REACTIVE PROGRAMMING

WITH NETBEANS AND JAVA 8

Stefan Reuter

STEFAN
REUTER

 @StefanReuter

- IT Architect
- Trainer and Consultant for trion development GmbH – www.trion.de

trion

www.trion.de

- Professional services
- Architecture
- Consulting
- Development
- Training

reactive programming
is programming with
asynchronous
data **streams**

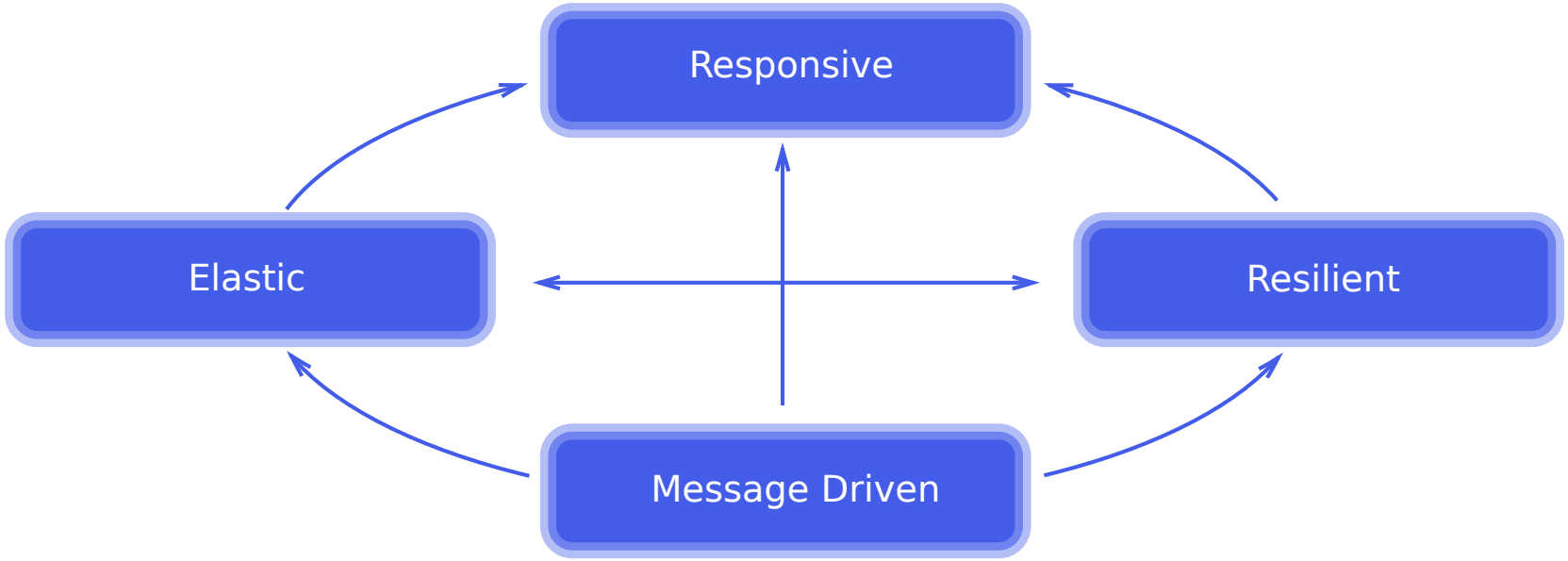
AGENDA

- Introduction
- Creating
- Subscribing
- Hot vs. Cold
- Integrating Existing Code
- Concurrency
- Composable Functions

AGENDA

- **Introduction**
- Creating
- Subscribing
- Hot vs. Cold
- Integrating Existing Code
- Concurrency
- Composable Functions

REACTIVE MANIFESTO



“[…] Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback. […]”

From: Reactive Manifesto, Version 2, September 2014

REACTIVE SCENARIOS

User Events

- Mouse movement and clicks
- Keyboard typing
- Changing GPS signals as a user moves with a device
- Touch events

