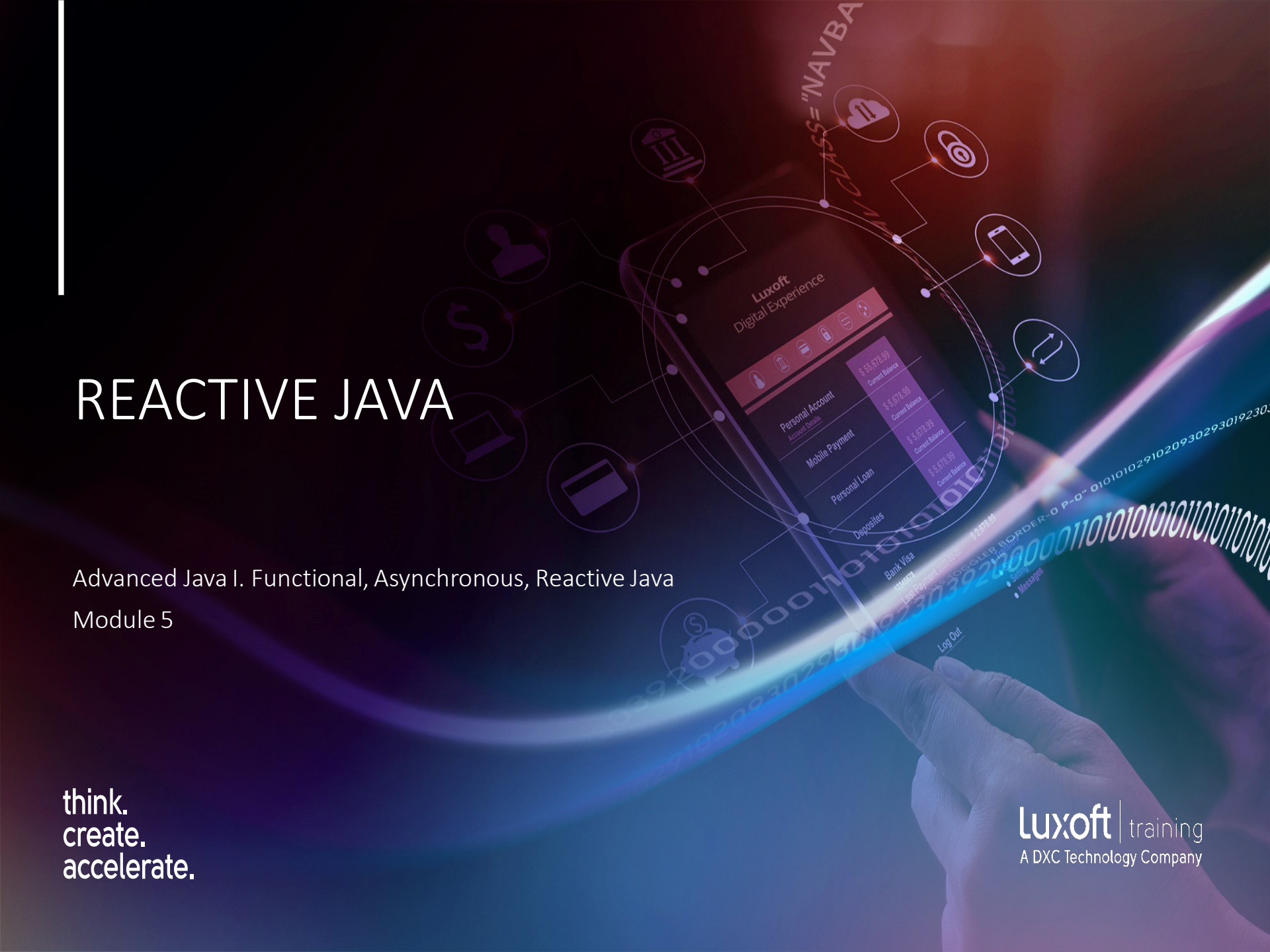
REACTIVE JAVA

Advanced Java I. Functional, Asynchronous, Reactive Java

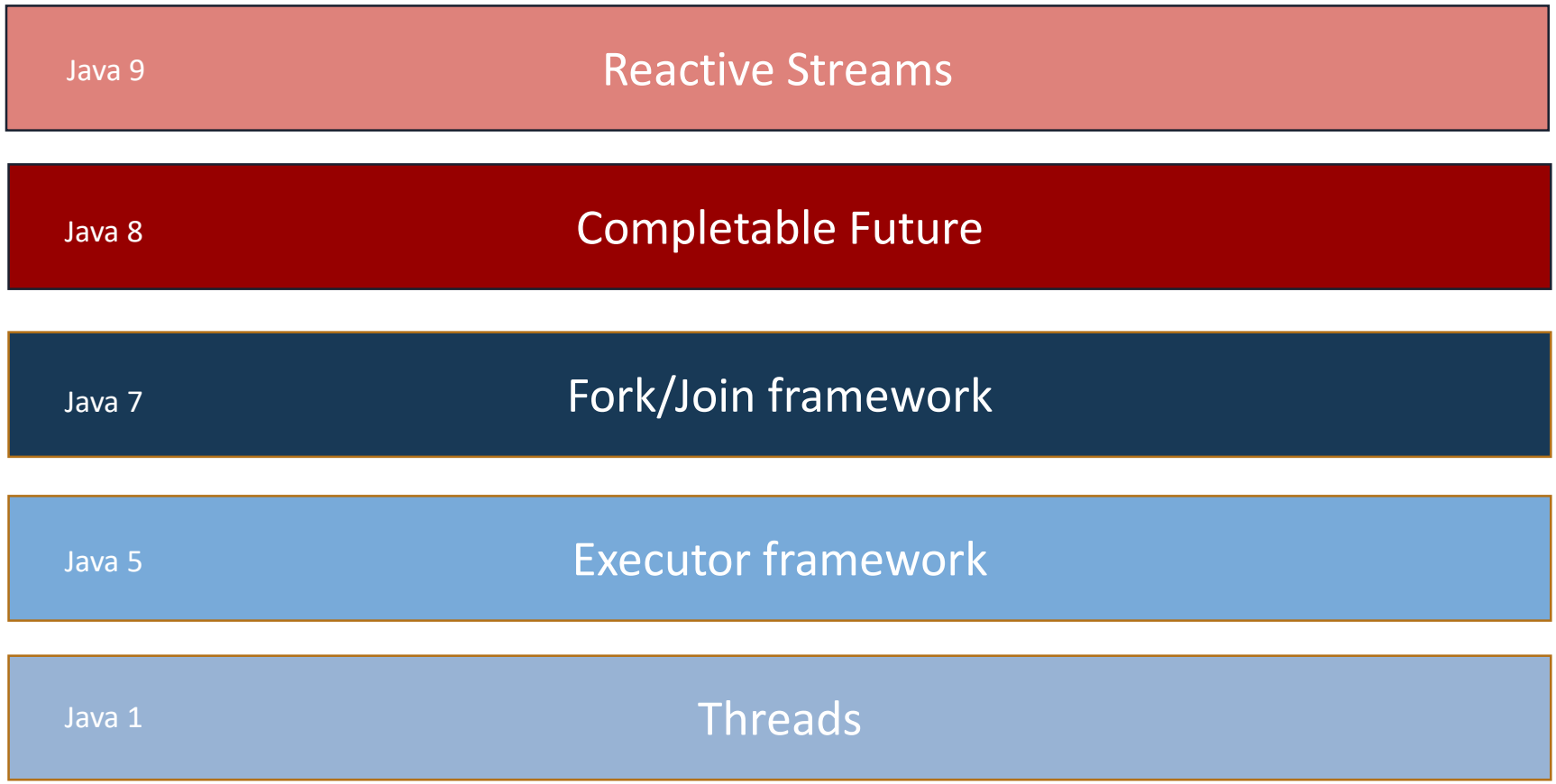
Module 5

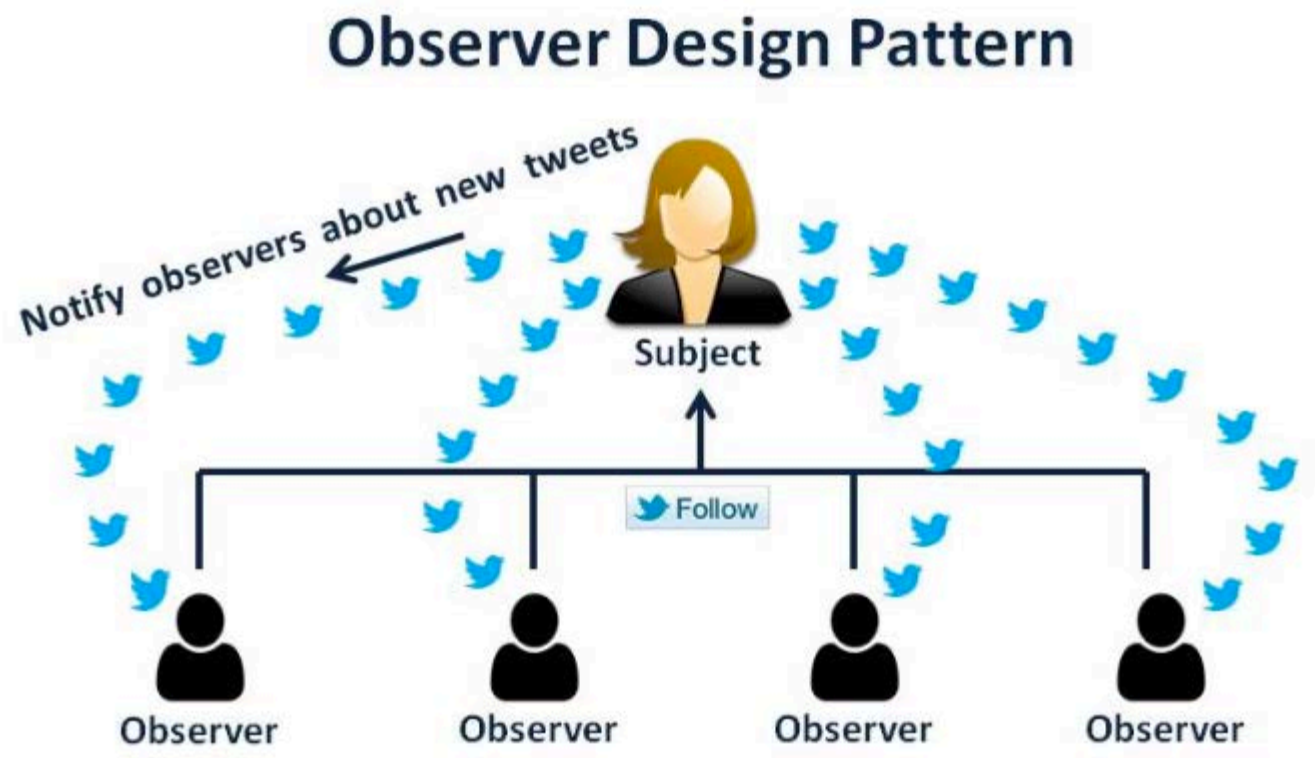


think.
create.
accelerate.

Luxoft | training
A DXC Technology Company

JAVA 9 REACTIVE STREAMS

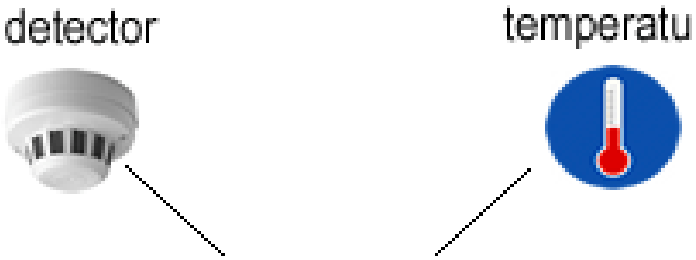




Publisher
Observable

Subscriber

WHAT IS REACTIVITY?



Observable variables

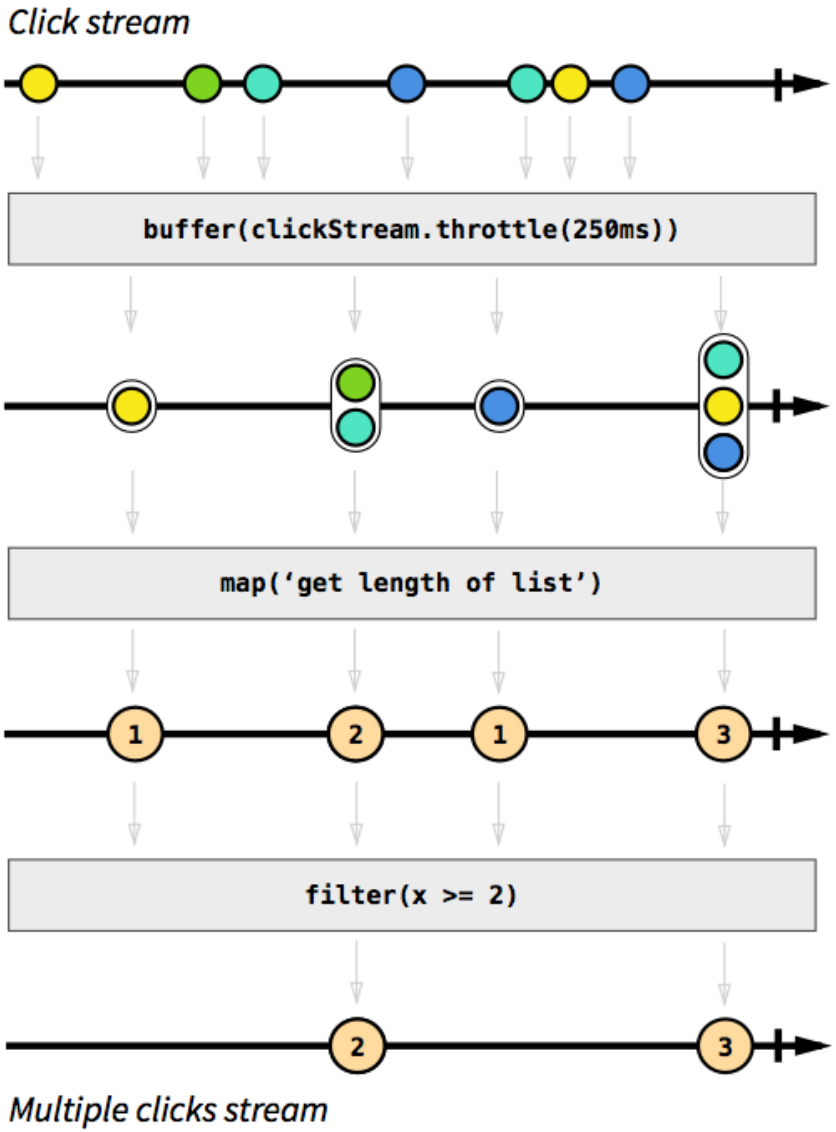
alarm.active = detector > X **&&** temperature **>**Y

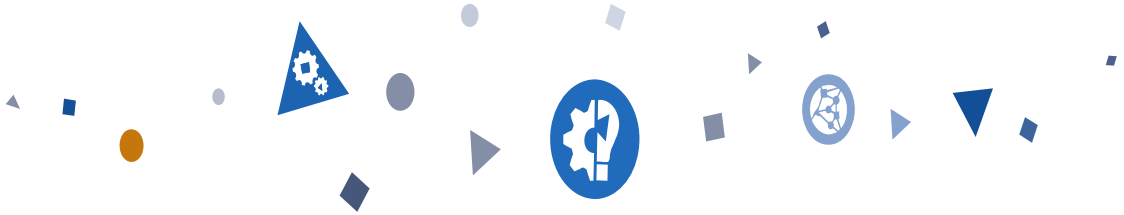
alarm.active

Reactive variable

REACTIVE APPROACH

TASK:
Get stream of "double click" events. Consider triple clicks as double clicks.

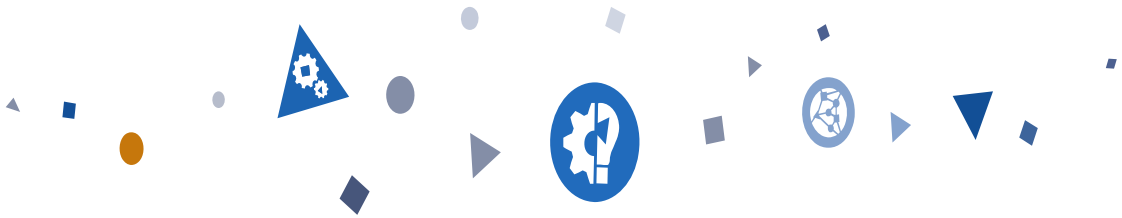




RX JAVA

MAVEN DEPENDENCY

```
<dependency>
  <groupId>io.reactivex.rxjava2</groupId>
  <artifactId>rxjava</artifactId>
  <version>2.2.16</version>
</dependency>
```



INTRODUCING OBSERVABLES

OBSERVABLE

- An `Observable` pushes objects.
- `Observable<T>` pushes objects of type T through a series of operators until it arrives at an `Observer` that consumes the items.

```
Observable<String> locations =
      Observable.just(“Bucharest",
            “Krakow", “Moscow",
            “Kiev", “Sofia"); //declaration

locations.subscribe(s -> System.out.println(s));
//effectively triggers the push of objects (emissions)
```

Result:
Bucharest
Krakow
Moscow
Kiev
Sofia

Example 1

WHAT IS AN OBSERVABLE?

	Single item	Many items
Blocking	T	Collection<T> Stream<T>
Non-blocking	CompletableFuture<T>	Observable<T>

- `Observable` is a push-based, composable non-blocking data stream.
- For a given `Observable<T>`, it pushes items (called emissions) of type `T` through a series of operators until it finally arrives at a final `Observer`, which consumes the items.

USING OPERATORS BETWEEN OBSERVABLE AND OBSERVER

```
Observable<String> locations =
      Observable.just(“Bucharest",
            “Krakow", “Moscow",
            “Kiev", “Sofia");

locations.map(s -> s.length())
         .subscribe(l ->
               System.out.println(l));
```

Result:

```
9
6
6
4
5
```

Example 2

OBSERVABLE PUSHING VALUES AT FIXED INTERVAL

```
Observable<Long> steps =
        Observable.interval(1,
                    TimeUnit.SECONDS);
steps.subscribe(l ->
      System.out.println(l));
```

Result:

0
1
2
3
4
5
6
7
8
9

Example 3

HOW DOES AN OBSERVABLE WORK?

An `Observable` works by passing three types of events:

- onNext()
- onComplete()
- onError()

USING OBSERVABLE.CREATE()

```
Observable<String> locations = Observable.create(location -> {
    location.onNext("Bucharest");
    location.onNext("Krakow");
    location.onNext("Moscow");
    location.onNext("Kiev");
    location.onNext("Sofia");
    location.onComplete();
});


locations.subscribe(
       s -> System.out.println("Location: " + s));
```

Results:

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia

Example 4

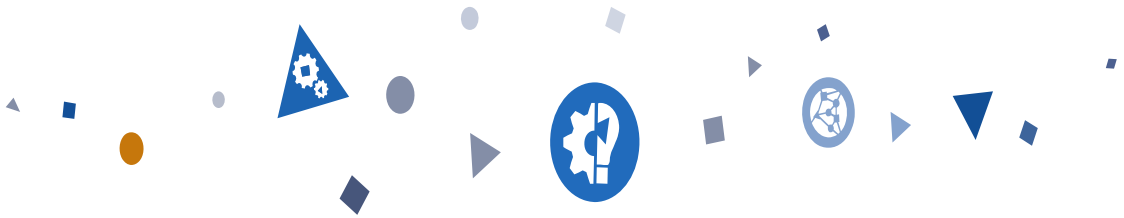
DERIVING NEW OBSERVABLES

```
Observable<String> locations =
      Observable.just("Bucharest",
           "Krakow", "Moscow",
           "Kiev", "Sofia");

locations.map(String::length)
         .filter(l -> l >= 5)
         .subscribe(l ->
             System.out.println(l));
```

Result

9
6
6
5

Example 5



INTRODUCING OBSERVERS

THE OBSERVER INTERFACE

An `Observer` works by passing three types of events:

- onSubscribe()
- onNext()
- onComplete()
- onError()

IMPLEMENTING AND SUBSCRIBING TO AN OBSERVER

```
Observable<String> locations =
   Observable.just("Bucharest", "Krakow", "Moscow", "Kiev", "Sofia");
Observer<Integer> observer = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
    }
    @Override
    public void onNext(Integer value) {
        System.out.println("Length: " + value);
    }
    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("Done.");
    }
};
locations.map(String::length).filter(l -> l >= 5)
        .subscribe(observer);
```


Example 6

IMPLEMENTING AND SUBSCRIBING TO AN OBSERVER

Results:
Length: 9
Length: 6
Length: 6
Length: 5
Done.

- The `Observer` receives emissions at the end of an `Observable` chain and serves as the endpoint where the emissions are consumed.
- The `onNext()` method receives each integer length emission and prints it.
- If an error occurs anywhere in our `Observable` chain, it will be pushed to the `onError()` implementation on `Observer`.
- When the source has no more emissions (after pushing `"Sofia"`), it will call `onComplete()` and print `Done.` to the console.

THE OVERLOADED SUBSCRIBE METHOD

The `subscribe()` method is overloaded to accept lambda arguments

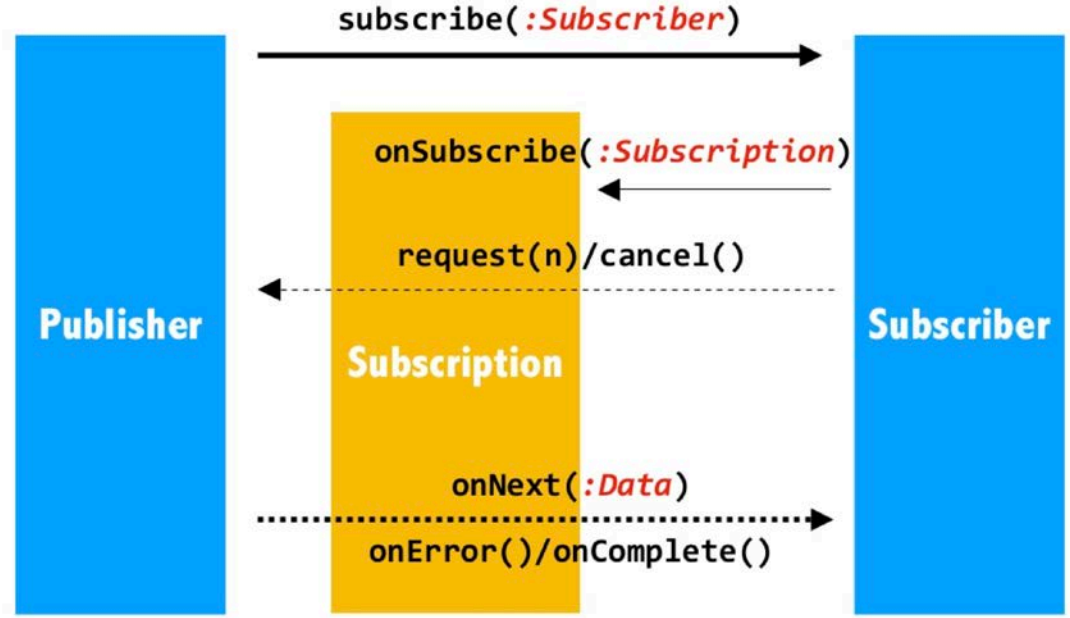
```
public final Disposable subscribe(Consumer<? super T> onNext)

public final Disposable subscribe(Consumer<? super T> onNext,
Consumer<? super Throwable> onError)

public final Disposable subscribe(Consumer<? super T> onNext,
Consumer<? super Throwable> onError, Action onComplete)

public final Disposable subscribe(Consumer<? super T> onNext,
Consumer<? super Throwable> onError, Action onComplete,
Consumer<? super Disposable> onSubscribe)

public final void subscribe(Observer<? super T> observer)
```

THE OVERLOADED SUBSCRIBE METHOD

```
Observable<String> locations =
        Observable.just("Bucharest", "Krakow", "Moscow", "Kiev",
                "Sofia");

locations.map(String::length).filter(l -> l >= 5)
        .subscribe(l -> System.out.println("Length: " + l),
                Throwable::printStackTrace,
                () -> System.out.println("Done."));


locations.map(String::length).filter(l -> l >= 5)
        .subscribe(l -> System.out.println("Length: " + l),
                Throwable::printStackTrace);
```

Examples 7, 8

REACTIVE STREAMS SPEC (JAVA 9)



Implementations:

RxJava

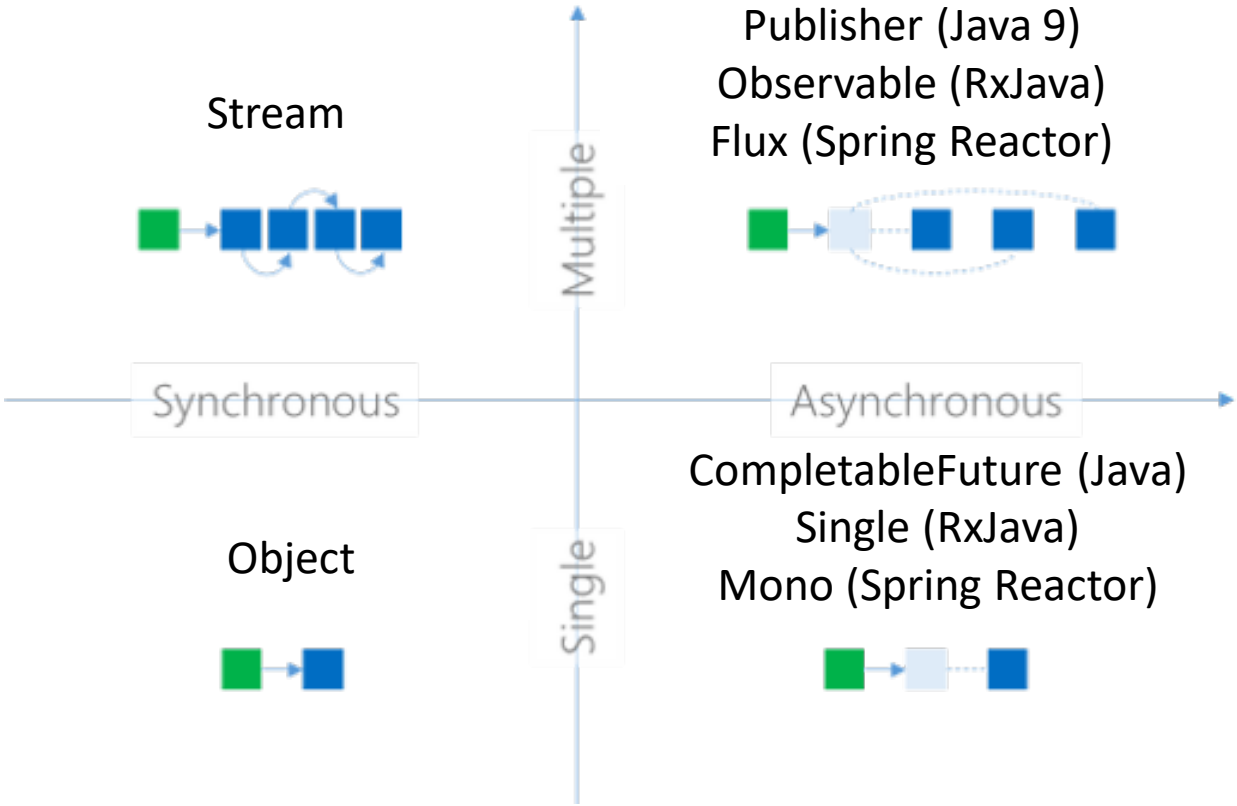
Reactor Core

Akka Streams

PUSH / PULL
Backpressure

```
public interface Publisher<T> {
   public void subscribe(
        Subscriber<? super T> s);
}

public interface Subscriber<T> {
   public void onSubscribe(Subscription s);
   public void onNext(T t);
   public void onError(Throwable t);
   public void onComplete();
}

public interface Subscription {
   public void request(long n);
   public void cancel();
}
public interface Processor<T, R>
  extends Subscriber<T>, Publisher<R> {
}
```

