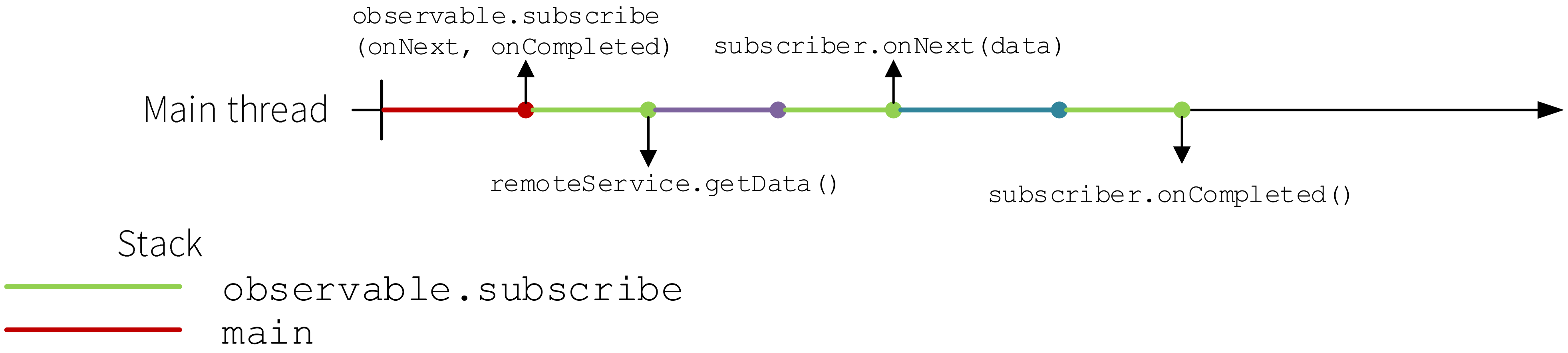
subscriber.onNext

observable.subscribe

main

Asynchronous Streams

“Out of the Box”



Main thread

observable.subscribe (onNext, onCompleted)

remoteService.getData()

subscriber.onNext(data)

subscriber.onCompleted()

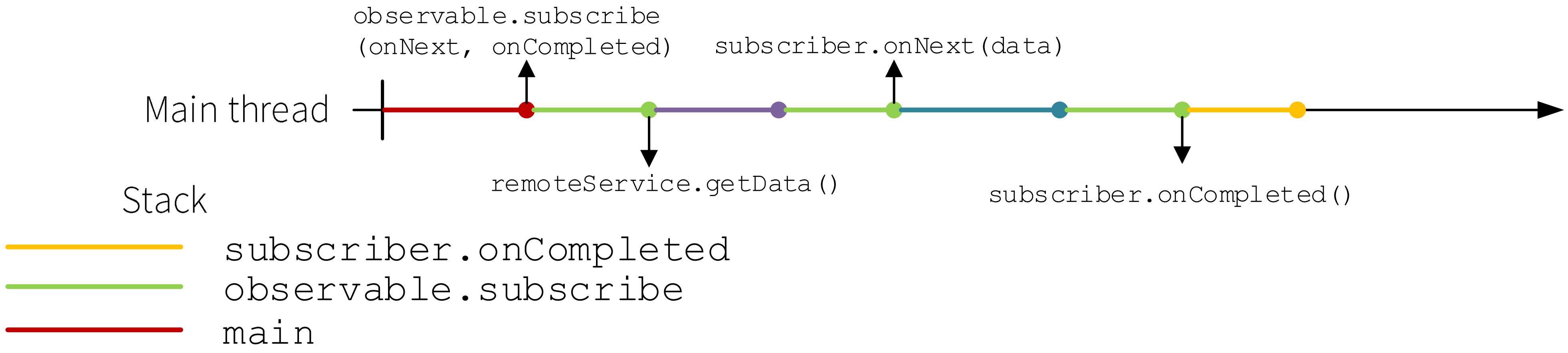
Stack

🟩 observable.subscribe

🟥 main

Asynchronous Streams

“Out of the Box”



Main thread

observable.subscribe
(onNext, onCompleted)

remoteService.getData()

subscriber.onNext(data)

subscriber.onCompleted()

