FREE eBook

LEARNING rx-java

Free unaffiliated eBook created from
Stack Overflow contributors.

#rx-java

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rx-java

It is an unofficial and free rx-java ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rx-java.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with rx-java

Remarks

This section provides a basic overview and superficial introduction to rx-java.

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

Learn more about RxJava on the Wiki Home.

Versions

Version	Status	Latest Stable Version	Release Date
1.x	Stable	1.3.0	2017-05-05
2.x	Stable	2.1.1	2017-06-21

Examples

Installation or Setup

rx-java set up

1. Gradle

```
compile 'io.reactivex:rxjava2:rxjava:2.1.1'
```

2. Maven

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.1.1</version>
</dependency>
```

3. Ivy

```
<dependency org="io.reactivex.rxjava2" name="rxjava" rev="2.1.1" />
```

4. Snapshots from JFrog

```
repositories {
maven { url 'https://oss.jfrog.org/libs-snapshot' }
```

```
}

dependencies {
    compile 'io.reactivex:rxjava:2.0.0-SNAPSHOT'
}
```

5. If you need to download the jars instead of using a build system, create a Maven `pom` file like this with the desired version:

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.netflix.rxjava.download</groupId>
    <artifactId>rxjava-download</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Simple POM to download rxjava and dependencies</name>
    <url>http://github.com/ReactiveX/RxJava</url>
    <dependencies>
        <dependency>
            <groupId>io.reactivex</groupId>
            <artifactId>rxjava</artifactId>
            <version>2.0.0</version>
            <scope/>
        </dependency>
    </dependencies>
</project>
```

Then execute:

```
$ mvn -f download-rxjava-pom.xml dependency:copy-dependencies
```

That command downloads `rxjava-*.jar` and its dependencies into `./target/dependency/`.

You need Java 6 or later.

Hello, World!

The following prints the message `Hello, World!` to console

```
public void hello() {
  Observable.just("Hello, World!") // create new observable
    .subscribe(new Action1<String>() { // subscribe and perform action

       @Override
       public void call(String st) {
         System.out.println(st);
       }

    });
}
```

Or using Java 8 lambda notation

```
public void hello() {
      Observable.just("Hello, World!") // create new observable
        .subscribe(onNext -> { // subscribe and perform action
             System.out.println(onNext);
        });
}
```

An introduction to RxJava

The core concepts of RxJava are its `Observables` and `Subscribers`. An `Observable` emits objects, while a `Subscriber` consumes them.

Observable

`Observable` is a class that implements the reactive design pattern. These Observables provide methods that allow consumers to subscribe to event changes. The event changes are triggered by the observable. There is no restriction to the number of subscribers that an `Observable` can have, or the number of objects that an `Observable` can emit.

Take for example:

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // Integer observable
Observable<String> stringObservable = Observable.just("Hello, ", "World", "!"); // String
observable
```

Here, an observable object called `integerObservable` and `stringObservable` are created from the factory method `just` provided by the Rx library. Notice that `Observable` is generic and can thus can emit any object.

Subscriber

A `Subscriber` is the consumer. A `Subscriber` can subscribe to ***only one*** observable. The `Observable` calls the `onNext()`, `onCompleted()`, and `onError()` methods of the `Subscriber`.

```
Subscriber<Integer> mSubscriber = new Subscriber<Integer>() {
        // NOTE THAT ALL THESE ARE CALLED BY THE OBSERVABLE
        @Override
        public void onCompleted() {
            // called when all objects are emitted
            System.out.println("onCompleted called!");
        }

        @Override
        public void onError(Throwable throwable) {
            // called when an error occurs during emitting objects
            System.out.println("onError called!");
        }

        @Override
        public void onNext(Integer integer) {
            // called for each object that is emitted
            System.out.println("onNext called with: " + integer);
        }
    };
```

Notice that `Subscriber` is also generic and can support any object. A `Subscriber` must subscribe to the observable by calling the `subscribe` method on the observable.

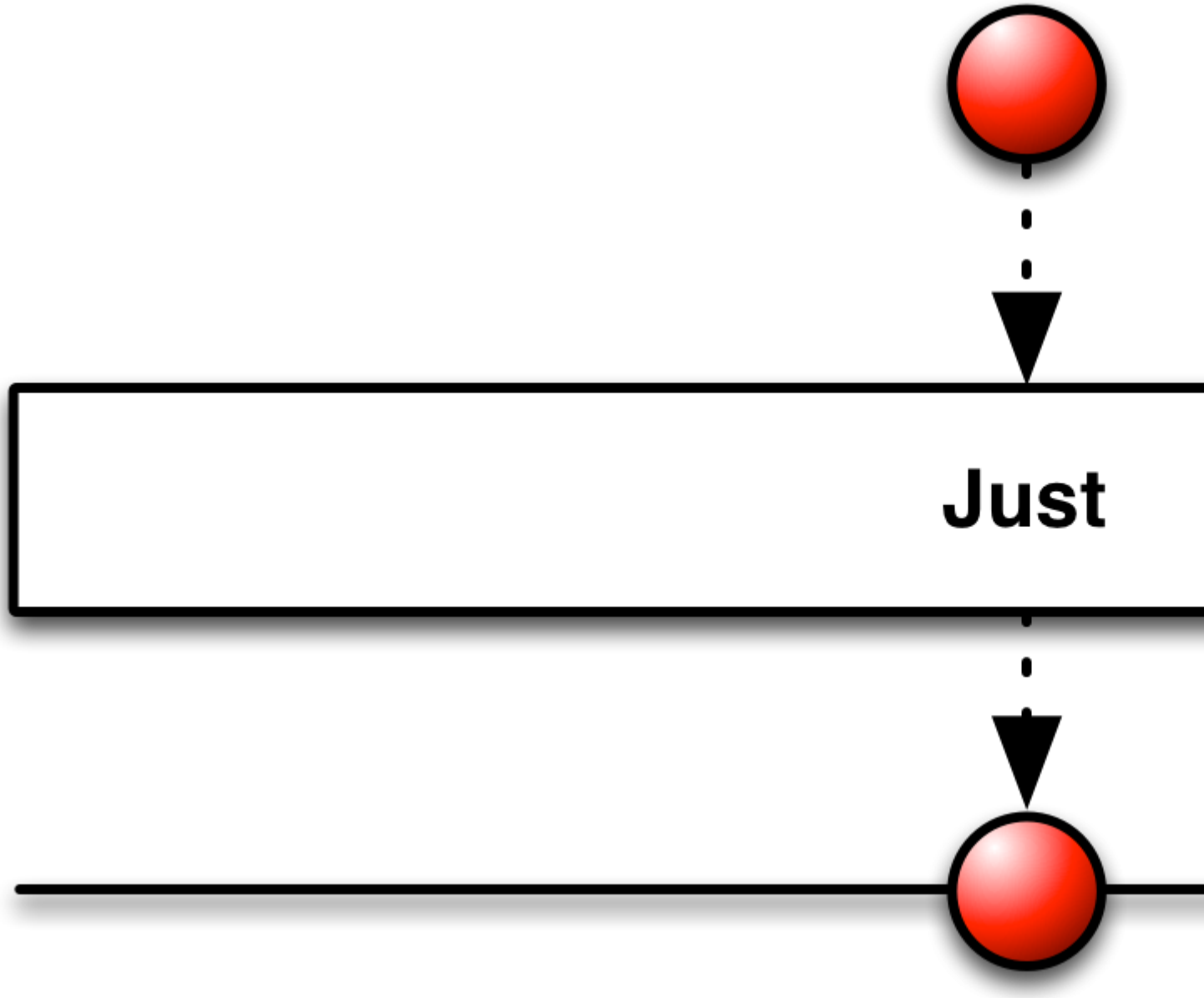
```
integerObservable.subscribe(mSubscriber);
```

The above, when run, will produce the following output:

```
onNext called with: 1
onNext called with: 2
onNext called with: 3
onCompleted called!
```

Understanding Marble Diagrams

An Observable can be thought of as just a stream of events. When you define an Observable, you have three listeners: onNext, onComplete and onError. onNext will be called every time the observable acquires a new value. onComplete will be called if the parent Observable notifies that it finished producing any more values. onError is called if an exception is thrown any time during the execution of the Observable chain. To show operators in Rx, the marble diagram is used to display what happens with a particular operation. Below is an example of a simple Observable operator "Just."



Marble diagrams have a horizontal block that represents the operation being performed, a vertical bar to represent the completed event, a X to represent an error, and any other shape represents a value. With that in mind, we can see that "Just" will just take our value and do an onNext and then finish with onComplete. There are a lot more operations then just "Just." You can see all the operations that are part of the ReactiveX project and there implementations in RxJava at the ReativeX site. There are also interactive marble diagrams via RxMarbles site.

Read Getting started with rx-java online: https://riptutorial.com/rx-java/topic/974/getting-started-with-rx-java

Chapter 2: Android with RxJava

Remarks

RxAndroid used to be a library with lot of features. It has been splitted in many different libraries moving from version 0.25.0 to 1.x.

A list of libraries that implement the features available before the 1.0 is maintained here.

Examples

RxAndroid - AndroidSchedulers

This is literally the only thing you need to start using RxJava on Android.

Include RxJava and RxAndroid in your gradle dependencies:

```
 // use the last version
compile 'io.reactivex.rxjava2:rxjava:2.1.1'
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

RxAndroid main addition to RxJava is a Scheduler for the Android Main Thread or UI Thread.