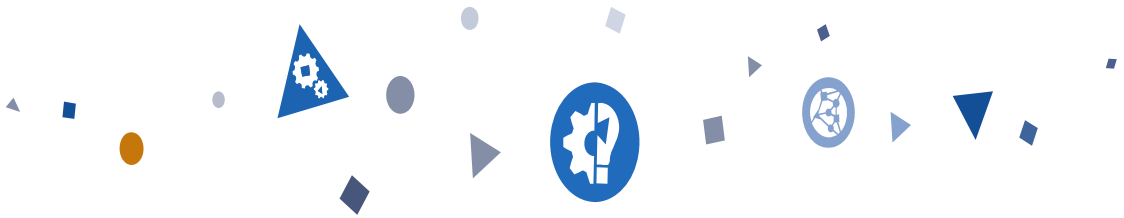
FUNCTION MAY RETURN…



Object < Stream < Observable
Object < CompletableFuture < Observable

SIDE EFFECT METHODS

Method	Functional Interface	Event
doOnSubscribe()	Consumer<T> onSubscribe	A subscriber subscribes to the Observable
doOnDispose()	Action	A subscriber unsubscribes from the subscription
doOnNext()	Consumer<T> onNext	The next item is emitted
doOnCompleted()	Action onComplete	The Observable will emit no more items
doOnError()	Consumer<Throwable> onError	An error occurred
doOnTerminate()	Action	Either an error occurred or the Observable will emit no more items - will be called before the termination methods
doFinally()	Action	Either an error occurred or the Observable will emit no more items - will be called after the termination methods
doOnEach()	Consumer<T> onNext, Consumer<Throwable> onError, Action onComplete, Action onAfterTerminate	Either an item was emitted, the Observable completes or an error occurred. The Notification object contains information about the type of event
doOnLifecycle()	Consumer<Disposable> onSubscribe, Action onDispose	Run at the moment of subscribing or disposing



ERROR HANDLING

ERROR HANDLING

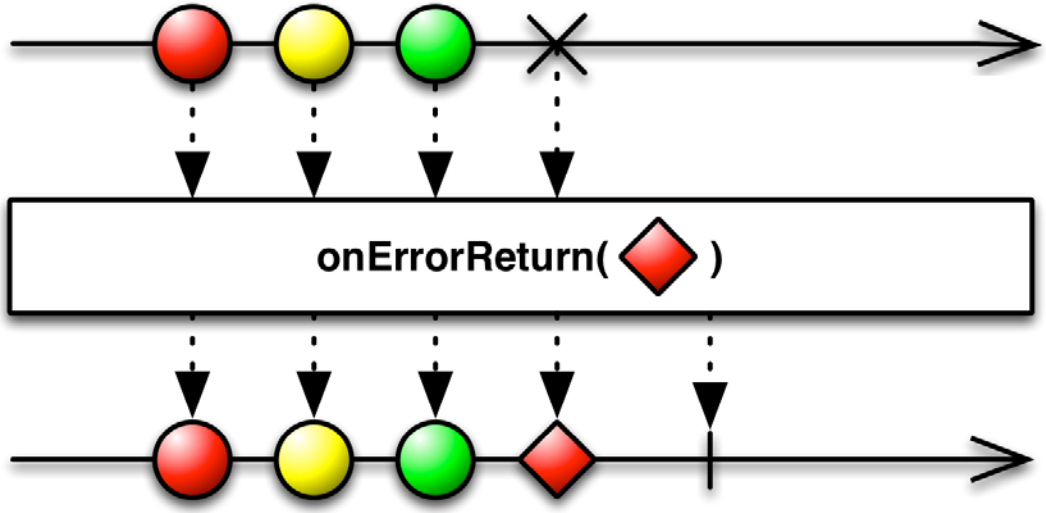
```
Observable.error(new IOException("Something went wrong"))
    .doOnError(error -> System.err.println("The error message is: " +
                                           error.getMessage()))
    .subscribe(
        x -> System.out.println("onNext should never be printed!"),
        Throwable::printStackTrace,
        () -> System.out.println("onComplete should never be printed!"));
```

Results:

The error message is: Something went wrong
java.io.IOException: Something went wrong

Example 08 onError

ONERRORRETURN




```
Observable<Integer> values =
Observable.create(o -> {
    o.onNext(1);
    o.onNext(2);
    o.onError(
        new Exception("some error"));
    o.onNext(3);
});

values.onErrorReturn(e -> 666)
  .subscribe(v -> System.out.println(v));
```

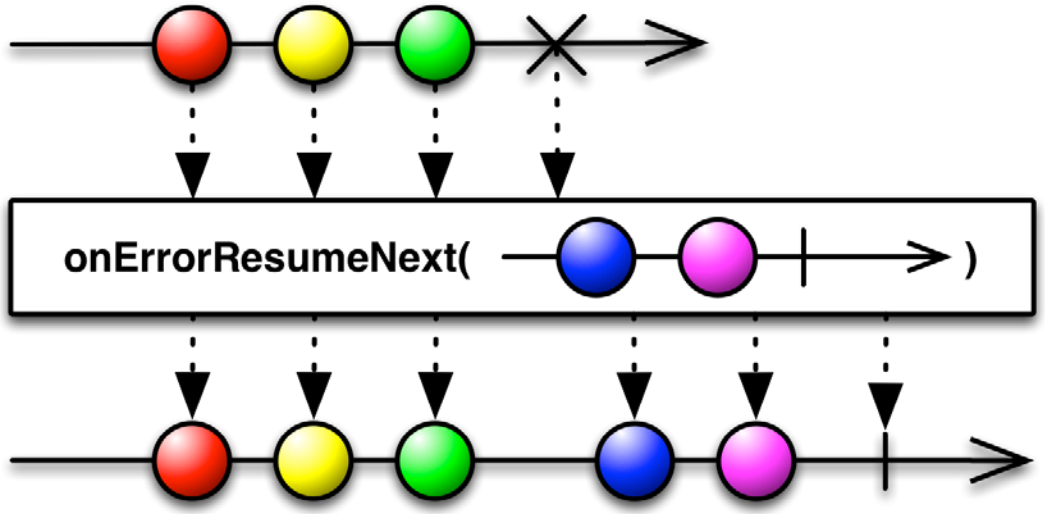
Results:

1
2
666

Example 08 onErrorReturn

ONERRORRESUMENEXT

The onErrorResumeNext allows you to **resume a failed sequence with another sequence**. The error will not appear in the resulting observable.




```
Observable<Integer> values =
        Observable.create(o -> {
    o.onNext(1);
    o.onNext(2);
    o.onError(new Exception("Oops"));
    o.onNext(3);
});

values
    .onErrorResumeNext(Observable.just(6,6,6))
    .subscribe(s-> System.out.println("next: "+s));
```

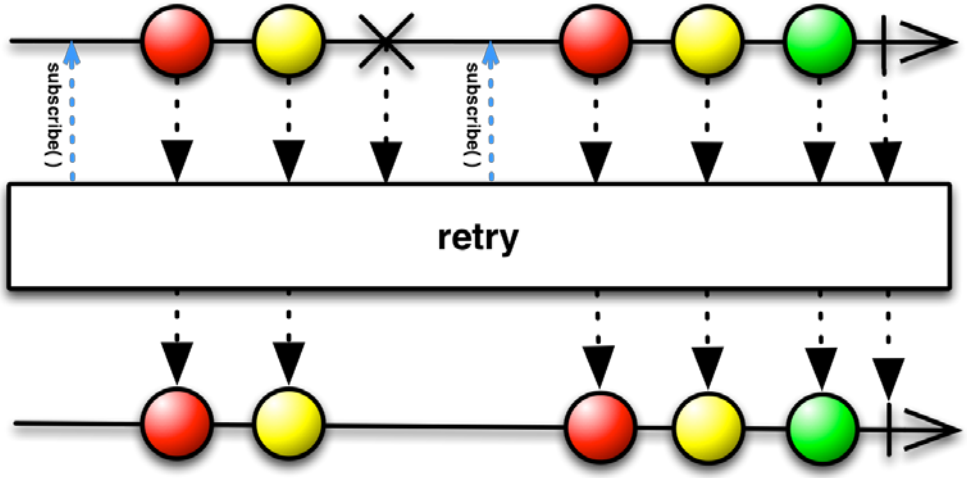
Results:

next: 1
next: 2
next: 6
next: 6
next: 6

Example 08 onErrorResume

RETRY

If the error is non-deterministic, it may make sense to retry. Retry re-subscribes to the source and emits everything again from the start.




```
Random random = new Random();
Observable<Integer> values = Observable.create(o -> {
   o.onNext(1 / (random.nextBoolean() ? 1 : 0));
});

values
   .retry(3)
   .subscribe(v -> System.out.println(v),
         e -> System.out.println("Error: "+e));
```

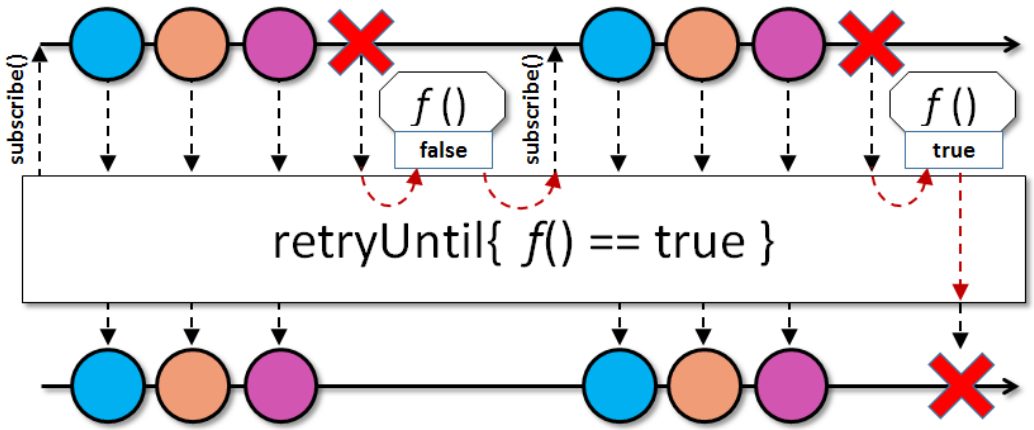
Results:

1

Example 08 retry

RETRY UNTIL

Retries until the given stop function returns true.

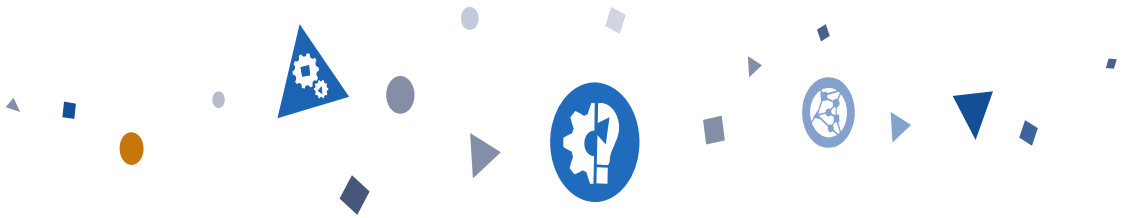



```
Random random = new Random();
LongAdder errorCounter = new LongAdder();
Observable<Integer> source = Observable.create(o -> {
    o.onNext(1 / (random.nextBoolean() ? 1 : 0));
});

source.doOnError((error) -> {
    errorCounter.increment();
    System.out.println("errorCounter "+errorCounter.intValue());
})
.retryUntil(() -> errorCounter.intValue() >= 2)
    .subscribe(x -> System.out.println("onNext: " + x),
        error -> System.err.println(
              "onError: " + error.getMessage()));
```

Results:

errorCounter 1
errorCounter 2
onError: / by zero

Example 08 retryUntil



HOT AND COLD OBSERVABLES

HOT AND COLD OBSERVABLES

- **Hot observables are pushing even when we are not subscribed** to them (e.g., UI events).
- **Cold observables start pushing only when we subscribe**. They start over if we subscribe again.
- Cold observables can be replayed for each observer.
- When we create an observer, we are setting up a whole new separate processing chain.
- A cold observable is **not shared**: each subscriber get its **own copy**.

Examples:

Cold	Hot
Async requests	Timers
DB queries	Push pub/sub
Reading file	UI events
Web service request	Stock prices

COLD OBSERVABLES

```
Observable<String> locations =

Observable.just("Bucharest",
"Krakow", "Moscow", "Kiev",
                "Sofia");

locations.subscribe(s ->
System.out.println("Location for
observer 1: " + s));

locations.subscribe(s ->
System.out.println("Location for
observer 2: " + s));
```

Result

```
Location for observer 1: Bucharest
Location for observer 1: Krakow
Location for observer 1: Moscow
Location for observer 1: Kiev
Location for observer 1: Sofia
Location for observer 2: Bucharest
Location for observer 2: Krakow
Location for observer 2: Moscow
Location for observer 2: Kiev
Location for observer 2: Sofia
```

Example 9

COLD OBSERVABLES

```
Observable<String> locations =

Observable.just("Bucharest",
"Krakow", "Moscow", "Kiev",
                "Sofia");

locations.subscribe(s ->
System.out.println("Location for
observer 1: " + s));

locations.map(String::length)
         .filter(l -> l >= 5)
         .subscribe(l ->
System.out.println("Length for
observer 2: " + l));
```

Result

```
Location for observer 1: Bucharest
Location for observer 1: Krakow
Location for observer 1: Moscow
Location for observer 1: Kiev
Location for observer 1: Sofia
Length for observer 2: 9
Length for observer 2: 6
Length for observer 2: 6
Length for observer 2: 5
```

Example 10

Observable sources that emit finite datasets are usually cold.
Example: SQL database queries.

COLD OBSERVABLE

```
Observable<Long> obs = Observable.interval(1, TimeUnit.SECONDS)
      .doOnNext(v-> System.out.println("Generated: "+v));
System.out.println("Observable created, no subscriptions...");
Thread.sleep(3000);

Disposable sub1 = obs.subscribe(s ->
      System.out.println("Observer 1: " + s));
System.out.println("Observer 1 subscribed...");

Thread.sleep(3000);
Disposable sub2 = obs.subscribe(l ->
      System.out.println("Observer 2: " + l));
System.out.println("Observer 2 subscribed...");

Thread.sleep(3000);
sub1.dispose();
System.out.println("Observer 1 disposed...");

Thread.sleep(3000);
sub2.dispose();
System.out.println("Observer 2 disposed...");

Thread.sleep(3000);
System.out.println("Finished.");
```

Example 11 cold

Result:

```
Observable created, no subscriptions...
Observer 1 subscribed...
Generated: 0
Observer 1: 0
Generated: 1
Observer 1: 1
Generated: 2
Observer 1: 2
Observer 2 subscribed...
Generated: 3
Observer 1: 3
Generated: 0
Observer 2: 0
Generated: 4
Observer 1: 4
Generated: 1
Observer 2: 1
Generated: 5
Observer 1: 5
Generated: 2
Observer 1 disposed...
Observer 2: 2
Generated: 3
Observer 2: 3
Generated: 4
Observer 2: 4
Observer 2 disposed...
Finished.
```

SHARED OBSERVABLE

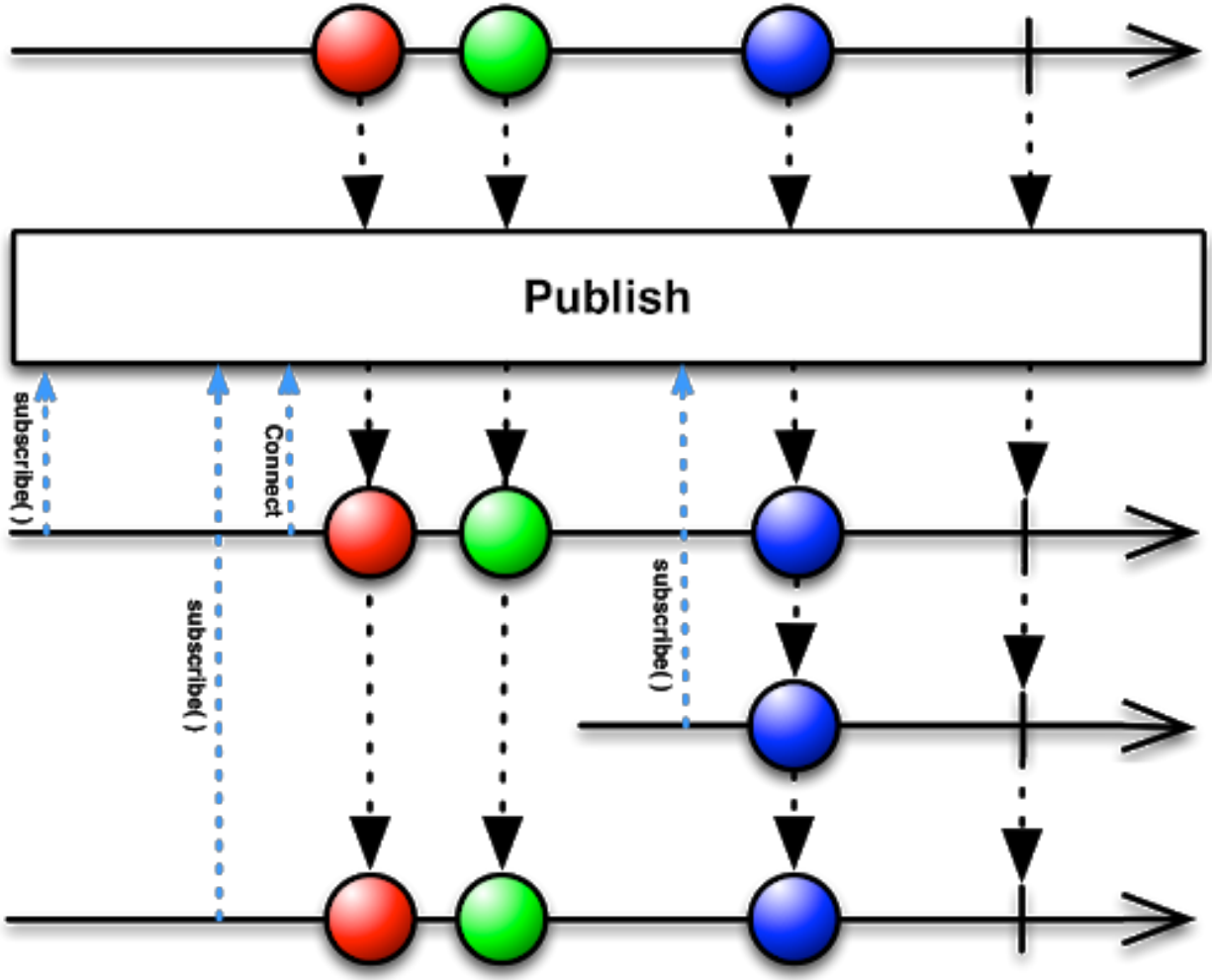
Operator share() makes hot observable from cold.

```
Observable<Long> obs =
      Observable.interval(1,
         TimeUnit.SECONDS)
         .doOnNext(v->
               System.out.println("Generated: "+v))
         .share();
Disposable sub1 = obs.subscribe(s ->
      System.out.println("Observer 1: " + s));
Disposable sub2 = obs.subscribe(l ->
      System.out.println("Observer 2: " + l));
Thread.sleep(3000);
sub1.dispose();
System.out.println("Observer 1 disposed...");
Thread.sleep(3000);
sub2.dispose();
System.out.println("Observer 2 disposed...");
Thread.sleep(3000);
System.out.println("Finished.");
```

Result:

```
Generated: 0
Observer 1: 0
Observer 2: 0
Generated: 1
Observer 1: 1
Observer 2: 1
Generated: 2
Observer 1: 2
Observer 2: 2
Observer 1 disposed...
Generated: 3
Observer 2: 3
Generated: 4
Observer 2: 4
Generated: 5
Observer 2: 5
Observer 2 disposed...
Finished.
```

CONNECTABLE OBSERVABLES



CONNECTABLE OBSERVABLE

```
ConnectableObservable<Long> obs =
      Observable.interval(1,
            TimeUnit.SECONDS)
            .doOnNext(v-> System.out.println("Generated: "+v))
            .publish();
Disposable sub1 = obs.subscribe(s ->
      System.out.println("Observer 1: " + s));
Disposable sub2 = obs.subscribe(l ->
      System.out.println("Observer 2: " + l));

System.out.println("Subscribed but not connected...");
Thread.sleep(3000);
obs.connect();
System.out.println("Connected...");
Thread.sleep(3000);
sub1.dispose();
System.out.println("Observer 1 disposed...");
Thread.sleep(3000);
sub2.dispose();
System.out.println("Observer 2 disposed...");
Thread.sleep(3000);
System.out.println("Finished...");
```

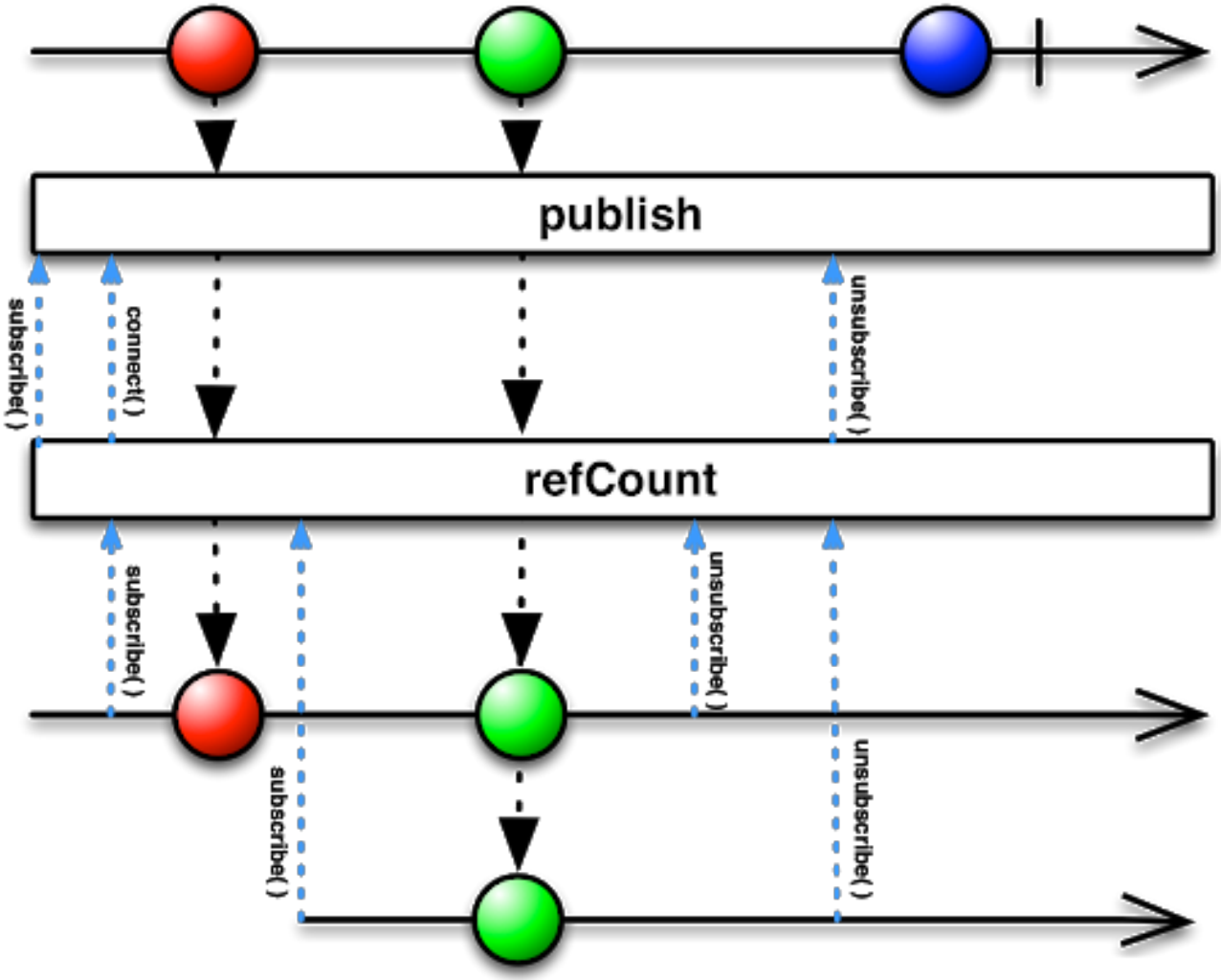
Result:

```
Subscribed but not connected...
Connected...
Generated: 0
Observer 1: 0
Observer 2: 0
Generated: 1
Observer 1: 1
Observer 2: 1
Generated: 2
Observer 2: 2
Observer 1 disposed...
Generated: 3
Observer 2: 3
Generated: 4
Observer 2: 4
Observer 2 disposed...
Generated: 5
Generated: 6
Generated: 7
Generated: 8
Finished...
```

Example 11connect

REFCOUNT

Make a Connectable Observable behave like an ordinary Observable.