Stack

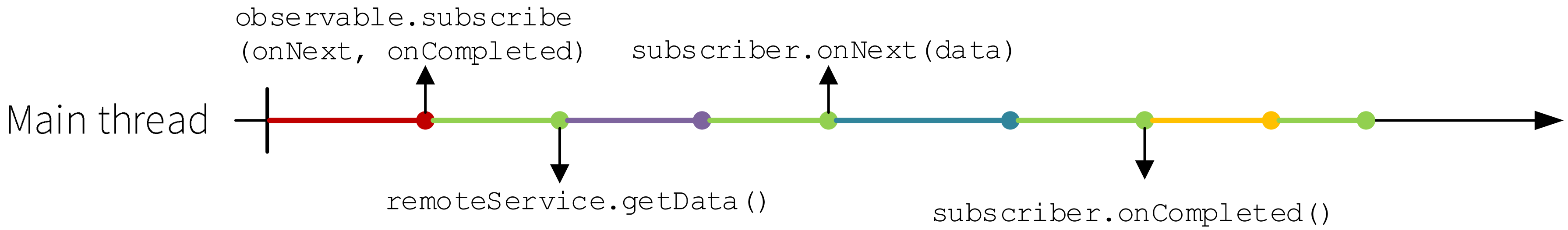
subscriber.onCompleted

observable.subscribe

main

Asynchronous Streams

“Out of the Box”



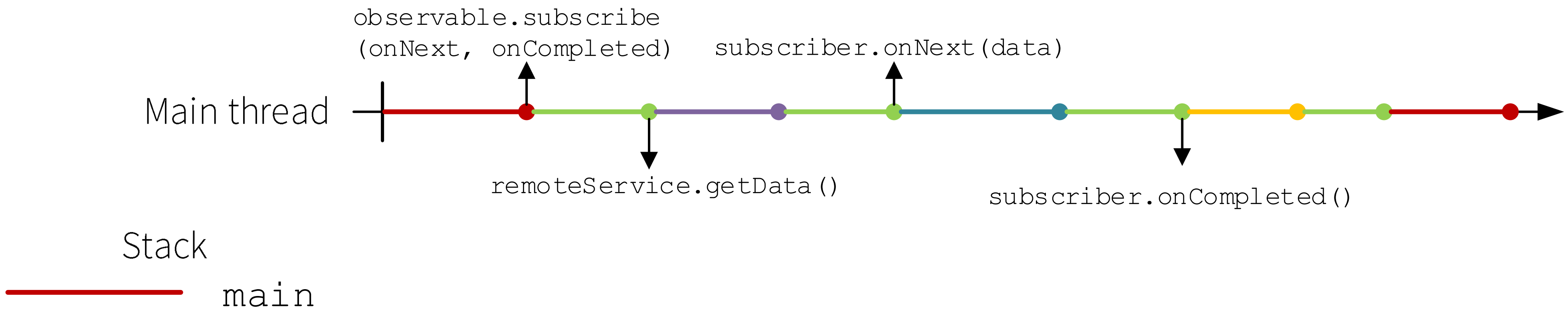
Stack

🟩 `observable.subscribe`

🟥 `main`

Asynchronous Streams

“Out of the Box”



observable.subscribe
(onNext, onCompleted)

subscriber.onNext(data)

Main thread

remoteService.getData()

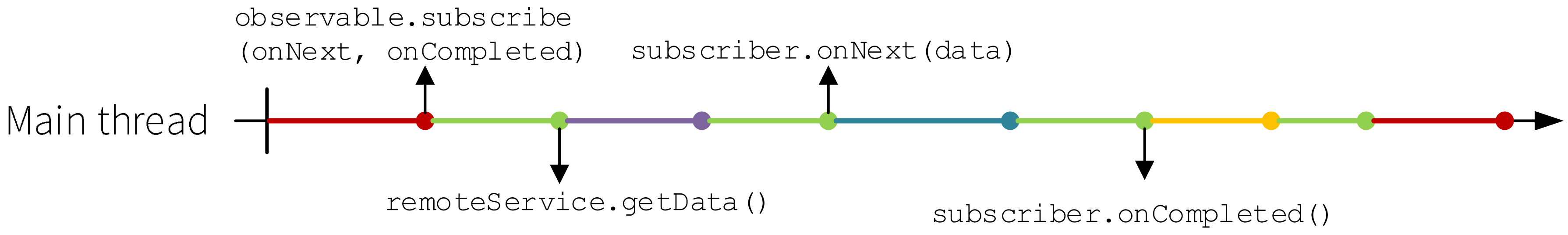
subscriber.onCompleted()