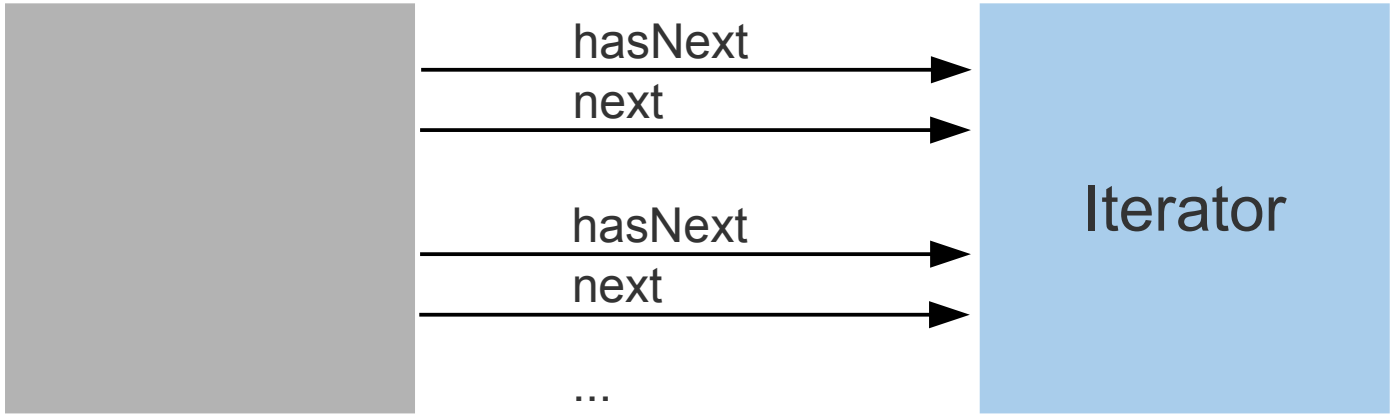
I/O

- Latency-bound I/O events from disk
- Data sent or received via network
- Distributed and cloud-based systems

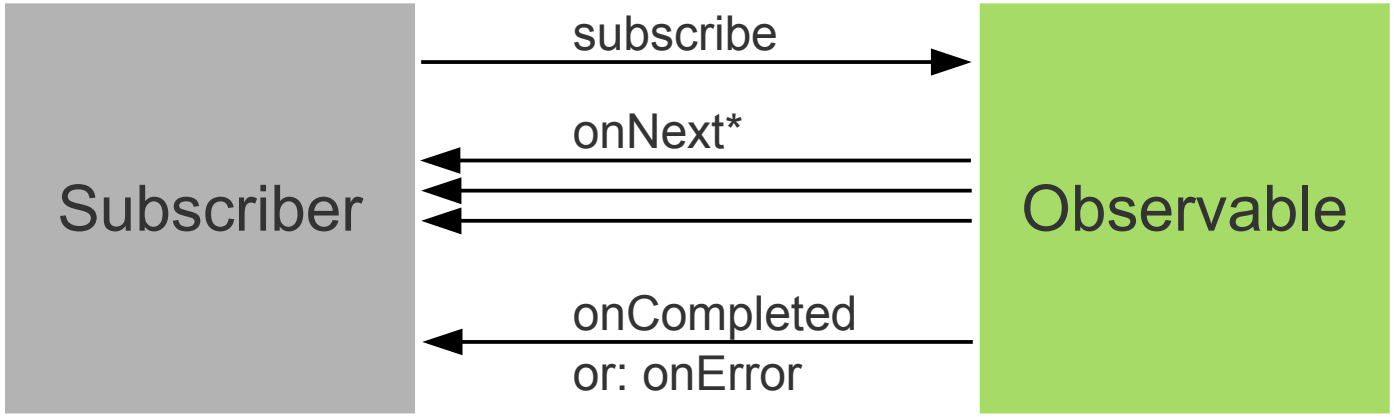
Push Events

- System events received from a server
- Signals received from hardware
- Events triggered by sensors
- Data received from IoT