REFCOUNT

```
ConnectableObservable<Long> obs =
        Observable.interval(1,
            TimeUnit.SECONDS)
            .doOnNext(v->
            System.out.println("Generated: "+v))
            .publish();
Observable<Long> refCountObs = obs.refCount();

Disposable sub1 = refCountObs.subscribe(s ->
        System.out.println("Observer 1: " + s));
Disposable sub2 = refCountObs.subscribe(l ->
        System.out.println("Observer 2: " + l));

Thread.sleep(3000);
sub1.dispose();
System.out.println("Observer 1 disposed...");
Thread.sleep(3000);
sub2.dispose();
System.out.println("Observer 2 disposed...");
Thread.sleep(3000);
System.out.println("Finished.");
```

Result:

```
Generated: 0
Observer 1: 0
Observer 2: 0
Generated: 1
Observer 1: 1
Observer 2: 1
Generated: 2
Observer 1 disposed...
Observer 2: 2
Generated: 3
Observer 2: 3
Generated: 4
Observer 2: 4
Generated: 5
Observer 2: 5
Observer 2 disposed...
Finished.
```

Example 11 refCount

AUTOCONNECT

```
Observable<Long> obs =
        Observable.interval(1, TimeUnit.SECONDS)
                .doOnNext(v->
System.out.println("Generated: "+v))
                .publish()
                .autoConnect();
System.out.println("Created, but no subscriptions...");
Thread.sleep(3000);
Disposable sub1 = obs.subscribe(s ->
        System.out.println("Observer 1: " + s));
System.out.println("Subscribed observer 1...");
Thread.sleep(3000);
Disposable sub2 = obs.subscribe(l ->
        System.out.println("Observer 2: " + l));
System.out.println("Subscribed observer 2...");
Thread.sleep(3000);
sub1.dispose();
System.out.println("Observer 1 disposed...");
Thread.sleep(3000);
sub2.dispose();
System.out.println("Observer 2 disposed...");
Thread.sleep(3000);
System.out.println("Finished...");
```

Results:

```
Created, but no subscriptions...
Subscribed observer 1...
Generated: 0
Observer 1: 0
Generated: 1
Observer 1: 1
Generated: 2
Observer 1: 2
Subscribed observer 2...
Generated: 3
Observer 1: 3
Observer 2: 3
Generated: 4
Observer 1: 4
Observer 2: 4
Generated: 5
Observer 1: 5
Observer 2: 5
Observer 1 disposed...
Generated: 6
Observer 2: 6
Generated: 7
Observer 2: 7
Generated: 8
Observer 2: 8
Observer 2 disposed...
Generated: 9
Generated: 10
Generated: 11
Finished...
```

Example 11 autoConnect

OBSERVABLE FACTORIES

- Observable.just(1,2,3)
- Observable.create()
- Observable.interval(1, TimeUnit.SECONDS)
- Observable.fromIterable(Arrays.asList(1,2,3))
- Observable.fromFuture()
- Observable.fromCallable(() -> "result")
- Observable.range(start, count)
- Observable.empty()
- Observable.never()

  // same as empty(), but do not generate onComplete
- Observable.error()
- Observable.defer(() -> Observable.range(start, count))

OBSERVABLE FACTORIES

```
Observable.fromIterable(new ArrayList<String>
(Arrays.asList("Bucharest", "Krakow", "Moscow", "Kiev",
     "Sofia"))).subscribe(s -> System.out.println("Location: " + s));

Observable.range(3, 4)
        .subscribe(s -> System.out.println("Counter: " + s));

Observable.empty().subscribe(System.out::println,
        Throwable::printStackTrace,
        () -> System.out.println("Done from empty."));

Observable.never().subscribe(System.out::println,
        Throwable::printStackTrace,
        () -> System.out.println("Done from never."));
Thread.sleep(5000);

Observable.error(new Exception("Forcing an error!"))
        .subscribe(s -> System.out.println("Received: " + s),
                Throwable::printStackTrace,
                () -> System.out.println("Done from error."));
```

OBSERVABLE FACTORIES

Result

```
Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Done from empty.
java.lang.Exception: Forcing an error!
        at com.luxoft.rxjava.Example12.main(Example12.java:35)
```

Example 12

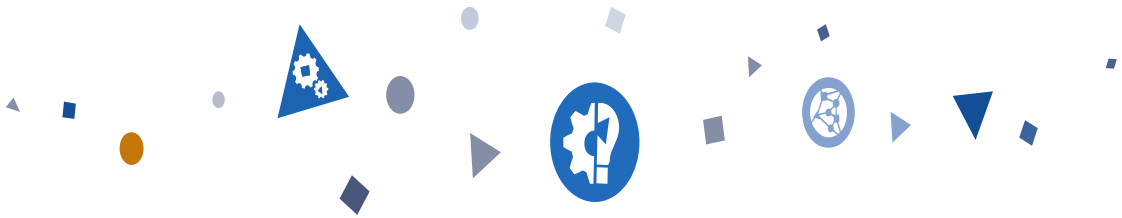
OBSERVABLE FACTORIES

```
private static int start = 1;
private static int count = 3;

public static void main(String[] args) {
    Observable<Integer> source =
        Observable.defer(() ->
            Observable.range(start, count));
    source.subscribe(i -> System.out.println(
                        "Observer 1: " + i));

    start = 4;
    count = 5;
    source.subscribe(i -> System.out.println(
                        "Observer 2: " + i));
}
```

Results:

Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 4
Observer 2: 5
Observer 2: 6
Observer 2: 7
Observer 2: 8

Example 13



SINGLE, MAYBE AND COMPLETABLE

SINGLE

- `Single<T>` is an `Observable<T>` that will only emit one item.
- It works like an `Observable`, but is limited only to operators that make sense for a single emission.
- It has its own `SingleObserver` interface:

```
interface SingleObserver<T> {
    void onSubscribe(Disposable d);
    void onSuccess(T value);
    void onError(Throwable error);
}
```

- `onSuccess()` consolidates `onNext()` and `onComplete()` into a single event that accepts the single emission

SINGLE

```
Single.just("Single")
      .map(String::length)
      .subscribe(System.out::println,
         Throwable::printStackTrace);

Observable<String> locations =
        Observable.just("Bucharest",
           "Krakow", "Moscow",
           "Kiev", "Sofia");

locations.first("Default item")
  .subscribe(System.out::println);
```

Results:

6

Bucharest

Example 14

MAYBE

- `Maybe` is just like a `Single` except that it allows no emission to occur at all (hence `Maybe`).

- It has its own `MaybeObserver` interface:

```
interface MaybeObserver<T> {
    void onSubscribe(Disposable d);
    void onSuccess(T value);
    void onError(Throwable error);
    void onComplete();
}
```

MAYBE

```
Maybe<String> justSource =
       Maybe.just("Bucharest");

justSource.subscribe(s ->
System.out.println("Receiver 1: " + s),
  Throwable::printStackTrace,
  () -> System.out.println("Done 1"));

Maybe<String> emptySource =
       Maybe.empty();

emptySource.subscribe(s ->
System.out.println("Receiver 2: " + s),
  Throwable::printStackTrace,
  () -> System.out.println("Done 2"));
```

Example 15

Result

Receiver 1: Bucharest

Done 2

COMPLETABLE

- `Completable` is concerned with an action being executed, but it does not receive any emissions.
- It does not have `onNext()` or `onSuccess()` to receive emissions, but it has `onError()` and `onComplete()`.
- It has its own `CompletableObserver` interface:

```
interface CompletableObserver<T> {
   void onSubscribe(Disposable d);
   void onError(Throwable error);
   void onComplete();
}
```

COMPLETABLE

```
public static void main(String[] args) {
    Completable.fromRunnable(() -> execute())
       .subscribe(() -> System.out.println("finish"));
}

public static void execute() {
    System.out.println("execute");
}
```

Results:

execute

finish

Example 16

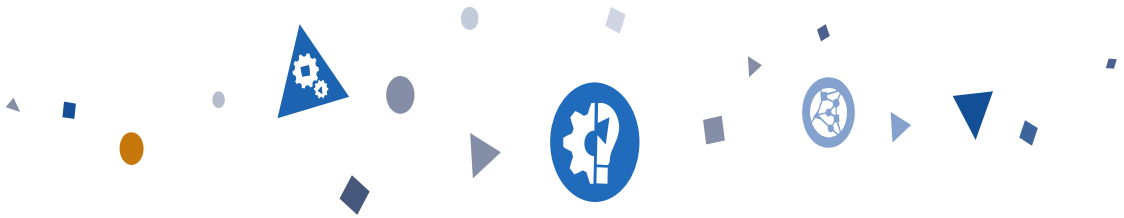
ONERRORCOMPLETE

```
Completable.fromAction(() -> {
    throw new IOException();
}).onErrorComplete(error -> {
    // Only ignore errors of type java.io.IOException.
    return error instanceof IOException;
}).subscribe(
    () -> System.out.println("IOException was ignored"),
    error -> System.err.println("onError should not be printed!"));
```

Results:

```
IOException was ignored
```

Example 16 onErrorComplete



DISPOSING

DISPOSING

- When you `subscribe()` to an `Observable` to receive emissions, an `Observable` chain is created to process the emissions.
- The finite `Observables` that call `onComplete()` will typically dispose of themselves safely when they are done.
- Working with infinite or long-running `Observables`, you may run into situations where you want to explicitly stop the emissions and dispose everything related to that subscription.
- An explicit disposal is necessary to prevent memory leaks.
- The `Disposable` is a link between an `Observable` and an active `Observer`, you can call its `dispose()` method to stop emissions and dispose of all resources used for that `Observer`.

DISPOSING

```
public interface Disposable {
   void dispose();
   boolean isDisposed();
}
```

- When you provide `onNext()`, `onComplete()`, and/or `onError()` lambdas as arguments to the `subscribe()` method, it will actually return a `Disposable`.
- You can use this to stop emissions at any time by calling its `dispose()` method.

DISPOSING

```
Observable<Long> observable =
        Observable.interval(1,
TimeUnit.SECONDS);
Disposable disposable =
        observable.subscribe(l ->
System.out.println("Item: " + l));

Thread.sleep(5000);
disposable.dispose();
Thread.sleep(5000);
```

Results:

Item: 0

Item: 1

Item: 2

Item: 3

Item: 4

Example 17

COMPOSITEDISPOSABLE

- If you have several subscriptions that need to be managed and disposed of, it can be helpful to use `CompositeDisposable`.
- It implements `Disposable`, but it internally holds a collection of disposables, which you can add to and then dispose all at once.

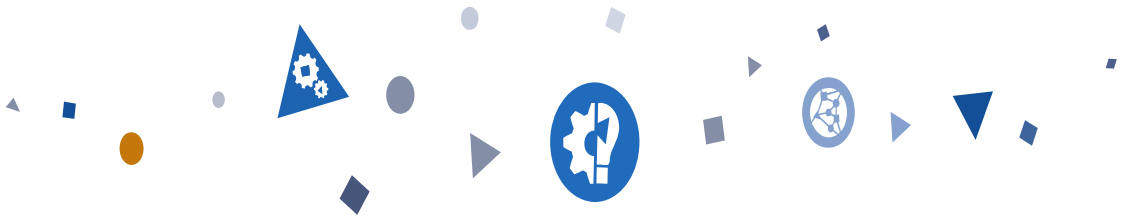
```
Observable<Long> observable =
        Observable.interval(1, TimeUnit.SECONDS);
Disposable disposable1 =
observable.subscribe(
      l -> System.out.println("Observer 1: " + l));
Disposable disposable2 =
observable.subscribe(
      l -> System.out.println("Observer 2: " + l));

disposables.addAll(disposable1, disposable2);
Thread.sleep(5000);
disposables.dispose();
Thread.sleep(5000);
```

Example 18

vsonkin@luxoft.com

PRACTICE: TASK 1

1) Create observable which will emit operations on bank account (cash deposit and withdrawal) at some random moments - use **create**(), **onNext**() and Thread.sleep() for delays: [+100, +1000, -200, -300]
2) Use **filter()** to show only deposits
3) Use **map()** to transform values in cents (multiply by 100)
4) Throw error in the observable with **onError()**, print error details by subscribing to onError() hook
5) Add **onComplete()** and add onComplete() hook to subscribe() method
6) Use Disposable to **unsubscribe** from the observable after some delay and check that you have stopped to receive messages
7) Use side effect methods to print elements before filtering, after filtering but before mapping and in case of error (use **doOnNext**(), **doOnError**())
8) In case of error replace it by 0 (use **onErrorReturn**())
9) Transform observable to **shared** and perform multiple subscriptions to it
10) Make observable **connectable**, subscribe to it using several subscription and **connect**() to it after delay
11) Postpone creating of observable (use **defer**()), make a pause before subscribing
12) Keep disposables from all subscriptions, make **composite** from all of them cancel all of them at once



OPERATORS

SUPPRESSING OPERATORS

Suppress emissions that fail to fulfill a criterion.

- filter
- take
- skip
- takeWhile
- skipWhile
- distinct
- distinctUntilChanged
- elementAt

FILTER OPERATOR

marbles – шарики


```
filter(x => x > 10)
```

FILTER OPERATOR

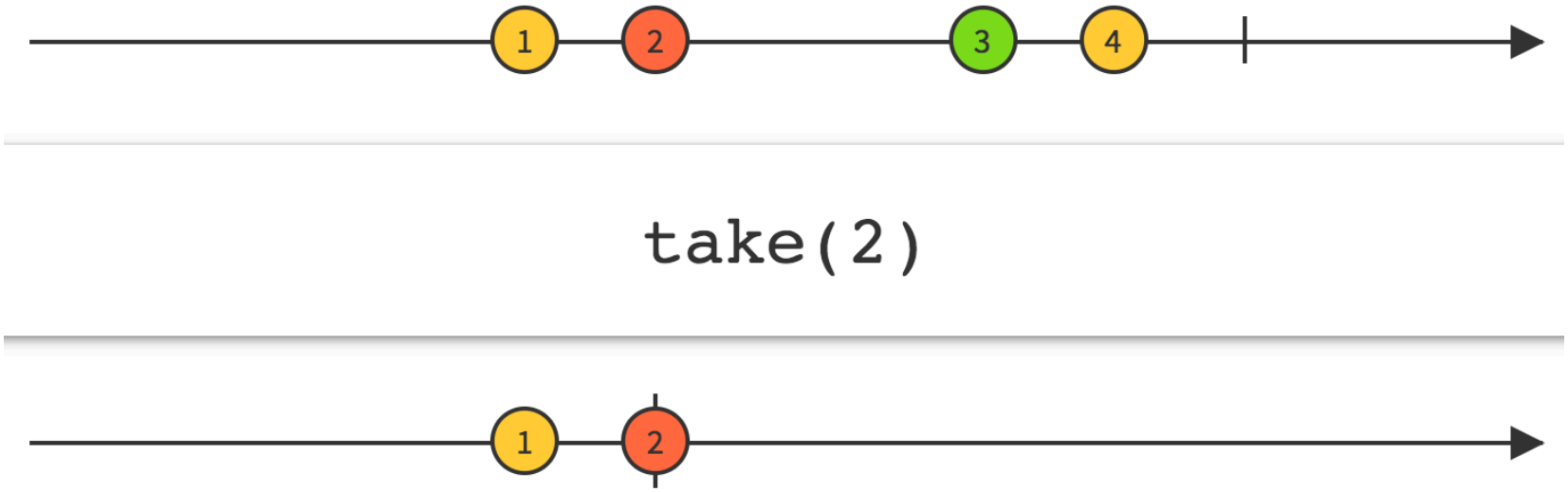
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .filter(s -> s.length() >= 5)
        .subscribe(s ->
System.out.println("Location: " + s));
```

Example 19

Result

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Sofia

TAKE OPERATOR



TAKE OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .take(2)
        .subscribe(s ->
System.out.println("Location: " + s));
```

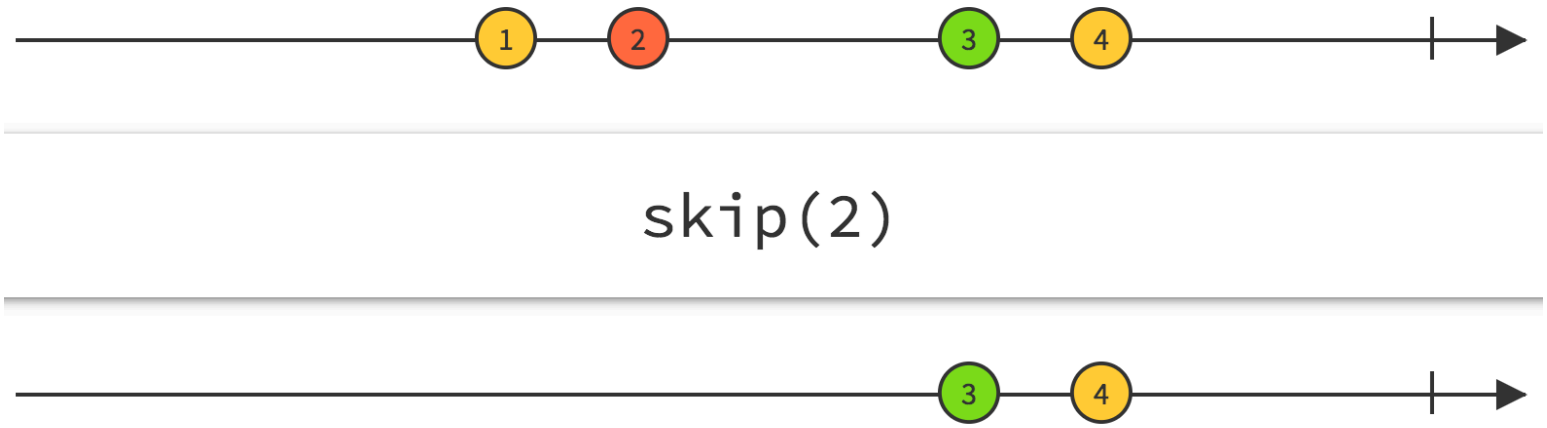
Example 20

Result

Location: Bucharest

Location: Krakow

SKIP OPERATOR



SKIP OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .skip(2)
          .subscribe(s ->
System.out.println("Location: " + s));
```

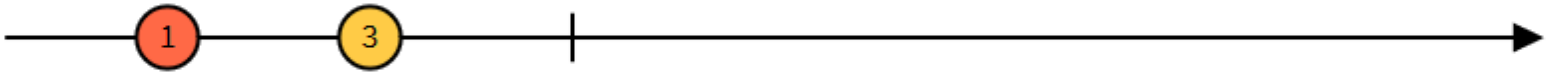
Example 21

Result

Location: Moscow
Location: Kiev
Location: Sofia

TAKEWHILE OPERATOR


```
takeWhile(x => x < 5)
```



TAKEWHILE OPERATOR

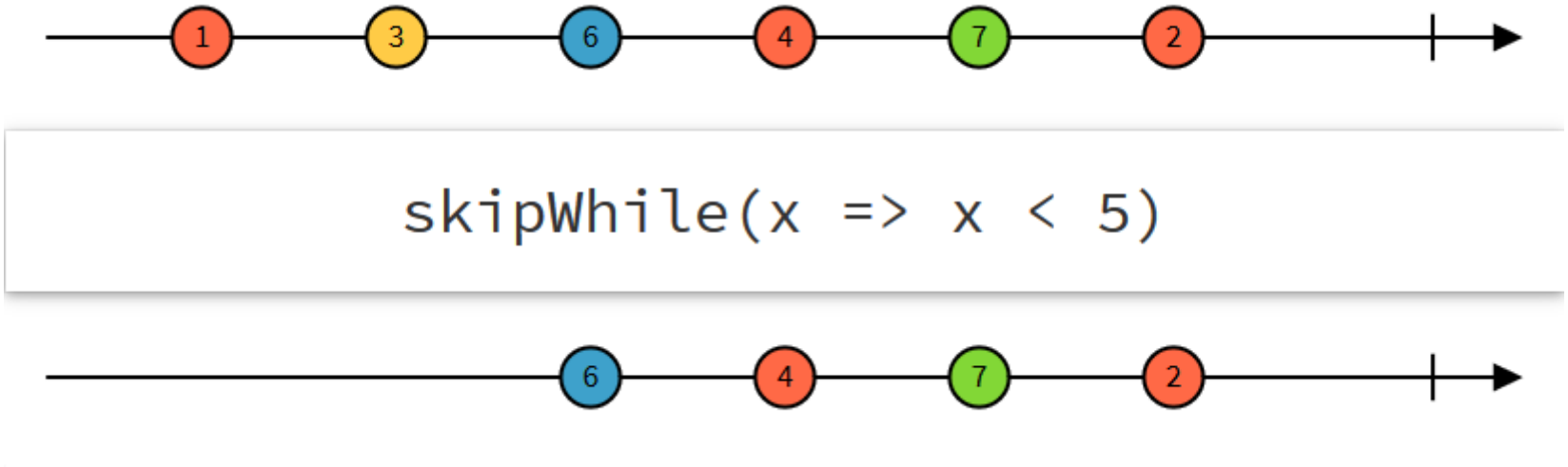
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .takeWhile(s -> s.length() >=5)
        .subscribe(s ->
System.out.println("Location: " + s));
```

Example 22

Result

Location: Bucharest
Location: Krakow
Location: Moscow

SKIPWHILE OPERATOR



SKIPWHILE OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .skipWhile(s -> s.length() >=5)
        .subscribe(s ->
System.out.println("Location: " + s));
```

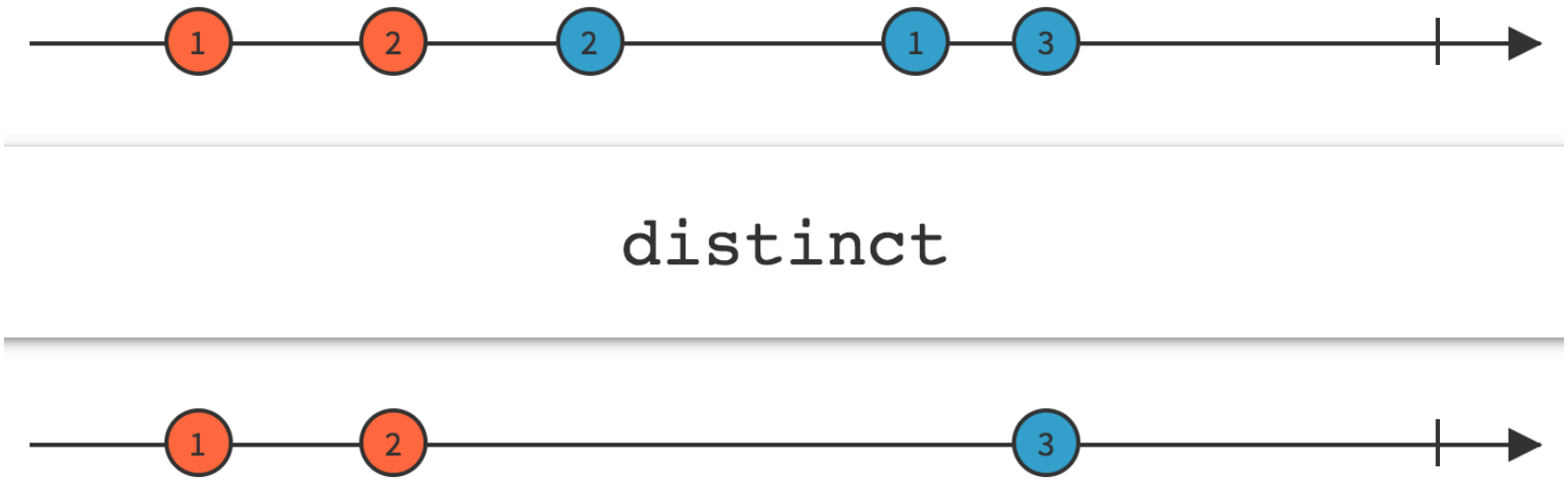
Example 23

Result

Location: Kiev

Location: Sofia

DISTINCT OPERATOR



DISTINCT OPERATOR

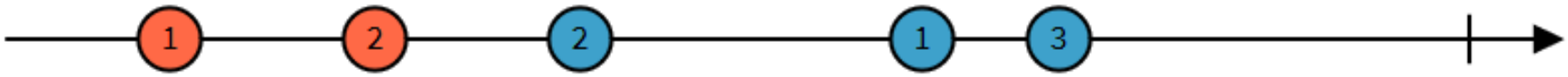
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Krakow", "Moscow",
                "Kiev", "Sofia")
        .distinct()
        .subscribe(s ->
System.out.println("Location: " + s));
```

Example 24

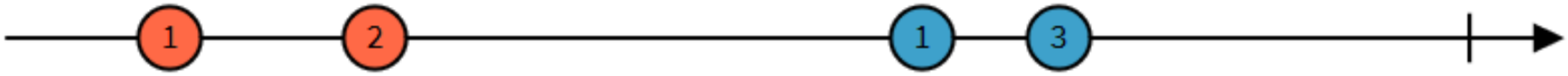
Result

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia

DISTINCTUNTILCHANGED OPERATOR



distinctUntilChanged



DISTINCTUNTILCHANGED OPERATOR

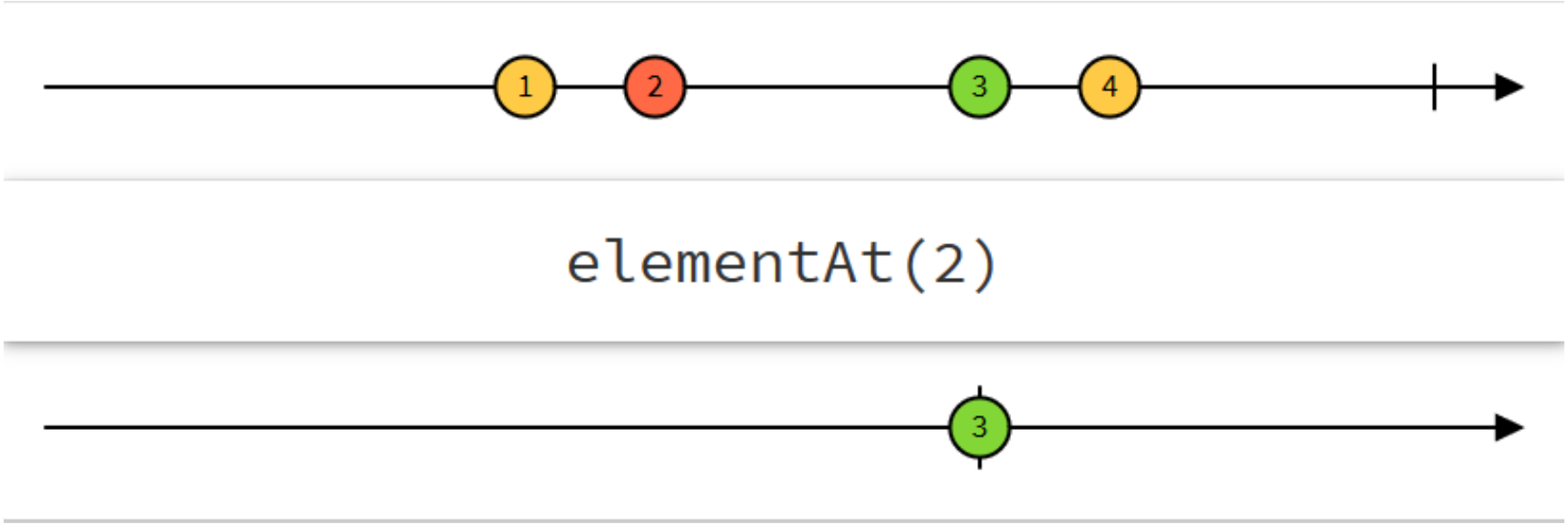
```
Observable.just("Bucharest", "Krakow",
                "Krakow", "Moscow", "Moscow",
                "Kiev", "Moscow", "Sofia")
        .distinctUntilChanged()
        .subscribe(s ->
System.out.println("Location: " + s));
```