Stack

main

Asynchronous Streams

“Out of the Box”



Stack empty

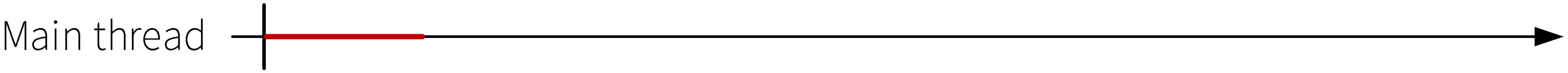
Asynchronous Streams

Let's get Asynchronous

- Thread handling with `Observable`
  - `subscribeOn(Scheduler)` - Thread Observable will run on
  - `observeOn(Scheduler)` - Thread Observer will run on
- Available `Schedulers`
  - `immediate` – use Caller Thread
  - `newThread` – do work on new Thread
  - `trampoline` – enqueue work on Caller Thread
  - `io` – Thread pool used for IO tasks
  - `computation` – Thread pool used for Computation tasks

Asynchronous Streams

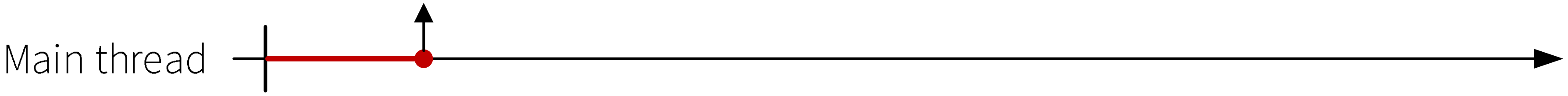
Asynchronous in practice



Asynchronous Streams

Asynchronous in practice

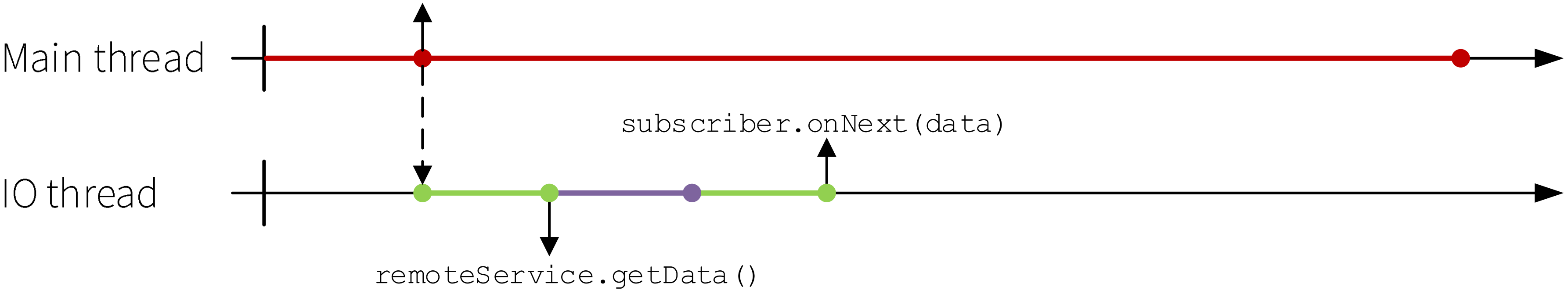
```
observable
.subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe(onNext, onCompleted)
```



Asynchronous Streams

Asynchronous in practice

```
observable
.subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe(onNext, onCompleted)
```