In your code:

```
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

Or you can create a Scheduler for a custom `Looper`:

```
Looper backgroundLooper = // ...
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.from(backgroundLooper))
        .subscribe(
            data -> doStuffOnMainThread(),
            error -> handleErrorOnMainThread()
        )
```

For most everything else you can refer to standard RxJava documentation.

RxLifecycle components

The RxLifecycle library makes it easier binding observable subscriptions to Android activities and

fragment lifecycle.

Keep in mind that forgetting to unsubscribe an Observable can cause memory leaks and keeping your activity / fragment alive event after it has been destroyed by the system.

Add the library to the dependencies:

```
// use the last version available
compile 'com.trello:rxlifecycle:0.6.1'
compile 'com.trello:rxlifecycle-components:0.6.1'
```

Then extends `Rx*` classes:

- `RxActivity` / `support.RxFragmentActivity` / `support.RxAppCompatActivity`
- `RxFragment` / `support.RxFragment`
- `RxDialogFragment` / `support.RxDialogFragment`
- `support.RxAppCompatDialogActivity`

You are all set, when you subscribe to an Observable you can now:

```
someObservable
    .compose(bindToLifecycle())
    .subscribe();
```

If you execute this in the `onCreate()` method of the activity it will automatically unsubscribed in the `onDestroy()`.

Tha same happens for:

- `onStart()` -> `onStop()`
- `onResume()` -> `onPause()`
- `onAttach()` -> `onDetach()` *(fragment only)*
- `onViewCreated()` -> `onDestroyView()` *(fragment only)*

As an alternative you can specify the event when you want the unsubscription to happen:

From an activity:

```
someObservable
    .compose(bindUntilEvent(ActivityEvent.DESTROY))
    .subscribe();
```

From a Fragment:

```
someObservable
    .compose(bindUntilEvent(FragmentEvent.DESTROY_VIEW))
    .subscribe();
```

You can also obtain the lifecycle observable using the method `lifecycle()` to listen lifecycle events directly.

RxLifecycle can also be used directly passing to it the lifecycle observable:

```
.compose(RxLifecycleAndroid.bindActivity(lifecycle))
```

If you need to handle `Single` or `Completable` you can do it by just adding respectively `forSingle()` or `forCompletable` after the bind method:

```
someSingle
    .compose(bindToLifecycle().forSingle())
    .subscribe();
```

It can also be used with Navi library.

Rxpermissions

This library allows the usage of RxJava with the new Android M permission model.

Add the library to the dependencies:

Rxjava

```
dependencies {
    compile 'com.tbruyelle.rxpermissions:rxpermissions:0.8.0@aar'
}
```

Rxjava2

```
dependencies {
    compile 'com.tbruyelle.rxpermissions2:rxpermissions:0.8.1@aar'
}
```

Usage

Example (with Retrolambda for brevity, but not required):

```
// Must be done during an initialization phase like onCreate
RxPermissions.getInstance(this)
    .request(Manifest.permission.CAMERA)
    .subscribe(granted -> {
        if (granted) { // Always true pre-M
           // I can control the camera now
        } else {
           // Oups permission denied
        }
    });
```

Read more: https://github.com/tbruyelle/RxPermissions.

Read Android with RxJava online: https://riptutorial.com/rx-java/topic/7125/android-with-rxjava

Chapter 3: Backpressure

Examples

Introduction

Backpressure is when in an `Observable` processing pipeline, some asynchronous stages can't process the values fast enough and need a way to tell the upstream producer to slow down.

The classic case of the need for backpressure is when the producer is a hot source:

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

source
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}

Thread.sleep(10_000);
```

In this example, the main thread will produce 1 million items to an end consumer which is processing it on a background thread. It is likely the `compute(int)` method takes some time but the overhead of the `Observable` operator chain may also add to the time it takes to process items. However, the producing thread with the for loop can't know this and keeps `onNext`ing.

Internally, asynchronous operators have buffers to hold such elements until they can be processed. In the classical Rx.NET and early RxJava, these buffers were unbounded, meaning that they would likely hold nearly all 1 million elements from the example. The problem starts when there are, for example, 1 billion elements or the same 1 million sequence appears 1000 times in a program, leading to `OutOfMemoryError` and generally slowdowns due to excessive GC overhead.

Similar to how error-handling became a first-class citizen and received operators to deal with it (via `onErrorXXX` operators), backpressure is another property of dataflows that the programmer has to think about and handle (via `onBackpressureXXX` operators).

Beyond the `PublishSubject`above, there are other operators that don't support backpressure, mostly due to functional reasons. For example, the operator `interval` emits values periodically, backpressuring it would lead to shifting in the period relative to a wall clock.

In modern RxJava, most asynchronous operators now have a bounded internal buffer, like `observeOn` above and any attempt to overflow this buffer will terminate the whole sequence with `MissingBackpressureException`. The documentation of each operator has a description about its backpressure behavior.

However, backpressure is present more subtly in regular cold sequences (which don't and

shouldn't yield `MissingBackpressureException`). If the first example is rewritten:

```
Observable.range(1, 1_000_000)
.observeOn(Schedulers.computation())
.subscribe(v -> compute(v), Throwable::printStackTrace);

Thread.sleep(10_000);
```

There is no error and everything runs smoothly with small memory usage. The reason for this is that many source operators can "generate" values on demand and thus the operator `observeOn` can tell the `range` generate at most so many values the `observeOn` buffer can hold at once without overflow.

This negotiation is based on the computer science concept of co-routines (I call you, you call me). The operator `range` sends a callback, in the form of an implementation of the `Producer` interface, to the `observeOn` by calling its (inner `Subscriber`'s) `setProducer`. In return, the `observeOn` calls `Producer.request(n)` with a value to tell the `range` it is allowed to produce (i.e., `onNext` it) that many **additional** elements. It is then the `observeOn`'s responsibility to call the `request` method in the right time and with the right value to keep the data flowing but not overflowing.

Expressing backpressure in end-consumers is rarely necessary (because they are synchronous in respect to their immediate upstream and backpressure naturally happens due to call-stack blocking), but it may be easier to understand the workings of it:

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(1);
    }

    public void onNext(Integer v) {
        compute(v);

        request(1);
    }

    @Override
    public void onError(Throwable ex) {
        ex.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("Done!");
    }
});
```

Here the `onStart` implementation indicates `range` to produce its first value, which is then received in `onNext`. Once the `compute(int)` finishes, the another value is then requested from `range`. In a naive implementation of `range`, such call would recursively call `onNext`, leading to `StackOverflowError` which is of course undesirable.

To prevent this, operators use so-called trampolining logic that prevents such reentrant calls. In `range`'s terms, it will remember that there was a `request(1)` call while it called `onNext()` and once `onNext()` returns, it will make another round and call `onNext()` with the next integer value. Therefore, if the two are swapped, the example still works the same:

```
@Override
public void onNext(Integer v) {
    request(1);

    compute(v);
}
```

However, this is not true for `onStart`. Although the `Observable` infrastructure guarantees it will be called at most once on each `Subscriber`, the call to `request(1)` may trigger the emission of an element right away. If one has initialization logic after the call to `request(1)` which is needed by `onNext`, you may end up with exceptions:

```
Observable.range(1, 1_000_000)
.subscribe(new Subscriber<Integer>() {

    String name;

    @Override
    public void onStart() {
        request(1);

        name = "RangeExample";
    }

    @Override
    public void onNext(Integer v) {
        compute(name.length + v);

        request(1);
    }

    // ... rest is the same
});
```

In this synchronous case, a `NullPointerException` will be thrown immediately while still executing `onStart`. A more subtle bug happens if the call to `request(1)` triggers an asynchronous call to `onNext` on some other thread and reading `name` in `onNext` races writing it in `onStart` post `request`.

Therefore, one should do all field initialization in `onStart` or even before that and call `request()` last. Implementations of `request()` in operators ensure proper happens-before relation (or in other terms, memory release or full fence) when necessary.

The onBackpressureXXX operators

Most developers encounter backpressure when their application fails with `MissingBackpressureException` and the exception usually points to the `observeOn` operator. The actual cause is usually the non-backpressured use of `PublishSubject`, `timer()` or `interval()` or custom operators created via `create()`.

There are several ways of dealing with such situations.

Increasing the buffer sizes

Sometimes such overflows happen due to bursty sources. Suddenly, the user taps the screen too quickly and `observeOn`'s default 16-element internal buffer on Android overflows.

Most backpressure-sensitive operators in the recent versions of RxJava now allow programmers to specify the size of their internal buffers. The relevant parameters are usually called `bufferSize`, `prefetch` or `capacityHint`. Given the overflowing example in the introduction, we can just increase the buffer size of `observeOn` to have enough room for all values.

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

source.observeOn(Schedulers.computation(), 1024 * 1024)
      .subscribe(e -> { }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Note however that generally, this may be only a temporary fix as the overflow can still happen if the source overproduces the predicted buffer size. In this case, one can use one of the following operators.

Batching/skipping values with standard operators

In case the source data can be processed more efficiently in batch, one can reduce the likelihood of `MissingBackpressureException` by using one of the standard batching operators (by size and/or by time).

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

source
      .buffer(1024)
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(list -> {
          list.parallelStream().map(e -> e * e).first();
      }, Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

If some of the values can be safely ignored, one can use the sampling (with time or another Observable) and throttling operators (`throttleFirst`, `throttleLast`, `throttleWithTimeout`).

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

source
      .sample(1, TimeUnit.MILLISECONDS)
```

```
      .observeOn(Schedulers.computation(), 1024)
      .subscribe(v -> compute(v), Throwable::printStackTrace);

for (int i = 0; i < 1_000_000; i++) {
    source.onNext(i);
}
```