Example 25

Result

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Moscow
Location: Sofia

ELEMENTAT OPERATOR



ELEMENTAT OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .elementAt(2)
          .subscribe(s ->
System.out.println("Location: " + s));
```

Example 26

Result

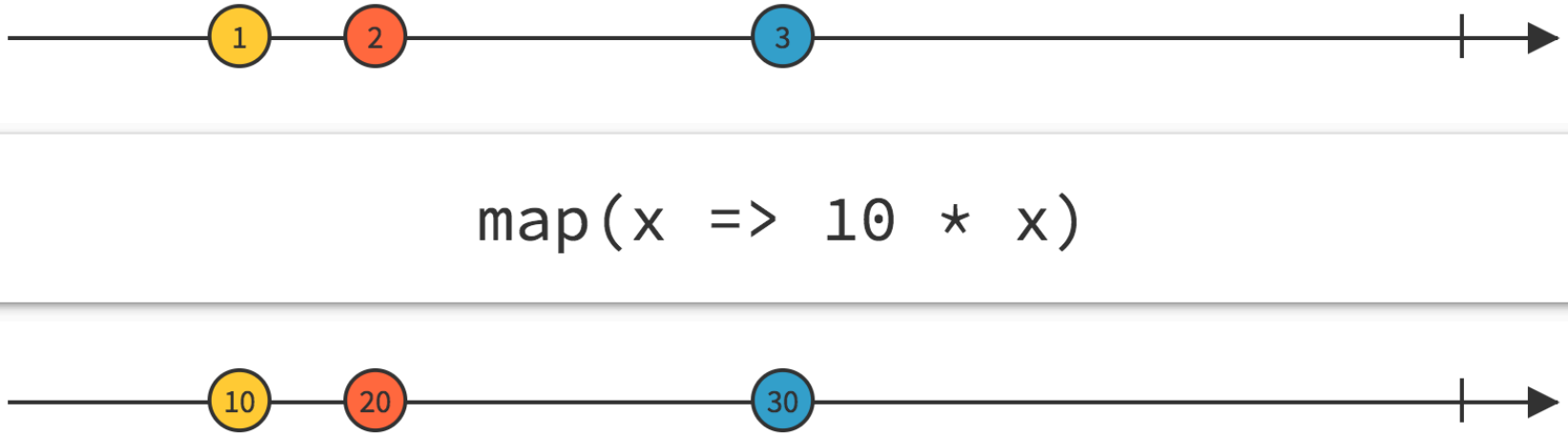
Location: Moscow

TRANSFORMING OPERATORS

A series of operators in an `Observable` chain is a stream of transformations.

- map
- startWith
- defaultIfEmpty
- sorted
- delay
- scan
- groupBy
- timestamp

MAP OPERATOR



MAP OPERATOR

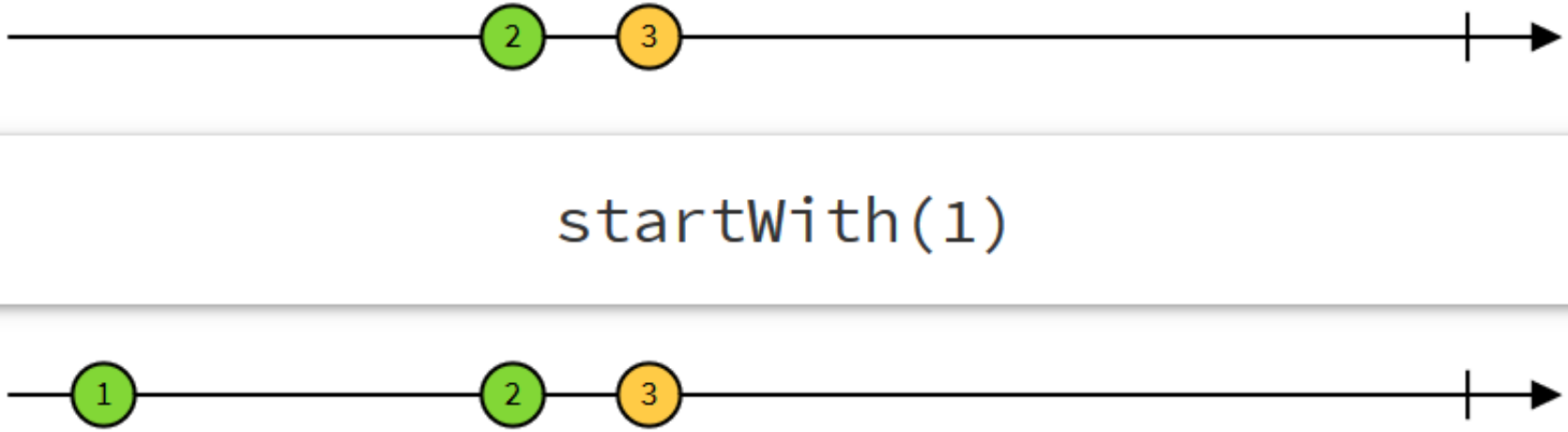
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .map(s->s.length())
          .subscribe(l -> System.out.println(
               "Length: " + l));
```

Results:

Length: 9
Length: 6
Length: 6
Length: 4
Length: 5

Example 27

STARTWITH OPERATOR



STARTWITH OPERATOR

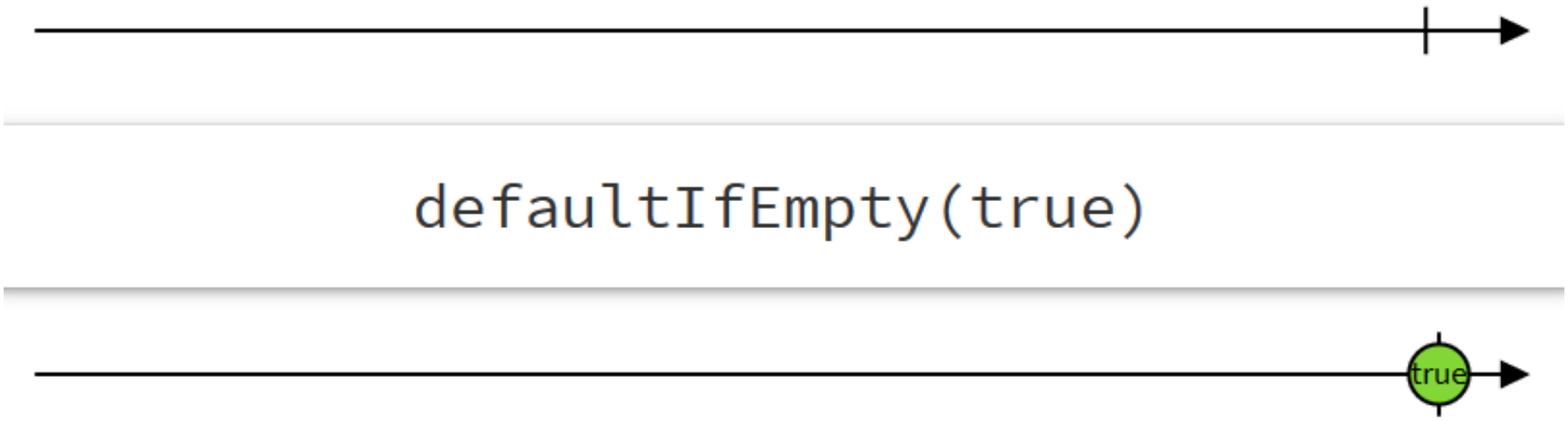
```
Observable.just("Bucharest", "Krakow",
        "Moscow", "Kiev", "Sofia")
.startWith("Berlin")
.subscribe(l -> System.out.println(
                "Location: " + l));
```

Results:

Location: Berlin
Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia

Example 28

DEFAULTIFEMPTY OPERATOR



Real-life use case:

we have to do a discount for the payment, but if the user is not registered in database, discount should be 0:

```
double discount = findUser()
                      .map(user -> user.getDiscount())
                      .defaultIfEmpty(0)
```

DEFAULTIFEMPTY OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .filter(s -> s.startsWith("A"))
        .defaultIfEmpty("None")
        .subscribe(l -> System.out.println(
             "Location: " + l));
```

Results:

Location: None

Example 29

SORTED OPERATOR

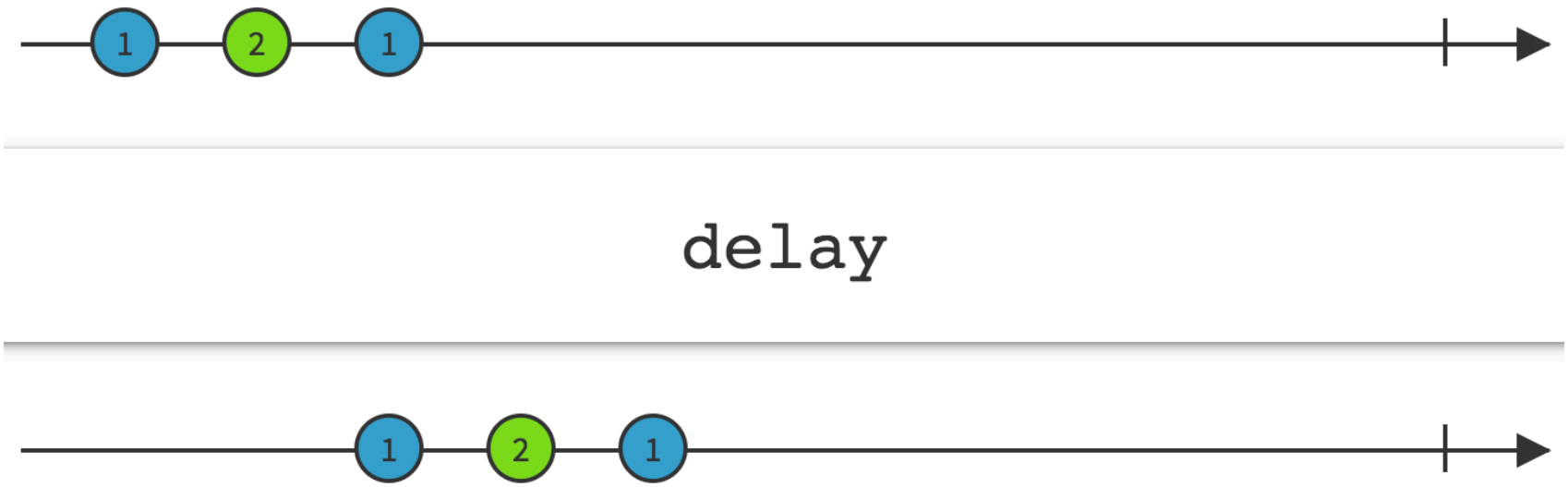
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .sorted()
          .subscribe(l -> System.out.println(
               "Location: " + l));
```

Example 30

Result

Location: Bucharest
Location: Kiev
Location: Krakow
Location: Moscow
Location: Sofia

DELAY OPERATOR



Real-life use case:

we get payments info as Observable, but should wait for some time before persisting to the database because it can be rejected during some timespan.

DELAY OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .delay(2, TimeUnit.SECONDS)
        .subscribe(l -> System.out.println(
            "Location: " + l));

Thread.sleep(5000);
```

Example 31

Result

Location: Bucharest
Location: Kiev
Location: Krakow
Location: Moscow
Location: Sofia

SCAN OPERATOR


```
scan((x, y) => x + y)
```

Real-life use case:

we get payments info, and want to have the actual account state, i.e.

Operation	*Account State*
	1000
+100	*1100*
-200	*900*

SCAN OPERATOR

```
Observable.range(1, 10)
          .scan((x, y) -> x+y)
          .subscribe(sum -> System.out.println(
               "Sum: " + sum));
```

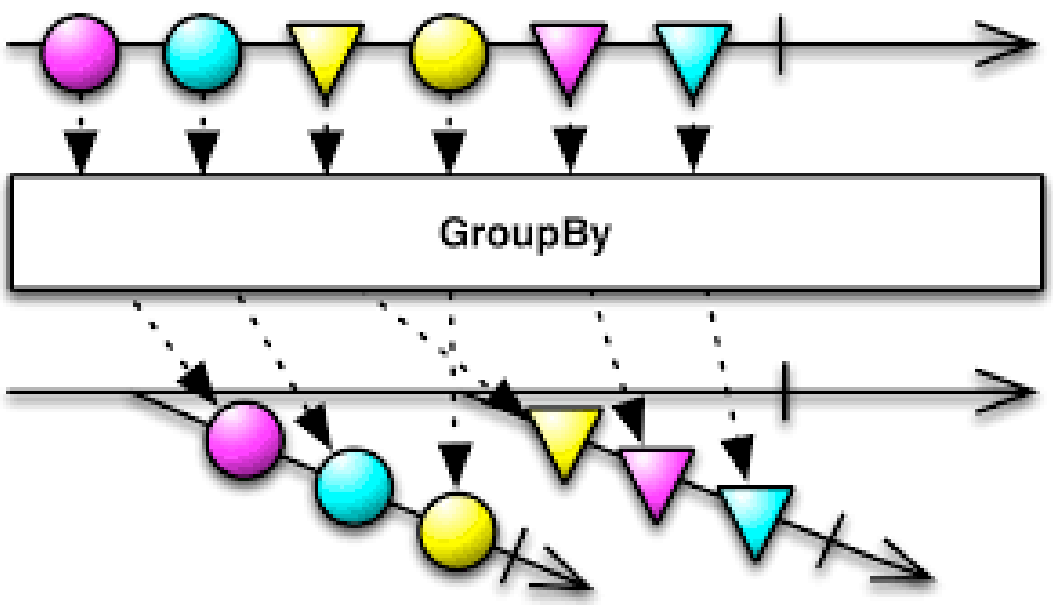
Example 32

Result

Sum: 1
Sum: 3
Sum: 6
Sum: 10
Sum: 15
Sum: 21
Sum: 28
Sum: 36
Sum: 45
Sum: 55

GROUPBY OPERATOR

Divide an Observable into a set of Observables that each emit a different subset of items from the original Observable.




```
Observable.just("Bucharest", "Krakow", "Moscow", "Kiev", "Sofia").
   groupBy(s -> s.length())
   .subscribe(group -> {
      System.out.println(group.getKey());
      group.subscribe(System.out::println);
   });
```

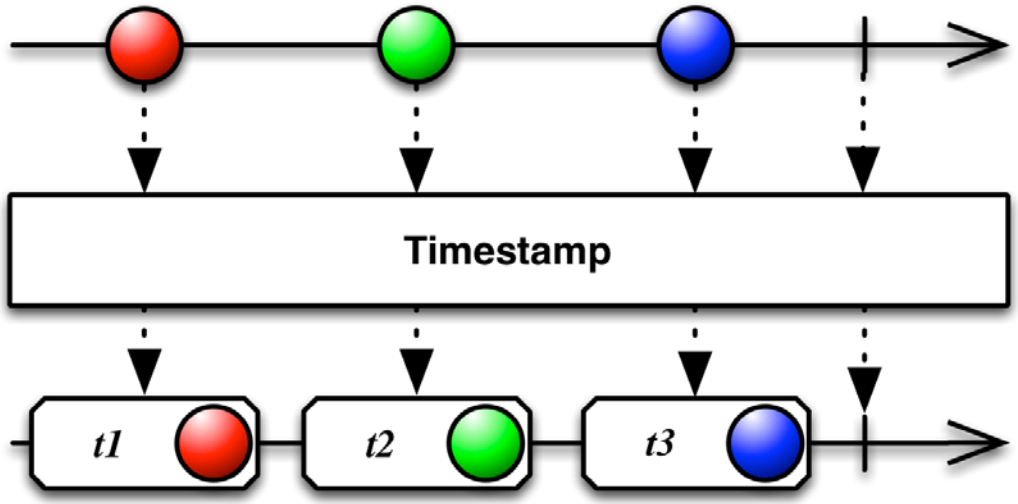
Results:

```
9
Bucharest
6
Krakow
Moscow
4
Kiev
5
Sofia
```

Example 32 groupBy

TIMESTAMP OPERATOR

Attach a timestamp to each item emitted by an Observable indicating when it was emitted.




```
Observable.interval(1, TimeUnit.SECONDS)
    .take(3)
    .timestamp()
    .blockingSubscribe(val -> {
        System.out.println(Instant.ofEpochMilli(val.time()));
        System.out.println(val.value());
    });
```

Results:

```
2020-06-03T22:07:57.468Z
0
2020-06-03T22:07:58.470Z
1
2020-06-03T22:07:59.469Z
2
```

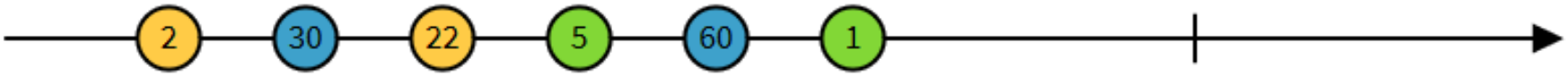
Example 32 timestamp

REDUCING OPERATORS

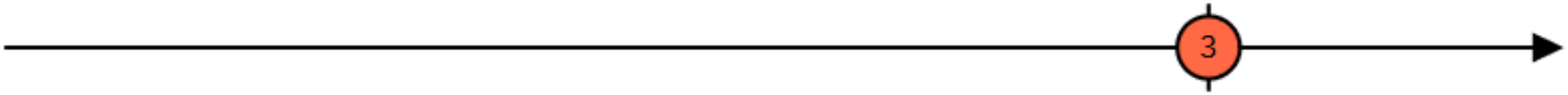
Take a series of emissions and consolidate them into a single emission.

- count
- reduce
- all
- any

COUNT OPERATOR




```
count(x => x > 10)
```



COUNT OPERATOR

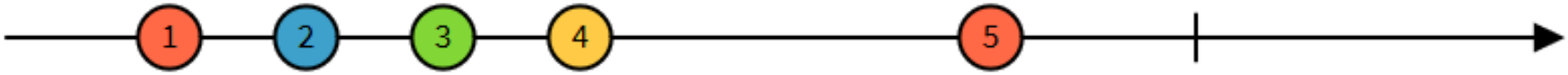
```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .count()
          .subscribe(counter ->
               System.out.println(
               “Counter: " + counter));
```

Example 33

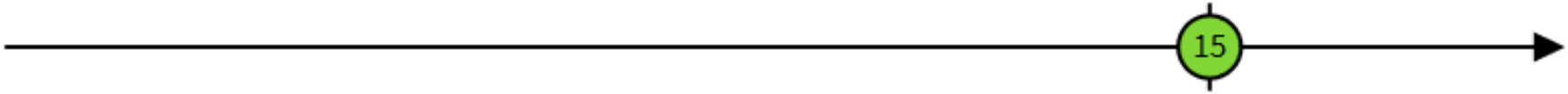
Result

Counter: 5

REDUCE OPERATOR




```
reduce((x, y) => x + y)
```



REDUCE OPERATOR

```
Observable.range(1, 10)
          .reduce((total, next) -> total + next)
          .subscribe(total ->
                System.out.println(
                "Total: " + total));
```

Example 34

Result

Total: 55

ALL OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .all(s -> s.length()>3)
          .subscribe(result ->
               System.out.println(
               “Result: " + result));
```

Example 35

Result

Result: true

ANY OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .any(s -> s.length()>6)
          .subscribe(result ->
               System.out.println(
               “Result: " + result));
```

Example 36

Result

Result: true

COLLECTION OPERATORS

Collection operators accumulate all emissions into a collection such as a list or map and then emit that entire collection as a single emission.

- toList
- toSortedList
- toMap
- toMultiMap

TOLIST OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .toList()
          .subscribe(list ->
               System.out.println(list));
```

Example 37

Result

[Bucharest, Krakow, Moscow, Kiev, Sofia]

TOSORTEDLIST OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .toSortedList()
          .subscribe(list ->
               System.out.println(list));
```

Example 38

Result

[Bucharest, Kiev, Krakow, Moscow, Sofia]

TOMAP OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
          .toMap(s->s.charAt(0))
          .subscribe(map ->
               System.out.println(map));
```

Example 39

Result

{B=Bucharest, S=Sofia, K=Kiev, M=Moscow}

TOMULTIMAP OPERATOR

```
Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
        .toMultimap(s->s.charAt(0))
        .subscribe(map ->
            System.out.println(map));
```

Example 40

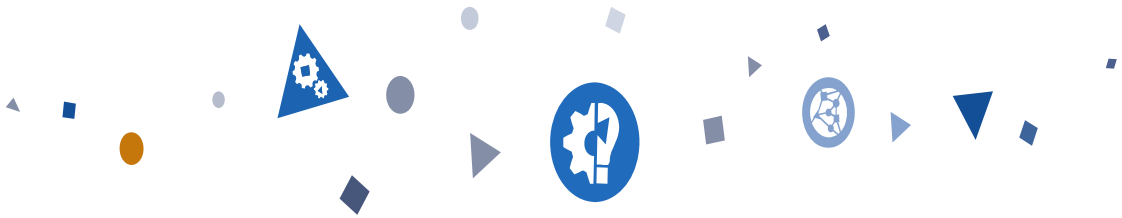
Result

{B=[Bucharest], S=[Sofia], K=[Krakow, Kiev], M=[Moscow]}

Real-life use case:

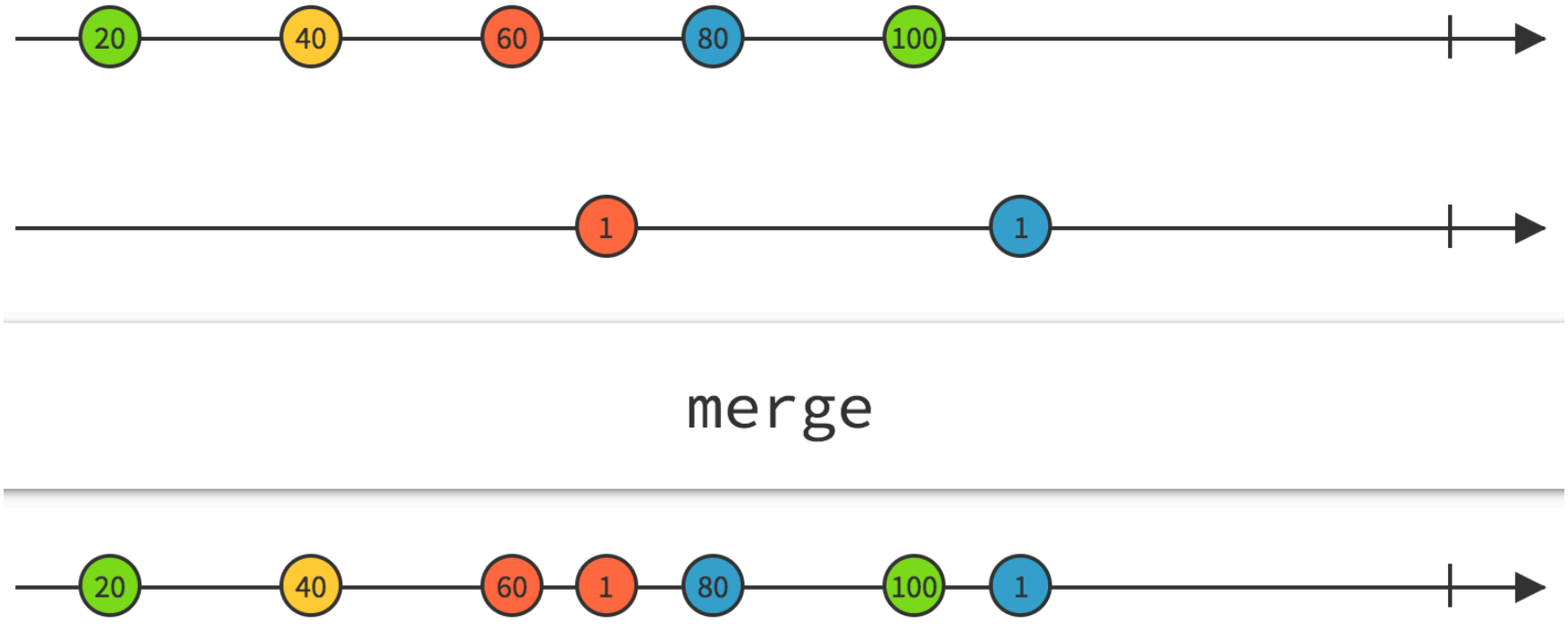
we need to sort out payments by category: ***incomes and expenses***

```
enum PaymentType { INCOME, EXPENSE };
Map<Payment> paymentsByType =
    payments.toMultiMap(p->p.amount>0 ?
            PaymentType.INCOME : PaymentType.EXPENSE);
```



COMBINING OBSERVABLES

MERGE OPERATOR



Real-life use case:

we have 2 sources of the payment info: internal bank payment system and interbank payment system, but we need observable having all the payments together:

*Observable<Payment> allPayments = Observable.**merge**(*
internalBankPayments, interbankPayments);

MERGE OPERATOR

```
Observable<String> locations =
Observable.just("Bucharest", "Krakow", "Moscow", "Kiev",
"Sofia");

Observable<String> locations2 =
        Observable.just("Berlin", "Stuttgart", "London");

Observable.merge(locations, locations2)
.subscribe(s -> System.out.println("Location: " + s));
```

Result:

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia
Location: Berlin
Location: Stuttgart
Location: London

Example 41

MERGEWITH OPERATOR

```
Observable<String> locations =
Observable.just("Bucharest", "Krakow",
"Moscow", "Kiev", "Sofia");

Observable<String> locations2 =
 Observable.just("Berlin",
       "Stuttgart", "London");

locations.mergeWith(locations2)
.subscribe(s ->
       System.out.println("Location: " + s));
```

Example 42

Result

Location: Bucharest
Location: Krakow
Location: Moscow
Location: Kiev
Location: Sofia
Location: Berlin
Location: Stuttgart
Location: London

MERGE OPERATOR

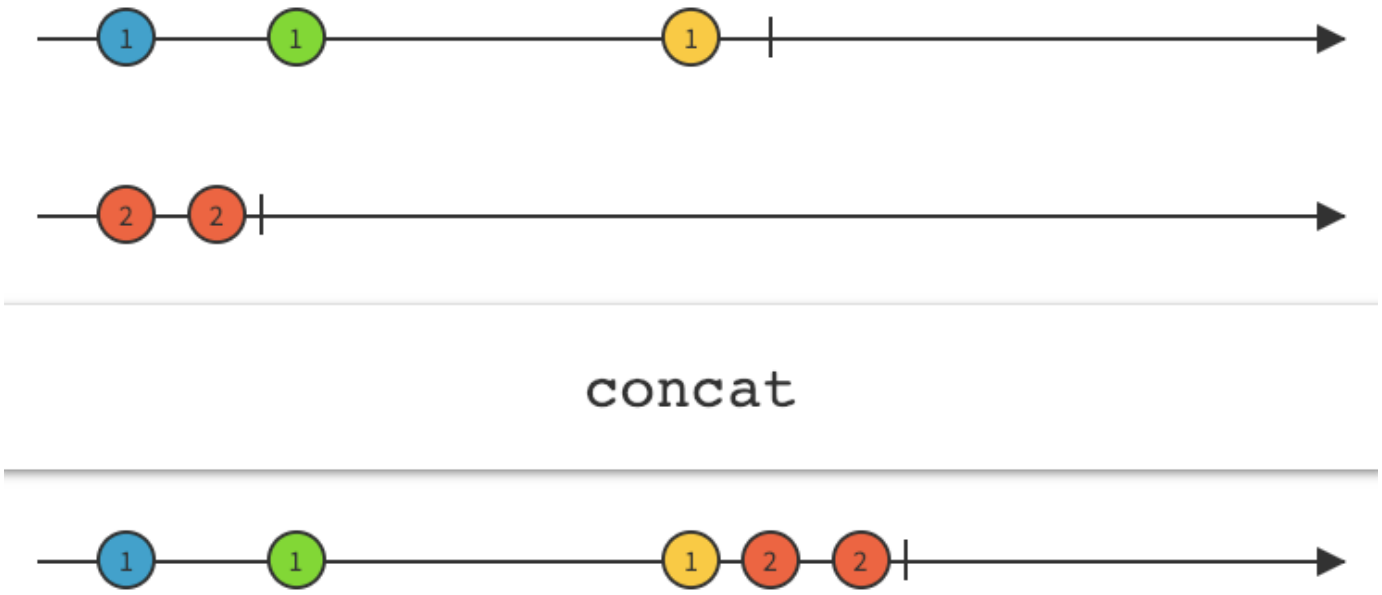
```
Observable<String> source1 =
Observable.interval(1, TimeUnit.SECONDS)
.map(l -> "Source1: " + (l + 1) + " s");

Observable<String> source2 =
Observable.interval(500, TimeUnit.MILLISECONDS)
.map(l -> "Source2: " + ((l + 1) * 500) + " ms");
Observable.merge(source1, source2)
        .subscribe(System.out::println);
Thread.sleep(2000);
```