JAVA.UTIL.ITERATOR



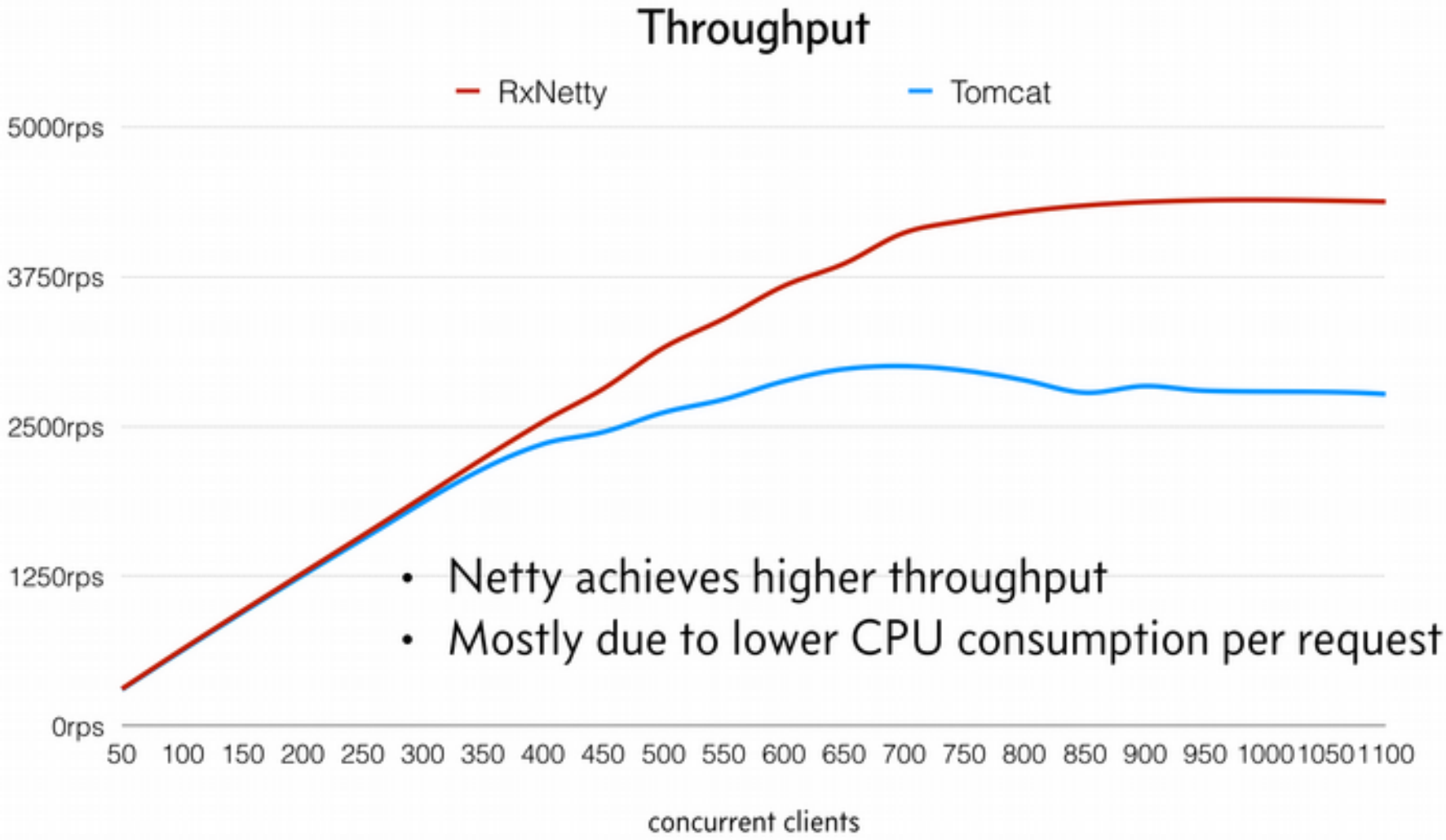
OBSERVABLE AND SUBSCRIBER



OnNext* (OnCompleted|OnError)?

ITERATOR vs. OBSERVABLE

Iterator	Observable
T next()	onNext(T t)
boolean hasNext()	onCompleted()
throws RuntimeException	onError(Throwable e)
▪ blocking ▪ synchronous ▪ pull	▪ non-blocking ▪ asynchronous ▪ push



Source: Reactive Programming with RxJava, O'Reilly 2016


CREATING

CREATING OBSERVABLES

Pre defined values

- Observable.just(T value)
- Observable.from(T[] values) / from(Iterable<T> values)
- Observable.range(int start, int count)
- Observable.interval()

Edge cases

- Observable.empty()
- Observable.never()
- Observable.error(Throwable exception)

Custom

- Observable.create(OnSubscribe<T> f)

```
Observable.create(subscriber -> {
  subscriber.onNext("Hello World");
  subscriber.onCompleted();
}
```


SUBSCRIBING

SUBSCRIBE (1/2)

- Observables do not emit items until someone subscribes
- Observables can have numerous subscribers
  (unlike a Java 8 stream that can only be consumed once)

```
Observable<Integer> o = Observable.range(1,3);

o.subscribe(System.out::println);
o.subscribe(System.out::println);
```