Note hovewer that these operators only reduce the rate of value reception by the downstream and thus they may still lead to `MissingBackpressureException`.

onBackpressureBuffer()

This operator in its parameterless form reintroduces an unbounded buffer between the upstream source and the downstream operator. Being unbounded means as long as the JVM doesn't run out of memory, it can handle almost any amount coming from a bursty source.

```
 Observable.range(1, 1_000_000)
           .onBackpressureBuffer()
           .observeOn(Schedulers.computation(), 8)
           .subscribe(e -> { }, Throwable::printStackTrace);
```

In this example, the `observeOn` goes with a very low buffer size yet there is no `MissingBackpressureException` as `onBackpressureBuffer` soaks up all the 1 million values and hands over small batches of it to `observeOn`.

Note however that `onBackpressureBuffer` consumes its source in an unbounded manner, that is, without applying any backpressure to it. This has the consequence that even a backpressure-supporting source such as `range` will be completely realized.

There are 4 additional overloads of `onBackpressureBuffer`

onBackpressureBuffer(int capacity)

This is a bounded version that signals `BufferOverflowError`in case its buffer reaches the given capacity.

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

The relevance of this operator is decreasing as more and more operators now allow setting their buffer sizes. For the rest, this gives an opportunity to "extend their internal buffer" by having a larger number with `onBackpressureBuffer` than their default.

onBackpressureBuffer(int capacity, Action0 onOverflow)

This overload calls a (shared) action in case an overflow happens. Its usefulness is rather limited as there is no other information provided about the overflow than the current call stack.

onBackpressureBuffer(int capacity, Action0 onOverflow, BackpressureOverflow.Strategy strategy)

This overload is actually more useful as it let's one define what to do in case the capacity has been reached. The `BackpressureOverflow.Strategy` is an interface actually but the class `BackpressureOverflow` offers 4 static fields with implementations of it representing typical actions:

- `ON_OVERFLOW_ERROR`: this is the default behavior of the previous two overloads, signalling a `BufferOverflowException`
- `ON_OVERFLOW_DEFAULT`: currently it is the same as `ON_OVERFLOW_ERROR`
- `ON_OVERFLOW_DROP_LATEST` : if an overflow would happen, the current value will be simply ignored and only the old values will be delivered once the downstream requests.
- `ON_OVERFLOW_DROP_OLDEST` : drops the oldest element in the buffer and adds the current value to it.

```
Observable.range(1, 1_000_000)
          .onBackpressureBuffer(16, () -> { },
              BufferOverflowStrategy.ON_OVERFLOW_DROP_OLDEST)
          .observeOn(Schedulers.computation())
          .subscribe(e -> { }, Throwable::printStackTrace);
```

Note that the last two strategies cause discontinuity in the stream as they drop out elements. In addition, they won't signal `BufferOverflowException`.

onBackpressureDrop()

Whenever the downstream is not ready to receive values, this operator will drop that elemenet from the sequence. One can think of it as a 0 capacity `onBackpressureBuffer` with strategy `ON_OVERFLOW_DROP_LATEST`.

This operator is useful when one can safely ignore values from a source (such as mouse moves or current GPS location signals) as there will be more up-to-date values later on.

```
component.mouseMoves()
.onBackpressureDrop()
.observeOn(Schedulers.computation(), 1)
.subscribe(event -> compute(event.x, event.y));
```

It may be useful in conjunction with the source operator `interval()`. For example, if one wants to perform some periodic background task but each iteration may last longer than the period, it is safe to drop the excess interval notification as there will be more later on:

```
Observable.interval(1, TimeUnit.MINUTES)
.onBackpressureDrop()
.observeOn(Schedulers.io())
.doOnNext(e -> networkCall.doStuff())
.subscribe(v -> { }, Throwable::printStackTrace);
```

There exist one overload of this operator: `onBackpressureDrop(Action1<? super T> onDrop)` where the

(shared) action is called with the value being dropped. This variant allows cleaning up the values themselves (e.g., releasing associated resources).

onBackpressureLatest()

The final operator keeps only the latest value and practically overwrites older, undelivered values. One can think of this as a variant of the `onBackpressureBuffer` with a capacity of 1 and strategy of `ON_OVERFLOW_DROP_OLDEST`.

Unlike `onBackpressureDrop` there is always a value available for consumption if the downstream happened to be lagging behind. This can be useful in some telemetry-like situations where the data may come in some bursty pattern but only the very latest is interesting for processing.

For example, if the user clicks a lot on the screen, we'd still want to react to its latest input.

```
component.mouseClicks()
.onBackpressureLatest()
.observeOn(Schedulers.computation())
.subscribe(event -> compute(event.x, event.y), Throwable::printStackTrace);
```

The use of `onBackpressureDrop` in this case would lead to a situation where the very last click gets dropped and leaves the user wondering why the business logic wasn't executed.

Creating backpressured data sources

Creating backpressured data sources is the relatively easier task when dealing with backpressure in general because the library already offers static methods on `Observable` that handle backpressure for the developer. We can distinguish two kinds of factory methods: cold "generators" that either return and generate elements based on downstream demand and hot "pushers" that usually bridge non-reactive and/or non-backpressurable data sources and layer some backpressure handling on top of them.

just

The most basic backpressure aware source is created via `just`:

```
Observable.just(1).subscribe(new Subscriber<Integer>() {
    @Override
    public void onStart() {
        request(0);
    }

    @Override
    public void onNext(Integer v) {
        System.out.println(v);
    }

    // the rest is omitted for brevity
}
```

Since we explicitly don't request in `onStart`, this will not print anything. `just` is great when there is a constant value we'd like to jump-start a sequence.

Unfortunately, `just` is often mistaken for a way to compute something dynamically to be consumed by `Subscriber`s:

```
int counter;

int computeValue() {
   return ++counter;
}

Observable<Integer> o = Observable.just(computeValue());

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

Surprising to some, this prints 1 twice instead of printing 1 and 2 respectively. If the call is rewritten, it becomes obvious why it works so:

```
int temp = computeValue();

Observable<Integer> o = Observable.just(temp);
```

The `computeValue` is called as part of the main routine and not in response to the subscribers subscribing.

fromCallable

What people actually need is the method `fromCallable`:

```
Observable<Integer> o = Observable.fromCallable(() -> computeValue());
```

Here the `computeValue` is executed only when a subscriber subscribes and for each of them, printing the expected 1 and 2. Naturally, `fromCallable` also properly supports backpressure and won't emit the computed value unless requested. Note however that the computation does happen anyway. In case the computation itself should be delayed until the downstream actually requests, we can use `just` with `map`:

```
Observable.just("This doesn't matter").map(ignored -> computeValue())...
```

`just` won't emit its constant value until requested when it is mapped to the result of the `computeValue`, still called for each subscriber individually.

from

If the data is already available as an array of objects, a list of objects or any `Iterable` source, the respective `from` overloads will handle the backpressure and emission of such sources:

```
Observable.from(Arrays.asList(1, 2, 3, 4, 5)).subscribe(System.out::println);
```

For convenience (and avoiding warnings about generic array creation) there are 2 to 10 argument overloads to `just` that internally delegate to `from`.

The `from(Iterable)` also gives an interesting opportunity. Many value generation can be expressed in a form of a state-machine. Each requested element triggers a state transition and computation of the returned value.

Writing such state machines as `Iterable`s is somewhat complicated (but still easier than writing an `Observable` for consuming it) and unlike C#, Java doesn't have any support from the compiler to build such state machines by simply writing classically looking code (with `yield return` and `yield break`). Some libraries offer some help, such as Google Guava's `AbstractIterable` and IxJava's `Ix.generate()` and `Ix.forloop()`. These are by themselves worthy of a full series so let's see some very basic `Iterable` source that repeats some constant value indefinitely:

```
Iterable<Integer> iterable = () -> new Iterator<Integer>() {
    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        return 1;
    }
};

Observable.from(iterable).take(5).subscribe(System.out::println);
```

If we'd consume the `iterator` via classic for-loop, that would result in an infinite loop. Since we build an `Observable` out of it, we can express our will to consume only the first 5 of it and then stop requesting anything. This is the true power of lazily evaluating and computing inside `Observable`s.

create(SyncOnSubscribe)

Sometimes, the data source to be converted into the reactive world itself is synchronous (blocking) and pull-like, that is, we have to call some `get` or `read` method to get the next piece of data. One could, of course, turn that into an `Iterable` but when such sources are associated with resources, we may leak those resources if the downstream unsubscribes the sequence before it would end.

To handle such cases, RxJava has the `SyncOnSubscribe` class. One can extend it and implement its methods or use one of its lambda-based factory methods to build an instance.