Example 43

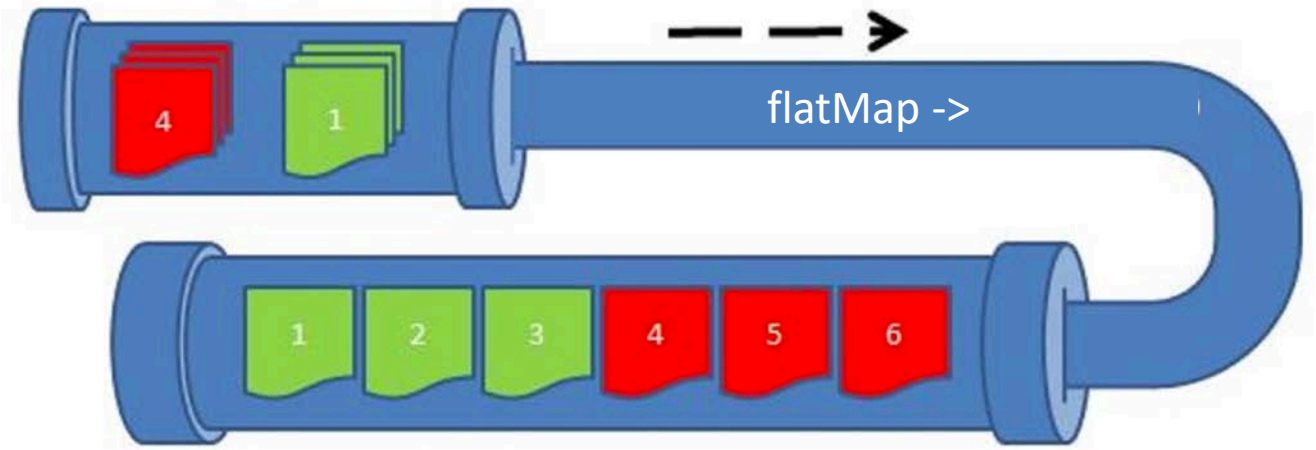
Result

Source2: 500 ms
Source2: 1000 ms
Source1: 1 s
Source2: 1500 ms
Source2: 2000 ms

CONCAT OPERATOR



FLATMAP OPERATOR



Real-life use case:

we have to find a client details for the payment in database, but clientsRepo.findUser(id) ***returns an Observable<User>****, but we need to have a* ***list of users****:*

```
Observable<Observable<User>> users = paymentInfo.map(
        p -> clientsRepo.findUser(p.user.id)); // that's not what we need!

Observable<User> users = paymentInfo.flatMap(
        p -> clientsRepo.findUser(p.user.id)); // much better
```

FLATMAP OPERATOR

```
public static Observable<String> remoteRequest(String s) {
   return Observable.just("("+s+")");
}

Observable<String> locations =
      Observable.just("Bucharest", "Krakow", "Moscow", "Kiev", "Sofia");
locations.flatMap(s -> remoteRequest(s))
      .subscribe(System.out::println);
```

Example 44 flatMap

FLATMAP ELEMENTS ORDER

flatMap() doesn't preserve order of the elements – first processed is the first in results!

```
static long delay = 1000;
public static Observable<String> remoteRequest(String s) {
   delay -= 100;
   return Observable.just("("+s+")")
         .delay(delay, TimeUnit.MILLISECONDS);
}

Observable<String> locations =
      Observable.just("Bucharest", "Krakow",
            "Moscow", "Kiev", "Sofia");
locations.flatMap(s -> remoteRequest(s))
      .blockingSubscribe(System.out::println);
```

Results:
(Sofia) (500)
(Kiev) (600)
(Moscow) (700)
(Krakow) (800)
(Bucharest) (900)

Example 44 flatMap2

CONCATMAP OPERATOR

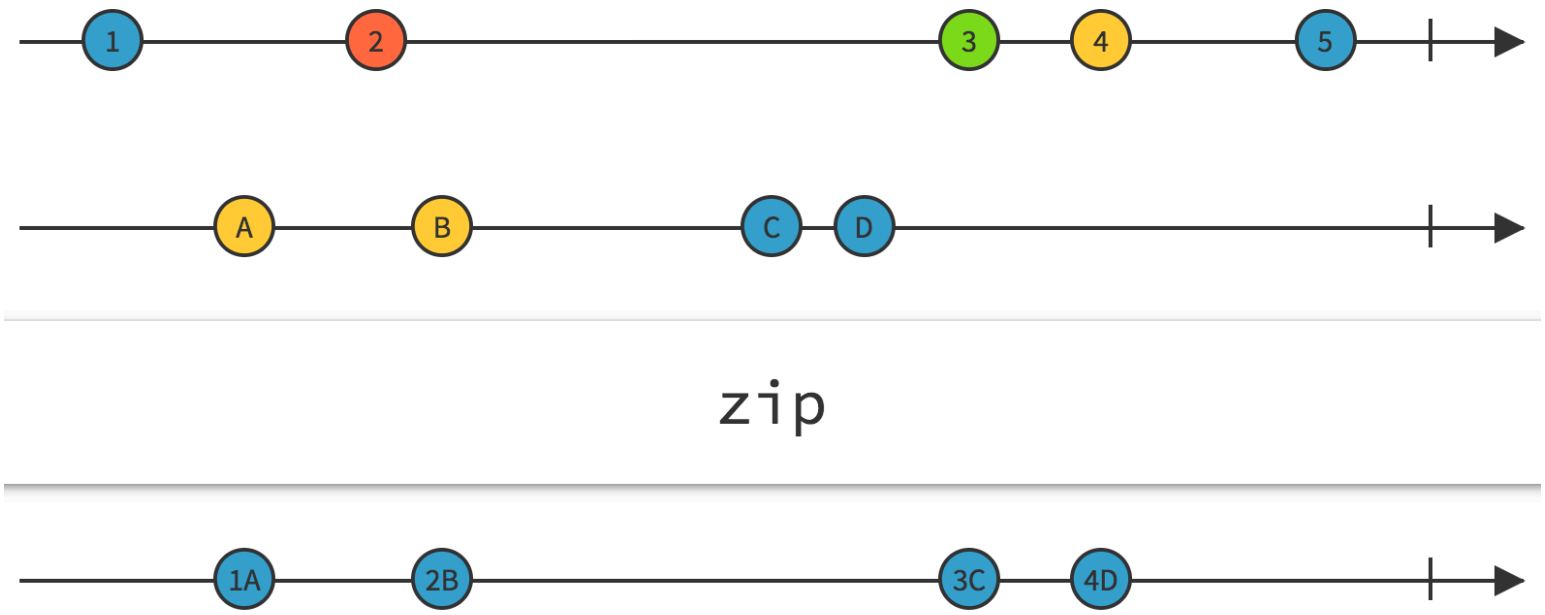
concatMap() does preserve order of the elements

```
static long delay = 1000;
public static Observable<String> remoteRequest(String s) {
   delay -= 100;
   return Observable.just("("+s+") ("+delay+")")
         .delay(delay, TimeUnit.MILLISECONDS);
}

Observable<String> locations =
      Observable.just("Bucharest", "Krakow",
            "Moscow", "Kiev", "Sofia");
locations.concatMap(s -> remoteRequest(s))
      .blockingSubscribe(System.out::println);
```

Results:
(Bucharest) (900)
(Krakow) (800)
(Moscow) (700)
(Kiev) (600)
(Sofia) (500)

Example 44 concatMap

ZIP OPERATOR



Real-life use case:

to commit a payment, we need to get information from 2 sources: request from the point of sale and approval from the client. We are using zip() to combine these 2 sources:

```
Observable<Payment> payments = Observable.zip(paymentRequests, paymentApprovals,
        (paymentRequest, paymentApproval) -> isValid(paymentApproval) &&
            paymentRequest.details.equals(paymentApproval.details) ?
                new Payment(paymentRequest) : Observable.empty());
```

ZIP OPERATOR

```
Observable<String> locations =
Observable.just("Bucharest",
"Krakow", "Moscow", "Kiev",
"Sofia");

Observable<Integer> integers =
Observable.range(1,4);
Observable.zip(locations, integers,
   (s,i) -> s + "-" + i)
.subscribe(System.out::println);
```

Result

Bucharest-1
Krakow-2
Moscow-3
Kiev-4

Example 45

ZIPWITH OPERATOR

```
Observable<String> locations =
Observable.just("Bucharest", "Krakow", "Moscow",
"Kiev", "Sofia");

locations.zipWith(
   Observable.interval(500, TimeUnit.MILLISECONDS),
         (item, interval) -> item)
.blockingSubscribe(System.out::println);
```

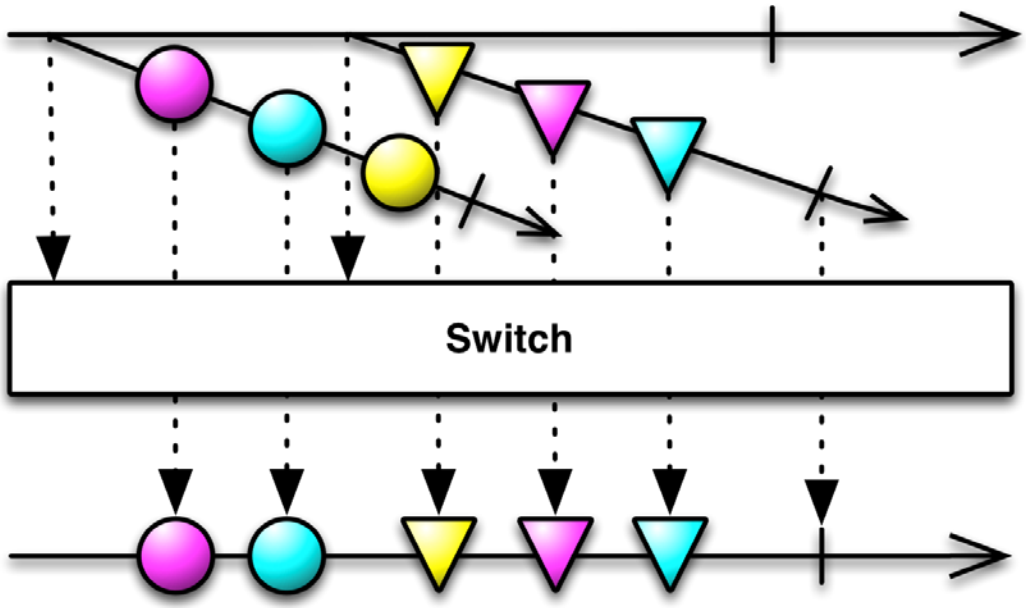
Results (0.5s delays between elements):

Bucharest
Krakow
Moscow
Kiev
Sofia

Example 45 zipWith

SWITCHMAP OPERATOR

*The **switchMap** operator is similar to **flatMap**, except that it retains the result of **only the latest observable**, discarding the previous ones.*

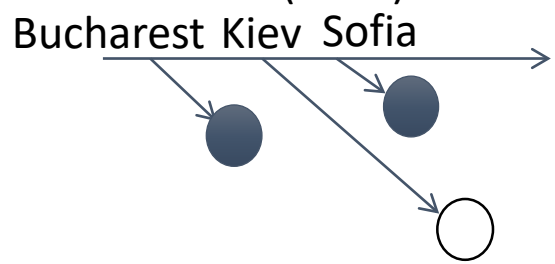



```
public static Observable<String>
  remoteRequest(String s) {
   return Observable.just("("+s+")")
         .delay(1, TimeUnit.SECONDS);
}
```

```
Observable<String> locations = Observable
      .just("Bucharest", "Krakow", "Moscow", "Kiev", "Sofia");
locations.zipWith(
   Observable.fromArray(200, 2100, 1500, 200, 100) // delays for locations
   .scan(Integer::sum)
   .delay(delay -> Observable.timer(delay, TimeUnit.MILLISECONDS)),
      (location, interval) -> location
)
.switchMap(s -> remoteRequest(s))
.blockingSubscribe(System.out::println);
```

Results:

(Bucharest)
(Krakow)
(Sofia)

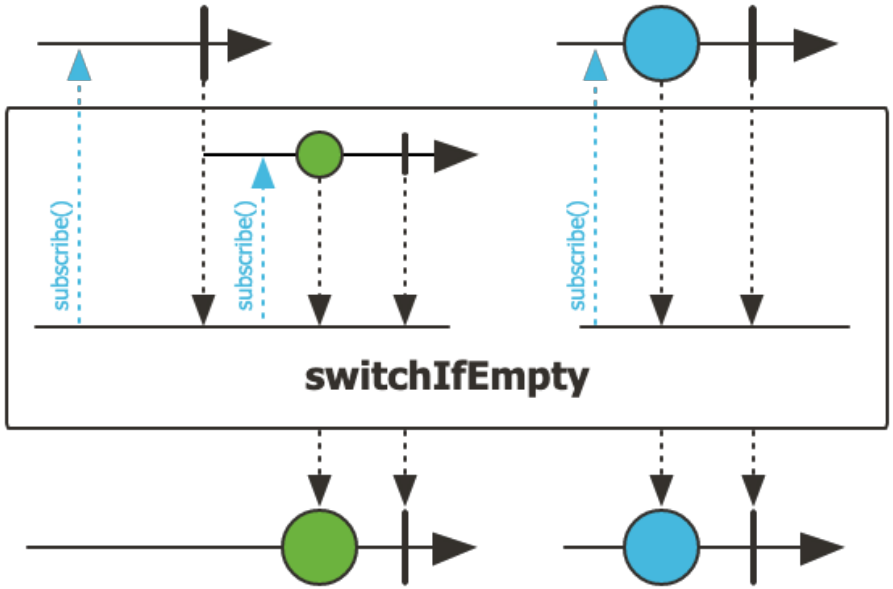


Example 45 switchMap

SWITCHMAPIFEMPTY OPERATOR

switchIfEmpty() specifies a different Observable to emit values from if the source Observable is empty.

```
// Case 1. If no info in cache, take it from network
Observable.defer(() -> fromCache())
.switchIfEmpty(fromNetwork())
.subscribe(System.out::println);

// Case 2. If no info in cache, throw error
Observable.defer(() -> fromCache())
      .switchIfEmpty(errorIfEmpty())
      .subscribe(System.out::println,
            err -> System.out.println("ERROR: " + err));
```



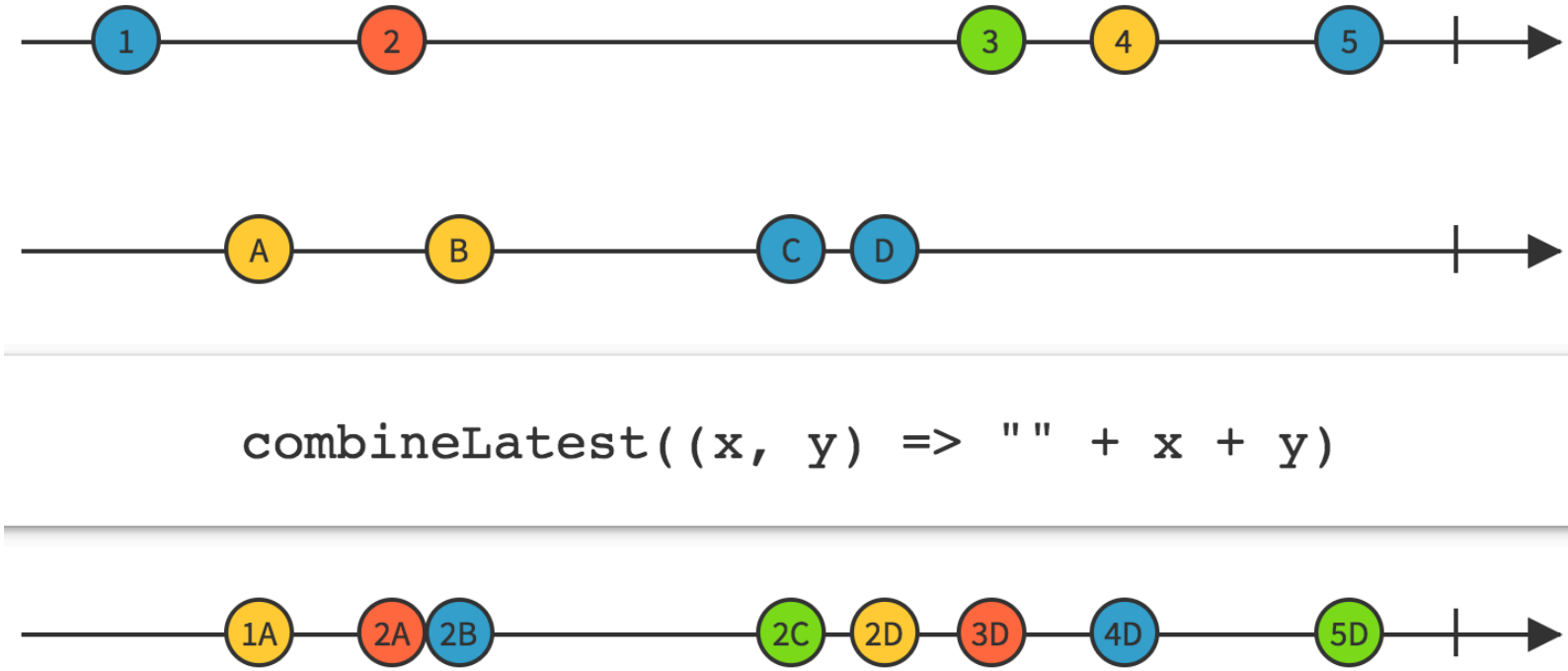
Results:

this
is
from
network

ERROR: java.lang.Exception: no data!

```
public static Observable<String> fromCache() {
   return Observable.<String>empty();
}
public static Observable<String> errorIfEmpty() {
   return Observable.error(
         new Exception("no data!"));
}
public static Observable<String> fromNetwork() {
   return Observable.just("this", "is",
         "from", "network");
}
```

COMBINELATEST OPERATOR



Real-life use case:

*we have an observable with **payments** and observable with **changes of the account info**. We need to take a **latest account details** and combine it with **every payment**.*

```
Observable<PaymentDetails> paymentDetails =
    Observable.combineLatest(payments, accountChanges,
        (payment, account) ->
            new PaymentDetails(payment.amount, account.number))
```

COMBINELATEST OPERATOR

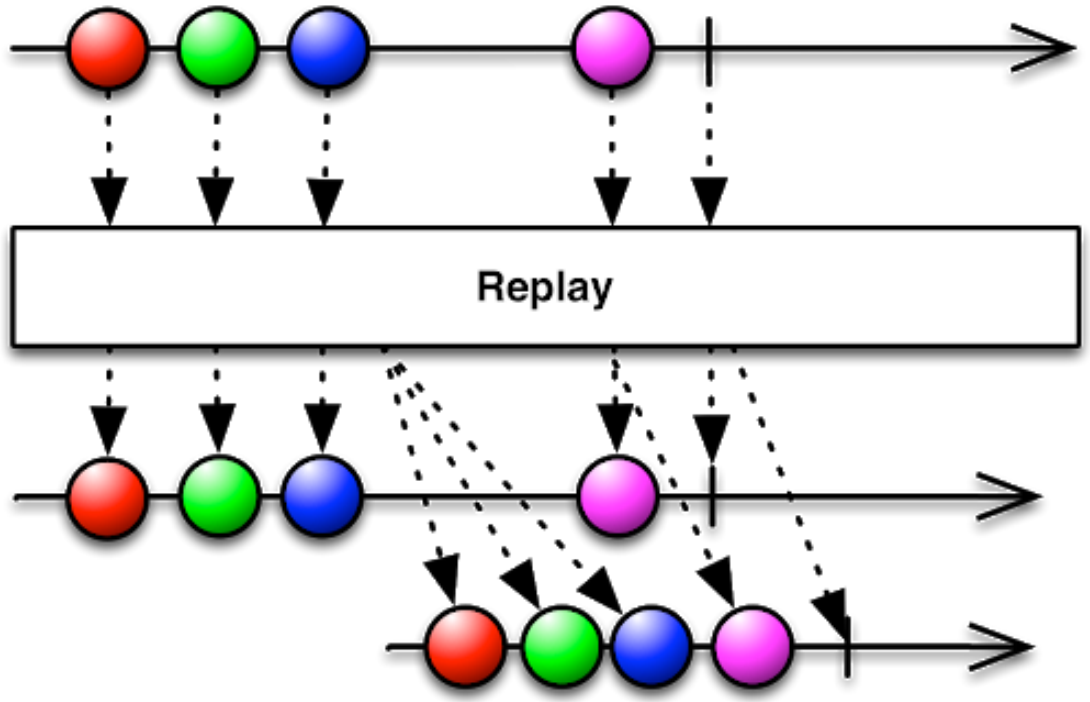
```
Observable<String> source1 =
        Observable.interval(1, TimeUnit.SECONDS)
        .map(l -> "Source1: " + (l + 1) + " s");
Observable<String> source2 =
        Observable.interval(400, TimeUnit.MILLISECONDS)
        .map(l -> "Source2: " + ((l + 1) * 400) + " ms");

Observable.combineLatest(source1, source2,
        (s1, s2) -> (s1 + "-"+ s2))
   .subscribe(System.out::println);

Thread.sleep(3000);
```

Example 46

Result:

Source1: 1 s-Source2: 800 ms
Source1: 1 s-Source2: 1200 ms
Source1: 1 s-Source2: 1600 ms
Source1: 2 s-Source2: 1600 ms
Source1: 2 s-Source2: 2000 ms
Source1: 2 s-Source2: 2400 ms
Source1: 2 s-Source2: 2800 ms
Source1: 3 s-Source2: 2800 ms

REPLAY OPERATOR



Real-life use case:

*if the client is subscribing for the payment alerts, we want him/her to receive payments for the **last 10 minutes**.*

```
Observable<Payment> paymentAlerts =
    payments.replay(10, TimeUnit.MINUTES ).autoConnect();
paymentAlerts.subscribe(payment -> sendAlertToUser(payment));
```

REPLAY OPERATOR

```
Observable<Long> seconds =
        Observable.interval(1,
                TimeUnit.SECONDS);

seconds.subscribe(i ->
System.out.println("Received 1: " + i));
Thread.sleep(3000);

seconds.subscribe(i ->
System.out.println("Received 2: " + i));

Thread.sleep(3000);
```

Example 47

Result

```
Received 1: 0
Received 1: 1
Received 1: 2
Received 1: 3
Received 2: 0
Received 2: 1
Received 1: 4
Received 1: 5
```

REPLAY OPERATOR

```
Observable<Long> seconds =
Observable.interval(1, TimeUnit.SECONDS)
        .replay()
        .autoConnect();

seconds.subscribe(i ->
System.out.println("Received 1: " + i));
Thread.sleep(3000);

seconds.subscribe(i ->
System.out.println("Received 2: " + i));

Thread.sleep(3000);
```

Example 48

Example 48 replay

Result

Received 1: 0
Received 1: 1
Received 1: 2
Received 2: 0
Received 2: 1
Received 2: 2
Received 1: 3
Received 2: 3
Received 1: 4
Received 2: 4
Received 1: 5
Received 2: 5

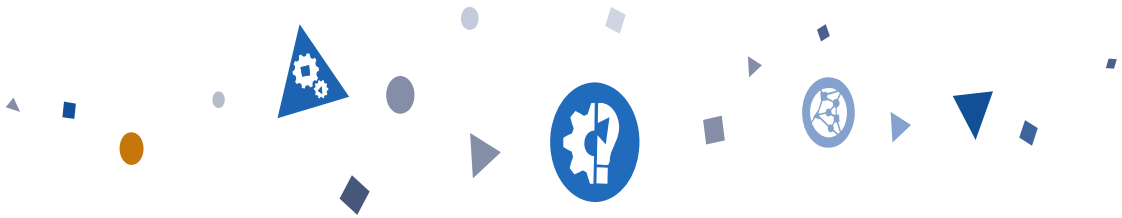
OPERATORS ARE GENERALLY NOT THREAD SAFE

- Most of the operators are not thread safe due to the following reasons:
  - Adding thread safety will increase the complexity during the implementation of the operators
  - The operator becomes hard to maintain with thread safety
  - When operators combine, it will be hard to maintain the flow and how they can be combined

THREAD SAFE OPERATORS

- Thread safe operators are basically the ones operating on multiple `Observables` to combine them: `merge()`, `zip()`, `combineLatest()`.

- When the observable emits the data concurrently, it is violating the `Observable` contract. In this case, we can use the `serialize()` operator, to serialize the emission from an `Observable`.



SUBJECTS

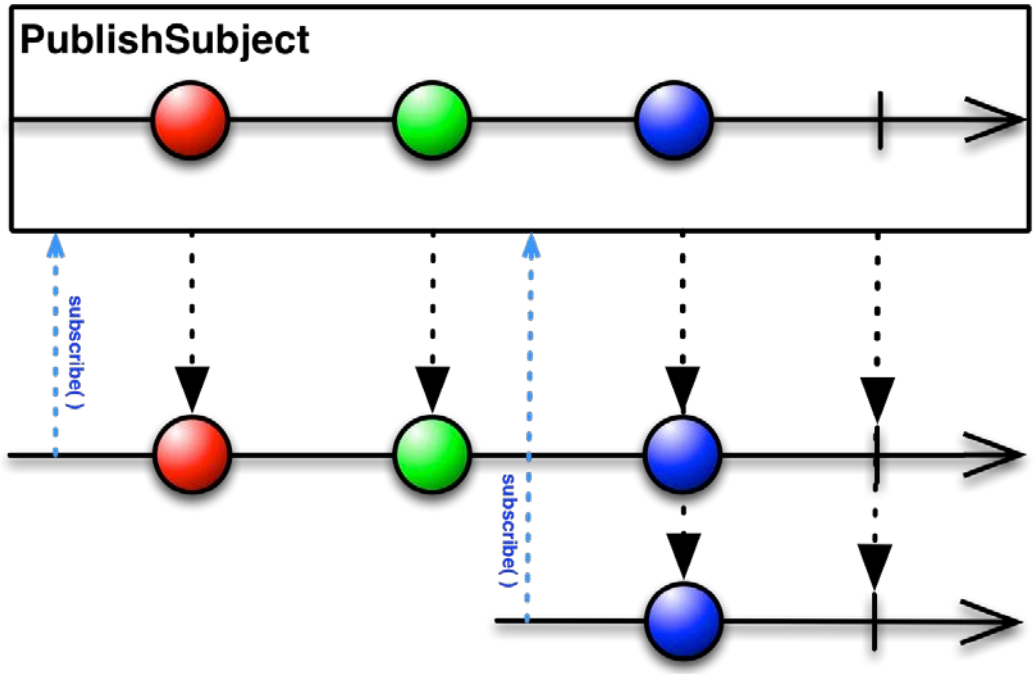
SUBJECT: OBSERVER AND OBSERVABLE

Observable: Assume that a professor is an observable. The professor teaches about some topic.
Observer: Assume that a student is an observer. The student observes the topic being taught by the professor.
Subject: Observable and Observer hybrid, as it both receives and emits events.