SUBSCRIBE (2/2)

```
o.subscribe(new Observer<Integer>() {
  public void onNext(Integer t) { }
  public void onError(Throwable e) { }
  public void onCompleted() { }
});

o.subscribe(System.out::println,
            Throwable::printStackTrace,
            this::completed);

o.subscribe(i -> logger.info("{}", i));
```

SHARING A SUBSCRIPTION

- ConnectableObservable created via publish()
- starts emitting items when connect() is called instead of when it is subscribed to

```
ConnectableObservable<> co = o.publish();

co.subscribe(…);
co.subscribe(…);
co.subscribe(…);

co.connect(); // or: co.refCount();
```

SUBSCRIPTION

- subscribing to an Observable returns a Subscription
- allows client code to cancel subscription and query status

```
Subscription subscription =
  tweets.subscribe(System.out::println);

//...

subscription.unsubscribe();

//subscription.isUnsubscribed() == true
```

SUBSCRIBER

- Subscriber is an abstract implementation of Observer and Subscription
- can unsubscribe itself

```
o.subscribe(new Subscriber<Tweet>() {
  @Override
  public void onNext(Tweet tweet) {
    if (tweet.getText().contains("Java")) {
      unsubscribe();
    }
  }
...
```

HOT vs. COLD

Cold

- Entirely lazy
- Never starts emiting before someone subscribes
- Every subscriber receives its own copy of the stream thus events are produced independently for each subscriber
- Examples: Observable.just(), from(), range()
- Subscribing often involves a side effect (e.g. db query)

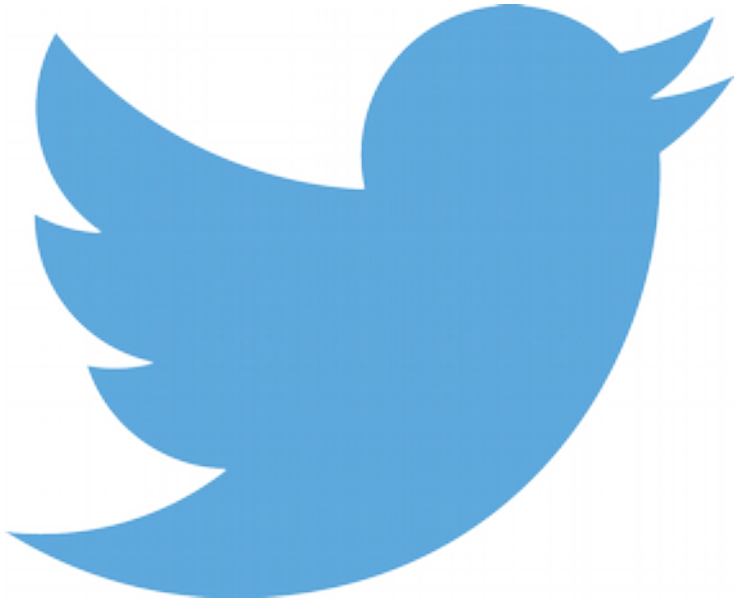
Hot

- Emits events whether subscribers are present or not
- Emits same events to all subscribers
- Examples: mouse movements, keyboard input, button clicks

Same API for hot and cold Observables


INTEGRATING EXISTING CODE

WRAPPING AN EXISTING ASYNC API



WRAPPING AN EXISTING SYNC API (1/3)

```
public Observable<Hotel> listHotels() {
  return Observable.from(
    query("SELECT * FROM HOTELS")
  );
}
```

- blocks until all data has been loaded
- is not lazy, i.e. loads data before there is any subscription

WRAPPING AN EXISTING SYNC API (2/3)

```
public Observable<Hotel> listHotels() {
  return Observable.defer(() ->
    Observable.from(
      query("SELECT * FROM HOTELS")
  ));
}
```

- Pass a lambda to Observable.defer()
- Database is no longer queried until someone subscribes

WRAPPING AN EXISTING SYNC API (3/3)

```
Observable<Booking> getBooking(String id) {
  return Observable.defer(() ->
    Observable.just(
       restTemplate.getForObject(
         "http://example.com/bookings/{id}",
         Booking.class, id)
  ));
}
```