```
SyncOnSubscribe<Integer, InputStream> binaryReader = SyncOnSubscribe.createStateful(
     () -> new FileInputStream("data.bin"),
     (inputstream, output) -> {
         try {
             int byte = inputstream.read();
             if (byte < 0) {
                 output.onCompleted();
```

```
            } else {
                output.onNext(byte);
            }
        } catch (IOException ex) {
            output.onError(ex);
        }
        return inputstream;
    },
    inputstream -> {
        try {
            inputstream.close();
        } catch (IOException ex) {
            RxJavaHooks.onError(ex);
        }
    }
);

Observable<Integer> o = Observable.create(binaryReader);
```

Generally, `SyncOnSubscribe` uses 3 callbacks.

The first callbacks allows one to create a per-subscriber state, such as the `FileInputStream` in the example; the file will be opened independently to each individual subscriber.

The second callback takes this state object and provides an output `Observer` whose `onXXX` methods can be called to emit values. This callback is executed as many times as the downstream requested. At each invocation, it has to call `onNext` at most once optionally followed by either `onError` or `onCompleted`. In the example we call `onCompleted()` if the read byte is negative, indicating and end of file, and call `onError` in case the read throws an `IOException`.

The final callback gets invoked when the downstream unsubscribes (closing the inputstream) or when the previous callback called the terminal methods; it allows freeing up resources. Since not all sources need all these features, the static methods of `SyncOnSubscribe` let's one create instances without them.

Unfortunately, many method calls across the JVM and other libraries throw checked exceptions and need to be wrapped into `try-catch`es as the functional interfaces used by this class don't allow throwing checked exceptions.

Of course, we can imitate other typical sources, such as an unbounded range with it:

```
SyncOnSubscribe.createStateful(
    () -> 0,
    (current, output) -> {
        output.onNext(current);
        return current + 1;
    },
    e -> { }
);
```

In this setup, the `current` starts out with `0` and next time the lambda is invoked, the parameter `current` now holds `1`.

There is a variant of `SyncOnSubscribe` called `AsyncOnSubscribe` that looks quite similar with the

exception that the middle callback also takes long value that represents the request amount from downstream and the callback should generate an `Observable` with the exact same length. This source then concatenates all these `Observable` into a single sequence.

```
AsyncOnSubscribe.createStateful(
    () -> 0,
    (state, requested, output) -> {
        output.onNext(Observable.range(state, (int)requested));
        return state + 1;
    },
    e -> { }
);
```

There is an ongoing (heated) discussion about the usefulness of this class and generally not recommended because it routinely breaks expectations about how it will actually emit those generated values and how it will respond to, or even what kind of request values it will receive in more complex consumer scenarios.

create(emitter)

Sometimes, the source to be wrapped into an `Observable` is already hot (such as mouse moves) or cold but not backpressurable in its API (such as an asynchronous network callback).

To handle such cases, a recent version of RxJava introduced the `create(emitter)` factory method. It takes two parameters:

- a callback that will be called with an instance of the `Emitter<T>` interface for each incoming subscriber,
- a `Emitter.BackpressureMode` enumeration that mandates the developer to specify the backpressure behavior to be applied. It has the usual modes, similar to `onBackpressureXXX` in addition to signalling a `MissingBackpressureException` or simply ignoring such overflow inside it altogether.

Note that it currently doesn't support additional parameters to those backpressure modes. If one needs those customization, using `NONE` as the backpressure mode and applying the relevant `onBackpressureXXX` on the resulting `Observable` is the way to go.

The first typical case for its use when one wants to interact with a push-based source, such as GUI events. Those APIs feature some form of `addListener`/`removeListener` calls that one can utilize:

```
Observable.create(emitter -> {
    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    emitter.setCancellation(() ->
        button.removeListener(al));

}, BackpressureMode.BUFFER);
```

The `Emitter` is relatively straightforward to use; one can call `onNext`, `onError` and `onCompleted` on it and the operator handles backpressure and unsubscription management on its own. In addition, if the wrapped API supports cancellation (such as the listener removal in the example), one can use the `setCancellation` (or `setSubscription` for `Subscription`-like resources) to register a cancellation callback that gets invoked when the downstream unsubscribes or the `onError`/`onCompleted` is called on the provided `Emitter`instance.

These methods allow only a single resource to be associated with the emitter at a time and setting a new one unsubscribes the old one automatically. If one has to handle multiple resources, create a `CompositeSubscription`, associate it with the emitter and then add further resources to the `CompositeSubscription` itself:

```
Observable.create(emitter -> {
    CompositeSubscription cs = new CompositeSubscription();

    Worker worker = Schedulers.computation().createWorker();

    ActionListener al = e -> {
        emitter.onNext(e);
    };

    button.addActionListener(al);

    cs.add(worker);
    cs.add(Subscriptions.create(() ->
        button.removeActionListener(al));

    emitter.setSubscription(cs);

}, BackpressureMode.BUFFER);
```

The second scenario usually involves some asynchronous, callback-based API that has to be converted into an `Observable`.

```
Observable.create(emitter -> {

    someAPI.remoteCall(new Callback<Data>() {
        @Override
        public void onSuccess(Data data) {
            emitter.onNext(data);
            emitter.onCompleted();
        }

        @Override
        public void onFailure(Exception error) {
            emitter.onError(error);
        }
    });

}, BackpressureMode.LATEST);
```

In this case, the delegation works the same way. Unfortunately, usually, these classical callback-style APIs don't support cancellation, but if they do, one can setup their cancellation just like in the previoius examples (with perhaps a more involved way though). Note the use of the `LATEST`

backpressure mode; if we know there will be only a single value, we don't need the `BUFFER` strategy as it allocates a default 128 element long buffer (that grows as necessary) that is never going to be fully utilized.

Read Backpressure online: https://riptutorial.com/rx-java/topic/2341/backpressure

Chapter 4: Observable

Examples

Create an Observable

There are several ways to create an Observable in RxJava. The most powerful way is to use the `Observable.create` method. But it's also the most **complicated way**. So you must **avoid using it**, as much as possible.

Emitting an exiting value

If you already have a value, you can use `Observable.just` to emit your value.

```
Observable.just("Hello World").subscribe(System.out::println);
```

Emitting a value that should be computed

If you want to emit a value that is not already computed, or that can take long to be computed, you can use `Observable.fromCallable` to emit your next value.

```
Observable.fromCallable(() -> longComputation()).subscribe(System.out::println);
```

`longComputation()` will only be called when you subscribe to your `Observable`. This way, the computation will be *lazy*.

Alternative way to Emitting a value that should be computed

`Observable.defer` builds an `Observable` just like `Observable.fromCallable` but it is used when you need to return an `Observable` instead of a value. It is useful when you want to manage the errors in your call.

```
Observable.defer(() -> {
          try {
                return Observable.just(longComputation());
          } catch(SpecificException e) {
                return Observable.error(e);
          }).subscribe(System.out::println);
```

Hot and Cold Observables

Observables are broadly categorised as `Hot` or `Cold`, depending on their emission behaviour. A `Cold Observable` is one which starts emitting upon request(subscription), whereas a `Hot Observable` is one that emits regardless of subscriptions.

Cold Observable

```
/* Demonstration of a Cold Observable */
Observable<Long> cold = Observable.interval(500, TimeUnit.MILLISECONDS); // emits a long every
500 milli seconds
cold.subscribe(l -> System.out.println("sub1, " + l)); // subscriber1
Thread.sleep(1000); // interval between the two subscribes
cold.subscribe(l -> System.out.println("sub2, " + l)); // subscriber2
```

The output of the above code looks like (may vary):

```
sub1, 0    -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 0    -> subscriber2 starts
sub1, 3
sub2, 1
sub1, 4
sub2, 2
```

Notice that even though `sub2` starts late, it receives values from the start. To conclude, a `Cold Observable` only emits items when requested for. Multiple request start multiple pipelines.

Hot Observable

Note: Hot observables emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not.

A `Cold Observale` can be converted to a `Hot Observable` with a simple `publish`.