PUBLISH SUBJECT

PublishSubject sends data to all its subscribers

```
PublishSubject<Integer> source =
    PublishSubject.create();

source.subscribe(v ->
System.out.println("Observer 1: "+v));

source.onNext(1);
source.onNext(2);
source.onNext(3);

source.subscribe(v ->
System.out.println("Observer 2: "+v));

source.onNext(4);
source.onComplete();
```



Result:

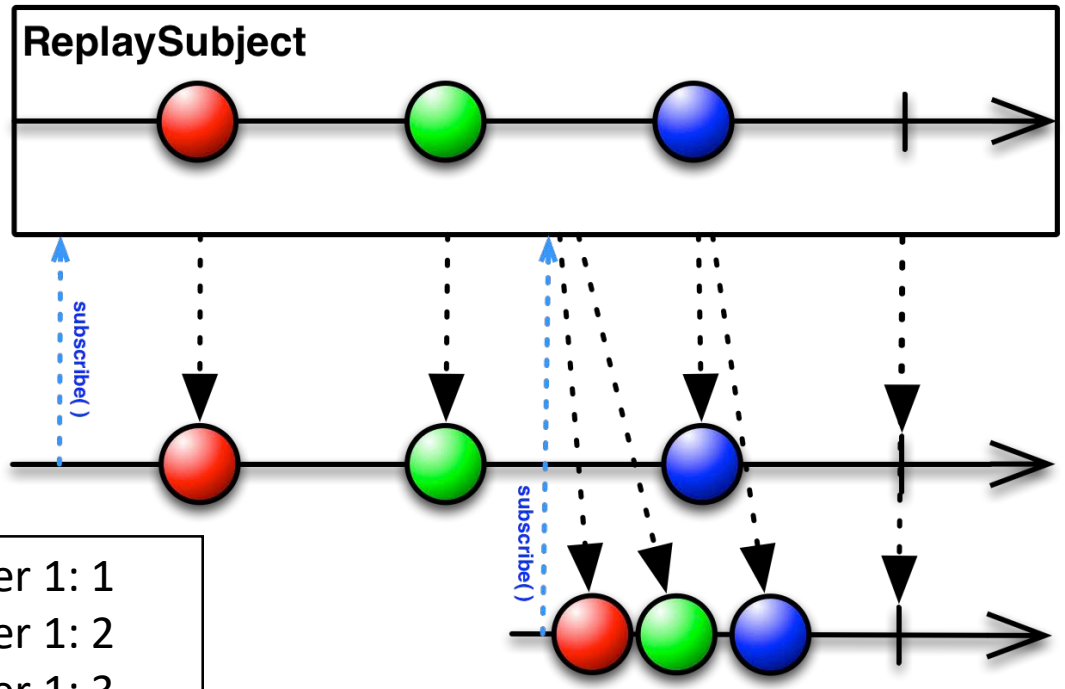
```
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 1: 4
Observer 2: 4
```

Example:
If a student entered late into the classroom, he just wants to listen from that point of time when he entered the classroom.

Example PublishSubject

REPLAY SUBJECT

ReplaySubject cache all data and sends it to every subscriber, regardless of when the observer subscribes




```
ReplaySubject<Integer> source =
ReplaySubject.create();

source.subscribe(v ->
  System.out.println(
    "Observer 1: "+v));

source.onNext(1);
source.onNext(2);
source.onNext(3);

source.subscribe(v ->
System.out.println("Observer 2: "+v));

source.onNext(4);
source.onComplete();
```

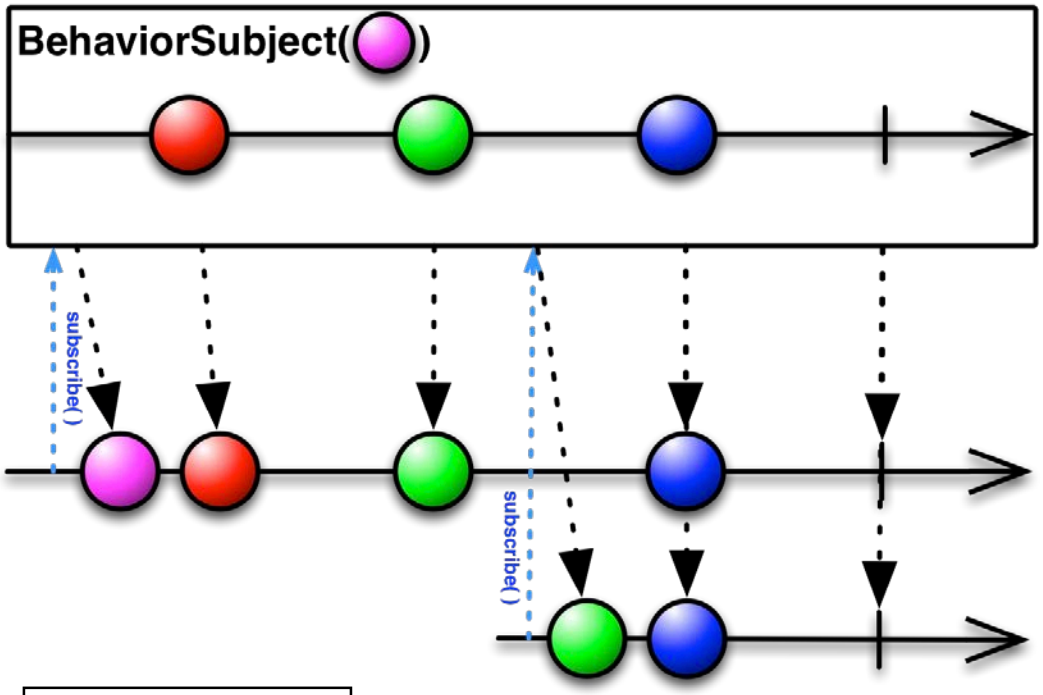
```
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 1
Observer 2: 2
Observer 2: 3
Observer 1: 4
Observer 2: 4
```

Example:
If a student entered late into the classroom, he wants to listen from the beginning.

Example ReplaySubject

BEHAVIOR SUBJECT

BehaviourSubject keeps only the last value.




```
BehaviorSubject<Integer> source =
BehaviorSubject.create();

source.subscribe(v ->
System.out.println("Observer 1: "+v));

source.onNext(1);
source.onNext(2);
source.onNext(3);

source.subscribe(v ->
System.out.println("Observer 2: "+v));

source.onNext(4);
source.onComplete();
```

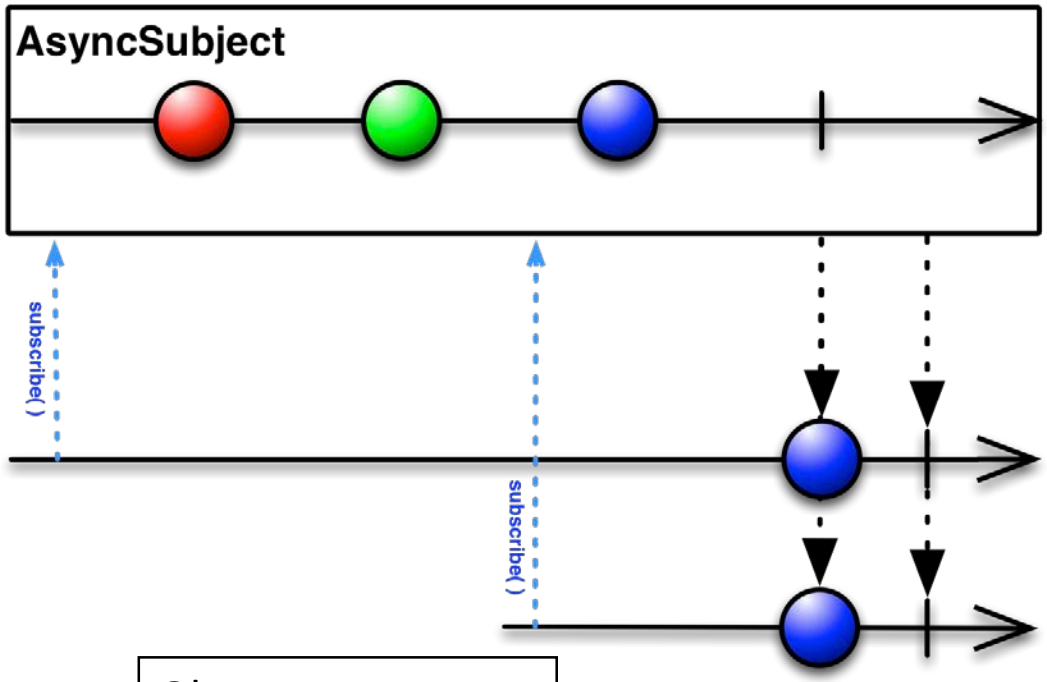
```
Observer 1: 1
Observer 1: 2
Observer 1: 3
Observer 2: 3
Observer 1: 4
Observer 2: 4
```

Example:
If a student entered late into the classroom, he wants to listen the most recent things (not from the beginning) being taught by the professor so that he gets the idea of the context.

Example BehaviorSubject

ASYNC SUBJECT

AsyncSubject sends data data only after completion.




```
AsyncSubject<Integer> source =
AsyncSubject.create();

source.subscribe(v ->
System.out.println("Observer 1: "+v));

source.onNext(1);
source.onNext(2);
source.onNext(3);

source.subscribe(v ->
System.out.println("Observer 2: "+v));

source.onNext(4);
source.onComplete();
```

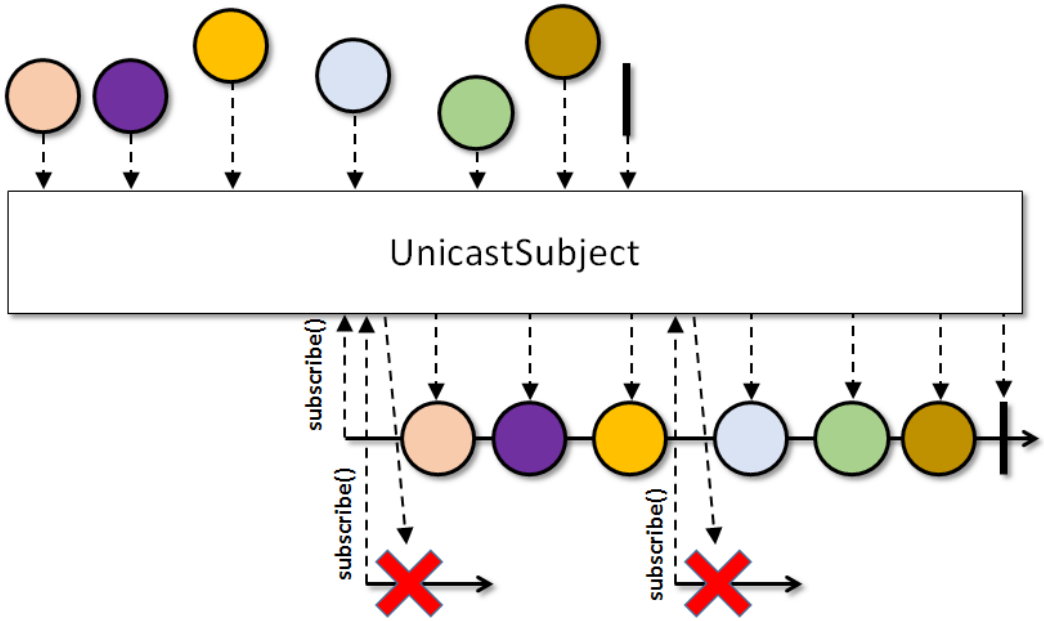
```
Observer 1: 4
Observer 2: 4
```

Example:
If a student entered at any point of time into the classroom, and he wants to listen only about the last thing (and only the last thing).

Example AsyncSubject

UNICAST SUBJECT

Unicast subject **buffers events until a single Subscriber arrives** and replays them to it and potentially switches to direct delivery once the Subscriber caught up. UnicastSubject can have **only one Subscriber**.

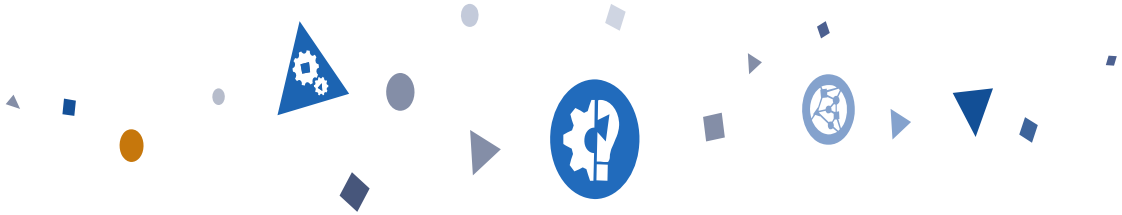



```
UnicastSubject<Integer> source =
        UnicastSubject.create();
source.subscribe(v -> System.out.println("Observer 1: "+v));
source.onNext(1);
source.onNext(2);
source.subscribe(
  v -> System.out.println("Observer 2: "+v),
       System.out::println);
source.onNext(3);
source.onNext(4);
source.onComplete();
```

Results:

```
Observer 1: 1
Observer 1: 2
Observer 1: 3
java.lang.IllegalStateException:
Only a single observer allowed.
Observer 1: 4
```

Example UnicastSubject



RX JAVA CONCURRENCY

SERIAL EXECUTION

```
public static void main(String[] args)
throws InterruptedException {
    Observable.just("Bucharest", "Krakow",
              "Moscow", "Kiev", "Sofia")
            .map(s -> longTask(s))
            .subscribe(System.out::println);
    Observable.range(1, 5)
            .map(i -> longTask(i))
            .subscribe(System.out::println);
}

public static <T> T longTask(T value)
throws InterruptedException {
    Thread.sleep(new Random().nextInt(2000));
    return value;
}
```

Example 49

Results:

Bucharest
Krakow
Moscow
Kiev
Sofia
1
2
3
4
5

CONCURRENT EXECUTION

```
Observable.just("Bucharest", "Krakow",
       "Moscow", "Kiev", "Sofia")
.subscribeOn(Schedulers.computation())
            .map(s -> longTask((s)))
.subscribe(System.out::println);

Observable.range(1, 5)
.subscribeOn(Schedulers.computation())
            .map(i -> longTask((i)))
.subscribe(System.out::println);
    Thread.sleep(20000);
```

Example 50

Results:

Bucharest
1
2
3
Krakow
4
Moscow
Kiev
5
Sofia

COMBINE OBSERVABLES ON DIFFERENT THREADS

```
Observable<String> locations =
        Observable.just("Bucharest", "Krakow",
                "Moscow", "Kiev", "Sofia")
.subscribeOn(Schedulers.computation())
                .map(s -> longTask((s)));

Observable<Integer> integers =
        Observable.range(1, 5)
.subscribeOn(Schedulers.computation())
                .map(i -> longTask((i)));

Observable.zip(locations, integers, (s,i) -> s + "-" + i)
        .subscribe(System.out::println);
Thread.sleep(10000);
```

Results:

Bucharest-1
Krakow-2
Moscow-3
Kiev-4
Sofia-5

Example 51

BLOCKING SUBSCRIBE

```
Observable.just("Bucharest", "Krakow",
        "Moscow", "Kiev", "Sofia")
.subscribeOn(Schedulers.computation())
.map(s -> longTask((s)))
.blockingSubscribe(System.out::println);
```

Example 52

Results:

Bucharest
Krakow
Moscow
Kiev
Sofia

SCHEDULERS

- RxJava implements its own concurrency abstraction called **Scheduler**.
- It defines methods and rules that an actual concurrency provider such as an `ExecutorService` must obey.
- Scheduler implementations can be found in the **`Schedulers`** static factory class.

SCHEDULERS

- For a given `Observer`, a `Scheduler` will provide a thread from its pool that will push the emissions.
- When `onComplete()` is called, the operation will be disposed and the thread will be given back to the pool, it may reused by another `Observer`.

SCHEDULERS

Scheduler name	Notes
`computation()`	Maintains a fixed number of threads based on the processor count, making it appropriate for computational tasks.
io()	Shared Scheduler instance intended for IO-bound work (databases, web requests, disk storage etc.)
newThread()	Creates a new Thread for each unit of work
single()	Single-thread-backed Scheduler instance for work requiring strongly-sequential execution
from(Executor executor)	Wraps an Executor into a new Scheduler instance and delegates schedule() calls to it.
trampoline()	Scheduler instance whose Scheduler.Worker instances queue work and execute them in a FIFO manner on one of the participating threads.

SCHEDULERS.NEWTHREAD()

```
Observable.interval(1, TimeUnit.SECONDS,
        Schedulers.newThread())
        .subscribe(i -> System.out.println("Received " + i
                + " on thread " +

Thread.currentThread().getName()));
Thread.sleep(5000);
```

Example 53

Result

Received 0 on thread RxNewThreadScheduler-1
Received 1 on thread RxNewThreadScheduler-1
Received 2 on thread RxNewThreadScheduler-1
Received 3 on thread RxNewThreadScheduler-1
Received 4 on thread RxNewThreadScheduler-1

SCHEDULERS.SINGLE()

```
Observable.interval(1, TimeUnit.SECONDS,
        Schedulers.single())
        .subscribe(i -> System.out.println(
             "Received " + i + " on thread "+
             Thread.currentThread().getName()));

Thread.sleep(5000);
```

Example 54

Result

Received 0 on thread RxSingleScheduler-1
Received 1 on thread RxSingleScheduler-1
Received 2 on thread RxSingleScheduler-1
Received 3 on thread RxSingleScheduler-1
Received 4 on thread RxSingleScheduler-1

SCHEDULERS

- You can build a `Scheduler` from a standard Java `ExecutorService` to have **more control over your thread** management policies.
- We can create a new fixed `ExecutorService` specified with a number of threads.
- Then, wrap it inside a `Scheduler` implementation by calling `Schedulers.from()`.

EXECUTORSERVICE SCHEDULER

```
ExecutorService executor =
        Executors.newFixedThreadPool(3);
Scheduler scheduler = Schedulers.from(executor);

Observable.just("Bucharest", "Krakow", "Moscow",
"Kiev", "Sofia")
        .subscribeOn(scheduler)
        .doFinally(executor::shutdown)
        .subscribe(System.out::println);
```

Example 55

Result:

Bucharest
Krakow
Moscow
Kiev
Sofia

MULTIPLE OBSERVERS TO THE SAME OBSERVABLE

```
Observable<String> locations = Observable.range(1, 5)
        .subscribeOn(Schedulers.computation());

locations.subscribe(i->
        System.out.println(i + " on thread " +
        Thread.currentThread().getName()));
locations.subscribe(i->
        System.out.println(i + " on thread " +
        Thread.currentThread().getName()));

Thread.sleep(1000);
```

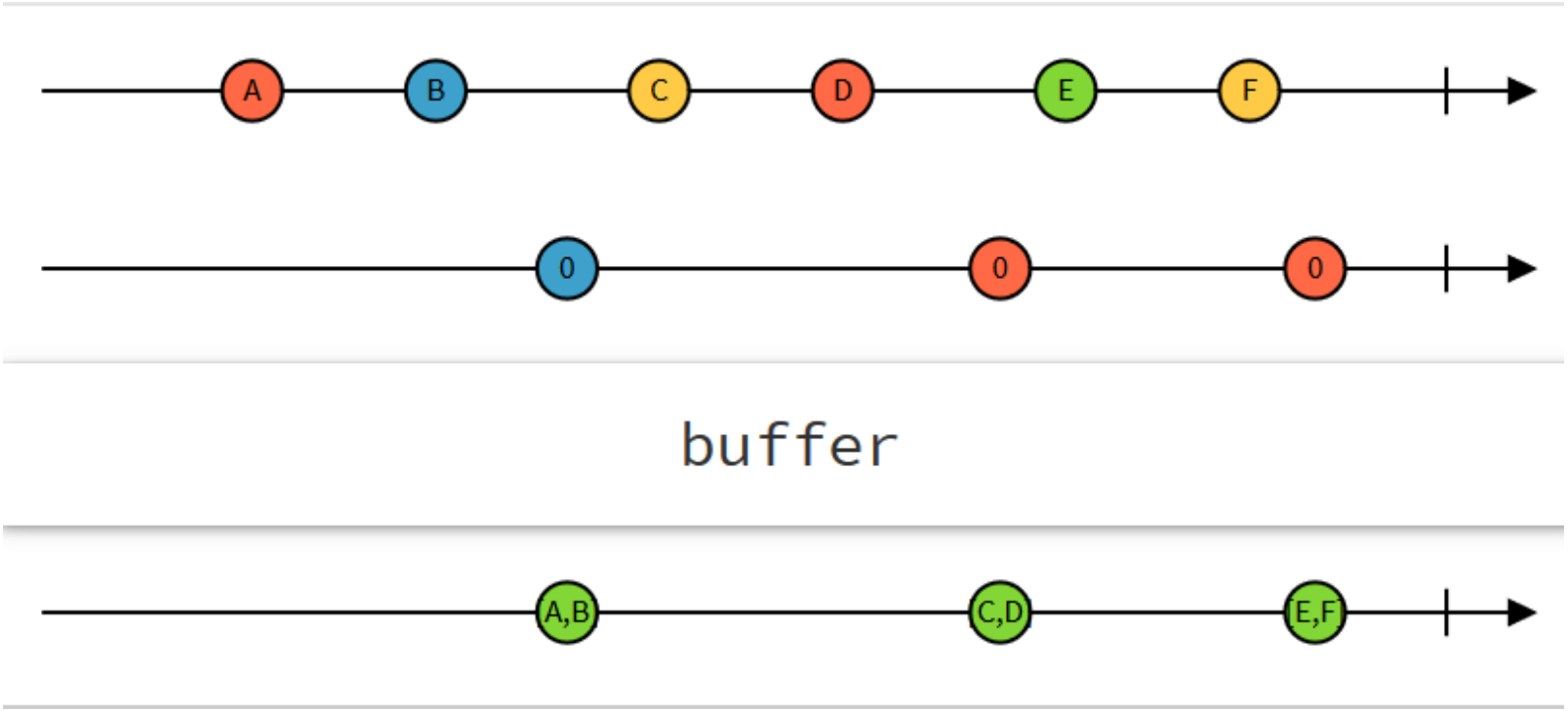
Example 56

Result:

1 on thread RxComputationThreadPool-2
1 on thread RxComputationThreadPool-1
2 on thread RxComputationThreadPool-2
2 on thread RxComputationThreadPool-1
…

BUFFERING AND WINDOWING

BUFFER OPERATOR



BUFFER OPERATOR

```
Observable.range(1,100)
          .buffer(16)
          .subscribe(System.out::println);
```

Example 57

Result

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
[33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48]
[49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]
[65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
[81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96]
[97, 98, 99, 100]

TIME-BASED BUFFERING

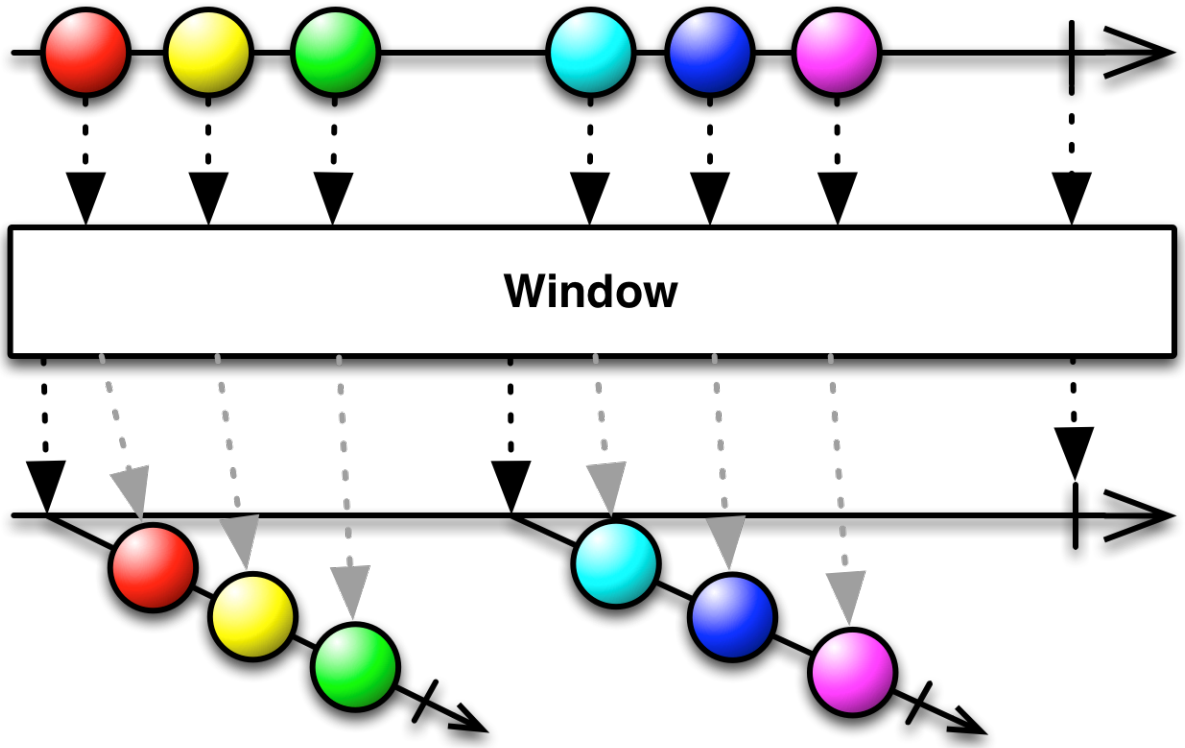
```
Observable.interval(300,
TimeUnit.MILLISECONDS)
        .map(i -> (i + 1) * 300)
        .buffer(1,
            TimeUnit.SECONDS)
.subscribe(System.out::println);
Thread.sleep(5000);
```

Result

[300, 600, 900]
[1200, 1500, 1800]
[2100, 2400, 2700]
[3000, 3300, 3600, 3900]
[4200, 4500, 4800]

Example 58

WINDOW OPERATOR



WINDOW OPERATOR

```
Observable.range(1, 50)
    .window(16)
    .flatMapSingle(obs ->
         obs.reduce("", (total, next) -> total
         + (total.equals("") ? "" : "|") + next))
         .subscribe(System.out::println);
```

Example 59

Result

1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16
17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32
33|34|35|36|37|38|39|40|41|42|43|44|45|46|47|48
49|50

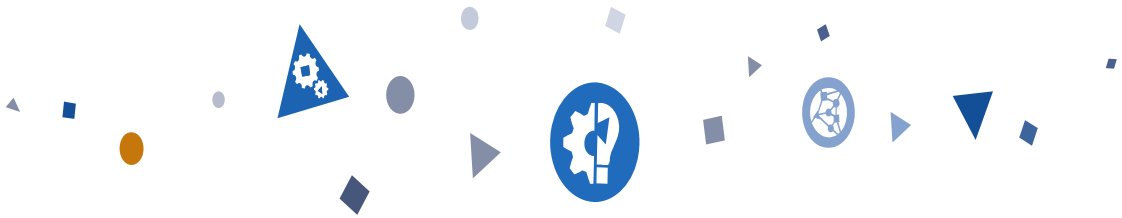
TIME-BASED WINDOWING

```
Observable.interval(300, TimeUnit.MILLISECONDS)
 .map(i -> (i + 1) * 300).window(1, TimeUnit.SECONDS)
 .flatMapSingle(obs -> obs.reduce("",
       (total, next) -> total +
       (total.equals("") ? "" : "|") + next))
 .subscribe(System.out::println);
Thread.sleep(5000);
```

Example 60

Result

300|600|900
1200|1500|1800
2100|2400|2700
3000|3300|3600|3900
4200|4500|4800

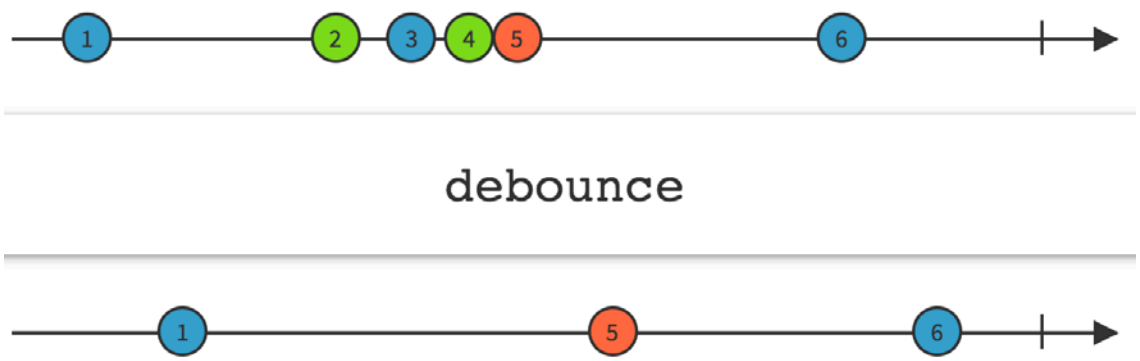


DEBOUNCE, THROTTLE AND SAMPLE

DEBOUNCE OPERATOR

The Debounce technique allow us to "group" multiple sequential calls in a single one.