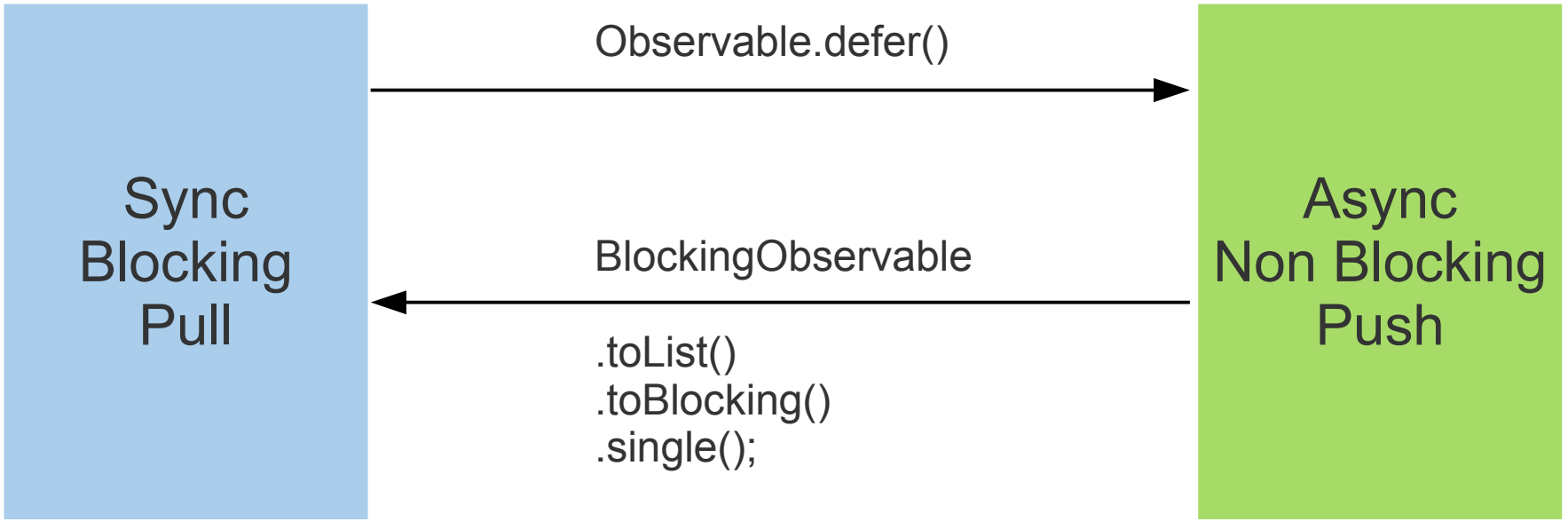
CONVERT TO BLOCKING

```
public Observable<Hotel> listHotels() {
  // ...
}

Observable<Hotel> o = listHotels();
Observable<List<Hotel>> hotelList = o.toList();
BlockingObservable<List<Hotel>> block = hotelList.toBlocking();
List<Hotel> people = block.single();

// short:
List<Hotel> o = listHotels()
  .toList()
  .toBlocking()
  .single();
```

SWITCH BETWEEN SYNC AND ASYNC




CONCURRENCY

USING SCHEDULERS

subscribeOn()

- can be called any time before subscribing
- the function passed to Observable.create() is executed in the thread provided by the given scheduler

```
Scheduler sched = Schedulers.newThread();
observable.subscribeOn(sched);
```

observeOn()

- controls which scheduler is used to invoke downstream subscribers occurring after observeOn()

create()

- scheduler can be passed as an additional argument to Observable.create()

SCHEDULERS (1/2)

newThread()

- starts a new thread each time subscribeOn() or observeOn() is called
- increased latency due to thread start
- usually not a good choice as threads are not reused

io()

- threads are reused
- unbounded pool of threads
- useful for I/O bound tasks which require very little CPU resources

computation()

- limits number of threads running in parallel
- default pool size == Runtime.availableProcessors()
- useful for tasks thar are entirely CPU-bound, i.e. they require computational power and have no blocking code

SCHEDULERS (2/2)

from(Executor exc)

- wrapper around java.util.concurrent.Executor

```
ExecutorService executor =
  Executors.newFixedThreadPool(10);
Scheduler s = Schedulers.from(executor);
```

immediate()

- invokes tasks in the client thread
- blocking
- usually not needed because this is the default behavior

test()

- only used for testing
- allows arbitrary changes to the clock for simulation

NOTES ON CONCURRENCY

- Callbacks will be invoked from only one thread at a time, but events can be emitted from many threads
- Concurrency often comes already from the source of events so explicit use of schedulers should be a last resort


COMPOSABLE FUNCTIONS

COMPOSABLE FUNCTIONS (1/2)

Transform	▪ map, flatMap ▪ groupBy, buffer ▪ window ▪ ...
Filter	▪ take, skip, last ▪ distinct ▪ filter ▪ ...
Combine	▪ concat, merge, zip, combineLatest ▪ multicast, publish ▪ switchOnNext ▪ ...