```
Observable.interval(500, TimeUnit.MILLISECONDS)
    .publish(); // publish converts cold to hot
```

`publish` returns a `ConnectableObservable` that adds functionalities to *connect* and *disconnect* from the observable.

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // returns ConnectableObservable
hot.connect(); // connect to subscribe

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
```

The above yields:

```
sub1, 0  -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2  -> subscriber2 starts
sub1, 3
sub2, 3
```

Notice that even though `sub2` starts observing late, it is in sync with `sub1`.
Disconnect is a little more complicated! Disconnect happens on the `Subscription` and not the `Observable`.

```
ConnectableObservable<Long> hot = Observable
                                    .interval(500, TimeUnit.MILLISECONDS)
                                    .publish(); // same as above
Subscription subscription = hot.connect(); // connect returns a subscription object, which we
store for further use

hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe(); // disconnect, or unsubscribe from subscription

System.out.println("reconnecting");
/* reconnect and redo */
subscription = hot.connect();
hot.subscribe(l -> System.out.println("sub1, " + l));
Thread.sleep(1000);
hot.subscribe(l -> System.out.println("sub2, " + l));
Thread.sleep(1000);
subscription.unsubscribe();
```

The above produces:

```
sub1, 0   -> subscriber1 starts
sub1, 1
sub1, 2
sub2, 2   -> subscriber2 starts
sub1, 3
sub2, 3
reconnecting  -> reconnect after unsubscribe
sub1, 0
...
```

Upon disconnect, the `Observable` essentially "terminates" and restarts when a new subscription is added.

`Hot Observable` can be used for creating an `EventBus`. Such EventBuses are generally light and super fast. The only downside of an RxBus is that all events must be manually implemented and passed to the bus.

Read Observable online: https://riptutorial.com/rx-java/topic/1418/observable

Chapter 5: Operators

Remarks

This document describes the basic behaviour of an operator.

Examples

Operators, an introduction

An operator can be used to manipulate the flow of objects from `Observable` to `Subscriber`.

```
Observable<Integer> integerObservable = Observable.just(1, 2, 3); // creating a simple Integer
observable
Subscriber<String> mSubscriber = new Subscriber<String>() {
    @Override
    public void onCompleted() {
        System.out.println("onCompleted called!");
    }

    @Override
    public void onError(Throwable throwable) {
        System.out.println("onError called");
    }

    @Override
    public void onNext(String string) {
        System.out.println("onNext called with: " + string);
    }
}; // a simple String subscriber

integerObservable
    .map(new Func1<Integer, String>() {
        @Override
        public String call(Integer integer) {
            switch (integer) {
                case 1:
                    return "one";
                case 2:
                    return "two";
                case 3:
                    return "three";
                default:
                    return "zero";
            }
        }
}).subscribe(mSubscriber);
```

The output would be:

```
onNext called with: one
onNext called with: two
```

```
onNext called with: three
onCompleted called!
```

The `map`operator changed the `Integer` observable to a `String` observable, thereby manipulating the flow of objects.

Operator Chaining

Multiple operators can be `chained` together to for more powerful transforms and manipulations.

```
integerObservable // emits 1, 2, 3
            .map(i -> i + 10) // adds 10 to each item; emits 11, 12, 13
            .filter(i -> i > 11) // emits items that satisfy condition; 12, 13
            .last() // emits last item in observable; 13
            // unlimited operators can be added ...
            .subscribe(System.out::println); // prints 13
```

Any number of operators can be added in between the `Observable` and `Subscriber`.

flatMap Operator

The `flatMap` operator help you to transform one event to another `Observable` (or transform an event to zero, one, or more events).

It's a perfect operator when you want to call another method which return an `Observable`

```
public Observable<String> perform(int i) {
     // ...
}

Observable.just(1, 2, 3)
          .flatMap(i -> perform(i))
          .subscribe(result -> System.out.println("result ->" + result);
```

`flatMap` will serialize `perform` subscriptions **but** events emited by `perform` may not be ordered. So you may receive events emitted by the last perform call **before** events from the first `perform` call (you should use `concatMap` instead).

If your creating another `Observable` in your subscriber, you **should** use `flatMap` instead. The main idea is : **never leave the Observable**

For example :

```
Observable.just(1, 2, 3)
          .subscribe(i -> perform(i));
```

can easily be replaced by :

```
Observable.just(1, 2, 3)
          .flatMap(i -> perform(i))
          .subscribe();
```

Reactivex.io documentation : http://reactivex.io/documentation/operators/flatmap.html

filter Operator

You can use the `filter` operator to filter out items from the values stream based on a result of a predicate method.

In other words, the items passing from the Observer to the Subscriber will be discarded based on the Function you pass `filter`, if the function returns `false` for a certain value, that value will be filtered out.

Example:

```
List<Integer> integers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

Observable.from(integers)
    .filter(number -> {
        return (number  % 2 == 0);
        // odd numbers will return false, that will cause them to be filtered
    })
    .map(i -> {
        return Math.pow(i, 2); // take each number and multiply by power of 2
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the remaining numbers
    });
```

This code will print out

```
0.0
4.0
16.0
36.0
64.0
```

map Operator

You can use the `map` operator to map the values of a stream to different values based on the outcome for each value from the function passed to `map`. The outcome stream is a new copy and will not modify the provided stream of values, the result stream will have the same length of the input stream but may be of different types.

The function passed to `.map()`, must return a value.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
Observable.from(numbers)
    .map(number -> {
        return number.toString(); // convert each integer into a string and return it
    })
    .subscribe(onNext -> {
         System.out.println(onNext); // print out the strings
```

```
    });
```

This code will print out

```
"1"
"2"
"3"
```

In this example the Observable accepted a `List<Integer>` the list will be transformed to a `List<String>` in the pipeline and the `.subscribe` will emit `String`'s

doOnNext operator

`doOnNext` operator called every time when source `Observable` emits an item. It can be used for debugging purposes, applying some action to the emitted item, logging, etc...

```
Observable.range(1, 3)
    .doOnNext(value -> System.out.println("before transform: " + value))
    .map(value -> value * 2)
    .doOnNext(value -> System.out.println("after transform: " + value))
    .subscribe();
```

In the example below `doOnNext` is never called because the source `Observable` emits nothing because `Observable.empty()` calls `onCompleted` after subscribing.

```
Observable.empty()
    .doOnNext(item -> System.out.println("item: " + item))
    .subscribe();
```

repeat operator

`repeat` operator allow to repeat whole sequence from source `Observable`.

```
Observable.just(1, 2, 3)
    .repeat()
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

Output of the example above

```
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
```

This sequence repeats infinite number of times and never completes.

To repeat sequence finite number of times just pass integer as an argument to `repeat` operator.

```
Observable.just(1, 2, 3)
    // Repeat three times and complete
    .repeat(3)
    .subscribe(
        next -> System.out.println("next: " + next),
        error -> System.out.println("error: " + error),
        () -> System.out.println("complete")
    );
```

This example prints

```
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
next: 1
next: 2
next: 3
complete
```

It is very important to understand that `repeat` operator resubscribes to source `Observable` when source `Observable` sequence completes. Let's rewrite example above using `Observable.create`.

```
Observable.<Integer>create(subscriber -> {

    //Same as Observable.just(1, 2, 3) but with output message
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
})
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

This example prints

```
Subscribed
next: 1
next: 2
next: 3
Subscribed
next: 1
next: 2
next: 3
Subscribed
next: 1
next: 2
```

```
next: 3
complete
```

When using operator chaining it is important to know that `repeat` operator repeats **whole sequence** rather than preceding operator.

```
Observable.<Integer>create(subscriber -> {
    System.out.println("Subscribed");
    subscriber.onNext(1);
    subscriber.onNext(2);
    subscriber.onNext(3);
    subscriber.onCompleted();
})
        .map(value -> value * 2) //First chain operator
        .map(value -> "modified " + value) //Second chain operator
        .repeat(3)
        .subscribe(
                next -> System.out.println("next: " + next),
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete")
        );
```

This example prints

```
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
Subscribed
next: modified 2
next: modified 4
next: modified 6
complete
```

This example shows that `repeat` operator repeats whole sequence resubscribing to `Observable` rather than repeating last `map` operator and it doesn't matter in which place in the sequence `repeat` operator used.

This sequence

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .repeat(3)
    .subscribe(
        /*....*/
    );
```

is equal to this sequence

```
Observable.<Integer>create(subscriber -> {
        //...
    })
    .repeat(3)
    .map(value -> value * 2) //First chain operator
    .map(value -> "modified " + value) //Second chain operator
    .subscribe(
        /*....*/
    );
```

Read Operators online: https://riptutorial.com/rx-java/topic/2316/operators

Chapter 6: Retrofit and RxJava

Examples

Set up Retrofit and RxJava

Retrofit2 comes with support for multiple pluggable execution mechanisms, one of them is RxJava.

To use retrofit with RxJava you first need to add the Retrofit RxJava adapter to your project:

```
compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'
```

then you need to add the adapter when building your retrofit instance:

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.example.com")
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
    .build();
```

In your interface when you define the API the return type should be `Observable` eg:

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Observable<List<Repo>> listRepos(@Path("user") String user);
}
```

You can also use `Single` instead of `Observable`.