Debouncing enforces that a function not be called again until a certain amount of time has passed without it being called. As in "***execute this function only if 100 milliseconds have passed without it being called***."




```
int[] delays = { 200, 1100, 1000, 200, 1400, 700 };
Observable.range(0, delays.length)
        .map(i -> delays[i])
        .scan(Integer::sum)
        .flatMap(delay -> Observable.just(delay)
            .delay(delay, TimeUnit.MILLISECONDS)
            .doOnNext(System.out::println))
        .debounce(1, TimeUnit.SECONDS)
        .subscribe(p -> System.out.println("debounced "+p));
Thread.sleep(5000);
```

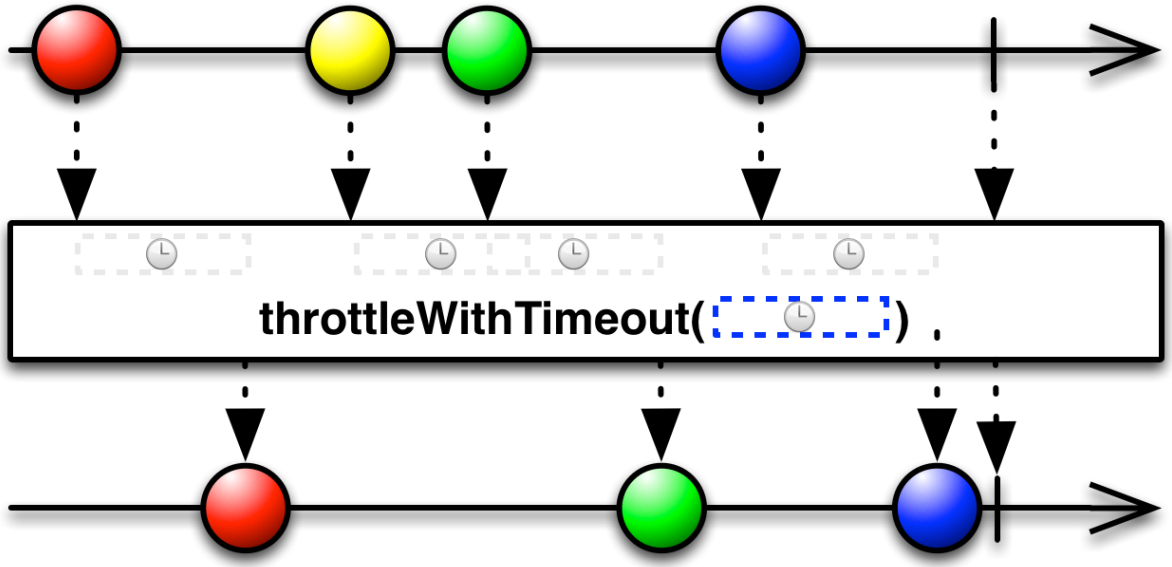
Results:

```
200
debounced 200
1300
debounced 1300
2300
2500
debounced 2500
3900
4600
debounced 4600
```

Example 60 debounce

THROTTLE OPERATOR

Throttle timeout - the length of the window of time that must pass after the emission of an item.




```
int[] delays = { 200, 1100, 1000,  200, 1400, 700 };
Observable.range(0, delays.length)
        .map(i -> delays[i])
        .scan(Integer::sum)
        .flatMap(delay -> Observable.just(delay)
           .delay(delay, TimeUnit.MILLISECONDS)
           .doOnNext(System.out::println))
        .throttleWithTimeout(1, TimeUnit.SECONDS)
        .subscribe(p -> System.out.println("throttle "+p));
Thread.sleep(5000);
```

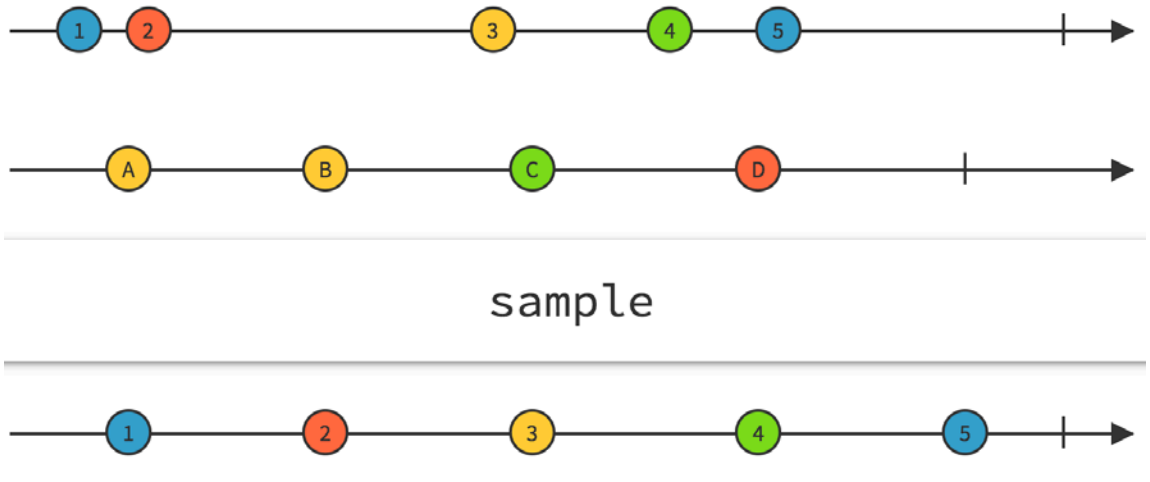
```
200
throttle 200
1300
2300
throttle 1300
2500
throttle 2500
3900
4600
throttle 4600
```

Example 60 throttle

SAMPLE OPERATOR

Sampling enforces a maximum number of times a function can be called over time. As in "***execute this function at most once every 100 milliseconds***."

Alias of this operator is **throttleLast**(interval, TimeUnit).




```
int[] delays = { 200, 1100, 1000, 200, 1400, 700 };
Observable.range(0, delays.length)
        .map(i -> delays[i])
        .scan(Integer::sum)
        .flatMap(delay -> Observable.just(delay)
            .delay(delay, TimeUnit.MILLISECONDS)
            .doOnNext(System.out::println))
        .sample(1, TimeUnit.SECONDS)
        .subscribe(p -> System.out.println("sample "+p));
Thread.sleep(5000);
```

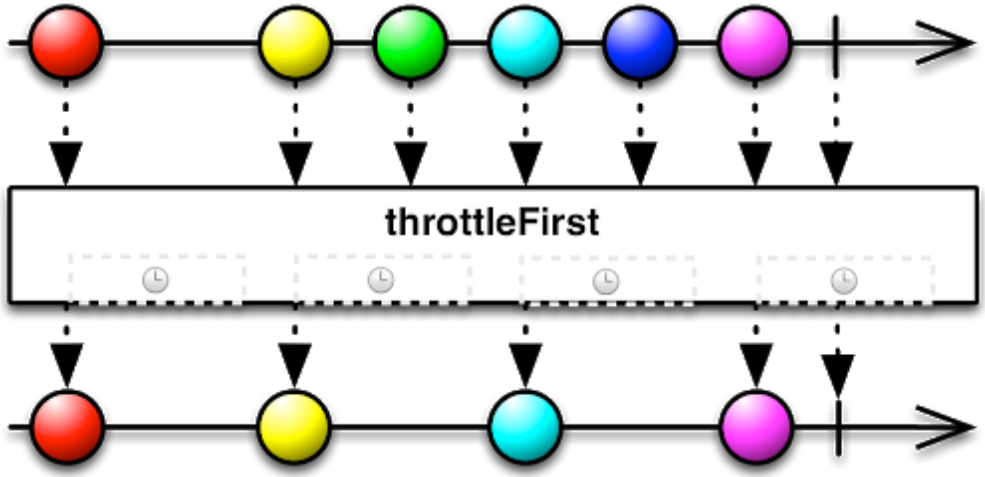
Results:

```
200
sample 200
1300
sample 1300
2300
2500
sample 2500
3900
sample 3900
4600
```

Example 60 sample

THROTTLEFIRST OPERATOR

Throttle timeout - the length of the window of time that must pass after the emission of an item. Throttle first emits the first value in interval.




```
int[] delays = { 200, 1100, 1000,  200, 1400, 700 };
Observable.range(0, delays.length)
        .map(i -> delays[i])
        .scan(Integer::sum)
        .flatMap(delay -> Observable.just(delay)
           .delay(delay, TimeUnit.MILLISECONDS)
           .doOnNext(System.out::println))
        .throttleFirst(1, TimeUnit.SECONDS)
        .subscribe(p -> System.out.println("throttle "+p));
Thread.sleep(5000);
```

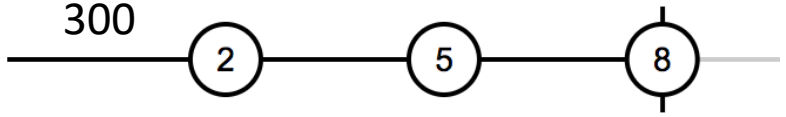
```
200
throttle 200
1300
throttle 1300
2300
2500
throttle 2500
3900
throttle 3900
4600
```

Example 60 throttleFirst

SAMPLE VS DEBOUNCE VS THROTTLE

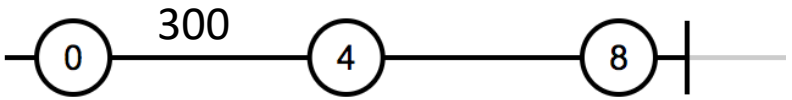
sampleTime - Emits the most recently emitted value from the source Observable within periodic time intervals.

Observable.interval(300).take(10).sampleTime(1000)



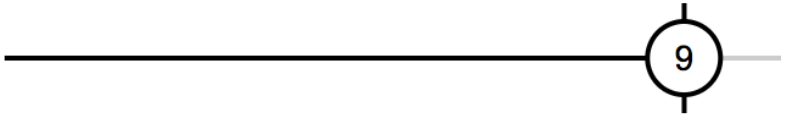
throttleTime - Emits a value from the source Observable, then ignores subsequent source values for duration milliseconds, then repeats this process.

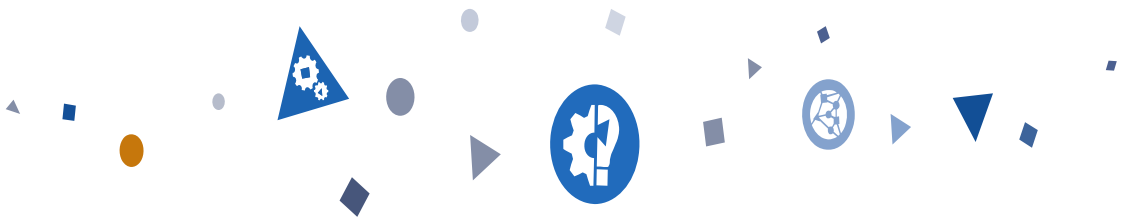
Observable.interval(300).take(10).throttleTime(1000)



debounceTime - Emits a value from the source Observable only after a particular time span has passed without another source emission.

Observable.interval(300).take(10).debounceTime(1000)

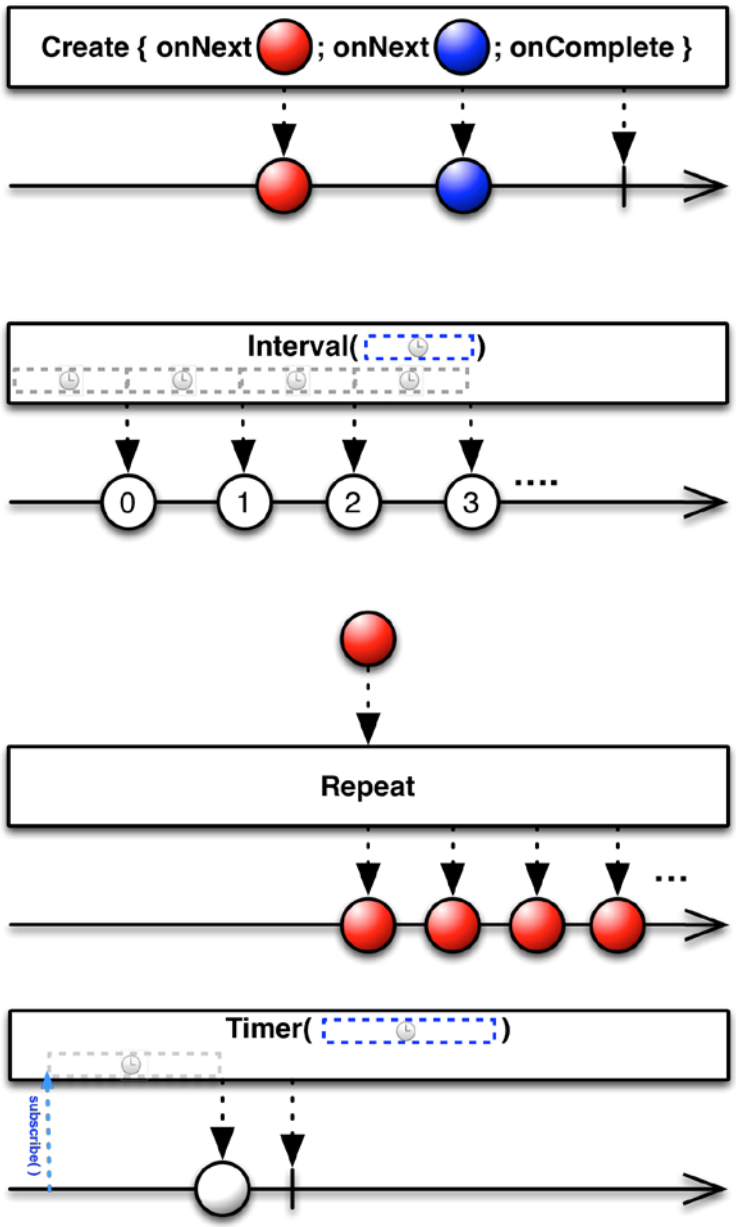




REACTIVE OPERATORS CATALOG

CREATING OBSERVABLES OPERATORS

Create — create an Observable from scratch by calling observer methods programmatically
Defer — do not create the Observable until the observer subscribes, and create a fresh Observable for each observer
Empty/Never/Throw — create Observables that have very precise and limited behavior
From — convert some other object or data structure into an Observable
Interval — creates Observable that emits a sequence of integers spaced by particular time interval
Just — convert an object or a set of objects into an Observable that emits that or those objects
Range — create an Observable that emits a range of sequential integers
Repeat — create an Observable that emits a particular item or sequence of items repeatedly
Start — create an Observable that emits the return value of a function
Timer — create an Observable that emits a single item after a given delay



TRANSFORMING OBSERVABLE

Buffer — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time

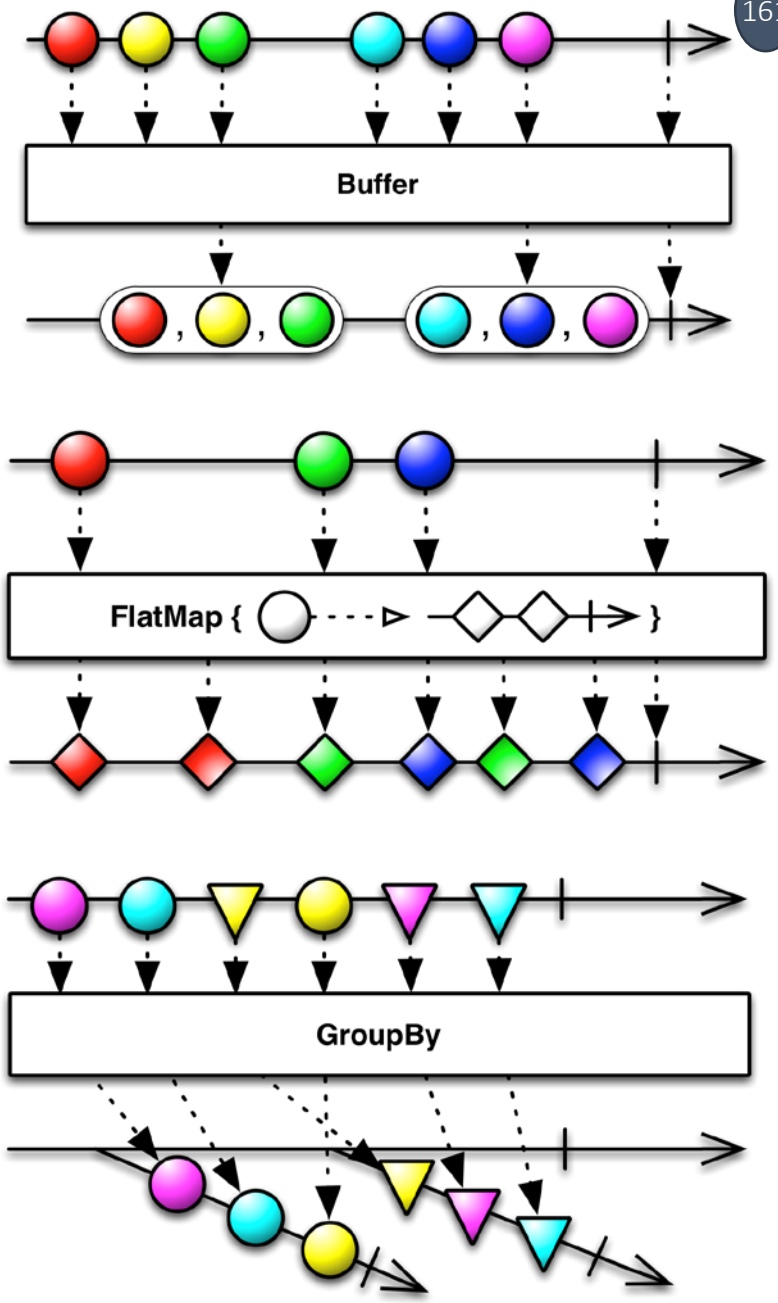
FlatMap — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable

GroupBy — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key

Map — transform the items emitted by an Observable by applying a function to each item

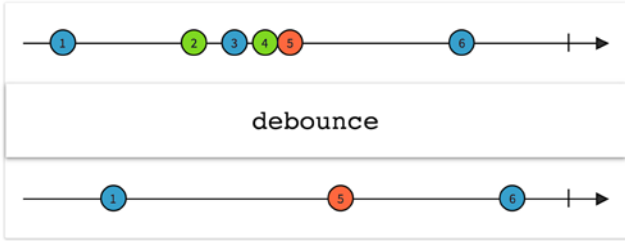
Scan — apply a function to each item emitted by an Observable, sequentially, and emit each successive value

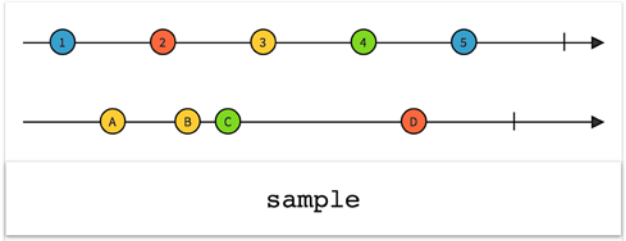
Window — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time

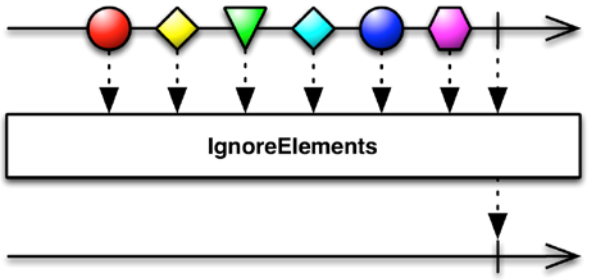


FILTERING OBSERVABLES

Debounce — only emit an item from an Observable if a particular timespan has passed without emitting another item
Distinct — suppress duplicate items emitted by an Observable
ElementAt — emit only item n emitted by an Observable
Filter — emit only those items from an Observable that pass a predicate test
First — emit only the first item, or the first item that meets a condition, from an Observable
IgnoreElements — do not emit any items from an Observable but mirror its termination notification
Last — emit only the last item emitted by an Observable
Sample — emit the most recent item emitted by an Observable within periodic time intervals
Skip — suppress the first n items emitted by an Observable
SkipLast — suppress the last n items emitted by an Observable
Take — emit only the first n items emitted by an Observable
TakeLast — emit only the last n items emitted by an Observable







COMBINING OBSERVABLES

CombineLatest — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function

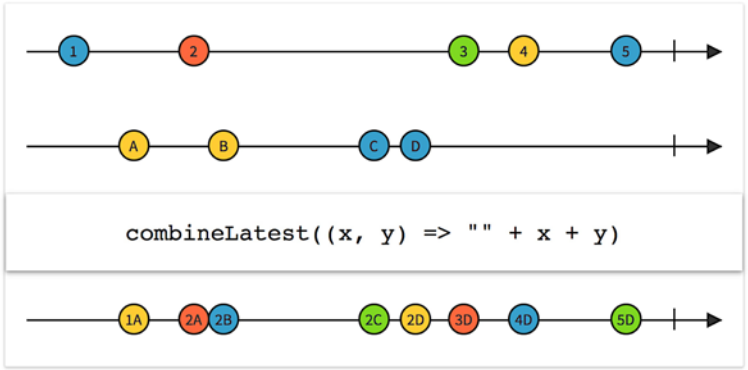
Join — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable

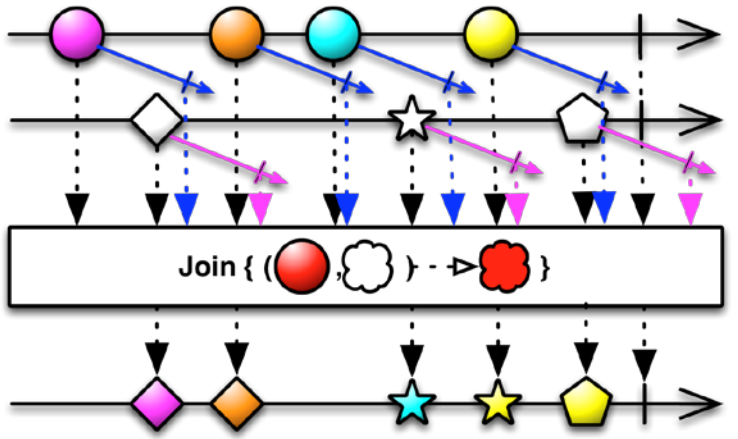
Merge — combine multiple Observables into one by merging their emissions

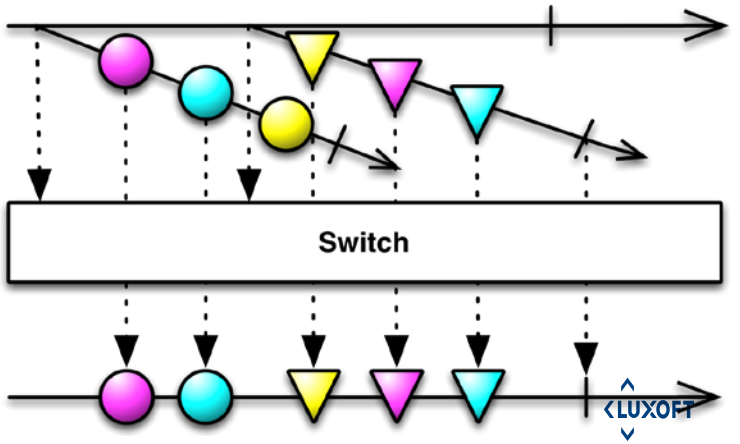
StartWith — emit a specified sequence of items before beginning to emit the items from the source Observable