COMPOSABLE FUNCTIONS (2/2)

Concurrency

- observeOn
- subscribeOn

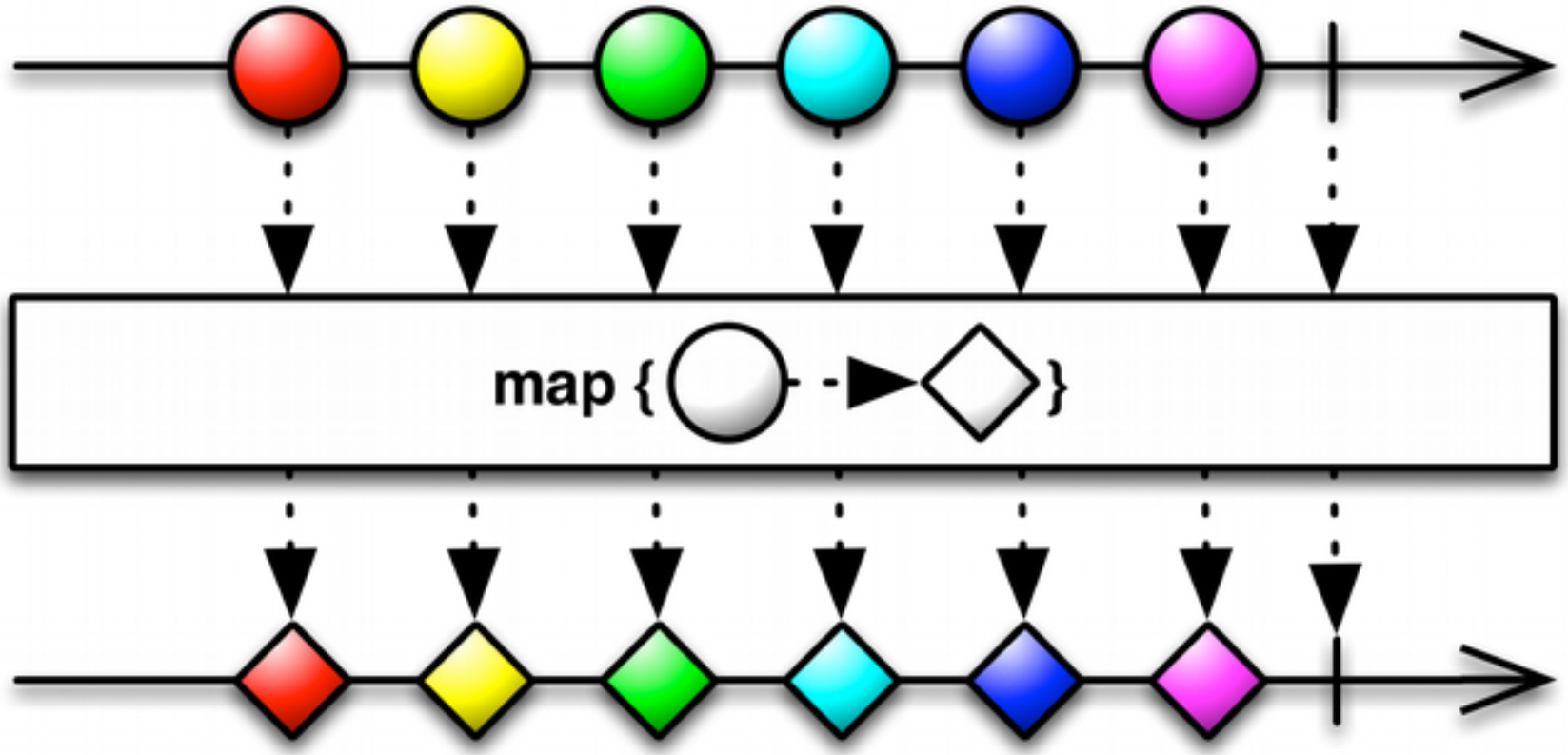
Error Handling

- onErrorReturn
- onErrorResumeNext
- retry
- ...

