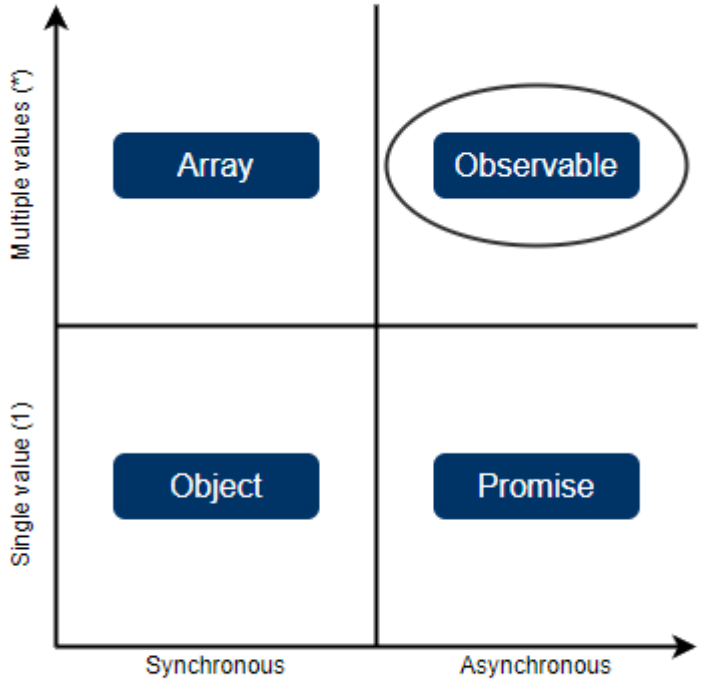
# Contents

## Reactive programming

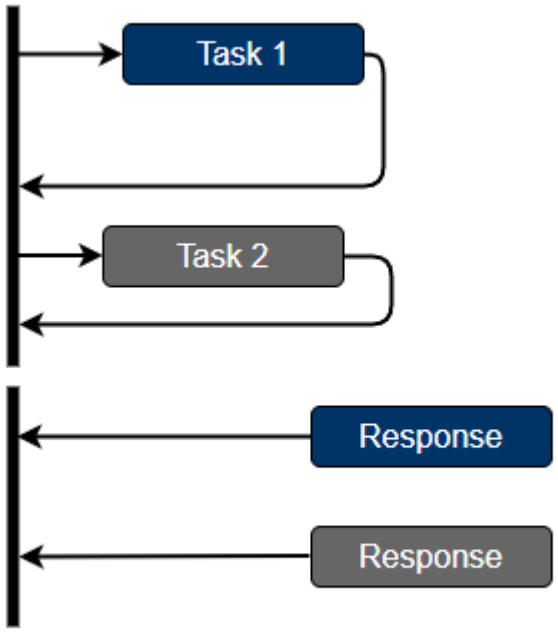
Реактивне програмування можна найпростіше визначити як асинхронне програмування зі спостережуваними потоками.




Reactive programming

# Asynchronous programming

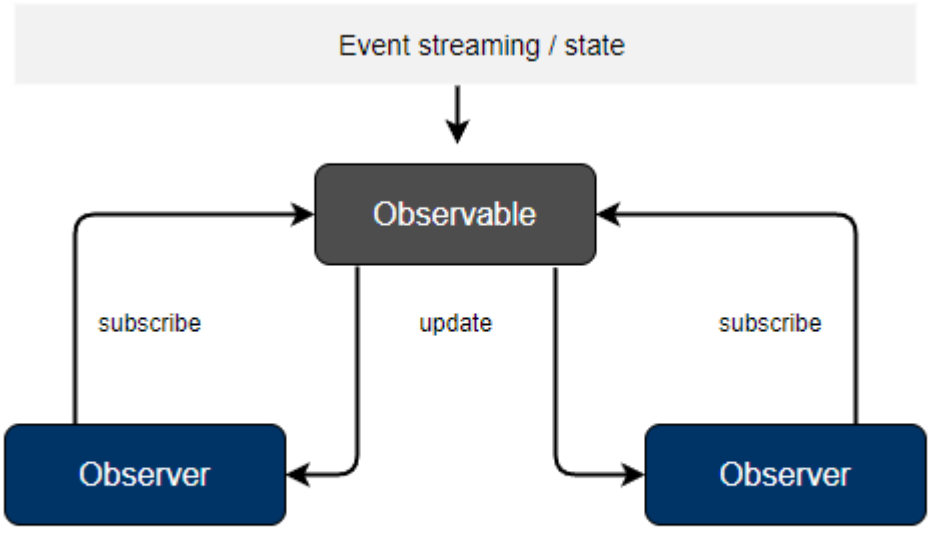
У контексті реактивного програмування асинхронне програмування є складним терміном. У традиційному розумінні, це програмування у неблокуючий спосіб, коли довготривалі завдання виконуються окремо від основного потоку програми. В іншому сенсі, це стиль програмування, керований подіями, де самі події є асинхронними і можуть надходити в будь-який момент часу.




Abstract level concept of asynchronous programming

# Observable

Observable відноситься до об'єкта, на який може підписатися будь-яка кількість спостерігачів, зацікавлених у його стані. Потім спостережуваний об'єкт надсилає оновлення своїм спостерігачам, коли стан об'єкта змінюється або коли настає якась подія. Це класичний патерн спостерігача.




Observable and observer

# Stream

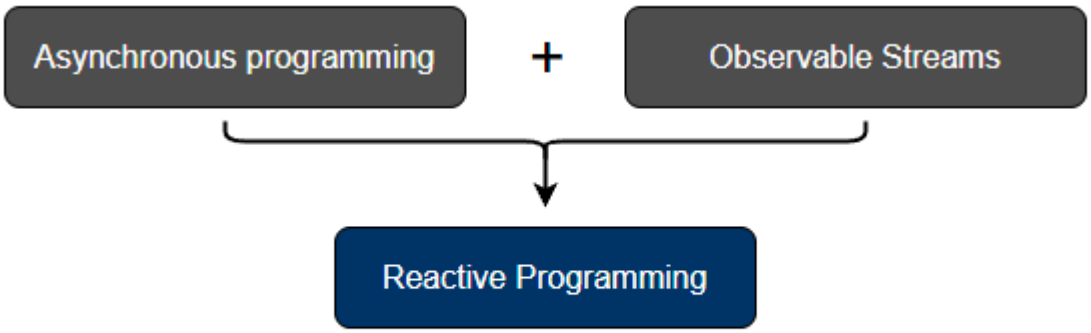
## Stream

A **stream** (or data stream) can be thought of as an ordered sequence of events. These events may arrive at any point in time, may have no defined beginning or end, and are often generated by sources external to our application.


Data stream

# Asynchronous programming with observable streams

Якщо переосмислити ці терміни, то спостережуваний потік - це послідовність подій, на яку можна підписатися і спостерігачі якої будуть сповіщатися про кожну вхідну подію. Асинхронне програмування зі спостережуваними потоками - це спосіб асинхронної обробки потоків даних за допомогою патерну спостерігачів, що базується на поштовхах, для забезпечення швидкої реакції програми.

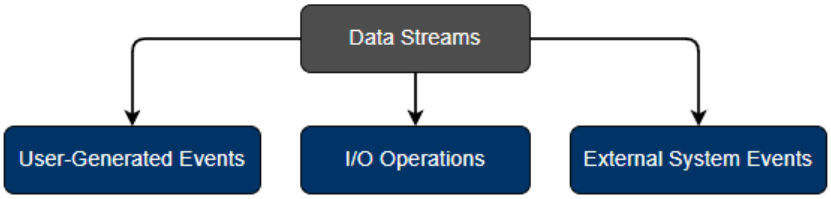



Asynchronous programming with observable streams

# Forms of data streams to handle

Android developers have undoubtedly dealt with many forms of data streams. Let's think of some examples of data streams that we might want to handle using reactive programming:

- **User-generated Events**: Click events, swipe events, keyboard input, and device rotations are just a few examples of events initiated by the user. If we consider each of these events across a timeline, we can see how they would form an ordered sequence of events, or a dynamic and potentially infinite data stream.
- **I/O Operations**: Network requests, file reads and writes, database accesses, and other I/O operations are probably the most common types of data streams we will apply reactive principles to in practice. I/O operations are asynchronous since they take some significant and uncertain amount of time to complete. The response, whether it be JSON, a byte array, or some plain-old Java objects, can then be treated as a data stream.




Different forms of data streams

- **External System Events**: Push notifications from the server, General Circulation Model (GCM) broadcasts, updates from device sensors, and events generated from external producers are similar to user-generated events in their dynamic and potentially infinite nature.
- **Just About Anything**: Really, just about anything can be modeled as a data stream. That's the reactive mantra. A single, scalar value or a list of objects, static data, or dynamic events can be made into a data stream in the reactive world.

> All of the previous examples are events that we want our program to take immediate action on. The point of reactive programming is to be reactive.

With reactive programming, data is a first-class citizen. The flow of data will drive the program as opposed to the thread of execution. The library we are using, RxJava, provides a thorough set of APIs for dealing with these data streams concisely and elegantly.

# RxJava in Action

## Sample RxJava code

Let’s take a look at some RxJava in action:

```
Observable<Car> carsObservable = getBestSellingCarsObservable();

    carsObservable.subscribeOn(Schedulers.io())
    .filter(car -> car.type == Car.Type.ALL_ELECTRIC)
    .filter(car -> car.price < 90000)
    .map(car -> car.year + " " + car.make + " " + car.model)
    .distinct()
    .take(5)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::updateUi);
```

Наведений вище код отримає 5 найкращих електромобілів вартістю до $90 000, а потім оновить інтерфейс користувача цими даними.

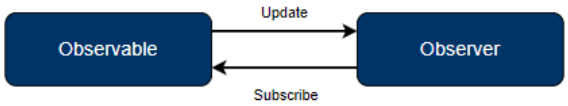
Тут багато чого відбувається, тому давайте подивимось, як він це робить по пунктах:

- **Line 1:** It all starts with our `Observable`, the source of our data. The `Observable` waits for an observer to subscribe to it, at which point it does some work and pushes data to its `Observer`. We intentionally abstract away the creation of the `carsObservable` here, and we'll cover how to create an `Observable` in the next chapter. For now, let's assume the work that the `Observable` will be doing is querying a REST API to get an ordered list of the top-selling cars.
- **Lines 3–10:** form essentially a data pipeline. Items emitted by the `Observable` will travel through the pipeline from top to bottom. Each of the functions in the pipeline is what we call an `Operator`; `Operators` modify the `Observable` stream, allowing us to massage the data until we get what we want. In our example, we start with an ordered list of all top-selling car models. By the end of our pipeline, we have just the top 5 selling electric cars that are under $90,000. The massaging of the data happens as follows:
    - **Line 3:** The `.subscribeOn(...)` function tells the `Observable` to do its work on a background thread. We do this so that we don't block Android's main UI thread with the network request performed by the `Observable`.
    - **Line 4:** The `.filter(...)` function filters the stream down to only items that represent electric cars. Any `Car` object that does not meet this criterion is discarded at this point and does not continue down the stream.
    - **Line 5:** The `.filter(...)` function further filters the electric cars to those below $90,000.
    - **Line 6:** The `.map(...)` function transforms the element from a `Car` object to a `String` consisting of the year, model, and make. From here on, this `String` takes the place of the `Car` in the stream.
    - **Line 7:** The `.distinct()` function removes any elements that we've already encountered before. Note that this uniqueness requirement applies to our `String` values and not our `Car` instances, which no longer exist at this point in our chain because of the previous `.map(...)` call.
    - **Line 8:** The `.take(5)` function ensures that at most **five elements** will be passed on. If five elements are emitted, the stream completes and emits no more items.
    - **Line 9**: The `.observeOn(...)` function switches our thread of execution back to the main UI thread. So far, we've been working on the background thread specified in **Line 3**. Now that we need to manipulate our views, however, we need to be back on the UI thread.
    - **Line 10**: The `.subscribe(...)` function is both the beginning and end of our data stream. It is the beginning because `.subscribe()` prompts the `Observable` to do its work and start emitting items. However, the parameter we pass to it is the `Observable`, which represents the end of the pipeline and defines some action to perform when it receives an item. In our case, the action will be to update the UI. The item received is the result of all of the transformations performed by the upstream operators.

# RxJava's Essential Characteristics

## Observer pattern

RxJava is a classic example of the observer pattern. We start with the `Observable`, which is the source of our data. Then we have one or more `Observer` elements, which subscribe to the `Observable` and are notified when there is a new event. This allows for a push-based mechanism, which is usually far more ideal than continually polling for new events.




Observer pattern in RxJava

RxJava adds to the traditional observer pattern by having the `Observable` signal completion and errors along with regular events, which we'll see in RxJava Core Components.

## Iterator pattern

With a traditional iterator pattern, the iterator *pulls* data from the underlying collection, which would implement some sort of Iterable interface. What Erik Meijer essentially did in creating Rx was flip the iterator pattern on its head, turning it into a *push*-based pattern.

The `Observable` was designed to be the dual of the `Iterable`. Instead of pulling data out of an `Iterable` with `.next()`, the `Observable` pushes data to an `Observer` using `.onNext()`. So, where the iterator pattern uses synchronous pulling of data, RxJava allows for asynchronous pushing of data, allowing code to be truly reactive.

## Functional programming

One of the most important aspects of RxJava is that it uses functional programming, in particular, with its `Operators`. **Functional programming** is programming with pure functions.

### Pure functions

A **pure function** is one that satisfies the following two conditions:

1. The function always returns the same value given the same input (that is, it's deterministic). This implies that the function cannot depend on any global state—for example, a Java class's member variable—nor any external resource, such as a web service.


2. The function does not cause any side effects. This means that the function cannot mutate any input parameter, update any global state, or interact with any external resource.

### Advantages

Functional programming leads to code that is:

**Declarative**: Instead of dictating *how* something is done, as is done with imperative programming, RxJava uses a declarative approach by describing *what* should be done (through our use of operators). This declarative style maps much more closely to how humans think, as opposed to how a computer executes, and allows us to much more easily reason about what the program is doing.

**Thread-safe**: Because functional programming avoids state mutation, it is inherently thread-safe. We have none of the problems that we normally see when introducing concurrency—for example, synchronization issues, race conditions, resource contention. Concurrency can be supported safely and easily with RxJava.