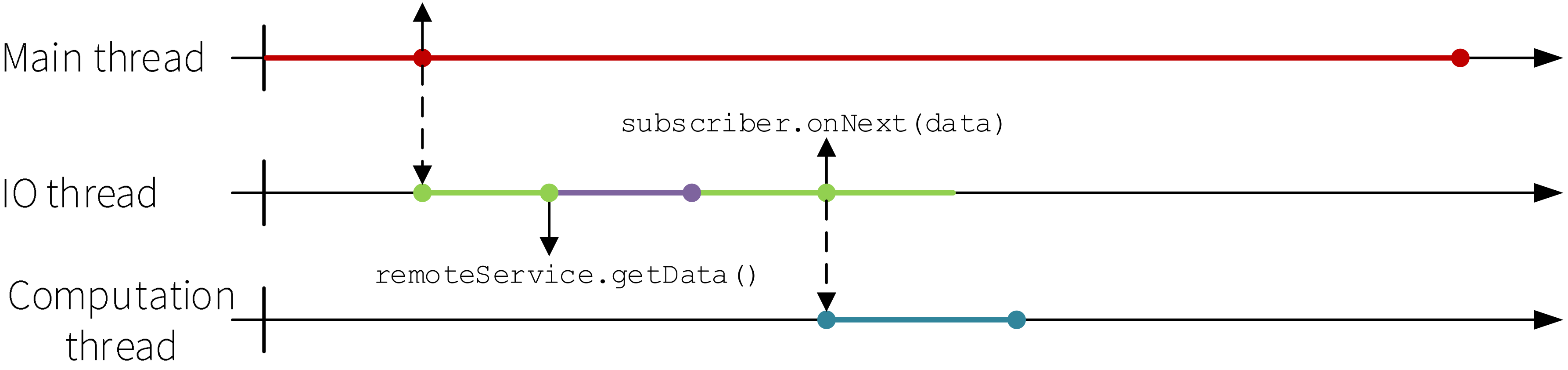
```
subscriber.onNext(data)
```

```
remoteService.getData()
```

Asynchronous Streams

Asynchronous in practice

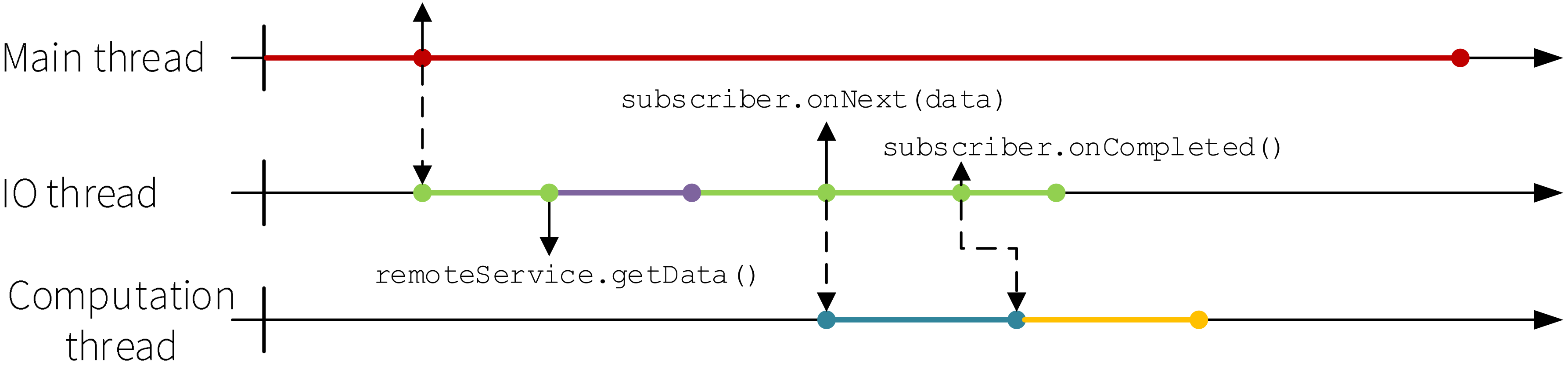
```
observable
.subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe(onNext, onCompleted)
```



Asynchronous Streams

Asynchronous in practice

```
observable
.subscribeOn(Schedulers.io())
.observeOn(Schedulers.computation())
.subscribe(onNext, onCompleted)
```



Main thread

subscriber.onNext(data)

subscriber.onCompleted()

IO thread

remoteService.getData()

Computation thread

Responsible Client

Being Reactive isn’t just about doing something fast, it’s about not doing it at all.

Or to be more precise, to do only what’s necessary.