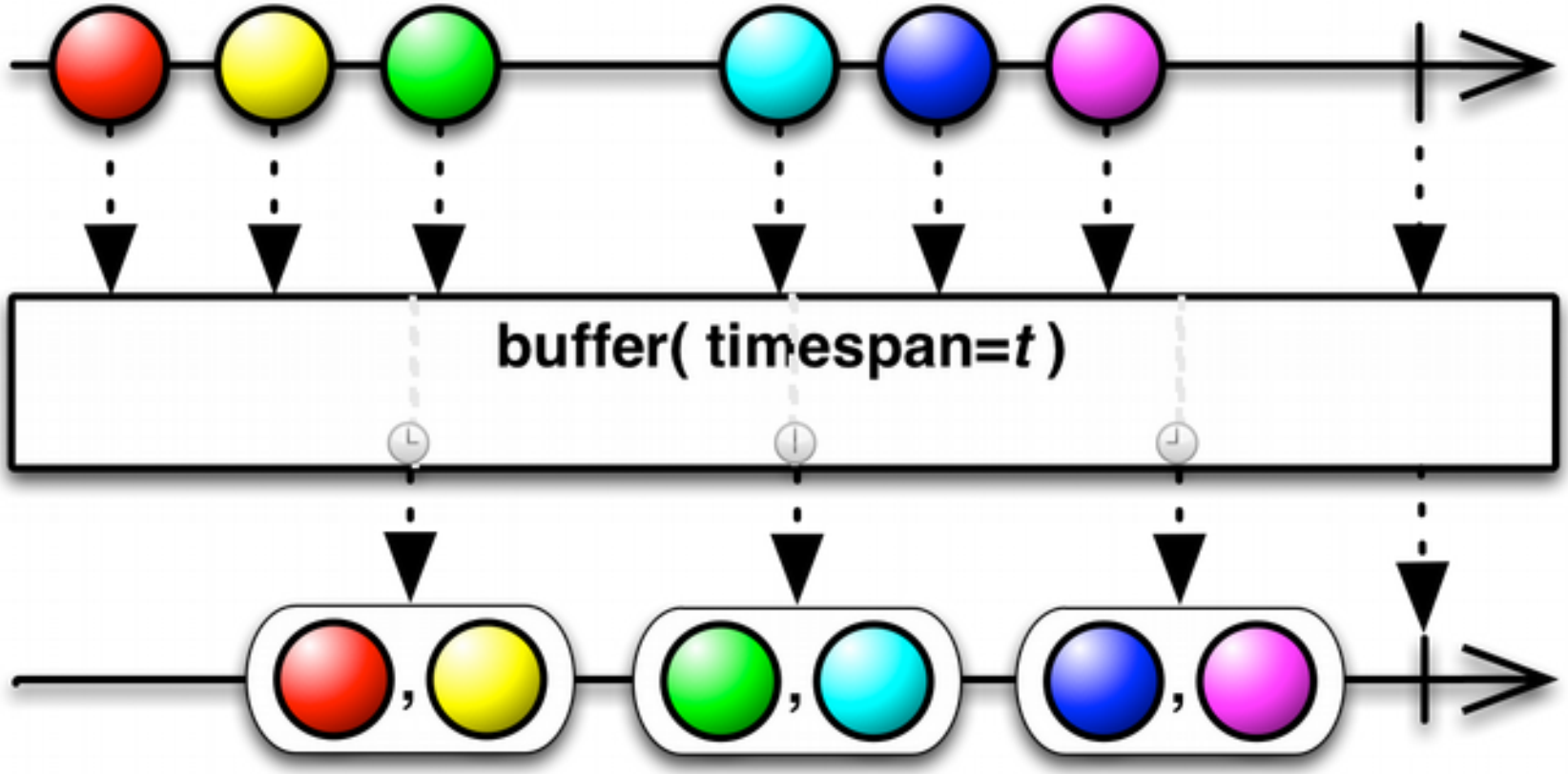
Custom

- any public class that implements the Operator interface
- or a subclass like Transformer
- most likely you will never need this!