**Testable**: Functional code becomes much easier to test because the functions are completely self-contained and deterministic. We don't need to mock the global state before executing unit tests. All we need is a set of input and expected output.

## Conciseness

The code below is extremely concise for how much it's doing. Imagine the number of lines that would be required to write this without RxJava.

```
Observable<Car> carsObservable = getBestSellingCarsObservable();

    carsObservable.subscribeOn(Schedulers.io())
    .filter(car -> car.type == Car.Type.ALL_ELECTRIC)
    .filter(car -> car.price < 90000)
    .map(car -> car.year + " " + car.make + " " + car.model)
    .distinct()
    .take(5)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::updateUi);
```

Getting the top five sub-$90,000 electric cars and updating the UI

Here are a few reasons behind the conciseness of the code:

1. A huge reason why the code is so concise is the use of lambdas. RxJava's functional style and lambdas go hand in hand. We recommend using lambdas in our Java/Android project regardless, but especially so if using RxJava. If you are not yet using Android Studio 3.+, we recommend using the Retrolambda library to add lambda support to the project. RxJava really isn't all that it can be without lambdas.

## Lazy vs. eager evaluation


Lazy evaluation has its pros and cons. In many cases, lazy evaluation is beneficial. For example, if the `Observable` queries a web service, we likely want the most up-to-date data, which means that execution should happen when the `Observer` subscribes, and not when the `Observable` is created. The same should apply for any `Observer` that subscribes to the `Observable`. No matter what time it subscribes, it should get the latest data.

However, if our `Observable` is retrieving data that will not change, then we do not want it to make a network round-trip for every subscriber. We'd prefer eager evaluation in this case and have the `Observable` get the data once and hold on to it. RxJava and its plentiful toolbox of APIs actually allows for this with the `.cache()` operator. In fact, we'll see how to dictate the laziness versus eagerness of an `Observable` more when we talk about `Observable` creation and cold versus hot `Observable`s in the chapter RxJava Core Components.

Lazy evaluation has its pros and cons. In many cases, lazy evaluation is beneficial. For example, if the Observable queries a web service, we likely want the most up-to-date data, which means that execution should happen when the Observer subscribes, and not when the Observable is created. The same should apply for any Observer that subscribes to the Observable. No matter what time it subscribes, it should get the latest data.

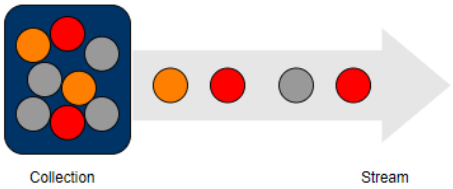
Лінива оцінка має свої плюси і мінуси. У багатьох випадках лінива оцінка вигідна. Наприклад, якщо спостережуваний об'єкт запитує веб-сервіс, ми, швидше за все, хочемо отримати найсвіжіші дані, а це означає, що виконання має відбуватися, коли об'єкт підписується, а не коли об'єкт створюється. Те ж саме має застосовуватися до будь-якого спостерігача, який підписується на спостережуваний об'єкт. Незалежно від того, коли він підписується, він повинен отримувати найсвіжіші дані.

## RxJava Core Components

# Streams

### Java's streams

If we think about it, a stream is not a new concept. A `Collection` in Java can be modeled as a stream where each element in the `Collection` is an item emitted in the stream.




A collection in Java modeled as a stream

On Android, click events can be a stream, location updates can be a stream, push notifications can be a stream, and so on.

Потоки в RxJava

У світі RxJava все можна моделювати як потік. Потік випромінює елементи з плином часу, і кожен елемент може бути перетворений або модифікований під час проходження через нього. Спостерігач або споживач може підписатися і виконати дію з кожного викиду, повернутого потоком. Крім того, потоки в RxJava дуже легко компонуються і навіть можуть бути об'єднані з іншими потоками для отримання бажаного результату.

Навіщо використовувати потоки RxJava?

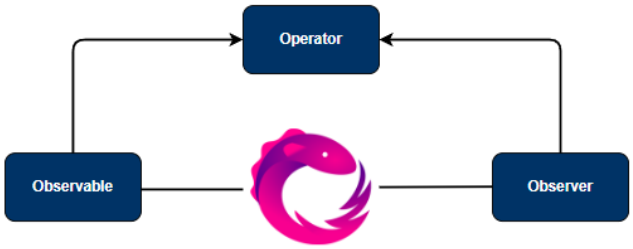
Якщо в RxJava все є потоком, а Java 8 підтримує потоки, то чому б просто не використовувати потоки Java 8?

RxJava дозволяє нам робити справжнє реактивне програмування на високому рівні, на відміну від потоків Java 8.

Інструментарій функціональних мовних конструкцій, таких як відображення, фільтрація та злиття, може бути спільним як для Java 8, так і для RxJava. Однак, з версією потоку RxJava набагато простіше працювати. Крім того, вона чудово справляється з асинхронністю.

# RxJava main constructs

Поняття потоку моделюється в RxJava за допомогою трьох основних конструкцій, які ми будемо називати "3 Os". Це спостережуваний об'єкт (Observable), спостерігач (Observer) та оператор (Operator).




The 3 Os of RxJava

The `Observable` is an entity that emits item(s) over time, or in other words, the stream. It can then be subscribed to at any point in time by an `Observer`, which will receive items that were pushed down along the stream. An operator, or sequence of operators, can then be inserted in between the `Observable` and `Observer` to perform actions that transform, filter, aggregate, and combine data emitted down the stream.

> The 3 Os of RxJava are:
>
> - The `Observable`, which emits items over time.
> - The `Observer`, which subscribes to receive those items.
> - The operator, which can be used to transform items emitted by the `Observable`, including the `Observable` itself.

# Observable and Observer

An `Observable` can emit any number of items. As mentioned, to receive or listen to these emitted items, an `Observer` needs to subscribe to the `Observable`. The `.subscribe()` method is defined by the `ObservableSource` interface, which is implemented by `Observable`.

```
1  public interface ObservableSource<T> {
2      void subscribe(Observer<? super T> observer);
3  }
```

ObservableSource interface

Once the `Observable` and `Observer` have been paired via the `.subscribe()` method, the `Observer` receives the following events through its defined methods:

- The `.onSubscribe()` method is called along with a `Disposable` object that may be used to unsubscribe from the `Observable` to stop receiving items.
- The `.onNext()` method is called when an item is emitted by the `Observable`.
- The `.onComplete()` method is called when the `Observable` has finished sending items.
- The `.onError()` method is called when an error is encountered within the `Observable`. The specific error is passed to this method.

## `.Next()`

The method `.onNext()` can be invoked zero, one, or multiple times by the `Observable` whereas the `.onComplete()` and `.onError()` methods are considered terminal events, meaning that if either of the two methods are called, no further method would be invoked on the `Observer`. This is a crucial contract of an `Observable` that must be enforced when creating an `Observable`.

The method `.subscribe()` is also an overloaded method of an `Observable`. Depending on the `Observable` being subscribed to, sometimes it is not necessary to provide a "full" `Observer` when subscribing. For example, if we know that the `Observable` will never invoke the `.onComplete()` method in the case of an infinite stream, we can choose one of the other overloaded methods. The other options are as follows, and note that all these versions will return a `Disposable` object:

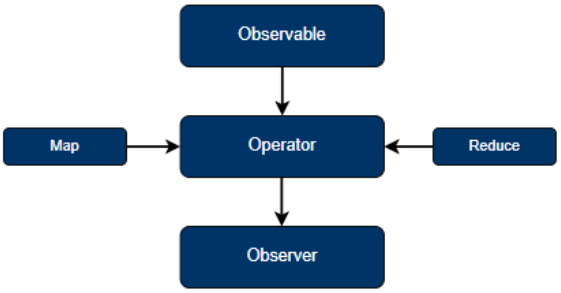
- The `.subscribe()` method: All methods are ignored. All errors will forward to the `RxJavaPlugins.onError()` handler. More information on this is in the chapter Error Handling.
- The `.subscribe(Consumer<? super T> onNext)` method: Only `.onNext()` events are received. All errors will be forwarded to the `RxJavaPlugins.onError()` handler.
- The `.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError)` method: Only `.onNext()` and `.onError()` events are received and `.onComplete()` is ignored.
- The `.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete)` method: All events are received.
- The `.subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete, Consumer<? super Disposable> onSubscribe)` method: All events are received. Additionally, a `Consumer` can be specified to receive the upstream's `Disposable` object.

Push vs. Pull

Дивлячись на визначення Observable та Observer, ми бачимо, що дизайн за своєю суттю є push-системою. Спостерігач реагує на отримання елемента, який був виштовхнутий на нього Спостерігачем, на якого він підписаний. Однак це не означає, що події, які надсилаються, є асинхронними. Насправді, за замовчуванням Observable є синхронним, що означає, що події будуть випромінюватися у потоці Observable у тому ж потоці, де викликається .subscribe().

# Operator

`Operator`s are very powerful constructs that allow us to declaratively modify item emissions of an `Observable` including the `Observable`/s themselves.




An abstraction of an Observable and Operator interaction

Through operator use, we can focus on the business logic that makes our applications interesting rather than concerning ourselves with low-level details of an imperative approach. Some of the most common operations found in functional programming, such as `map`, `filter`, `reduce`, and so on, can also be applied to an `Observable` stream.

## `.map()`: An example

Let's take a look at `.map()` as an example:

```
Observable<Integer> intObservable =
    Observable.create((ObservableOnSubscribe<Integer>) emitter -> {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
        emitter.onNext(4);
        emitter.onNext(5);
        emitter.onComplete();
    });

intObservable.map(val -> val * 3)
    .subscribe(i -> {
        // Will receive the following values in order: 3, 6, 9, 12, 15
    });
```

.map() operator

The code snippet above would take each emission from the `Observable` and multiply each by 3, producing the stream 3, 6, 9, 12, 15. Let's say we wanted to only receive even numbers. This can be achieved simply by chaining a `.filter()` operation.

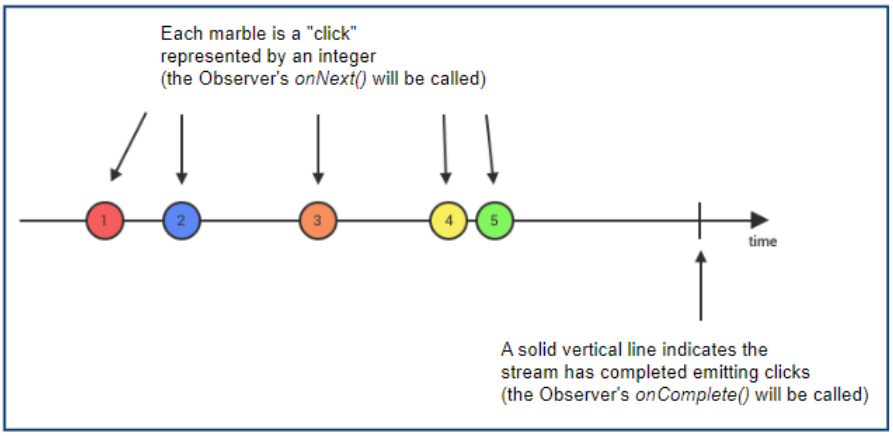
```
intObservable.map(val -> val * 3)
    .filter(val -> val % 2 == 0)
    .subscribe(i -> {
        // Will receive the following values in order: 6, 12
    });
```

# Marble Diagrams

Мармурова діаграма є поширеним візуальним засобом для Observables. Такі діаграми широко використовуються у вікі RxJava, і ми побачимо декілька з них протягом курсу.

## `OnComplete()`

Say we have an `Observable` emitting view clicks that are represented by an integer value:



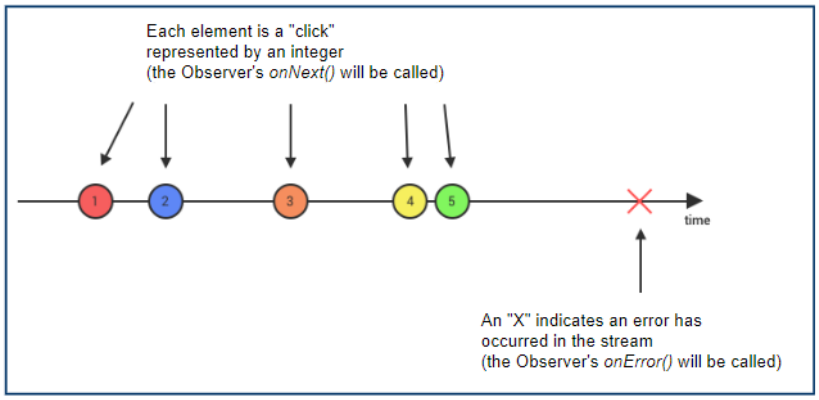

Marble diagram of an Observable completed successfully

The above marble diagram depicts clicks propagated down the stream as it is received over time. The clicks are represented by an integer, the value of which doesn't matter and the stream completes by invoking the `.onComplete()` method of an `Observer`.

> In Android, this successful completion might mean that the user has backgrounded the app or completed the desired action on the screen, and thus click events are no longer received.

## `OnError()`

On the other hand, say an error occurred to the `Observable` while view click events were being propagated. For example, a view animation failed and so no further clicks should be propagated. In this case, the `.onError()` method of an `Observer` would be invoked and no further events should occur down the stream. In this event, the diagram would display an error using an "X."