Switch — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables

Zip — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function



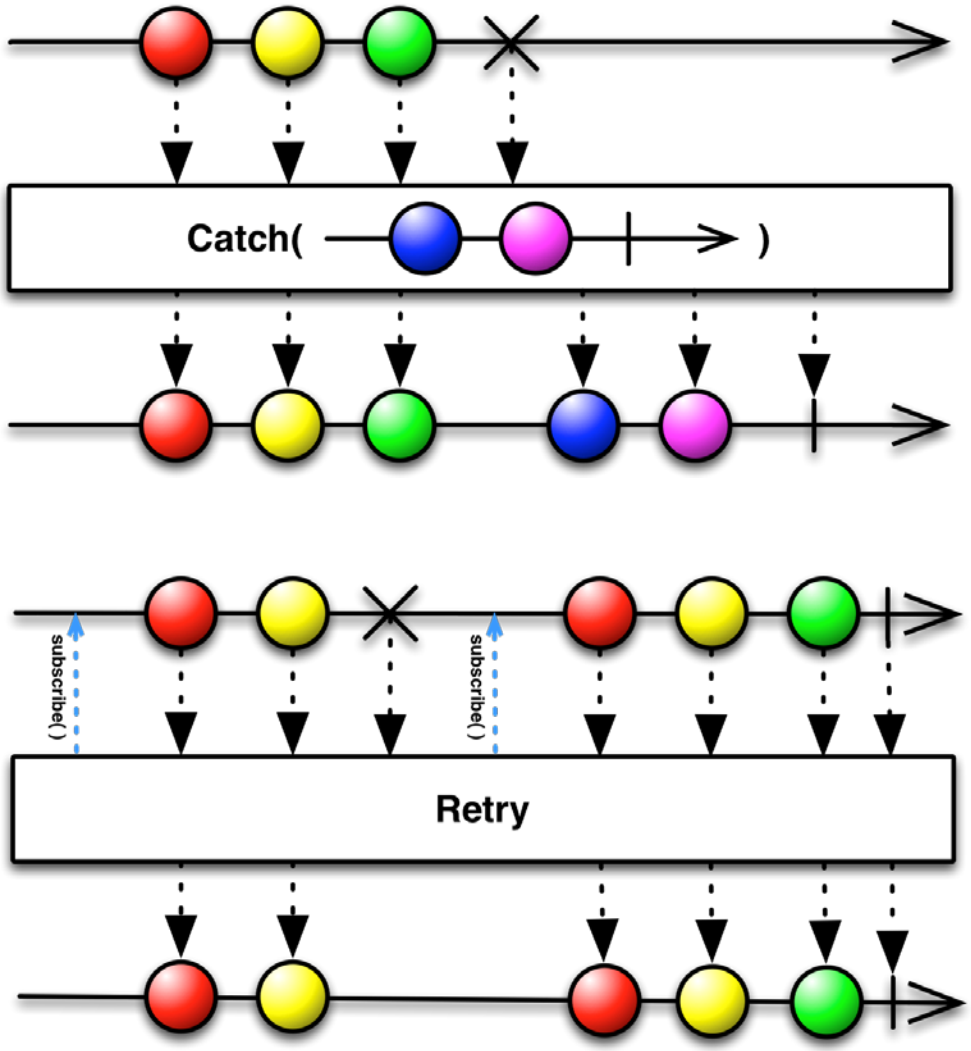




ERROR HANDLING

Catch — recover from an onError notification by continuing the sequence without error (**onErrorResumeNext**, **onErrorReturn**, **onExceptionResumeNext**)

Retry — if a source Observable sends an onError notification, resubscribe to it in the hopes that it will complete without error



CONDITIONAL AND BOOLEAN

All (every) — determine whether all items emitted by an Observable meet some criteria

Amb — given two or more source Observables, emit all of the items from only the first of these Observables to emit an item

Contains — determine whether an Observable emits a particular item or not

DefaultIfEmpty — emit items from the source Observable, or a default item if the source Observable emits nothing

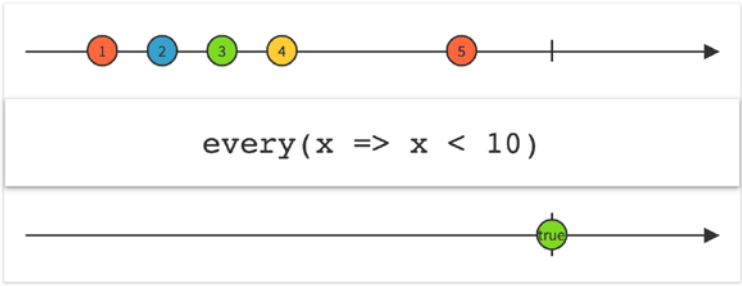
SequenceEqual — determine whether two Observables emit the same sequence of items

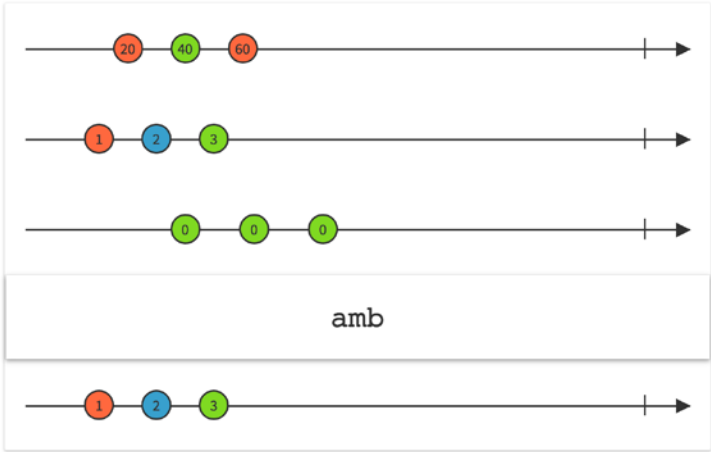
SkipUntil — discard items emitted by an Observable until a second Observable emits an item

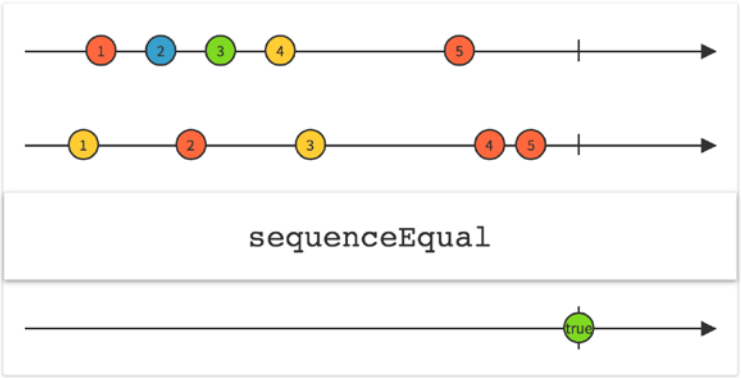
SkipWhile — discard items emitted by an Observable until a specified condition becomes false

TakeUntil — discard items emitted by an Observable after a second Observable emits an item or terminates

TakeWhile — discard items emitted by an Observable after a specified condition becomes false

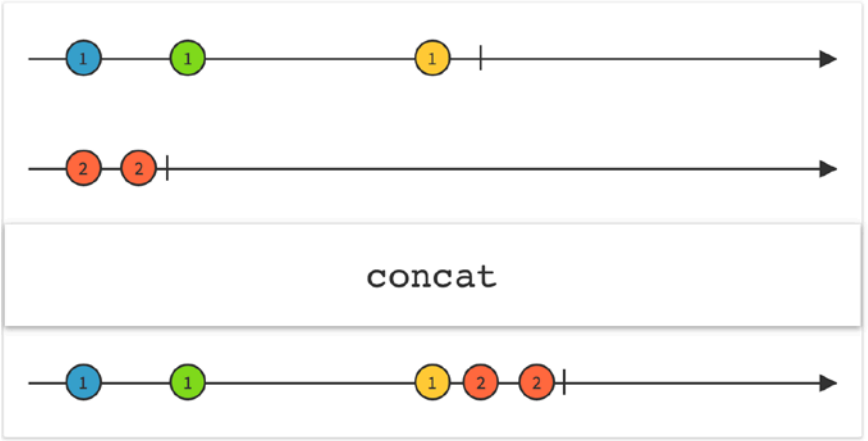


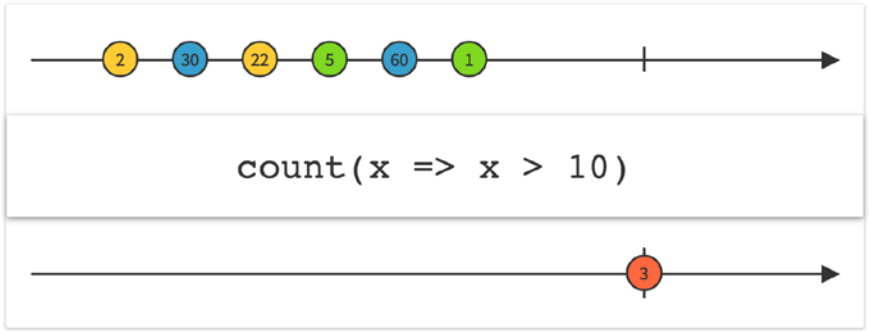


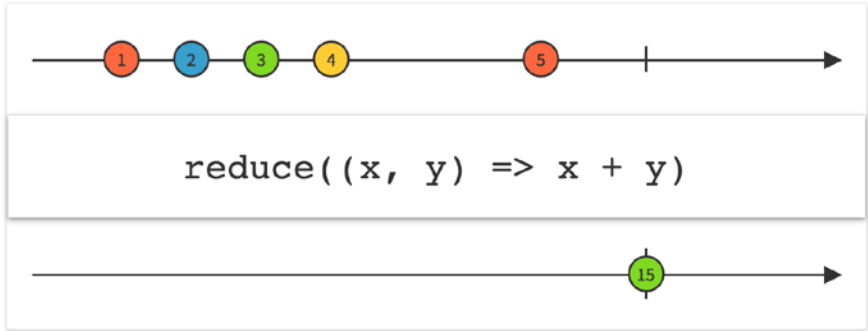


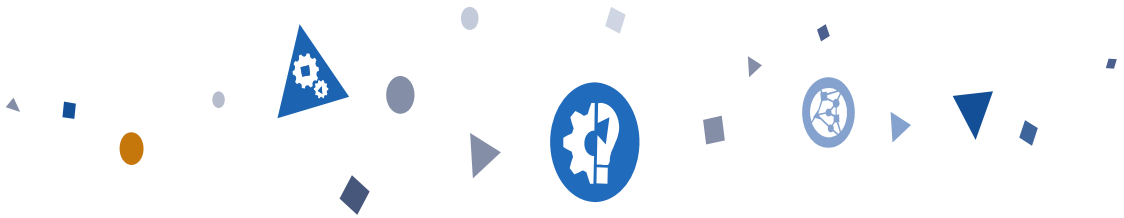
MATHEMATICAL AND AGGREGATE

Average — calculates the average of numbers emitted by an Observable and emits this average
Concat — emit the emissions from two or more Observables without interleaving them
Count — count the number of items emitted by the source Observable and emit only this value
Max — determine, and emit, the maximum-valued item emitted by an Observable
Min — determine, and emit, the minimum-valued item emitted by an Observable
Reduce — apply a function to each item emitted by an Observable, sequentially, and emit the final value
Sum — calculate the sum of numbers emitted by an Observable and emit this sum









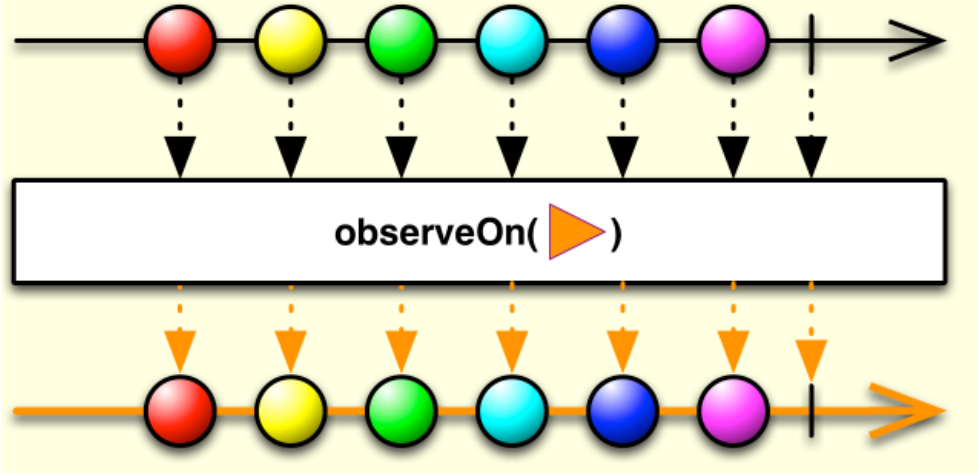
BACKPRESSURE AND FLOWABLES

BACKPRESSURE

- For most of the cases where a source is producing emissions faster than the downstream can process them, you can require the source to slow down and emit at a pace that agrees with the downstream operations.
- This is backpressure or flow control, and it can be enabled by using a `Flowable` instead of an `Observable`.

OBSERVEON

observeOn modifies a Publisher to perform its emissions and notifications on a specified Scheduler, asynchronously.



Backpressure: This operator honors backpressure from downstream and expects it from the source Publisher. Violating this expectation will lead to MissingBackpressureException.

FLOWABLE

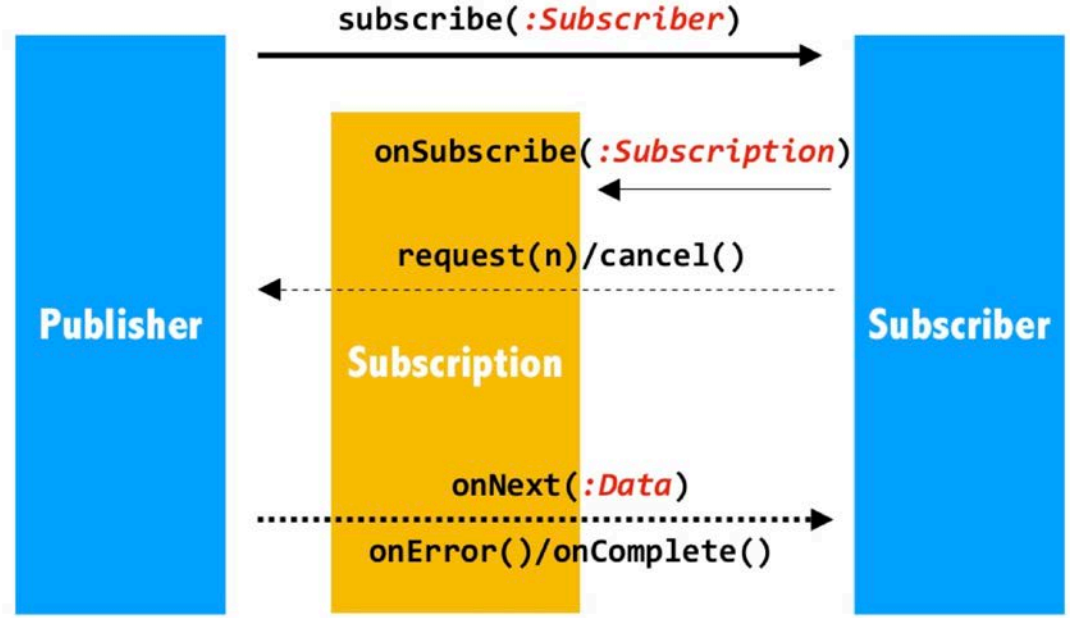
```
public static void main(String[] args)
  throws InterruptedException {
    Flowable
        .range(1, Integer.MAX_VALUE)
        .map(Counter::new)
        .observeOn(Schedulers.computation())
        .subscribe(counter -> {
             Thread.sleep(30);
             System.out.println("Received " +
                counter.count);
             });
    Thread.sleep(20000);
}
static final class Counter {
    int count;
    Counter(int count) {
        this.count = count;
         System.out.println("Creating " + count);
    }
}
```

Result:

Creating 1
…
Creating 128
Received 1
…
Received 96
Creating 129
….
Creating 224
Received 97
…
Received 192
Creating 225
…

Example 61

REACTIVE STREAMS SPEC: SUBSCRIPTION INTERFACE WITH REQUEST(N)



PUSH / PULL
Backpressure

```
public interface Publisher<T> {
   public void subscribe(
        Subscriber<? super T> s);
}

public interface Subscriber<T> {
   public void onSubscribe(Subscription s);
   public void onNext(T t);
   public void onError(Throwable t);
   public void onComplete();
}

public interface Subscription {
   public void request(long n);
   public void cancel();
}
public interface Processor<T, R>
  extends Subscriber<T>, Publisher<R> {
}
```

FLOWABLE VS. OBSERVABLE

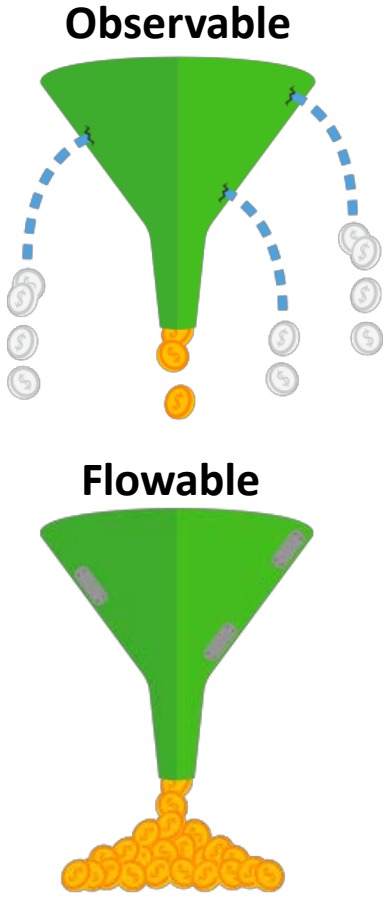
Flowable has backpressure because of **request method** of Subscription.
Observable does not have backpressure.

In Observable:
Disposable subscribe(Consumer<? super T> onNext)
In Flowable (in addition to same as Observable):
void subscribe(Subscriber<? super T> s);

```
public interface Disposable {
   void dispose();
   boolean isDisposed();
}
```

```
public interface Subscription {
   void request(long n);
   void cancel();
}
```

```
public interface Subscriber<T> {
   void onSubscribe(Subscription s);
   void onNext(T next);
   void onError(Throwable t);
   void onComplete();
}
```

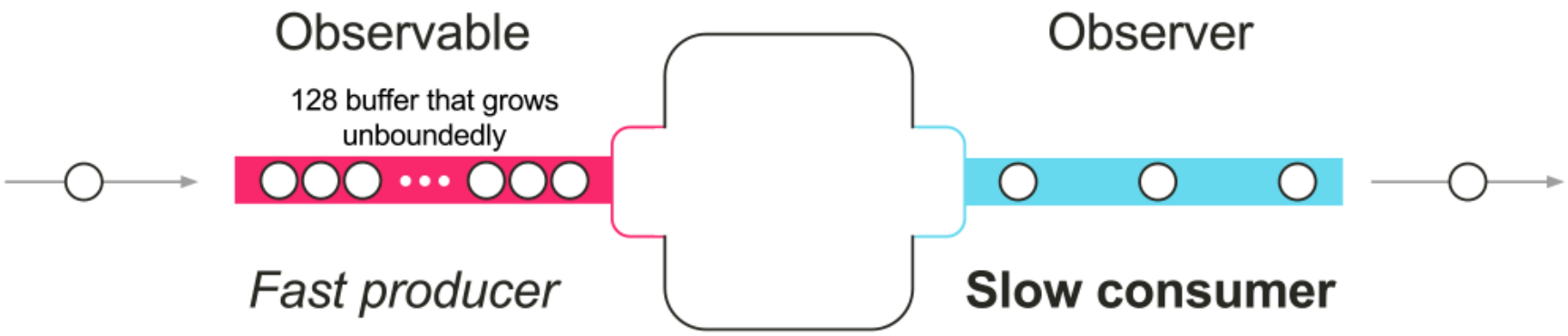


Example 61 FlowableSubscriber

BACKPRESSUREMODE.BUFFER

BackpressureMode.BUFFER

In this mode, an unbounded buffer with an initial size of 128 is created. Items emitted too quickly are buffered unboundedly. If the buffer isn't drained, items continue to accumulate until memory is exhausted. This results in an OutOfMemoryException.



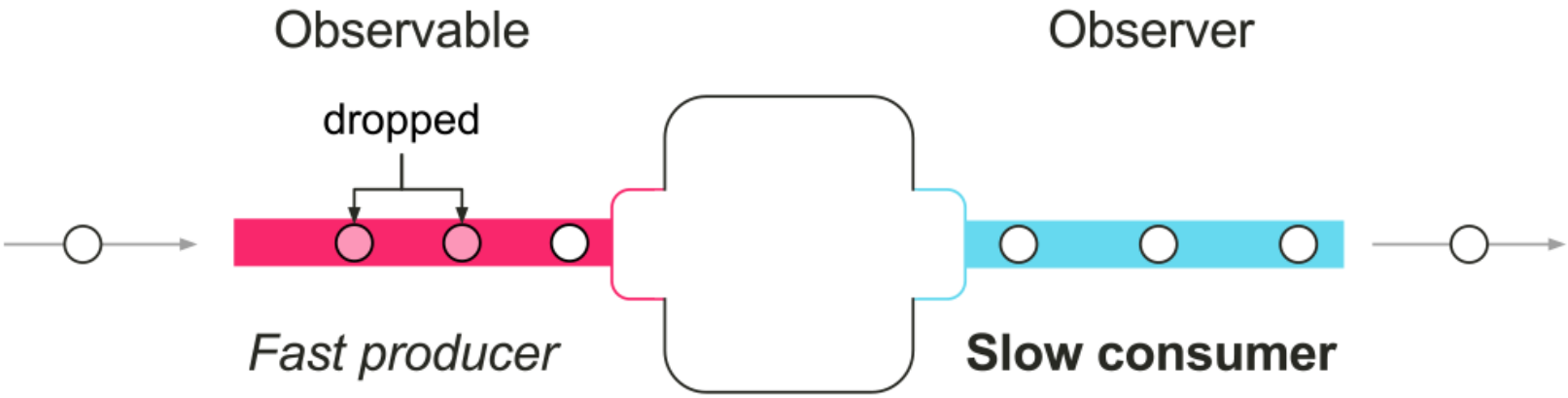
Buffer is a default backpressure strategy, but we are able to configure size of the buffer.

Example 62

BACKPRESSUREMODE.DROP

BackpressureMode.DROP

This mode uses a fixed buffer of size 1. If the downstream observable can't keep up, the first item is buffered and subsequent emissions are dropped. When the consumer is ready to take the next value, it receives the first value emitted by the source Observable.

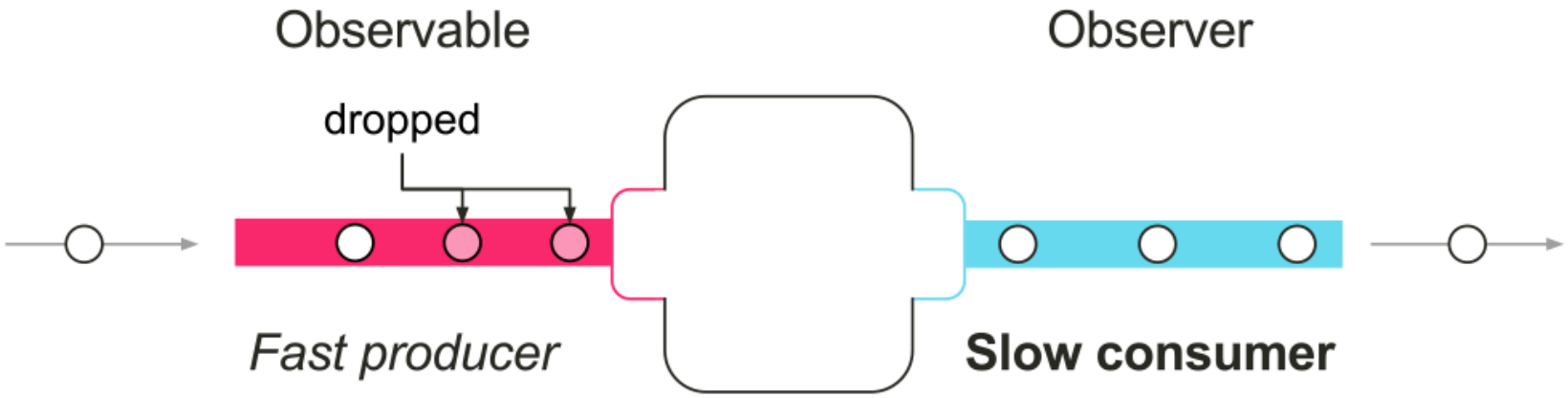


Example 63

BACKPRESSUREMODE.LATEST

BackpressureMode.LATEST

This mode is similar to BackpressureMode.DROP because it also uses a fixed buffer of size 1. However, rather than buffering the first item and dropping subsequent items, BackpressureMode.LATEST replaces the item in the buffer with the latest emission. When the consumer is ready to take the next value, it receives the latest value emitted by the source Observable.



Example 64

CONVERT OBSERVABLE TO FLOWABLE

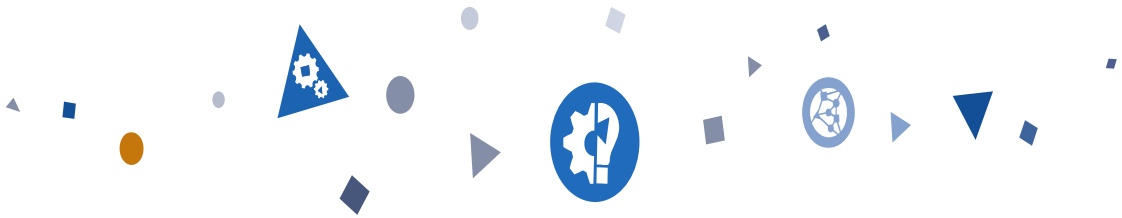
toFlowable() converts the current Observable into a Flowable by applying the specified backpressure strategy.

```
Observable.range(1, 1_000_000)
        .toFlowable(BackpressureStrategy.BUFFER)
        .map(Counter::new)
        .observeOn(Schedulers.io())
        .subscribe(counter -> {
            Thread.sleep(30);
            System.out.println("Received " + counter.count);
        }, Throwable::printStackTrace);
```

Example 65

BACKPRESSURE STRATEGIES

Strategy	Explanation
BUFFER	Buffers all the values of `onNext()` until it has not been consumed by the downstream.
DROP	Drops the most recent value given by `onNext()` if the consumer is not able to keep that.
ERROR	If the downstream is not able to keep up, `MissingBackpressureException` is thrown.
LATEST	The latest value (if not used by the `Subscriber`) will be overwritten by the recent value of `onNext()`.
MISSING	The `onNext()` method events are written without buffering.



TESTING RX JAVA

TESTOBSERVER

```
Observable<Integer> source = Observable.range(1, 5);

TestObserver<Integer> testObserver = new TestObserver<>();
testObserver.assertNotSubscribed();

source.subscribe(testObserver);
testObserver.assertSubscribed();

testObserver.awaitTerminalEvent();
testObserver.assertComplete();
testObserver.assertNoErrors();

testObserver.assertValueCount(5);
testObserver.assertValues(1, 2, 3, 4, 5);
```

Example 66

TESTSCHEDULER

- Tests that deal with time-driven sources.
- Simulate time elapses rather than experiencing them.
- `TestScheduler` is a `Scheduler` implementation that allows to fast-forward by a specific amount of elapsed time
- We can do any assertions after each fast-forward to see what events have occurred.

TESTSCHEDULER

```
TestScheduler testScheduler = new TestScheduler();
TestObserver<Long> testObserver = new TestObserver<>();

Observable<Long> minuteTicker =
        Observable.interval(1, TimeUnit.MINUTES,
                testScheduler);
minuteTicker.subscribe(testObserver);

testScheduler.advanceTimeBy(30, TimeUnit.SECONDS);
testObserver.assertValueCount(0);
testScheduler.advanceTimeTo(70, TimeUnit.SECONDS);

testObserver.assertValueCount(1);

testScheduler.advanceTimeTo(90, TimeUnit.MINUTES);
testObserver.assertValueCount(90);
```

Example 67

Think about
how data should flow
instead of
what you do to make it flow

Thank You!

think.
create.
accelerate.