MAP



BUFFER



TWEETS PER SECOND

```
Observable<Integer> o = tweets()
   .buffer(1, TimeUnit.SECONDS)
   .map(items -> items.size())
```

```
Observable<Integer> itemsPerSecond
                                  (Observable<?> o) {
   return o.buffer(1, TimeUnit.SECONDS)
           .map(items -> items.size());
}
```

IMPLEMENTATIONS ON THE JVM

RxJava

- Reactive Extensions (ReactiveX.io) for the JVM
- Zero Dependencies
- Polyglot (Scala, Groovy, Clojure and Kotlin)
- RxJava 2: Java 8+ and Reactive Streams compatible

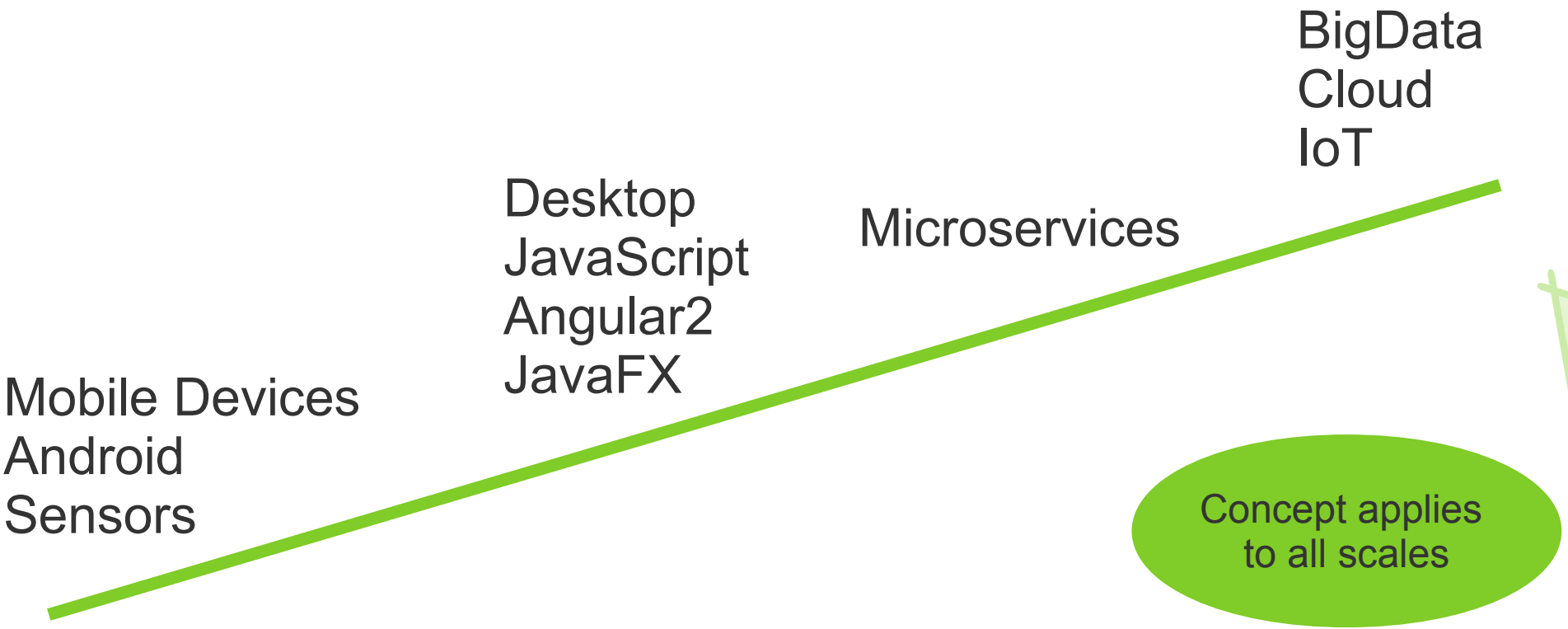
Project Reactor 2.5

- Spring Ecosystem
- Reactive Streams compatible
- Will become part of Spring Framework 5.0

Java 9 j.u.c.Flow

- Flow.Processor, Publisher, Subscriber and Subscription
- Interfaces correspond to Reactive Streams specification

SMALL TO LARGE SCALE



SUMMARY

reactive programming is a powerful option that is available today

- You can start using reactive programming right away as you can always convert back using BlockingObservable
- Have a look at http://reactivex.io/
- Check supporting libraries like RxNetty and RxJs

Questions?

Thank you
for your attention

I am looking forward to answer your questions at

stefan.reuter@trion.de
@StefanReuter

BACKUP

RxNetty HTTP CLIENT

```
SocketAddress serverAddress = ...
Charset charset = Charset.defaultCharset();

HttpClient.newClient(serverAddress)
  .enableWireLogging(LogLevel.DEBUG)
  .createGet("/api/talks")
  .doOnNext(resp -> logger.info(resp.toString()))
  .flatMap(
    resp -> resp.getContent()
                .map(bb -> bb.toString(charset))
  )
  .toBlocking()
  .forEach(logger::info);
```