Making serial requests

RxJava is handy when making serial request. If you want to use the result from one request to make another you can use the `flatMap` operator:

```
api.getRepo(repoId).flatMap(repo -> api.getUser(repo.getOwnerId())
    .subscribe(/*do something with the result*/);
```

Making parallel requests

You can use the `zip` operator to make request in parallel and combine the results eg:

```
Observable.zip(api.getRepo(repoId1), api.getRepo(repoId2), (repo1, repo2) ->
    {
        //here you can combine the results
    }).subscribe(/*do something with the result*/);
```

Read Retrofit and RxJava online: https://riptutorial.com/rx-java/topic/2950/retrofit-and-rxjava

Chapter 7: RxJava2 Flowable and Subscriber

Introduction

This topic shows examples and documentation with regard to the reactive concepts of Flowable and Subscriber that were introduced in rxjava version2

Remarks

the example needs rxjava2 as a dependency, the maven coordinates for the used version are:

```
    <dependency>
        <groupId>io.reactivex.rxjava2</groupId>
        <artifactId>rxjava</artifactId>
        <version>2.0.8</version>
    </dependency>
```

Examples

producer consumer example with backpressure support in the producer

The `TestProducer`from this example produces `Integer`objects in a given range and pushes them to its `Subscriber`. It extends the `Flowable<Integer>` class. For a new subscriber, it creates a `Subscription` object whose `request(long)` method is used to create and publish the Integer values.

It is important for the `Subscription` that is passed to the `subscriber` that the `request()` method which calls `onNext()`on the subscriber can be recursively called from within this `onNext()` call. To prevent a stack overflow, the shown implementation uses the `outStandingRequests` counter and the `isProducing` flag.

```
class TestProducer extends Flowable<Integer> {
    static final Logger logger = LoggerFactory.getLogger(TestProducer.class);
    final int from, to;

    public TestProducer(int from, int to) {
        this.from = from;
        this.to = to;
    }

    @Override
    protected void subscribeActual(Subscriber<? super Integer> subscriber) {
        subscriber.onSubscribe(new Subscription() {

            /** the next value. */
            public int next = from;
            /** cancellation flag. */
            private volatile boolean cancelled = false;
            private volatile boolean isProducing = false;
            private AtomicLong outStandingRequests = new AtomicLong(0);
```

```
            @Override
            public void request(long n) {
                if (!cancelled) {

                    outStandingRequests.addAndGet(n);

                    // check if already fulfilling request to prevent call  between request()
an subscriber .onNext()
                    if (isProducing) {
                        return;
                    }

                    // start producing
                    isProducing = true;

                    while (outStandingRequests.get() > 0) {
                        if (next > to) {
                            logger.info("producer finished");
                            subscriber.onComplete();
                            break;
                        }
                        subscriber.onNext(next++);
                        outStandingRequests.decrementAndGet();
                    }
                    isProducing = false;
                }
            }

            @Override
            public void cancel() {
                cancelled = true;
            }
        });
    }
}
```

The Consumer in this example extends `DefaultSubscriber<Integer>` and on start and after consuming an Integer requests the next one. On consuming the Integer values, there is a little delay, so the backpressure will be built up for the producer.

```
class TestConsumer extends DefaultSubscriber<Integer> {

    private static final Logger logger = LoggerFactory.getLogger(TestConsumer.class);

    @Override
    protected void onStart() {
        request(1);
    }

    @Override
    public void onNext(Integer i) {
        logger.info("consuming {}", i);
        if (0 == (i % 5)) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException ignored) {
                // can be ignored, just used for pausing
            }
```

```
        }
        request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        logger.error("error received", throwable);
    }

    @Override
    public void onComplete() {
        logger.info("consumer finished");
    }
}
```

in the following main method of a test class the producer and consumer are created and wired up:

```
public static void main(String[] args) {
    try {
        final TestProducer testProducer = new TestProducer(1, 1_000);
        final TestConsumer testConsumer = new TestConsumer();

        testProducer
                .subscribeOn(Schedulers.computation())
                .observeOn(Schedulers.single())
                .blockingSubscribe(testConsumer);

    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

When running the example, the logfile shows that the consumer runs continuously, while the producer only gets active when the internal Flowable buffer of rxjava2 needs to be refilled.

Read RxJava2 Flowable and Subscriber online: https://riptutorial.com/rx-java/topic/9810/rxjava2-flowable-and-subscriber

Chapter 8: Schedulers

Examples

Basic Examples

Schedulers are an RxJava abstraction about processing unit. A scheduler can be backed by a Executor service, but you can implement your own scheduler implementation.

A `Scheduler` should meet this requirement :

- Should process undelayed task sequencially (FIFO order)
- Task can be delayed

A `Scheduler` can be used as parameter in some operators (example : `delay`), or used with the `subscribeOn` / `observeOn` method.

With some operator, the `Scheduler` will be used to process the task of the specific operator. For example, `delay` will schedule a delayed task that will emit the next value. This is a `Scheduler` that will retain and execute it later.

The `subscribeOn` can be used once per `Observable`. It will define in which `Scheduler` the code of the subscription will be executer.

The `observeOn` can be used multiple times per `Observable`. It will define in which `Scheduler` will be used to execute all tasks defined **after** the `observeOn` method. `observeOn` will help you to perform thread hopping.

subscribeOn specific Scheduler

```
// this lambda will be executed in the `Schedulers.io()`
Observable.fromCallable(() -> Thread.currentThread().getName())
          .subscribeOn(Schedulers.io())
          .subscribe(System.out::println);
```

observeOn with specific Scheduler

```
Observable.fromCallable(() -> "Thread -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.io())
         .map(str -> str + " -> " + Thread.currentThread().getName())
          // next tasks will be executed in the computation scheduler
         .observeOn(Schedulers.computation())
         .map(str -> str + " -> " + Thread.currentThread().getName())
         // next tasks will be executed in the io scheduler
         .observeOn(Schedulers.newThread())
         .subscribe(str -> System.out.println(str + " -> " +
Thread.currentThread().getName()));
```

Specifying a specific Scheduler with an operator

Some operators can take a `Scheduler` as parameter.

```
Observable.just(1)
          // the onNext method of the delay operator will be executed in a new thread
          .delay(1, TimeUnit.SECONDS, Schedulers.newThread())
          .subscribe(System.out::println);
```

Publish To Subscriber:

```
TestScheduler testScheduler = Schedulers.test();
EventBus sut = new DefaultEventBus(testScheduler);
TestSubscriber<Event> subscriber = new TestSubscriber<Event>();
sut.get().subscribe(subscriber);
sut.publish(event);
testScheduler.advanceTimeBy(1, TimeUnit.SECONDS);
```

Thread Pool:

```
this.poolName = schedulerFig.getIoSchedulerName();
final int poolSize = schedulerFig.getMaxIoThreads();
final BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(poolSize);
final MaxSizeThreadPool threadPool = new MaxSizeThreadPool( queue, poolSize );
this.scheduler = Schedulers.from(threadPool);
```

Web Socket Observable:

```
final Subscription subscribe = socket.webSocketObservable()
        .subscribeOn(Schedulers.io())
        .doOnNext(new Action1<RxEvent>() {
            @Override
            public void call(RxEvent rxEvent) {
                System.out.println("Event: " + rxEvent);
            }
        })
        .subscribe();
```

Read Schedulers online: https://riptutorial.com/rx-java/topic/2321/schedulers

Chapter 9: Subjects

Syntax

- Subject<T, R> subject = AsyncSubject.create(); // Default AsyncSubject
- Subject<T, R> subject = BehaviorSubject.create(); // Default BehaviorSubject
- Subject<T, R> subject = PublishSubject.create(); // Default PublishSubject
- Subject<T, R> subject = ReplaySubject.create(); // Default ReplaySubject
- mySafeSubject = new SerializedSubject(unSafeSubject); // Convert an unsafeSubject to a safeSubject - generally for multi threaded Subjects

Parameters

Parameters	Details
T	Input type
R	Output type

Remarks

This documentation provides details and explanations about `Subject`. For more information and further reading, please visit the official documentation.

Examples

Basic Subjects

A `Subject` in RxJava is a class that is both an `Observable` and an `Observer`. This basically means that it can act as an `Observable` and pass inputs to subscribers and as an `Observer` to get inputs from another Observable.

```
Subject<String, String> subject = PublishSubject.create();
subject.subscribe(System.out::print);
subject.onNext("Hello, World!");
```

The above prints "Hello, World!" to console using `Subjects`.

Explanation

1. The first line of code defines a new `Subject` of type `PublishSubject`

```
Subject<String, String> subject = PublishSubject.create();
    |      |       |       |                  |
```

```
subject<input, output>  name   = default publish subject
```

2. The second line subscribes to the subject, showing the `Observer` behaviour.

```
subject.subscribe(System.out::print);
```

   This enables the `Subject` to take inputs like a regular subscriber

3. The third line calls the `onNext` method of the subject, showing the `Observable` behaviour.