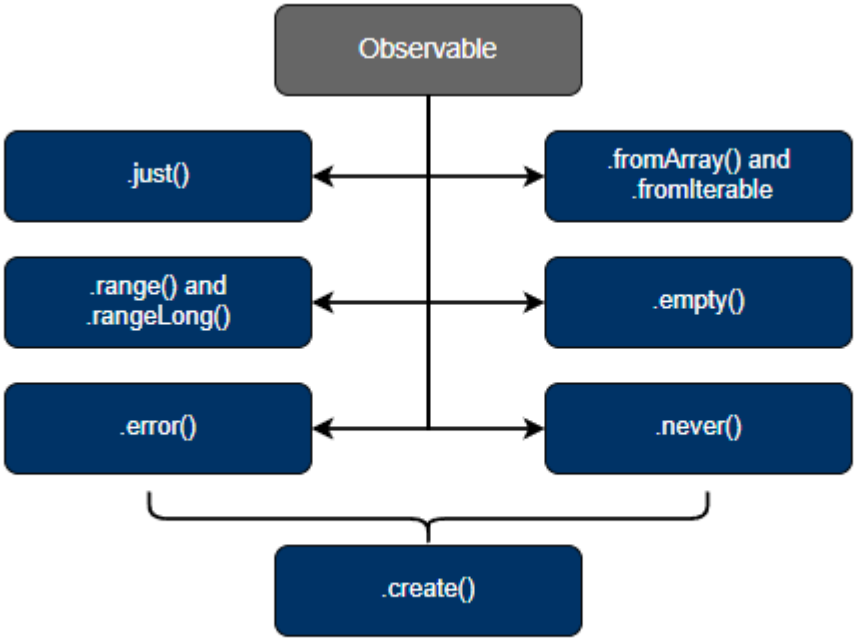



Marble diagram of an Observable with an error

# Creating an Observable

Різні способи створення Observable

Тепер, коли ми маємо базове, високорівневе розуміння різних компонентів RxJava, давайте занурimося в Observable трохи глибше, а саме в різні способи його створення. Ми розглянули найбільш багатослівний спосіб створення Observable (.create()). Однак, існують простіші та зручніші API, які нам доступні.




Different methods of creating Observables

## `.just()`

The `Observable.just(T item)` method is one of the most common ways to wrap any object to an `Observable`. It creates an `Observable` of type `T` that just emits the provided `item` via an `Observer`'s `.onNext()` method and then completes via `.onComplete()`. The `Observable.just()` is also overloaded, and we can specify anywhere from one to nine items to emit.

Example usage is below:

```
1  Observable.just(1, 2, 3).subscribe(val -> {
2      // val will be 1, 2, 3
3  });
```

Observable.just()

## `.fromArray()` and `.fromIterable()`

The `Observable.fromArray(T... items)` and `Observable.fromIterable(Iterable<? extends T> iterable)` methods are used to create an `Observable` from an array and an `Iterable` (for example `List`, `Map`, `Set`, and so on), respectively. Upon subscription, the resulting `Observable` will emit the items in the array or `Iterable` and complete after emitting all the values.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);

Integer[] array = new Integer[] {1, 2, 3};

Observable.fromArray(array).subscribe(val -> {
    // val will be 1, 2, 3
});

Observable.fromIterable(numbers).subscribe(val -> {
    // val will be 1, 2
});
```

Observable.fromArray() and Observable.fromIterable()

## `.range()` and `.rangeLong()`

The `Observable.range(int start, int count)` and `Observable.rangeLong(long start, long count)` methods create an `Observable<Integer>` and `Observable<Long>`, respectively. Both emit a range of values starting from `start` up to, but not including, `start + count`.

```
Observable.range(1,5).subscribe(val -> {
    // val will be 1, 2, 3, 4, 5
});

Observable.rangeLong(1,5).subscribe(val -> {
    // val will be 1, 2, 3, 4, 5
});
```

Observable.range() and Observable.rangeLong()

## `.empty()`

The `Observable.empty()` method returns no items and immediately invokes the `.onComplete()` method of an `Observer` when subscribed to. By itself, it's not very useful, so it's commonly used in conjunction with other operators.

```
Observable<String> empty = Observable.empty();
empty.subscribe(
        // returns no items and invokes onComplete()
);
```

Observable.empty()

### `.error()`

The `Observable.error(Throwable exception)` method wraps an exception and invokes the `.onError()` method of an `Observer` when subscribed to.

```
Observable err = Observable.error(new Exception("An Exception"));
err.subscribe(
    // invokes onError method
);
```

Observable.error()

### `.never()`

The method creates an `Observable` that never invokes any of an `Observer`'s methods when subscribed to. This is primarily used for testing purposes.

```
Observable.never().subscribe(val -> {
  // Never invokes any of the Observer's method
});
```

Observable.never()

### `.create()`

All of the above methods for creating an `Observable` can be replicated by using `.create()`. For example, `Observable.just()` can be implemented as following:

```
public static <T> Observable just(T item) {
    return Observable.create(emitter -> {
        emitter.onNext(item);
        emitter.onComplete();
    });
}
```

Implementing Observable.just() by using .create()

# Cold vs. Hot Observables

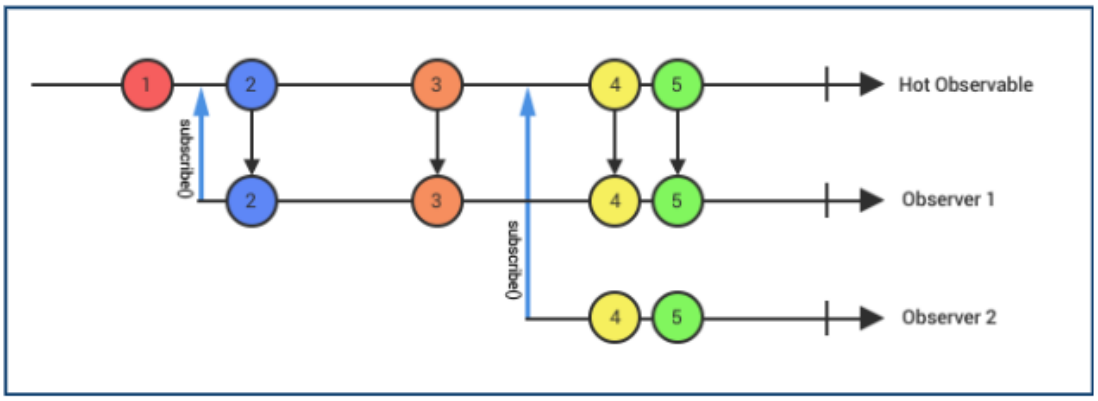
Cold observable

Observable вважається холодним, якщо він не надсилає активних повідомлень. Він починає випромінювати елементи лише тоді, коли на нього підписуються. Кожна підписка на холодну обсерваторію призводить до того, що вона починає випускати послідовність від початку до кінця для кожного спостерігача.

Загалом, холодний Observable - це те, що нам потрібно, якщо ми хочемо отримати повну копію подій, що випромінюються у потоці. Це те, що нам потрібно для операцій, які слід викликати лише за наявності підписки: мережеві операції, запити до бази даних, файловий ввід/вивід і так далі.

Hot observable (гарячий спостерігач)

Спостерігач отримує події, починаючи з моменту підписки, і не отримує жодної з подій, які були відправлені до підписки. Гаряча модель Observable обробляє тимчасові процеси: подію кліку, шину подій тощо.




Hot Observable

## Cold vs. hot Observables

The main differences between a cold and a hot `Observable` are as follows:

- While a cold `Observable` is subscriber-dependent, a hot `Observable` is not.
- A hot `Observable` has its own timeline and actively emits items, regardless of if there are any subscribers.

# Lazy Emissions

## `Observable.fromCallable()`

The `Observable.fromCallable(Callable<? extends T> callable)` method creates an `Observable` that, when subscribed to, invokes the function supplied and then emits the value returned by that function.

Constructing an `Observable` with `.fromCallable()` won't actually do the work specified inside the `Callable`. All it does is define the work to be done. As such, `.fromCallable()` is the ideal construction method to be used for any potentially long-running, UI-blocking operation. For example, we can wrap a network call returning a `User` object by returning the value of the network call inside the function provided to `.fromCallable()`.

```
Observable<User> observable = Observable.fromCallable(() -> {
    return apiService.getUserWithId(123);
});
observable.subscribe(user -> {
    // Got user with ID 123
});
```

Using Observable.fromCallable()

## `Observable.defer()`

The `Observable.defer(Callable <? extends ObservableSource<? extends T>> supplier)` method is another `Observable` creation method available for deferring a potentially long-running, UI-blocking operation.

The `.defer()` method creates an `Observable` that calls an `ObservableSource` factory to create an `Observable` for each new `Observer` that subscribes.

The above code for retrieving a `User` by `ID` can then be rewritten as:

```
Observable<User> observable = Observable.defer(() -> {
    return Observable.just(apiService.getUserWithId(123));
});
observable.subscribe(user -> {
    // Got user with ID 123
});
```

Using Observable.defer()

## `.fromCallable` vs. `.defer()`

Here are a few differences between `.fromCallable` and `.defer()`:

- The main distinction between `.defer()` and `.fromCallable()` is that the former returns an `Observable` in the supplied function.
- The main reason to use `.defer()` over `.fromCallable()` is that the creation of an `Observable` in itself is a blocking operation. However, these should be rare instances, and `.fromCallable()` should suffice.

Ось декілька відмінностей між .fromCallable і .defer():

Основна відмінність між .defer() та .fromCallable() полягає у тому, що перша повертає Observable у наданій функції.

Основною причиною використання .defer() замість .fromCallable() є те, що створення Observable саме по собі є блокувальною операцією. Однак, це мають бути рідкісні випадки, і .fromCallable() має бути достатньо.

## When to use `.fromCallable()` and `.defer()`?

The `.fromCallable()` and `.defer()` methods should be used if the evaluation of a value takes some time to compute and we would like to delay that computation up until the time of subscription. In contrast, if `.just()` is used, we would not be delaying that computation up until the time of subscription. Instead, the value would be computed immediately.

Operators

# Transform: Map and Filter

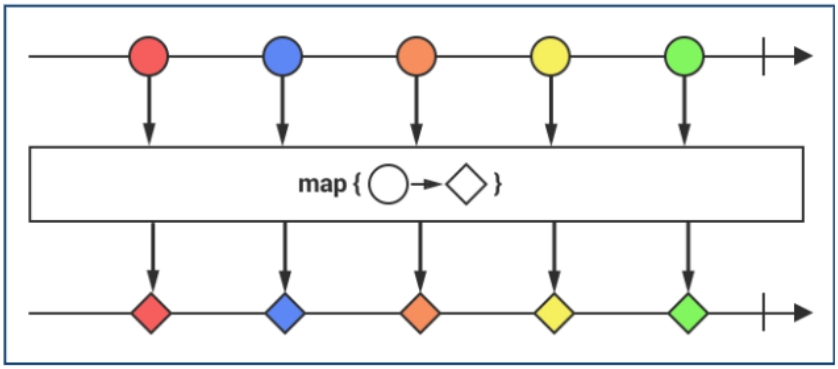
## Transforming and filtering

Operators that transform or filter are some of the handiest operators that are available because they do not require timing or buffering. The operations are applied as soon as a new element is produced in the stream.

In the previous chapter, we looked at `map` and `filter` to apply a simple transformation and filtering of data. The following marble diagrams demonstrate how each one affects emitted items before they are propagated to subscribers.

### Map

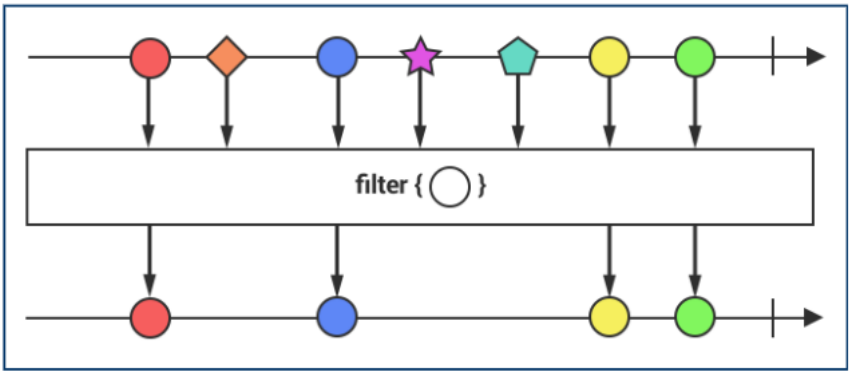
The `.map()` operator works by applying the mapper function provided on each item emitted upstream and passes the return value from that function downstream.




Marble diagram of .map()

### Filter

The `.filter()` operator works by applying the predicate on each emission. If the return value of the predicate is true, the item continues downstream. otherwise, it is filtered out.

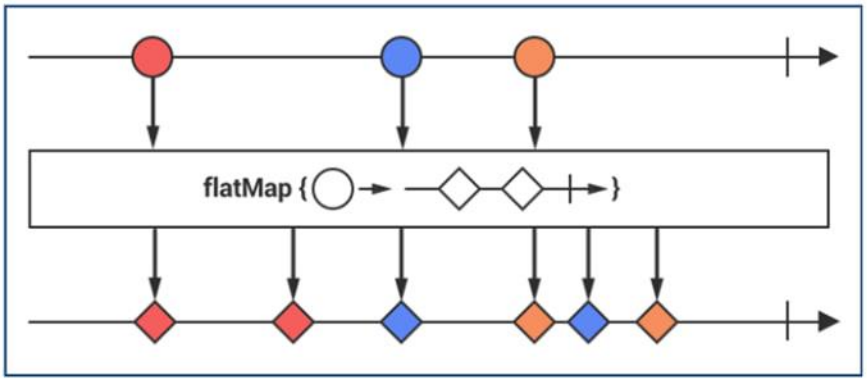



Marble diagram of .filter()

# Transform: FlatMap and ConcatMap

## FlatMap

The `.flatMap()` operator is another very common operator used to transform emissions from an `Observable`. The stream is transformed by applying a function that we specify to each emitted item. However, unlike `.map()`, the function specified should return an object of type `Observable`. The returned `Observable` is then immediately subscribed to and the sequence emitted by that `Observable` is then merged and flattened along with subsequent emissions, which are then passed along to the `Observer`.



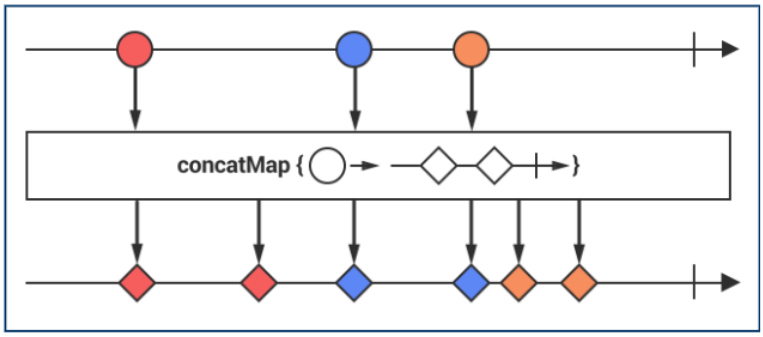

Marble diagram of .flatMap()

потік трансформується шляхом застосування функції, яку ми вказуємо до кожного елемента, що випромінюється. **Однак, на відміну від .map(), вказана функція повинна повертати об'єкт типу Observable. Повернутий об'єкт Observable негайно підписується, і послідовність, яку він випромінює, об'єднується і згладжується разом з наступними випромінюваннями, які потім передаються до Observer'а.**

### .map() vs. .flatMap()

Використовуючи .map(), Спостерігач отримує об'єкт типу Observable<UserDetail> і, таким чином, **повинен підписатися на нього, щоб отримувати його викиди**. З .flatMap() це не так. **Спостерігач отримує UserDetail, який у нашому прикладі є результатом, що його цікавить.**

The `.concatMap()` operator transforms emissions by invoking a function that we specify. Similar to `.flatMap()`, the function we provide should also return an `Observable`. Emissions from `.flatMap()`, however, can be interleaved because it immediately subscribes to emitted `Observable` objects from the function provided. See `.flatMap()` diagram for reference and visualization. The `.concatMap()` operator, on the other hand, does not interleave emissions. Instead, it waits for the previous `Observable` to complete before subscribing to subsequent ones.




Marble diagram of .concatMap()

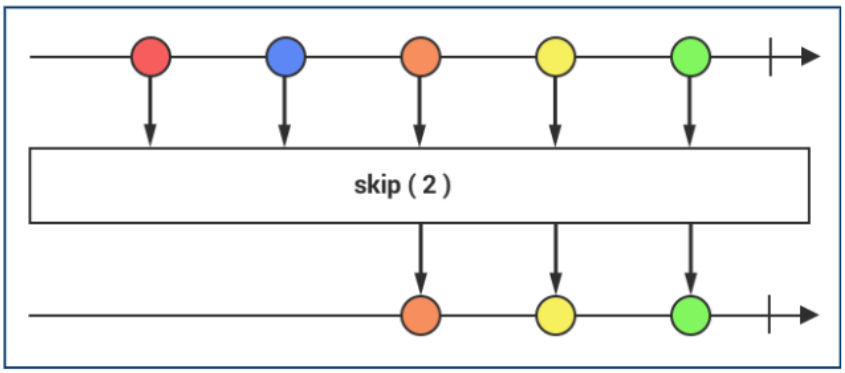
Зауваження: Зверніть увагу на вплив використання .concatMap() на продуктивність. Якщо повернутий Observable з .concatMap() потребує деякого часу для завершення, наступний випромінюваний Observable буде змушений чекати, перш ніж на нього підпишуться. Цей компроміс слід враховувати при прийнятті рішення про використання .flatMap() на противагу .concatMap().

# Transform: Skip, Take, First, and Last

## Skip

Other handy operators allow us to consume only a subset of the items emitted by the sequence.

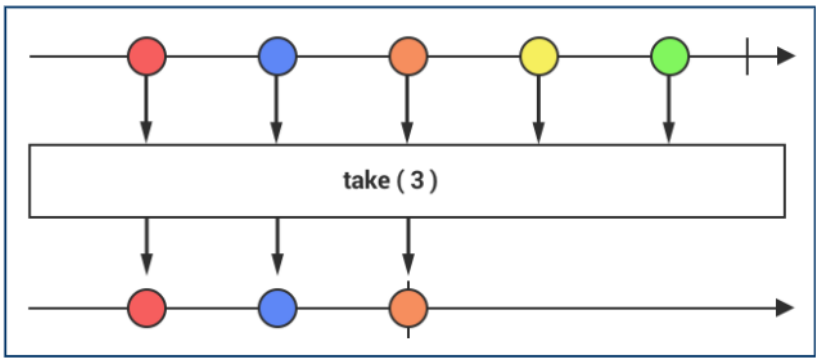
The `.skip(n)` operator ignores the first `n` items emitted by the sequence and emits the remainder of the sequence. The `.skip()` operator is also overloaded with `.skip(time, timeUnit)`, which skips any item(s) emitted before the specified time window.




Marble diagram of .skip()

## Take

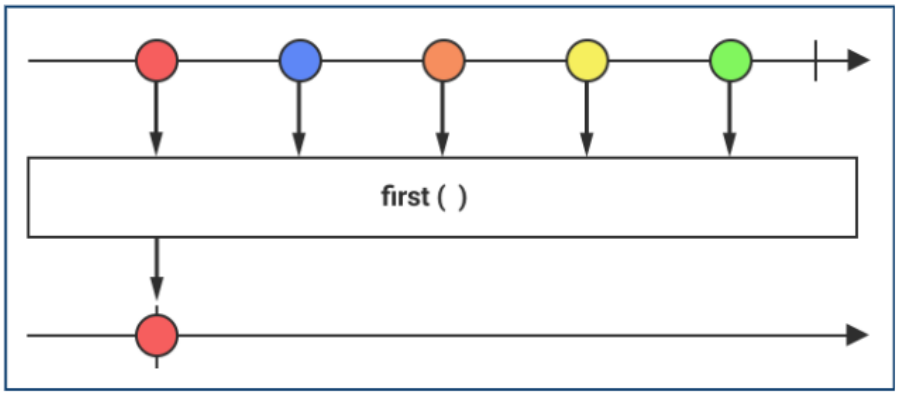
Mirroring `.skip()`, `.take(n)` takes the first `n` items emitted by the sequence and completes immediately after. Similarly, `.take(time, timeUnit)` only takes the item(s) emitted in the specified time window and ignores the rest.




Marble diagram of .take()

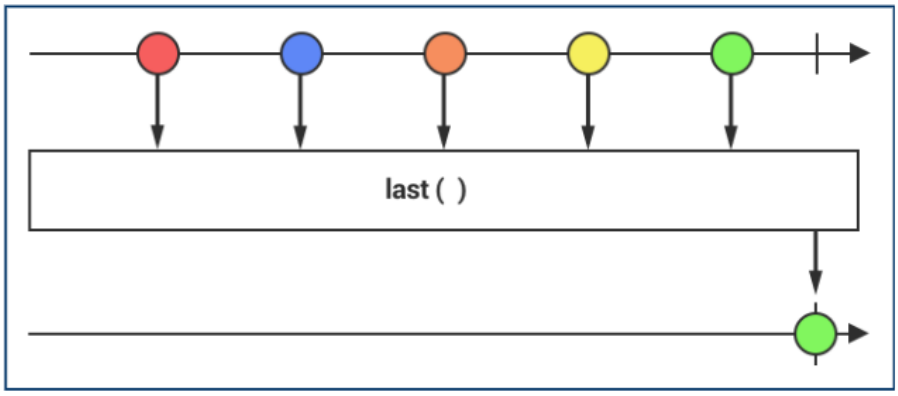
## First and last

The `.first()` and `.last()` operators are fairly self-explanatory. The `.first()` operator returns the first item emitted by the sequence.




Marble diagram of .first()

The `.last()` operator returns the last item emitted by the sequence. For `.last()`, the item is not received immediately after it is emitted. Rather, it is received after the upstream sequence completes.

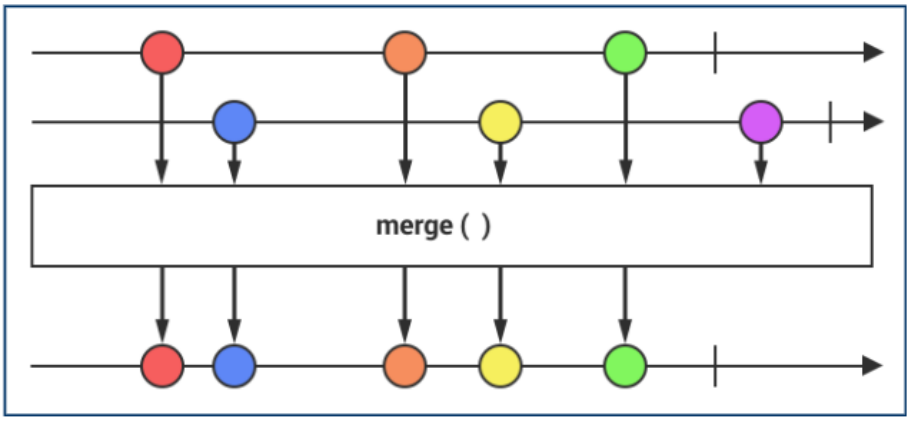



Marble diagram of .last()

# Combine: Merge, Concat, and Zip

## Merge

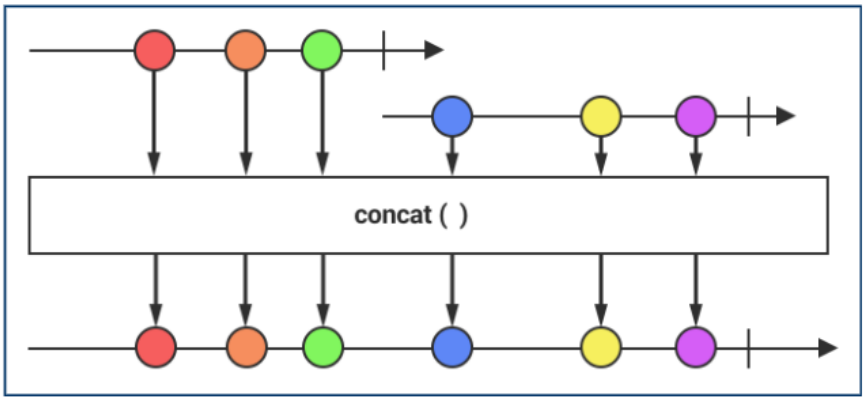
There are several overloaded flavors of the `.merge()` operator, all of which allow the merging of multiple `Observable` streams into a single stream.




Marble diagram of .merge()

## Concat

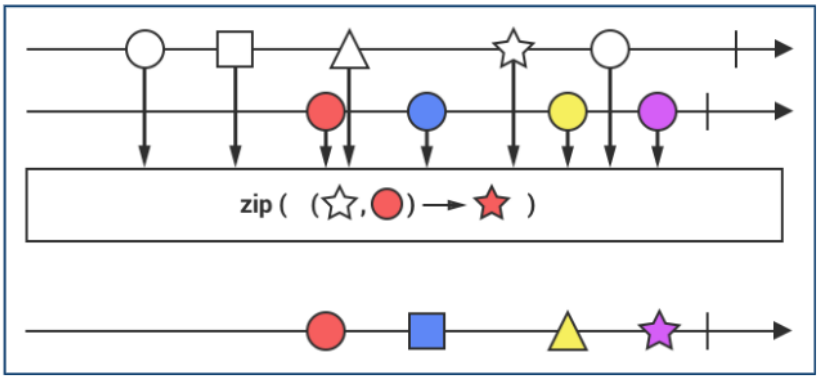
If the interleaving behavior is undesired, the `.concat()` operator may be used to *concatenate* the results rather than *merge* them. In effect, the passed-in `Observable` instances are subscribed to serially as each one completes rather than simultaneously. However, if the preceding `Observable` does not complete, or in other words, its infinite stream, the subsequent `Observable` will not be subscribed to.




Marble diagram of .concat()

## Zip

The `.zip()` function enables pair-wise zipping of emissions from multiple different `Observable` objects. It works by combining the emission from each `Observable` using the specified combiner function. This is especially handy when we have two data sources of different types that we would like to combine into a single `Observable` source.




Marble diagram of .zip()

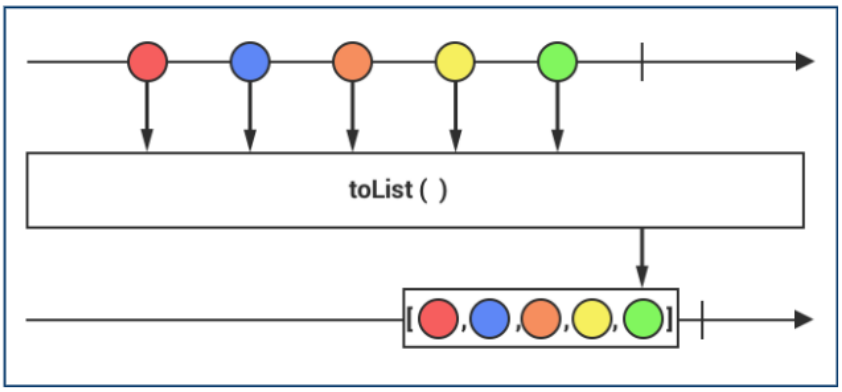
Each zipped item is emitted in a strict sequence. The result of the first zipped item is the combination of the first item in each stream, the second zipped item is the combination of the second item in each stream, and so on. This implies that the zipped `Observable` will emit the same number of items emitted by the `Observable` that has the fewest items. If one sequence does not emit any items, the zipped `Observable` will also not emit any items.

Функція .zip() дозволяє попарно об'єднувати випромінювання від декількох різних об'єктів спостереження. Вона працює шляхом об'єднання викидів від кожного спостережуваного об'єкта за допомогою вказаної функції-комбінатора. Це особливо зручно, коли ми маємо два джерела даних різних типів, які ми хочемо об'єднати в одне джерело спостережуваного об'єкта.

# Aggregate: ToList, Reduce, and Collect

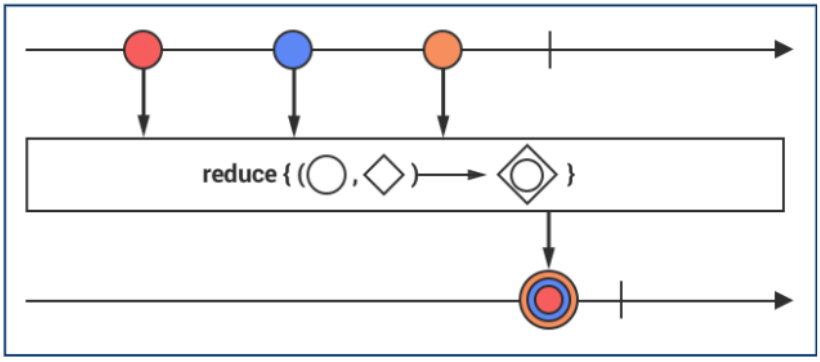
## ToList

The `.toList()` operator converts an `Observable<T>` into an `Observable<List<T>>`. This is useful if the `Observer` ultimately cares about a `List<T>` but the stream itself pushes objects of type `T`.




Marble diagram of .toList()

The `.reduce()` function operates by applying an accumulator function on each emission to ultimately reduce the stream to a single emission.




Marble diagram of .reduce()

For example, say we have a stream that emits integers and we're interested in computing the summation of those integers. For that, we can use `.reduce()` to accumulate the sum.

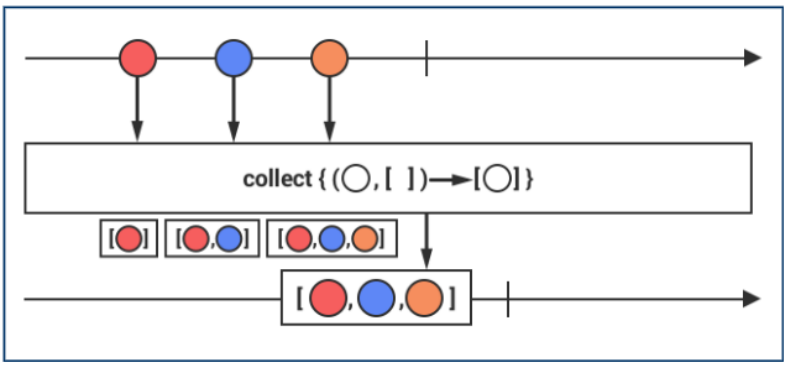
```
Observable<Integer> intStream = // ...
intStream.reduce(
    (i1, i2) -> i1 + i2
).subscribe(total -> {
    Log.d(TAG, "Total is: " + total);
});
```

Using .reduce() to accumulate the sum of integers

In our example, `i1` is the summation of previous emissions, and `i2` is the current integer emission. The starting value for `i1` is `0`, but `.reduce()` also has an overloaded version where a different starting (seed) value may be specified. The `.reduce()` function, however, should not be used for accumulating data into a mutable data structure. For this, we should use the `.collect()` function.

## Collect

The `.collect()` function operates very similarly to `.reduce()`, but instead of taking in an accumulator function, we pass it a `Callable` that emits the mutable data structure, followed by a `BiConsumer` that modifies the mutable data structure.




Marble diagram of .collect()

For example, say we had an `Observable` that emits a list of words. We can accumulate the list of words in a `StringBuilder`:

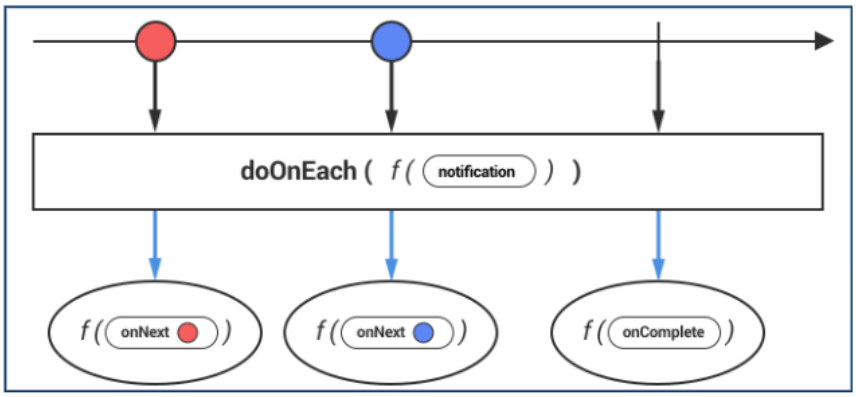
```
final Observable<String> wordStream = // ...
wordStream.collect(() -> {
    return new StringBuilder();
}, (stringBuilder, s) -> {
    stringBuilder.append(" ");
    stringBuilder.append(s);
}).subscribe(stringBuilder -> {
    Log.d(TAG, stringBuilder.toString());
});
```

Using .collect() to accumulate the list of words in a StringBuilder

# Utility Operators: DoOnEach, and Cache

## DoOnEach

The `.doOnEach()` function is an `Operator` that enables listening to events that occur in an `Observable` stream, or in other words when `.onNext()`, `.onError()` and `.onComplete()` are invoked by providing an `Observer`. Generally, this operator is used if we would like to perform a side-effect when an event occurs. There are also several `.doX()` operators, such as `.doOnNext()`, `.doOnError()`, and `.doOnComplete()`, available if we're interested in listening only to a specific event.




Marble diagram of .doOnEach()

Some common usages for `.doX()` operators are for logging purposes. Say we had a chain of operators in a stream and we would like to log each time an `Operator` is about to be applied.

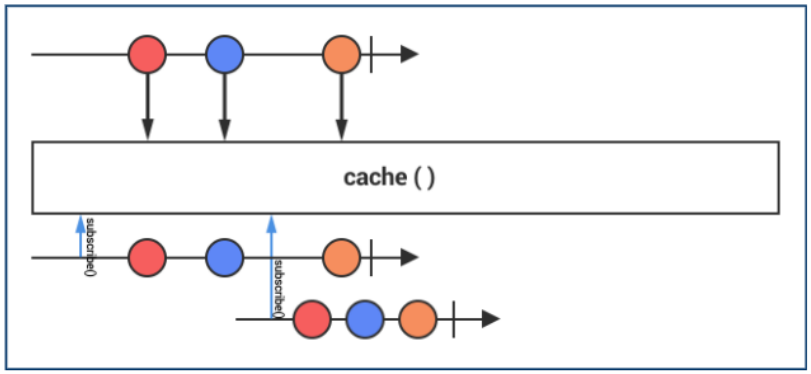
```
Observable<Integer> intStream = Observable.range(1, 3);
intStream.doOnNext(i -> Log.d(TAG, "Emitted: " + i))
    .map(i -> i * 2)
    .doOnNext(i -> Log.d(TAG, "map(): " + i))
    .filter(i -> i % 2 == 0)
    .doOnNext(i -> Log.d(TAG, "filter(): " + i))
    .subscribe(i -> {
        Log.d(TAG, "onNext(): " + i);
    });
```

Logging each time an operator is about to be applied

Функція .doOnEach() - це оператор, який дозволяє прослуховувати події, що відбуваються у потоці Observable, або, іншими словами, коли викликаються .onNext(), .onError() і .onComplete(), надаючи Observer. Зазвичай цей оператор використовується, якщо ми хочемо виконати побічний ефект, коли відбувається подія. Існує також декілька операторів .doX(), таких як .doOnNext(), .doOnError() і .doOnComplete(), які можна використовувати, якщо ми зацікавлені у прослуховуванні лише певної події.

## Cache

The `.cache()` function is a handy operator that subscribes lazily to an `Observable` and caches all of the events emitted in the `Observable`. This is so that when another `Observer` subscribes at a later point, the cached events will be replayed. The `.cache()` function can be used when we would like to reuse already computed events versus having to recompute the events all over again.




Marble diagram of .cache()

Функція .cache() є зручним оператором, який ліниво підписується на спостережуваний об'єкт і кешує всі події, випромінювані цим об'єктом. Це робиться для того, щоб пізніше, коли інший спостерігач підпишеться на Observable, кешовані події були відтворені. Функція .cache() може бути використана, коли ми хочемо повторно використовувати вже обчислені події замість того, щоб обчислювати їх заново.

# Reusing Operator Chains

## Repeating operator chains

As such, we might find ourselves repeating the same chain of operators in parts of our codebase. For example, say it was a very common action in one of our classes to filter by some condition, skip the first filtered item, and follow that by taking the first result after skipping:

```
someObservable.filter(this::isConditionSatisfied)
    .skip(1)
    .take(1)
    // ...continue with other operations here
```

Filtering by some condition, skipping the first filtered item, and taking the first result after skipping

In the spirit of the **DRY principle**—meaning don't repeat yourself—these repeated operator chains can be shared and reused by using an `ObservableTransformer` and passing that as a parameter to the `.compose()` operator.

## `ObservableTransformer`

An `ObservableTransformer` allows us to compose and transform and `Observable`. In our example above, we can encapsulate the three operators, namely, `.filter()`, `.skip()` and `.take()`, inside of an `ObservableTransformer` and reuse it across other `Observable` objects.

```
public class MyTransformer <T> implements ObservableTransformer<T, T> {

    private final Predicate<? super T> filterVal;
    private final int skipVal;
    private final int takeVal;

    MyTransformer(
        Predicate<? super T> filterVal,
        int skipVal,
        int takeVal
    ) {
        this.filterVal = filterVal;
        this.skipVal = skipVal;
        this.takeVal = takeVal;
    }

    @Override
    public ObservableSource<T> apply(Observable<T> upstream) {
        return upstream.filter(filterVal)
                       .skip(skipVal)
                       .take(takeVal);
    }
}
```

ObservableTransformer

ObservableTransformer дозволяє нам компонувати і трансформувати Observable і Observable. У нашому прикладі вище ми можемо інкапсулювати три оператори, а саме .filter(), .skip() і .take(), всередині ObservableTransformer і повторно використовувати його в інших об'єктах Observable.

Клас MyTransformer реалізує ObservableTransformer і має перевизначений метод .apply(), куди ми передаємо вихідний Observable. У ньому ми застосовуємо необхідні перетворення і повертаємо отриманий Observable.

Для цього ми створюємо екземпляр MyTransformer і передаємо його через .compose():

```
someIntObservable.compose(
    new MyTransformer<Integer>(this::isConditionSatisfied, 1, 1)
)
    // ...continue with other operations here
```

## Multithreading

# Default Synchronicity of RxJava

Поширеною помилкою є думка, що RxJava за своєю природою є асинхронною. Можливо, причиною цієї плутанини є те, що Observable-об'єкти за своєю природою є push-, а не pull-об'єктами. Але, насправді, потоки Observable за замовчуванням є синхронними, якщо не вказано інше.

Враховуючи, що потоки Observable за замовчуванням синхронні, чи можете ви здогадатися про порядок операторів Log у наступному прикладі?

```
1  Observable<Integer> integerObservable = Observable.create(source -> {
2      Log.d(TAG, "In subscribe");
3      source.onNext(1);
4      source.onNext(2);
5      source.onNext(3);
6      source.onComplete();
7  });
8  Log.d(TAG, "Created Observable");
9
10 Log.d(TAG, "Subscribing to Observable");
11 integerObservable.subscribe(i -> Log.d(TAG, "In onNext(): " + i));
12
13 Log.d(TAG, "Finished");
```

Default synchronicity of Observable streams

The above code prints out the following:

```
1  Created Observable
2  Subscribing to Observable
3  In subscribe
4  In onNext(): 1
5  In onNext(): 2
6  In onNext(): 3
7  Finished
```

Output

# Adding Asynchronicity

## Using `Thread`

There are many ways we can add asynchronicity so that the function within `.create()` is invoked on a separate thread from the calling thread. As mentioned earlier, RxJava is agnostic regarding how asynchronicity is accomplished. One way is to make use of a basic `Thread`.

Let's take our previous example. Instead of immediately calling `.onNext()` in the function provided in `.create()`, we will:

1. Wrap the `.onNext()` and `.onComplete()` calls in a `Runnable`.
2. Provide the `Runnable` to a new `Thread`.
3. Start the `Thread`.

```
Observable<Integer> integerObservable = Observable.create(source -> {
    Log.d(TAG, "In subscribe");
    new Thread(() -> {
        source.onNext(1);
        source.onNext(2);
        source.onNext(3);
        source.onComplete();
    }).start();
});
Log.d(TAG, "Created Observable");

Log.d(TAG, "Subscribing to Observable");
integerObservable.subscribe(i -> Log.d(TAG, "In onNext(): " + i));

Log.d(TAG, "Finished");
```

Using a Thread to add asynchronicity

With this change, we now get:

```
Created Observable
Subscribing to Observable
In subscribe
Finished
In onNext(): 1
In onNext(): 2
In onNext(): 3
```

Output

Як бачимо, оператор Finished викликається перед операторами onNext(). Насправді, порядок виконання цих операторів вже не є детермінованим, і наша початкова послідовність журналу все ще можлива. Все залежить від того, коли ОС вирішить виконати перемикання контексту, яке може змінюватися від одного виконання до іншого.

Такий підхід до неблокування підписки є абсолютно правильним, доки виклик .onNext(), .onError() або .onComplete() виконується з одного потоку і він є потокобезпечним. Якщо цю угоду порушено, деякі оператори, які залежать від цього припущення, можуть не працювати. Іншими словами, ми не можемо випромінювати події з декількох потоків.

# Schedulers

## What are schedulers?

A `Scheduler` is a multithreading construct introduced in RxJava that can run a scheduled unit of work on a thread. Simplistically, you can think of a `Scheduler` as a thread pool, and when a task needs to be executed, it takes a single thread from its pool and runs the necessary task.


`Scheduler` constructs are used in conjunction with `.subscribeOn()` and `.observeOn()`, two operators that specify where a particular operation should execute. When a `Scheduler` is specified along the `Observable` chain, the `Scheduler` provides a worker that runs the actual block of code. The thread that the scheduled work executes on depends on the `Scheduler` in use.

### Types of Schedulers

Існує декілька типів операторів Планувальника, доступних через фабричні методи в класі Schedulers:

Оператор Schedulers.newThread(), як випливає з його назви, повертає планувальник, який створює новий потік для кожної запланованої одиниці роботи. Як наслідок, використання Scheduler.newThread() є дорогим, оскільки кожного разу створюється новий потік - повторного використання не відбувається.

Оператор Schedulers.from(Executor виконавець) повертає Планувальник, який підтримується вказаним Виконавцем.

Оператор Schedulers.single() повертає Планувальник, який підтримується одним потоком. Якщо в Schedulers.single() заплановано декілька завдань, вони будуть заплановані послідовно і можуть бути виконані в порядку черговості.

Оператор Schedulers.io() надає планувальник, призначений для роботи, пов'язаної з вводом/виводом, наприклад, мережевих викликів, транзакцій з базами даних тощо. Базова реалізація використовує пул потоків Executor, який розширюється за потреби. При виконанні завдань вводу/виводу ми не виконуємо роботу, що вимагає багато процесорних ресурсів. Замість цього, ми зазвичай чекаємо на ресурси або сервіси, які можуть знаходитися на іншому

хості, тому можливість створювати і призначати новий потік для кожного завдання є ідеальною.

Оператор Schedulers.computation() надає планувальник, призначений для обчислювальної роботи, що вимагає великих ресурсів процесора, наприклад, обробки довгого списку, зміни розміру зображення і так далі. Цей планувальник використовує пул потоків, розмір якого обмежений кількістю доступних ядер. Оскільки ми виконуємо роботу, що вимагає багато процесорних ресурсів, створення більшої кількості потоків, ніж є ядер, фактично погіршить продуктивність. Як створення потоків, так і перемикання контексту призводять до значних накладних витрат, не допомагаючи прискорити нашу роботу, що вимагає багато процесорних ресурсів. Планувальник .computation() ніколи не слід використовувати для вводу/виводу, а планувальник .io() ніколи не слід використовувати для обчислювальної роботи; ввід/вивід і обчислення блокуються з абсолютно різних причин і тому вимагають абсолютно різних стратегій багатопоточності.

Оператор Schedulers.trampoline() надає планувальник, який планує роботу у поточному потоці - або іншими словами, використання цього планувальника заблокує поточний потік. Заплановані завдання не виконуються негайно, а ставляться у чергу і виконуються після завершення поточної одиниці роботи. Цей планувальник зазвичай використовується при реалізації рекурсії, щоб уникнути нескінченного зростання стеку викликів.

# Schedulers with Operators

### `.subscribeOn()`

The `.subscribeOn()` operator is used in the `Observable` chain to dictate where the `Observable` should operate–for example, the function inside of `.create()`. Rewriting the previous example using a `Scheduler` instead gives us:

```
Observable<Integer> integerObservable = Observable.create(source -> {
    Log.d(TAG, "In subscribe");
    source.onNext(1);
    source.onNext(2);
    source.onNext(3);
    source.onComplete();
});
Log.d(TAG, "Created Observable");

Log.d(TAG, "Subscribing to Observable");
integerObservable.subscribeOn(Schedulers.newThread())
                 .subscribe(i -> Log.e(TAG, "In onNext(): " + i));

Log.d(TAG, "Finished");
```

Using a Scheduler

Running the above code is similar to using a `Thread` in that the operations inside `.create()` now occurs in a separate thread provided by `Schedulers.newThread()`. The benefit of this approach over using a `Thread` is that tacking on a `Scheduler` to specify where the `Observable` should execute is declarative. We no longer have to worry about the low-level details of how to create and run a thread. Furthermore, we can decouple the `Observable` creation from the threading. The subscriber can determine the thread on which to run the `Observable` rather than having that decision baked into the `Observable` itself.

### `.observeOn()`

Using `.subscribeOn()`, we are able to modify where an `Observable` does its work. But say, for example, that we performed a network call to retrieve the information of a `User` using `Schedulers.io()`. Now, we want to update the UI once the network call succeeds. Can we spot the bug in the following line of code?

```
// `userObservable` retrieves a User object over the network
Observable<User> userObservable = // ...

userObservable.subscribeOn(Schedulers.io())
              .subscribe(user -> textView.setText(user.getName()));
```

Retrieving a User's information and updating the UI

As we might have noticed in the preceding example, all subsequent operations after `.subscribeOn()` run on the thread provided by the specified `Scheduler`. In other words, the call to `textView.setText(user.getName())` runs on a thread other than the UI thread, which causes the app to crash with a `CalledFromWrongThreadException`.

# Scheduler Behavior

## .subscribeOn()

Як зазначалося раніше, не має значення, де в ланцюжку операторів з'являється виклик .subscribeOn(). Однак важливо зазначити, що застосовується лише перший виклик .subscribeOn(). Будь-які наступні виклики .subscribeOn() не матимуть жодного ефекту. Часто

об'єкти Observable створюються як частина бібліотеки або API і надаються клієнту через інтерфейс. У документації цього інтерфейсу має бути чітко зазначено, чи повернутий об'єкт Observable вже було передано якомусь планувальнику за допомогою .subscribeOn(), оскільки якщо так, то клієнт більше не зможе виконати власний виклик .subscribeOn().

### .observeOn()

Оператор .observeOn(), з іншого боку, багато в чому протилежний .subscribeOn(). У той час як .subscribeOn() впливає на джерело Observable вище за течією, .observeOn() впливає на оператори нижче за течією. В той час як тільки перший .subscribeOn() має силу, .observeOn() можна викликати декілька разів, при цьому кожен виклик перемикає потік для наступних операторів, доки не зустрінеться інший виклик .observeOn().

Для поширених випадків використання - наприклад, коли спостерігач оновлює інтерфейс користувача - ми рекомендуємо розміщувати виклик .observeOn() якомога ближче до виклику .subscribe() (тобто якомога пізніше в ланцюжку операторів).

# Achieving True Concurrency

## The basics of true concurrency

If you noticed in our example of using `.subscribeOn()`, events received from upstream are not processed on separate threads. All emissions were executed on the same thread, `RxCachedThreadScheduler-1`. To achieve true concurrency, we need the help of `.flatMap()`.

### Using `.flatMap()`

As we've seen in the previous chapter, we are able to return a new `Observable` given an upstream emission by using `.flatMap()`. The returned `Observable` from `.flatMap()` is completely independent. We can even specify a separate `Scheduler` for that `Observable` to operate in. In this way, we can impose concurrent behavior in an `Observable` chain.

By modifying the code above to use `.flatMap()` instead of `.map()`, we get:

```
Observable.fromArray(usernames)
    .subscribeOn(Schedulers.io())
    .flatMap(username ->
        Observable.fromCallable(() ->
            networkClient.fetchUser(username)
        ).subscribeOn(Schedulers.io())
    )
    .subscribe(user -> print("Got user: " + user.username));
}
```

Fetching User objects using .flatMap()

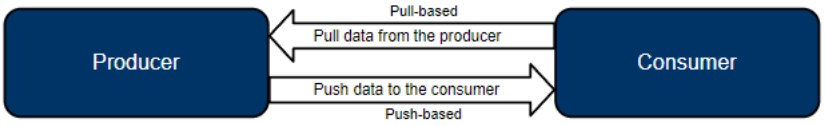
Running the above code gives us the following output:

```
RxCachedThreadScheduler-3 : Got user: jacob
RxCachedThreadScheduler-2 : Got user: mike
RxCachedThreadScheduler-1 : Got user: john
```

Output

# Bridging the Non-reactive and Reactive Worlds

One of the challenges of adding RxJava as one of the libraries to projects is that it fundamentally changes the way that we reason about our code.




RxJava requires us to think about data as being pushed rather than being *pulled*. We covered this topic in our earlier lesson, Push vs. Pull. (https://www.educative.io/collection/page/10370001/6549293806125056/6254001416306688#Push-vs.-Pull)). While the concept itself is simple, changing a full codebase that is based on a pull paradigm can be a bit daunting. Although consistency is always ideal, we might not always have the privilege to make this transition throughout our entire code base all at once, and so an incremental approach may be required.

# RxJava's Consequences: Method Signature

## Issue: Method signature

After making the changes, the first point to note is that the method signature is no longer the same. This may not be a big deal if this method call is only used in a few places and it's easy to propagate the changes up to other areas of the stack. However, if it breaks clients relying on this method, that's problematic, and the method signature should be reverted.

## Solution: Using `.blockingX()`

To address the first issue from our change, we can make use of any of the `.blockingX()` operators available to an `Observable`. Essentially, a `.blockingX()` operator blocks the calling thread until an item is emitted downstream.

A few notable blocking operators that are available are:

- `.blockingFirst()` blocks the calling thread and returns the first element emitted by the `Observable`.
- `.blockingNext()` returns an `Iterable` which allows us to iterate through all the emissions by the `Observable`. Each iteration will block until the `Observable` emits a next item.
- The `.blockingLast()` operator blocks the calling thread and returns the last element emitted by the `Observable`.
- The `.blockingSingle()` operator blocks the calling thread until the `Observable` emits an element followed by an `.onComplete()` event. Otherwise, an `Exception` is thrown.
- The `.blockingSubscribe()` operator blocks until a terminal event is received. All elements that are received through `.onNext()` are ignored.

Після внесення змін перше, на що слід звернути увагу, це те, що сигнатура методу вже не та, що раніше. Це може не мати великого значення, якщо цей виклик методу використовується лише в декількох місцях, і зміни легко поширити на інші області стеку. Однак, якщо це порушує роботу клієнтів, які покладаються на цей метод, це проблематично, і сигнатуру методу слід повернути.

Щоб вирішити першу проблему з нашої зміни, ми можемо використати будь-який з операторів .blockingX(), доступних для спостережуваного об'єкта. По суті, оператор .blockingX() блокує викликаючий потік доти, доки не буде випущено елемент нижче за течією.

Single - це спеціальний тип Observable, який генерує один елемент. Він відрізняється від Observable, який може випромінювати будь-яку з 3 подій: .onNext(), onCompleted() та .onError(). Натомість, Single має лише .onSuccess() та .onError(). Використання Single є кращим, коли ми знаємо, що буде виведено лише один елемент, тому що він надає користувачеві чіткіші наміри щодо того, які дані буде виведено у потоці (один елемент).

# RxJava's Consequences: Laziness

RxJava розроблено з урахуванням лінощів. Перегляньте Ліниві викиди для нагадування. Тобто, не слід виконувати довгих операцій, коли немає підписників на Observable. З цією модифікацією це припущення більше не є вірним, оскільки UserCache.getAllUsers() викликається ще до того, як з'являться підписники. Незалежно від того, де знаходиться резервний кеш - у пам'яті чи на диску, цю операцію слід відкласти, доки на нього не підпишеться Спостерігач, щоб залишатися вірним принципам RxJava.

Як згадувалося раніше, RxJava була розроблена з урахуванням лінощів. Тобто, довготривалі операції повинні бути максимально відкладені до моменту виклику .subscribe() на Observable. Щоб зробити наше рішення лінивим, ми можемо скористатися методами створення Observable .fromCallable() або .defer().

Оператор .defer() отримує джерело ObservableSource, яке, по суті, є фабрикою, що повертає Observable щоразу, коли на нього підписано зовнішній Observable. У нашому випадку ми хочемо повернути Observable.fromItereable(User.getAllUsers()) щоразу, коли на нього підписується Observer.

# Reactive Everything: Long Operations

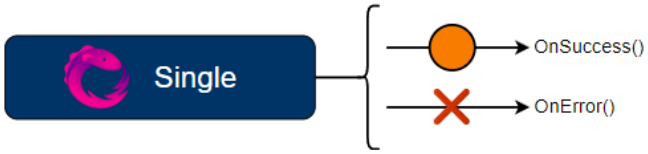
З попередніх прикладів ми бачили, що можемо обернути будь-який об'єкт в Observable і переходити між нереактивним і реактивним станами за допомогою операторів .blockingX(). Крім того, ми можемо затримати виконання операції, обернувши її в метод створення Observable .fromCallable() або .defer(). Використовуючи ці конструкції, ми можемо почати перетворювати області Android-додатку на реактивні.

Починати використовувати RxJava варто завжди, коли у нас є процес, який займає багато часу на обчислення, наприклад, мережеві виклики, читання і запис на диск, обробка растрових зображень і так далі.

# Reactive Everything: Completable, Single, and Maybe

## Single

In our previous example in Leaving the Reactive World, we looked at another base reactive type called a `Single`. As mentioned earlier, a `Single` is a special type of `Observable` that only emits a *single* item rather than a *stream* of items.



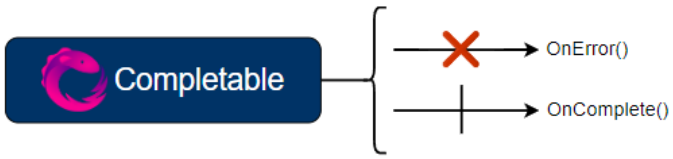

A Single emits either an element or an error

Consequently, an observer to a `Single`, a `SingleObserver`, has a different signature and only receives the following events:

- The `.onSubscribe()` function is invoked on subscription and a `Disposable` object is passed along, which can be used to unsubscribe from the `Single`.
- The `.onSuccess()` function is invoked when the `Single` has emitted an item.
- The `.onError()` function is invoked when the `Single` has encountered an error.

## Completable

As we've seen in the previous example of writing text to the file system, a `Completable` is another base reactive type that notifies an observer whether or not an operation succeeds. It does not emit an item. Instead, it simply tells the observer if the operation passes or fails.



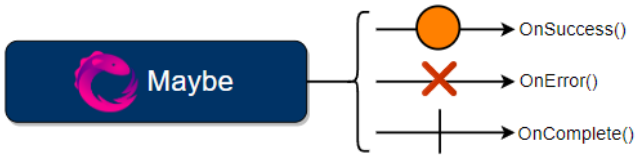

A Completable notifies whether or not an operation succeeds

An observer to a `Completable`, a `CompletableObserver`, has the following events:

- The `.onSubscribe()` function is invoked on subscription and a `Disposable` object is passed along that can be used to unsubscribe from the `Completable`.
- The `.onComplete()` function is invoked when the `Completable` completes.
- The `.onError()` function is invoked when the `Completable` has encountered an error.

## Maybe

A `Maybe` can be considered a mix of a `Single` and a `Completable`. It's like a `Single` in that one item may be emitted, or like a `Completable` in that zero items may be emitted.




Maybe emits either one item or no items

Its observer, a `MaybeObserver`, has the following events:

- The `.onSubscribe()` function is invoked on subscription and a `Disposable` object is passed along, which can be used to unsubscribe from the `Maybe`.
- The `.onComplete()` function is invoked when the `Maybe` completes. This method is invoked when the `Maybe` does not emit an item.
- The `.onSuccess(T value)` function is invoked when the `Maybe` emits a single item.
- The `.onError()` function is invoked when the `Completable` has encountered an error.

## Converting reactive types

However, if another type is required, say, when using the operator `.zip()` with a `Single` and with an `Observable`, it's possible to convert between the two types through transformational methods. For example, `Single`, `Maybe`, and `Completable` have the `.toObservable()` method which converts the type to an `Observable`.

Conversely, an `Observable` may be converted to another base reactive type depending on the operator. For instance, as we've seen in a previous example, invoking `.toList()` on an `Observable` returns a `Single`.

# Reactive Everything: Replacing Callbacks

## Using `.create()`

The `.create()` operator allows us to explicitly invoke any of the available methods of an `Observer` (`.onNext()`, `.onComplete()` or `.onError()`) as we receive updates from our data source. As we've seen, to use `.create()`, we pass in an `ObservableOnSubscribe`, which receives an `ObservableEmitter` whenever an `Observer` subscribes. Using the received emitter, we can then perform all the necessary set-up calls to start receiving updates and then invoke the appropriate emitter event.

Let's look at an Android-specific example where we might want to apply these principles: a device's accelerometer events. First, let's create a `DeviceSensorManager` class that can be queried for an `Observable` that emits accelerometer events.

Оператор .create() дозволяє нам явно викликати будь-який з доступних методів Observer (.onNext(), .onComplete() або .onError()), коли ми отримуємо оновлення від нашого джерела даних. Як ми бачили, для використання .create() ми передаємо ObservableOnSubscribe, який отримує ObservableEmitter щоразу, коли Observer підписується. Використовуючи отриманий емітер, ми можемо виконати всі необхідні виклики налаштування, щоб почати отримувати оновлення, а потім викликати відповідну подію емітера.

### Convert a hot Observable to a cold Observable

What might happen if multiple observers decide to subscribe to this `Observable`?

Since `.create()` returns a cold `Observable`, the function inside `.create()` would be repeated on each new subscription, which means a new location update request would be made. Clearly, this is not what we want and instead, we would like to share one subscription but have as many observers as we would like. To accomplish this, we need a way to convert the *cold* `Observable` to a *hot* `Observable`.

# Reactive Everything: Multicasting

### Using `.publish()`

The simplest mechanism to convert a given cold `Observable` to a hot `Observable` is by using the method `.publish()`. Calling `.publish()` returns a `ConnectableObservable`, which is a special type of `Observable` wherein each `Observer` shares the same underlying resource. In other words, using `.publish()`, we are able to multicast, or share to multiple observers.

Subscribing to a `ConnectableObservable`, however, does not begin emitting items.

## Subject

A `Subject` behaves both as an `Observable` and as an `Observer`. Using a `Subject` is an alternative to multicasting the same resource to multiple subscribers. Implementation-wise, we would want to expose the `Subject` as an `Observable` to clients, or keep it as a `Subject`, to the provider.

### Types of `Subject`

RxJava has a few other varieties of `Subject` types:

- **`PublishSubject`** emits to an observer only items that are emitted after the time of the subscription.
- **`AsyncSubject`** emits to an observer only the last value after the `Observable` successfully completes.
- **`BehaviorSubject`** upon subscription emits to an observer the most recently emitted item and any subsequent items.
- **`ReplaySubject`** emits to an observer all previously emitted items and any subsequent items.

# Reactive Everything: View Events

---

Using RxBinding, we can turn `View` click events into an `Observable`.