Responsible Client

Being Responsible

- Observable works only when someone's listening
  - `subscribe` triggers Observable Stream
- Client (Consumer of Stream) tells us when he's done listening
  - `unsubscribe`

Responsible Client

Two flavors of Unsubscribing

- Client (Consumer) is unsubscribed from “outside”

```
Subscription subscription = observableInstance.subscribe(
        (Long message) -> {/*onNext*/},
        (Throwable error) -> {/*onError*/},
        () -> {/*onCompleted*/});
// Do some logic;
subscription.unsubscribe();
```

Responsible Client

Two flavors of Unsubscribing

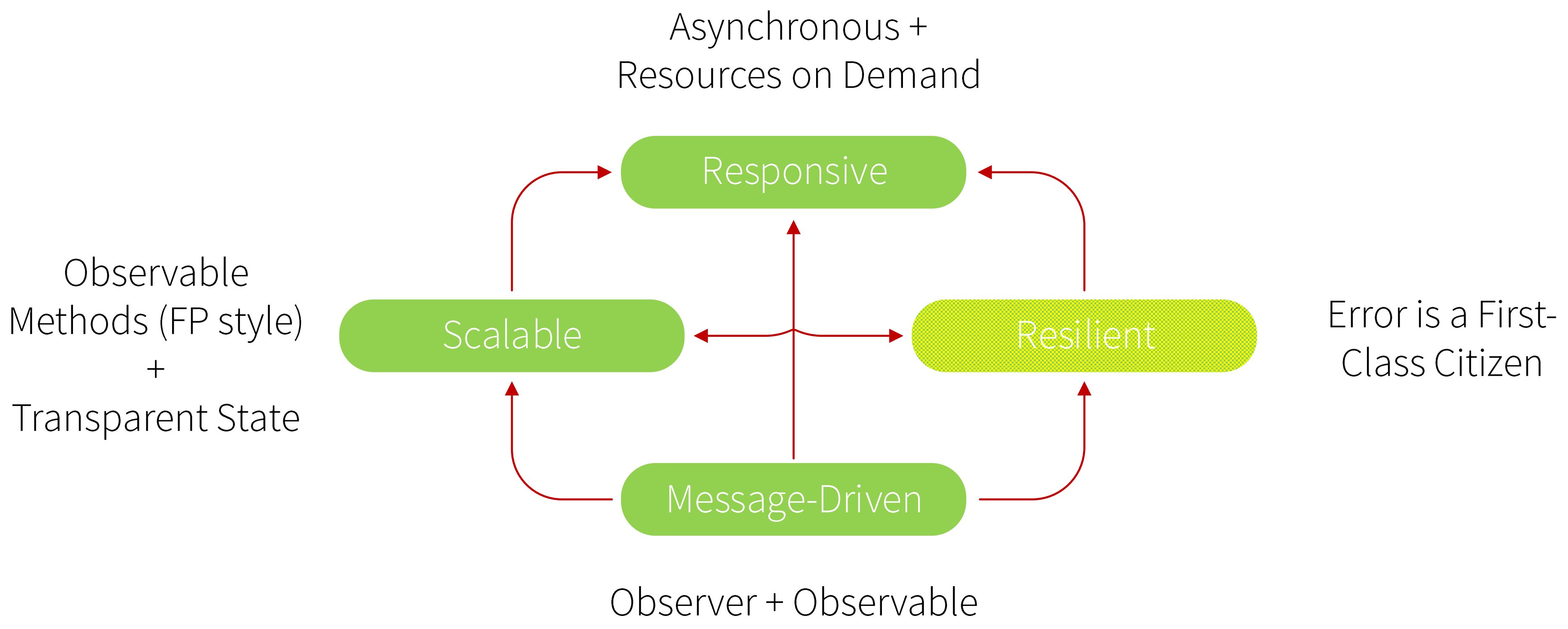
- Client (Consumer) is unsubscribed from “inside”

```
observableInstance.subscribe(new Subscriber<Long>() {
    @Override
    public void onNext(Long message) {
        // Do something on each Message received
        unsubscribe();
    }

    @Override
    public void onError(Throwable e) {
        // Do something on Error
    }

    @Override
    public void onCompleted() {
        // Do something when Observable completes
    }
});
```

OOP + RxJava, State of the Union


Asynchronous +
Resources on Demand

Observable
Methods (FP style)
+
Transparent State

Error is a First-Class Citizen

Observer + Observable

THANKS FOR YOUR TIME!

Q & hopefully A