```
subject.onNext("Hello, World!");
```

   This enables the `Subject` to give inputs to all subscribing to it.

Types

A `Subject` (in RxJava) can be of any of these four types:

- AsyncSubject
- BehaviorSubject
- PublishSubject
- ReplaySubject

Also, a `Subject` can be of type `SerializedSubject`. This type ensures that the `Subject` does not violate to the *Observable Contract* (which specifies that all calls must be Serialized)

Further reading:

- To Use or Not to Use Subject from Dave Sexton's blog

PublishSubject

`PublishSubject` emits to an `Observer` only those items that are emitted by the source `Observable` subsequent to the time of the subscription.

A simple `PublishSubject` example:

```
Observable<Long> clock = Observable.interval(500, TimeUnit.MILLISECONDS);
Subject<Long, Long> subjectLong = PublishSubject.create();

clock.subscribe(subjectLong);

System.out.println("sub1 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub1 -> " + l));
Thread.sleep(3000);
System.out.println("sub2 subscribing...");
subjectLong.subscribe(l -> System.out.println("sub2 -> " + l));
Thread.sleep(5000);
```

Output:

```
sub1 subscribing...
sub1 -> 0
sub1 -> 1
sub2 subscribing...
sub1 -> 2
sub2 -> 2
sub1 -> 3
sub2 -> 3
```

In the above example, a `PublishSubject` subscribes to an `Observable` which acts like a clock, and emits items(Long) every 500 milli seconds. As seen in the output, the `PublishSubject` passes on the vales it gets from the source (`clock`) to its subscribers(`sub1` and `sub2`).

A `PublishSubject` can start emitting items as soon as it is created, without any observer, which runs the risk of one or more items being lost till a observer can sunscribe.

```
createClock(); // 3 lines moved for brevity. same as above example

Thread.sleep(5000); // introduces a delay before first subscribe

sub1andsub2(); // 6 lines moved for brevity. same as above example
```

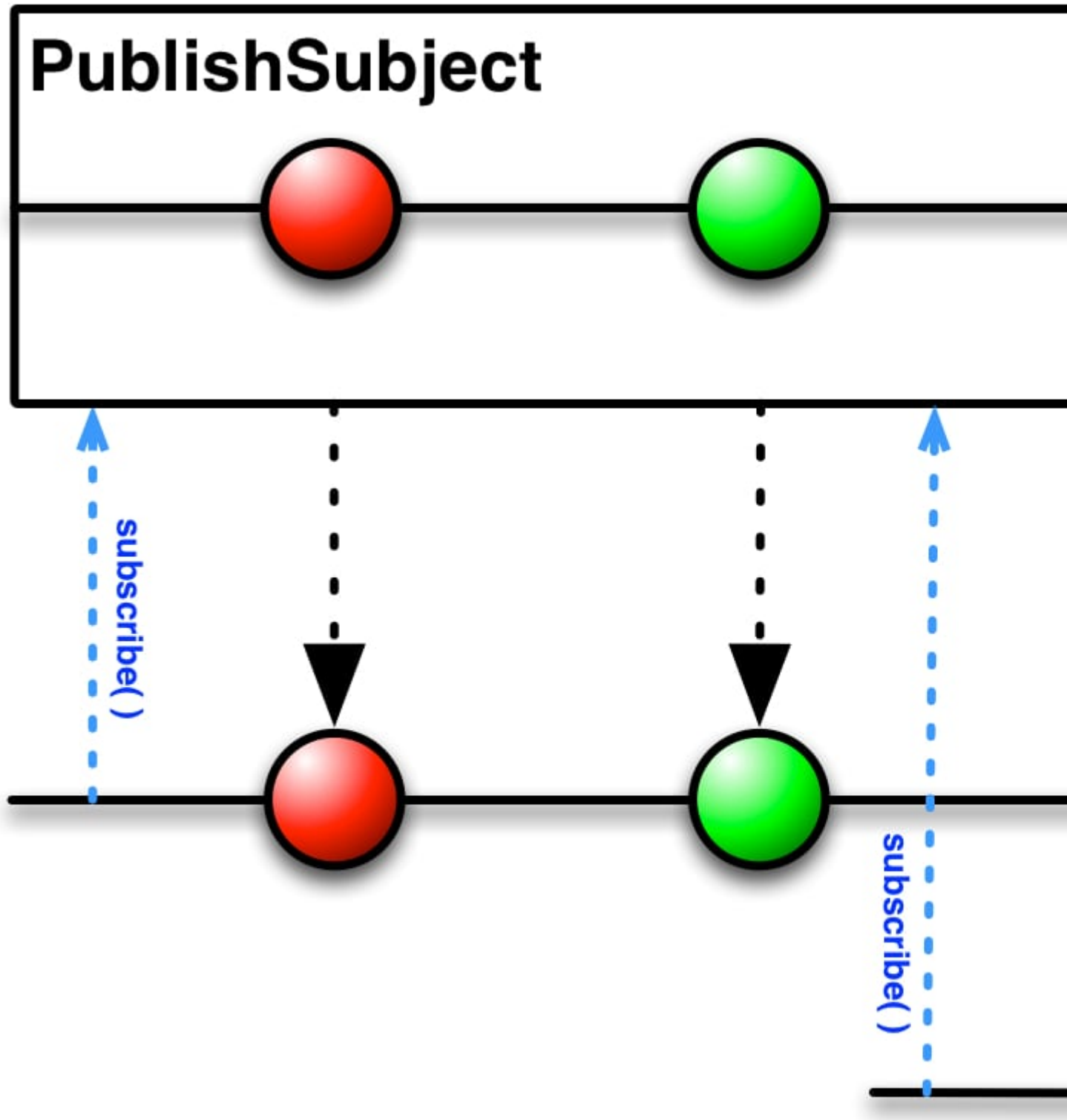
Output:

```
sub1 subscribing...
sub1 -> 10
sub1 -> 11
sub2 subscribing...
sub1 -> 12
sub2 -> 12
sub1 -> 13
sub2 -> 13
```

Notice that `sub1` emits values starting from `10`. The 5 second delay introduced caused a *loss* of items. These cannot be reproduces. This essentially makes `PublishSubject` a `Hot Observable`.

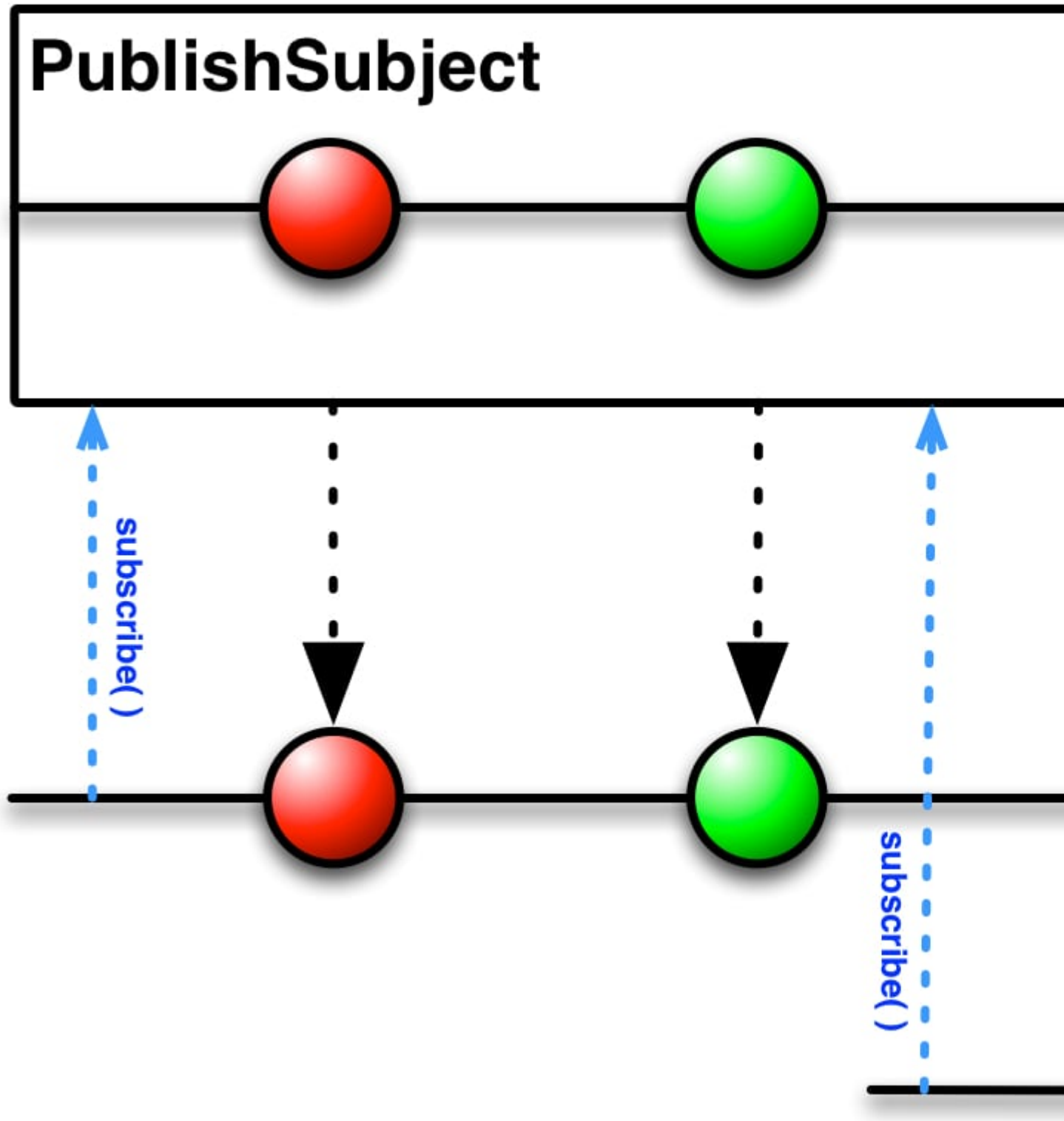
Also, note that if an observer subscribes to the `PublishSubject` after it has emitted ***n*** items, these ***n*** items *cannot* be reproduced for this observer.

Below is the marble diagram of `PublishSubject`



The `PublishSubject` emits items to all that have subscribed, at any point of time before the `onCompleted` of the source `Observable` is called.

If the source `Observable` terminates with an error, the `PublishSubject` will not emit any items to subsequent observers, but will simply pass along the error notification from the source Observable.



Use Case

Suppose you want to create an application that will monitor the stock prices of a certain company and forward it to all clients who request for it.