```
Observable<Object> clicks = RxView.clicks(view);
clicks.subscribe(obj -> {
    // Get click events here
});
```

Turning View click events into an Observable using RxBinding

## Example

This example is quite simple, but the benefits of using Rxjava become apparent when used in conjunction with operators. Say, for example, we wanted to detect double clicks, or two clicks that happen within 300 milliseconds apart.

```
Observable<Object> clicks = RxView.clicks(view);

Observable<List<Object>> clickAggregate =
    clicks.buffer(300, TimeUnit.MILLISECONDS);

Observable<List<Object>> doubleClicks =
    clickAggregate.filter(list -> list.size() >= 2);

doubleClicks.subscribe(o -> {
    Timber.d("Received double clicks.");
});
```

Detecting double clicks using RxJava

Normally, the above method calls to the click `Observable` would be chained, but for educational purposes, we've separated it out line-by-line so we can understand the transformations at each step.

1. First, we convert click events from the `view` to an `Observable` using `RxView.clicks()`.
2. After that, we use the operator `.buffer()` which buffers items that occur within a provided timespan, in this case 300 milliseconds, into a `List<Object>`.
3. Afterward, we simply filter each emission of the `.buffer()` operation using `.filter()` to check if the buffered list contains 2 or more items. If it does, then a double click was just performed.

# Disposable and the Activity/Fragment Lifecycle

Досі ми не звертали уваги на ключовий компонент реактивного потоку: "Disposable" (одноразовий). Disposable повертається в результаті підписки, щоб ми могли контролювати, коли від неї відписатися. Це робиться для того, щоб Observable, який лежить в основі, міг припинити випромінювати події. Якщо цього не зробити, це може призвести до небажаних витоків пам'яті, які важко відстежити.

У випадку з життєвим циклом Activity або Fragment, найкращим місцем для скасування підписки буде момент, коли Activity або Fragment більше не потрібні - тобто, у життєвому циклі .onDestroy().

```
public class MyActivity extends Activity {
    Disposable disposable;

    protected void onCreate(Bundle savedInstanceState) {
        this.disposable = // ...subscribe to some Observable
    }

    protected void onDestroy() {
        if (!disposable.isDisposed()) {
            disposable.dispose();
        }
    }
}
```

## RxLifecycle

Alternatively, an open source library called RxLifecycle, created by Trello, can be used to automatically unsubscribe from an `Observable` once an `Activity` or `Fragment` lifecycle event occurs. This way, we don't have to manage the `Disposable` object explicitly. Instead, RxLifecycle handles it for us.

To bind a `Disposable` with RxLifecycle, we can specify a lifecycle event when an `Observable` should be unsubscribed, for example, on the `.onDestroy()` of an `Activity`:

```
observable
    .compose(RxLifecycle.bindUntilEvent(
        lifecycle,
        ActivityEvent.DESTROY
    ))
    .subscribe();
```

Binding a Disposable with RxLifeCycle by specifying a lifecycle event

Or, we can let RxLifecycle decide when it should unsubscribe from an `Observable`. In this case, it will unsubscribe at the opposing lifecycle event from when the `Observable` was subscribed, for example, `.onCreate()`/`.onDestroy()`, `.onStart()`/`.onStop()` or `.onResume()`/`.onPause()`.

```
observable
    .compose(
        RxLifecycleAndroid.bindActivity(lifecycle)
    )
    .subscribe();
```

Letting RxLifecycle decide when to unsubscribe from an Observable

In both examples, lifecycle represents an `Observable<ActivityEvent>` that can be obtained by subclassing `RxActivity`/`RxFragment`:

```
public class MyActivity extends RxActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        observable
            .compose(RxLifecycle.bindUntilEvent(
                lifecycle,
                ActivityEvent.DESTROY
            ))
            .subscribe();
    }
}
```

Obtaining Observable<ActivityEvent> by Subclassing RxActivity

## Backpressure

Щоб вирішити цю проблему, нам потрібен спосіб повідомити видобувну компанію про те, що вона повинна зменшити виробництво, поки споживач, який знаходиться нижче за течією, не зможе встигнути за переробкою продукції.

Механізм, за допомогою якого ми можемо повідомити видобувну компанію про те, що вона повинна сповільнити своє виробництво, називається протитиском. Однак, використовуючи об'єкт Observable, ми не можемо застосувати зворотний тиск, тому що об'єкти Observable не призначені для підтримки зворотного тиску. Для цього нам знадобиться інший базовий реактивний клас, який називається Flowable.

# Flowable

Flowable - це базовий реактивний клас, який підтримує протитиск. Якщо ми знаємо, що маємо справу з потоком, до якого потрібно додати протитиск, то Flowable - це те, що нам потрібно.

Flowable підтримує наступні стратегії протитиску:

BackpressureStrategy.ERROR генерує виключення MissingBackpressureException у випадку, коли потік не встигає за викидами елементів.

BackpressureStrategy.BUFFER` буферизує всі елементи до того часу, поки наступний потік не споживе їх. Розмір буферу за замовчуванням 128.

BackpressureStrategy.DROP скидає останній елемент, якщо наступний не встигає за ним. BackpressureStrategy.LATEST зберігає лише останній елемент, перезаписуючи будь-яке попереднє значення, якщо наступний не встигає.

BackpressureStrategy.MISSING вказує на те, що протитиск не додається до потоку. Це еквівалентно використанню спостережуваного елемента.

Отже, коли ми хочемо вибрати Flowable замість Observable? Вікі RxJava пропонує наступні загальні рекомендації.

**Коли варто використовувати Observable**

Максимальний потік не перевищує 1000 елементів: іншими словами, у нас так мало елементів з часом, що у вашому додатку практично немає шансів на виникнення OOME (OutOfMemoryError - помилка виходу з пам'яті).

Ми маємо справу з подіями графічного інтерфейсу, такими як рухи миші або дотики: вони рідко піддаються розумному зворотному тиску і трапляються не так вже й часто. Можливо, ви зможете впоратися з частотою елементів 1,000 Гц або менше за допомогою Observable, але все одно варто використовувати дискретизацію та дебаунсинг.

Наш потік по суті синхронний, але наша платформа не підтримує Java-потоки, або ми втрачаємо їхні можливості. Використання Observable загалом має менші накладні витрати, ніж

Flowable. (Ми також можемо розглянути IxJava, який оптимізовано для змінних потоків з підтримкою Java 6+).

**Коли варто використовувати Flowable**

Нам доводиться мати справу з більш ніж 10 000 елементів, які десь генеруються певним чином. Таким чином, ланцюжок може сказати джерелу обмежити кількість елементів, які він генерує.

Читання (парсинг) файлів з диска за своєю суттю є блокуванням і витягуванням, що добре працює з протитиском. Ми контролюємо, наприклад, скільки рядків ми зчитуємо з цього файлу для заданого обсягу запиту.

Читання з бази даних через Java Database Connectivity (JDBC) також базується на блокуванні та витягуванні і контролюється нами шляхом виклику ResultSet.next() для кожного наступного запиту.

Нам доводиться використовувати мережевий (потоковий) ввід-вивід, коли або мережа допомагає, або використовуваний протокол підтримує запит певної логічної кількості даних.

Багато блокуючих та/або заснованих на витягуванні джерел даних, можливо, в майбутньому отримають неблокуючий реактивний API/драйвер.

# Subscriber

Використовуйте subscriber, щоб дозволити тонкий контроль над тим, як споживач повідомляє виробника про те, що він може споживати більше даних. На додаток до стандартних методів спостерігача .onNext(T), .onError(Throwable) та .onComplete(), Subscriber має додатковий метод .onSubscribe(Subscription), який викликається при підписці на виробника (Flowable). Об'єкт Subscription, отриманий з .onSubscribe(Subscription), потім використовується для керування потоком продукції від виробника через його метод .request(long). Як тільки викликається .onSubscribe(), ми повинні негайно запросити дані у виробника через .request(long), щоб почати отримувати елементи.

# Throttling and Buffering

Троттлінг

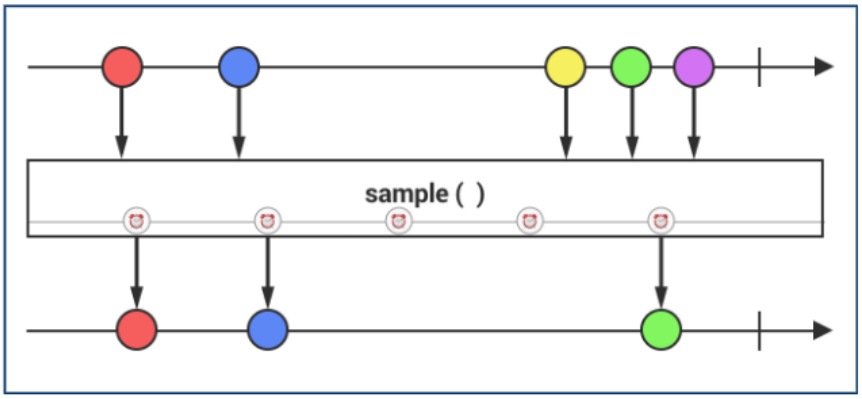
Згадаймо приклад DeviceSensorManager, який ми обговорювали раніше. Ми бачили, що швидкість, з якою надходять події від акселерометра, визначається значенням SensorManager.SENSOR_DELAY_NORMAL (тобто близько 20 мілісекунд).

Припустимо, ми хочемо споживати події зі значно меншою швидкістю. Як ми можемо це зробити? Ми можемо підключити окремий Listener з потрібною швидкістю, але що, якщо ми

хочемо використовувати той самий Observable? Для цього є кілька варіантів, один з яких - використання оператора фільтрації .sample().

## Sample

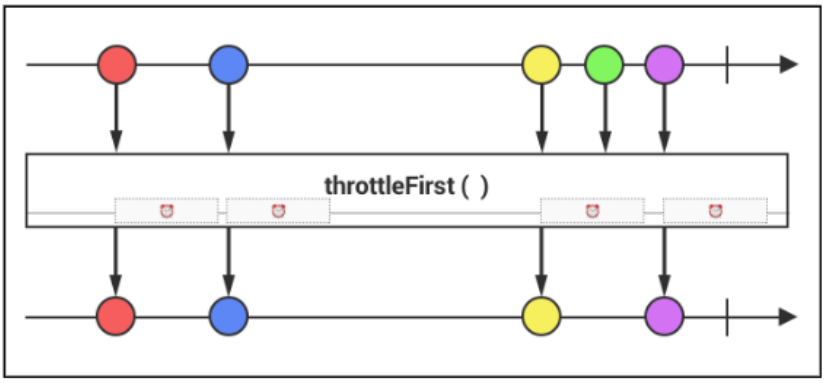
The `.sample(long, TimeUnit)` method works by periodically sampling events at a specified rate by emitting the latest item in a period while dropping all other items.



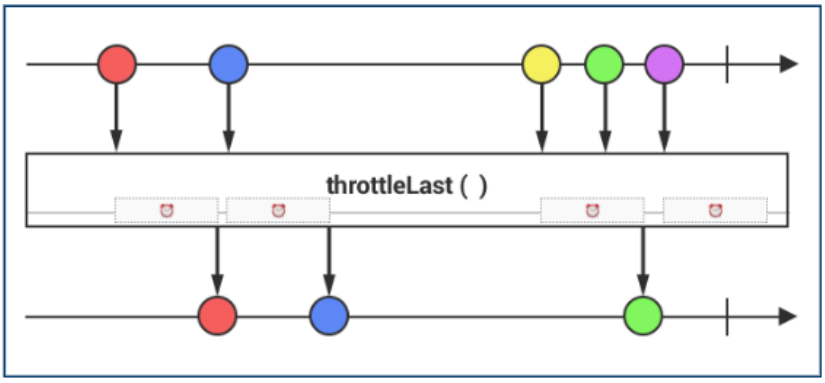

Marble diagram of .sample()

## Throttle

On the other hand, the `.throttleFirst()` and `.throttleLast()` methods can be used to only emit the first or last item, respectively, over a sequential time period.




Marble diagram of .throttleFirst()




Marble diagram of .throttleLast()

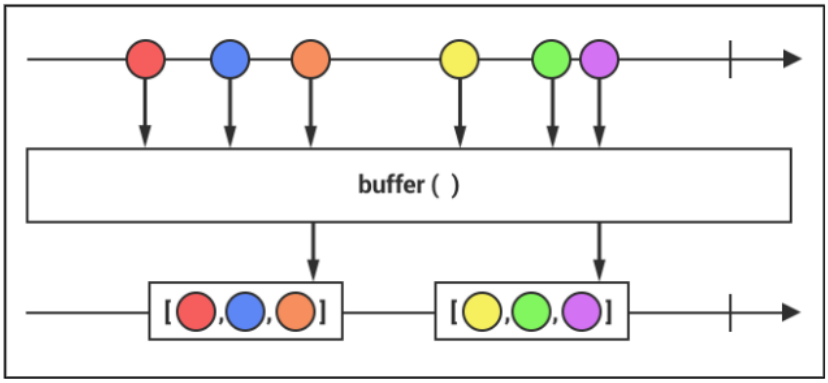
Буферизація

Інша стратегія, яка допомагає зменшити проблеми з балакучими виробниками, полягає у буферизації або збиранні викидів і видачі результату нижче за течією з меншою частотою. Двома корисними операторами для досягнення цієї мети є .buffer() та .window().

## Buffer

As the name implies, `.buffer(...)` allows us to buffer emissions from upstream into a list. The `.buffer()` operator is overloaded to support buffering given different criteria. A few handy ones are:

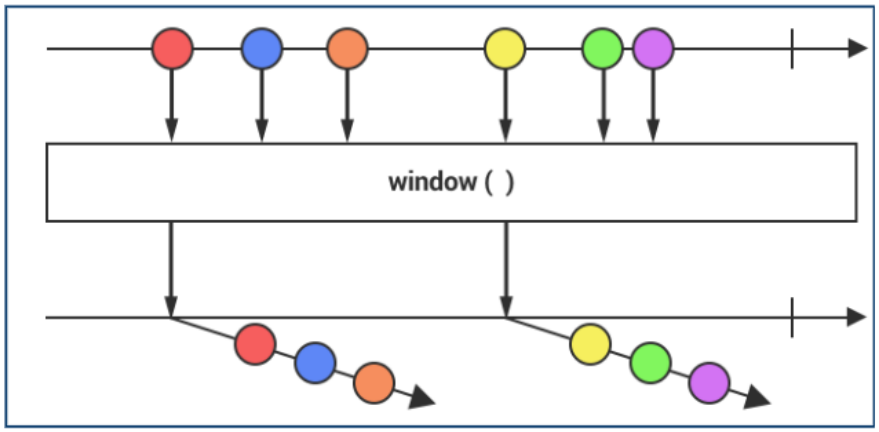
- The `Observable.buffer(int count)` operator: Buffers *count* items.
- The `Observable.buffer(ObservableSource<Type> boundary)` operator: Buffers items in between emissions from boundary.
- The `Observable.buffer(long timespan, TimeUnit timeUnit)` operator: Buffers items that are emitted in a given time span.




Marble diagram of .buffer()

## Window

The `.window()` operator is similar to `.buffer()`, in that it allows us to emit `Observable` objects instead of lists. The `.window()` operator is also overloaded and allows buffers given different criteria.




Marble diagram of .window()

# Errors Along the Chain

Як ми бачили раніше, однією з подій, яку може отримати Observer, є подія .onError(Throwable). Сповіщення .onError() вважається термінальним, тобто після його виклику Observable більше не буде генерувати жодних інших подій. Існує декілька способів викликати сповіщення .onError(). Деякі з них є явними, наприклад, виклик .onError() на Emitter:

```
Observable<Integer> observable = Observable.create(emitter -> {
    emitter.onError(new Exception("An error occurred."));
});
observable.subscribe(i -> {
}, throwable -> {
    Log.e(TAG, "onError(): " + throwable.getMessage());
});
```

Invoking the .onError() on an Emitter

Or, by returning an `Observable.error()` wherever an `Observable` is expected, such as in the `.flatMap()` operator:

```
Observable<Integer> observable =
    Observable.range(0, 10).flatMap(i -> {
        return (i == 3)
            ? Observable.error(new Exception("An error occurred"))
            : Observable.just(i);
    });
observable.subscribe(i -> {
}, throwable -> {
    Log.e(TAG, "onError(): " + throwable.getMessage());
});
```

Returning an Observable.error()

## Fatal errors

However, there are some `Exceptions` that are too fatal to recover from and are not propagated down to the `Observer`, that is, instances of the following classes: `VirtualMachineError`, `ThreadDeath`, and `LinkageError`). Instead, these Exceptions are thrown from the thread where the `Exception` occurred.

```
Observable<Integer> observable =
    Observable.range(0, 10).subscribeOn(
        Schedulers.io()
    ).map(i -> {
        if (i == 3) {
           throw new OutOfMemoryError();
        }
        return i * 2;
    });

observable.observeOn(
    Schedulers.computation()
).subscribe(i -> {
}, throwable -> {
    Log.e("onError(): " + throwable.getMessage());
});
```

OutOfMemoryError() not captured in the .OnError() call

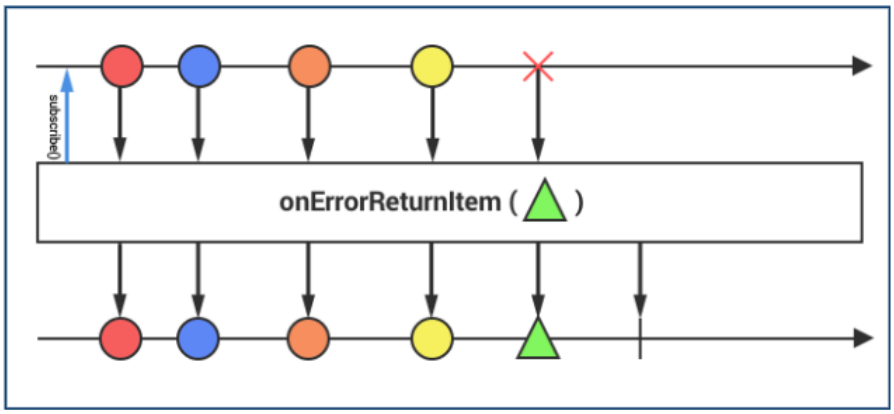
# Error-handling Operators

## Using error-handling operators

To handle the case of fetching an image over the network, we can make use of any of the following error-handling operators.

### `Observable.onErrorReturnItem()`

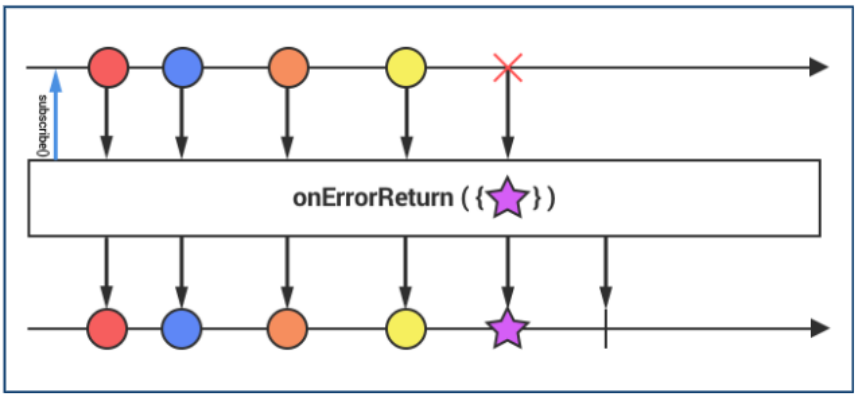
`Observable.onErrorReturnItem(T item)`: When an error is received upstream, this operator returns the item provided to it downstream instead of the received error.




Marble diagram of .onErrorReturnItem()

### `Observable.onErrorReturn()`

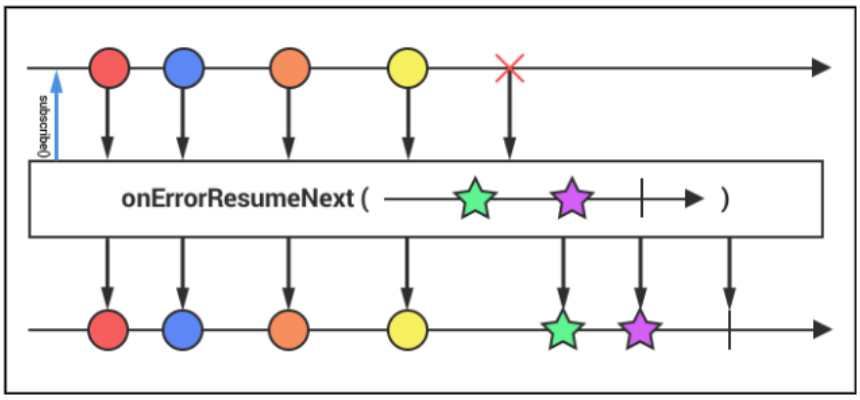
The `Observable.onErrorReturn(Function<? super Throwable, ? extends T> valueSupplier)` method: When an error is received upstream, this operator invokes the function provided and returns the result of that function downstream instead of the received error. This operator is preferred if we want to return a different item depending on the error.




Marble diagram of .onErrorReturn()

### `Observable.onErrorResumeNext()`

The`Observable.onErrorResumeNext(ObservableSource<? extends T> next))` method: This id similar to `.onErrorReturnItem()` but instead of returning an item, it passes control to the provided `Observable` instead of passing the received error.

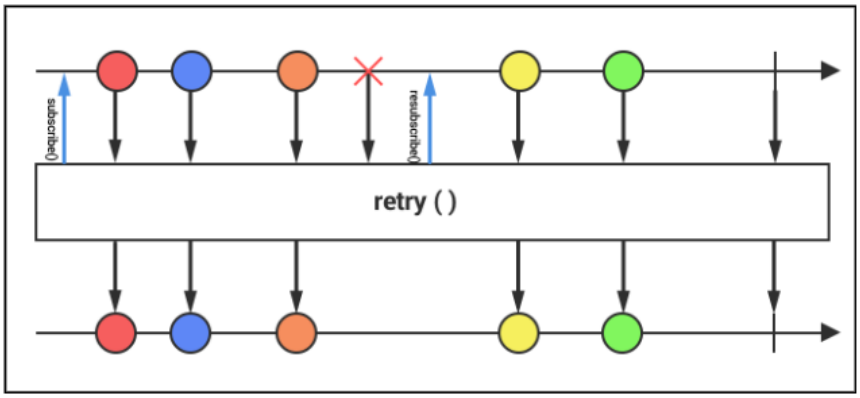



Marble diagram of .onErrorResumeNext()

# Retry Operators

### `Observable.retry()`

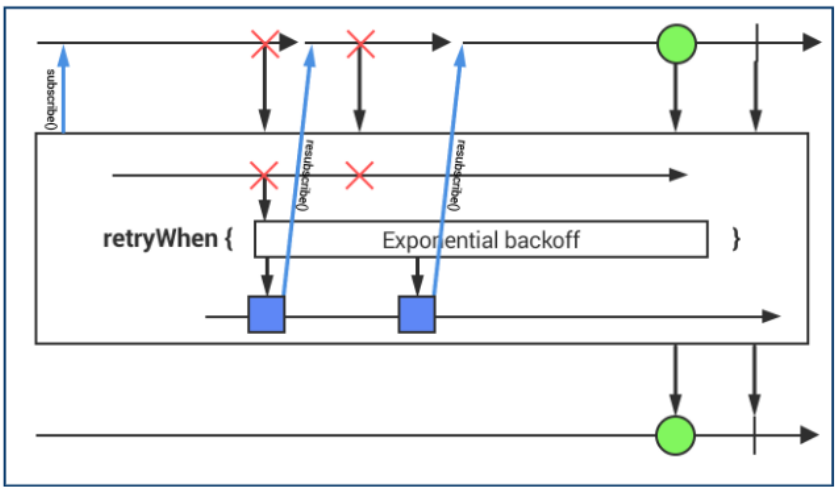
A straightforward approach to retry a failed operation is to use `.retry()`operator intercepts error notifications received upstream and does not pass those through its instances of `Observer`. Instead, it will resubscribe to the source `Observable` and give it the opportunity to complete its sequence without error.




Marble diagram of .retry()

### `Observable.retryWhen()`

One common way of retrying when it comes to networking issues is to use **exponential backoff** with the rate of retries. That is, the delay of a subsequent retry would be some multiple of the delay of the previous one. To do this, we can make use of `.retryWhen()`.




Marble diagram of .retryWhen()

# Undelivered Errors

Обробка недоставлених помилок

Метод RxJavaPlugins.onError() викликається у внутрішньому середовищі RxJava для обробки недоставленої помилки. По суті, цей метод .onError() за замовчуванням поводиться так, як можна було б очікувати, якби помилка виникла поза контекстом RxJava. Трасування стеку

помилки роздруковується, а обробка делегується обробнику виключень UncaughtExceptionHandler потоку, в якому сталася помилка.

Альтернативно, споживач повідомлень про помилки за замовчуванням також може бути наданий для обробки недоставлених помилок за допомогою RxJavaPlugins.setErrorHandler(Consumer<Throwable>). Таким чином, у RxJava буде створено статичну змінну, яка передаватиме недоставлені помилки наданому обробнику, а не використовуватиме RxjavaPlugins.onError(). Цей підхід є кращим, оскільки ми можемо передати помилку до модуля журналювання нашого додатку.