```
/* Dummy stock prices */
Observable<Integer> prices = Observable.just(11, 12, 14, 11, 10, 12, 15, 11, 10);

/* Your server */
PublishSubject<Integer> watcher = PublishSubject.create();
/* subscribe to listen to stock price changes and push to observers/clients */
```

```
prices.subscribe(watcher);

/* Client application */
stockWatcher = getWatcherInstance(); // gets subject
Subscription steve = stockWatcher.subscribe(i -> System.out.println("steve watching " + i));
Thread.sleep(1000);
System.out.println("steve stops watching");
steve.unsubscribe();
```

In the above example use case, the `PublishSubject` acts as a bridge to pass on the values from your server to all the clients that subscribe to your `watcher`.

Further reading:

- PublishSubject javadocs
- Blog by Thomas Nield (Advanced reading)

Read Subjects online: https://riptutorial.com/rx-java/topic/3287/subjects

Chapter 10: Unit Testing

Remarks

Because all the Schedulers methods are static, unit tests utilizing the RxJava hooks cannot be ran in parallel on the same JVM instance. If they where, one TestScheduler would be removed in the middle of a unit test. That is basically the downside of using the Schedulers class.

Examples

TestSubscriber

TestSubscribers allow you to avoid the work creating your own Subscriber or subscribe Action<?> to verify that certain values where delivered, how many there are, if the Observable completed, an exception was raised and a whole lot more.

Getting Started

This example just shows an assertion that the values 1,2,3 and 4 where passed into the Observable via onNext.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3,4); // Success
```

`assertValues` asserts that the count is correct. If you were to only pass some of the values, the assert would fail.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
ts.assertValues(1,2,3); // Fail
```

`assertValues` uses the `equals` method when doing asserts. This lets you easily test classes that are treated as data.

```
TestSubscriber<Object> ts = TestSubscriber.create();
Observable.just(new Object(), new Object()).subscribe(ts);
ts.assertValues(new Object(), new Object()); // Fail
```

This example shows a class that has a equals defined and asserting the values from the Observable.

```
public class Room {

    public String floor;
```

```
    public String number;

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (o instanceof Room) {
            Room that = (Room) o;
            return (this.floor.equals(that.floor))
                    && (this.number.equals(that.number));
        }
        return false;
    }
}

TestSubscriber<Room> ts = TestSubscriber.create();
Observable.just(new Room("1", "10")).subscribe(ts);
ts.assertValue(new Room("1", "10"); // Success
```

Also take note that we use the shorter `assertValue` because we only need to check for one item.

Getting all events

If need be you can also ask for all the events as a list.

```
TestSubscriber<Integer> ts = TestSubscriber.create();
Observable.just(1,2,3,4).subscribe(ts);
List<Integer> onNextEvents = ts.getOnNextEvents();
List<Throwable> onErrorEvents = ts.getOnErrorEvents();
List<Notification<Integer>> onCompletedEvents = ts.getOnCompletedEvents();
```

Asserting on events

If you want to do more extensive tests on your events, you can combine `getOnNextEvents` (or `getOn*Events`) with your favorite assertion library:

```
Observable<Integer> obs = Observable.just(1,2,3,4)
    .filter( x -> x % 2 == 0);

// note that we instanciate TestSubscriber via the constructor here
TestSubscriber<Integer> ts = new TestSubscriber();
obs.subscribe(ts);

// Note that we are not using Observable#forEach here
// but java.lang.Iterable#forEach.
// You should never use Observable#forEach unless you know
// exactly what you're doing
ts.getOnNextEvents()
    .forEach( integer -> assertTrue(integer % 2 == 0));
```

Testing Observable#error

You can make sure that the correct exception class is emitted:

```
Observable<Integer> obs = Observable.error(new Exception("I am a Teapot"));

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertError(Exception.class);
```

You can also make sure that the exact Exception was thrown:

```
Exception e = new Exception("I am a Teapot");
Observable<Integer> obs = Observable.error(e);

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertError(e);
```

TestScheduler

TestSchedulers allows you to control time and execution of Observables instead of having to do busy waits, joining threads or anything to manipulate system time. This is VERY important if you want to write unit tests that are predictable, consistent and fast. Because you are manipulating time, there is no longer the chance that a thread got starved, that your test fails on a slower machine or that you waste execution time busy waiting for a result.

TestSchedulers can be provided via the overload that takes a Scheduler for any RxJava operations.

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);

try {
    Thread.sleep(TimeUnit.SECONDS.toMillis(11));
} catch (InterruptedException ignored) { }
subscriber.assertValues(1,2,3); // fails

testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success
```

The TestScheduler is pretty basic. It only consists of three methods.

```
testScheduler.advanceTimeBy(amount, timeUnit);
testScheduler.advanceTimeTo(when, timeUnit);
testScheduler.triggerActions();
```

This lets you manipulate when the TestScheduler should fire all the actions pertaining to some time in the future.

While passing the scheduler works, this is not how the TestScheduler is commonly used because of how ineffective it is. Passing schedulers into classes ends up providing a lot of extra code for little gain. Instead, you can hook into RxJava's Schedulers.io()/computation()/etc. This is done with RxJava's Hooks. This lets you define what gets returned from a call from one of the Schedulers methods.

```
public final class TestSchedulers {

    public static TestScheduler test() {
        final TestScheduler testScheduler = new TestScheduler();
        RxJavaHooks.reset();
        RxJavaHooks.setOnComputationScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnIOScheduler((scheduler) -> {
            return testScheduler;
        });
        RxJavaHooks.setOnNewThreadScheduler((scheduler) -> {
            return testScheduler;
        });
        return testScheduler;
    }
}
```

This class allows the user to get the test scheduler that will be hooked up for all calls to Schedulers. A unit test would just need to get this scheduler in its setup. It is highly recommend aquiring it in the setup and not as any plain old field because your TestScheduler might try to triggerActions in from another unit test when you advance time. Now our example above becomes

```
TestScheduler testScheduler = new TestScheduler();
TestSubscriber<Integer> subscriber = TestSubscriber.create();
Observable.just(1,2,3)
          .delay(10, TimeUnit.SECONDS, testScheduler)
          .subscribe(subscriber);
testScheduler.advanceTimeBy(9, TimeUnit.SECONDS);
subscriber.assertValues(); // success (delay hasn't finished)
testScheduler.advanceTimeBy(10, TimeUnit.SECONDS);
subscriber.assertValues(1,2,3); // success (delay has finished)
```

That's how you can effectively remove the system clock from your unit test (at least as far as RxJava is concerned)

Read Unit Testing online: https://riptutorial.com/rx-java/topic/5207/unit-testing

Credits

S. No	Chapters	Contributors
1	Getting started with rx-java	Buttink, Community, dimsuz, Dmitry Avtonomov, Hans Wurst, hello_world, Omar Al Halabi, Saulius Next, Sneh Pandya, svarog, Tom
2	Android with RxJava	akarnokd, Athafoud, Daniele Segato, Eugen Martynov, Geng Jiawen, Sneh Pandya
3	Backpressure	akarnokd, Bartek Lipinski, Chris A, Cristian, dwursteisen, Niklas, Sebas LG
4	Observable	Aki K, dwursteisen, hello_world, JonesV
5	Operators	dwursteisen, hello_world, svarog, Vadeg
6	Retrofit and RxJava	LordRaydenMK
7	RxJava2 Flowable and Subscriber	P.J.Meisch
8	Schedulers	dwursteisen, Gal Dreiman
9	Subjects	hello_world, mavHarsha
10	Unit Testing	Buttink, Sir Celsius