Contents

[Reactive programming 3](#_Toc149848389)

[Asynchronous programming 3](#_Toc149848390)

[Observable 4](#_Toc149848391)

[Stream 5](#_Toc149848392)

[Asynchronous programming with observable streams 5](#_Toc149848393)

[Forms of data streams to handle 5](#_Toc149848394)

[RxJava in Action 6](#_Toc149848395)

[RxJava’s Essential Characteristics 8](#_Toc149848396)

[Observer pattern 9](#_Toc149848397)

[Iterator pattern 9](#_Toc149848398)

[Functional programming 10](#_Toc149848399)

[Conciseness 11](#_Toc149848400)

[Lazy vs. eager evaluation 11](#_Toc149848401)

[RxJava Core Components 12](#_Toc149848402)

[Streams 12](#_Toc149848403)

[RxJava main constructs 13](#_Toc149848404)

[Observable and Observer 13](#_Toc149848405)

[Operator 15](#_Toc149848406)

[Marble Diagrams 15](#_Toc149848407)

[Creating an Observable 17](#_Toc149848408)

[Cold vs. Hot Observables 19](#_Toc149848409)

[Lazy Emissions 21](#_Toc149848410)

[Operators 23](#_Toc149848411)

[Transform: Map and Filter 23](#_Toc149848412)

[Transform: FlatMap and ConcatMap 23](#_Toc149848413)

[.map() vs. .flatMap() 24](#_Toc149848414)

[Transform: Skip, Take, First, and Last 25](#_Toc149848415)

[Combine: Merge, Concat, and Zip 27](#_Toc149848416)

[Aggregate: ToList, Reduce, and Collect 30](#_Toc149848417)

[Utility Operators: DoOnEach, and Cache 32](#_Toc149848418)

[Reusing Operator Chains 33](#_Toc149848419)

[Multithreading 35](#_Toc149848420)

[Default Synchronicity of RxJava 35](#_Toc149848421)

[Adding Asynchronicity 35](#_Toc149848422)

[Schedulers 37](#_Toc149848423)

[Types of Schedulers 37](#_Toc149848424)

[Schedulers with Operators 38](#_Toc149848425)

[Scheduler Behavior 39](#_Toc149848426)

[.subscribeOn() 39](#_Toc149848427)

[.observeOn() 40](#_Toc149848428)

[Achieving True Concurrency 40](#_Toc149848429)

[Reactive Modeling on Android 42](#_Toc149848430)

[Bridging the Non-reactive and Reactive Worlds 42](#_Toc149848431)

[RxJava's Consequences: Method Signature 42](#_Toc149848432)

[RxJava's Consequences: Laziness 43](#_Toc149848433)

[Reactive Everything: Long Operations 43](#_Toc149848434)

[Reactive Everything: Completable, Single, and Maybe 44](#_Toc149848435)

[Reactive Everything: Replacing Callbacks 45](#_Toc149848436)

[Reactive Everything: Multicasting 46](#_Toc149848437)

[Reactive Everything: View Events 47](#_Toc149848438)

[Disposable and the Activity/Fragment Lifecycle 48](#_Toc149848439)

[Backpressure 50](#_Toc149848440)

[Flowable 50](#_Toc149848441)

[Subscriber 51](#_Toc149848442)

[Throttling and Buffering 51](#_Toc149848443)

[Error handling 54](#_Toc149848444)

[Errors Along the Chain 54](#_Toc149848445)

[Error-handling Operators 55](#_Toc149848446)

[Retry Operators 57](#_Toc149848447)

[Undelivered Errors 58](#_Toc149848448)

# Reactive programming

Реактивне програмування можна найпростіше визначити як асинхронне програмування зі спостережуваними потоками. A diagram of a computer program

Description automatically generated

## 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.

A diagram of a task

Description automatically generated

## 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.

A diagram of a function

Description automatically generated

## Stream

A screenshot of a computer

Description automatically generated

## 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.

A diagram of a program

Description automatically generated

## Forms of data streams to handle

A screenshot of a computer

Description automatically generated

## RxJava in Action

A screenshot of a computer code

Description automatically generated

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:

A text on a white background

Description automatically generatedA screenshot of a computer program

Description automatically generated

## RxJava’s Essential Characteristics

### Observer pattern

A diagram of a diagram

Description automatically generated

### Iterator pattern

A close-up of a text

Description automatically generated

### Functional programming

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

### Conciseness

A screenshot of a computer program

Description automatically generated

### Lazy vs. eager evaluation

A screenshot of a text

Description automatically generated

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

A close-up of a diagram

Description automatically generated

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). A screenshot of a computer

Description automatically generated

## Observable and Observer

A screenshot of a computer

Description automatically generated

A close-up of a text

Description automatically generated

A screenshot of a computer program

Description automatically generated

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

A diagram of a company

Description automatically generated

A screenshot of a computer program

Description automatically generated

## 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.

A screenshot of a computer

Description automatically generatedA diagram of a line of colored dots and arrows

Description automatically generated

## Creating an Observable

Різні способи створення Observable

Тепер, коли ми маємо базове, високорівневе розуміння різних компонентів RxJava, давайте зануримося 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. A diagram of a program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

## 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.

A diagram of a diagram

Description automatically generatedA close up of text

Description automatically generated

## Lazy Emissions

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generatedA close-up of text

Description automatically generated

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.

A close up of a computer screen

Description automatically generated

# Operators

## Transform: Map and Filter

A screenshot of a diagram

Description automatically generatedA diagram of a diagram

Description automatically generated

## Transform: FlatMap and ConcatMap

A diagram of a diagram

Description automatically generated

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.**

A diagram of a diagram

Description automatically generated

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

A diagram of a graph

Description automatically generated

A diagram of a diagram

Description automatically generatedA screenshot of a computer

Description automatically generated

## Combine: Merge, Concat, and Zip

A diagram of a diagram

Description automatically generated

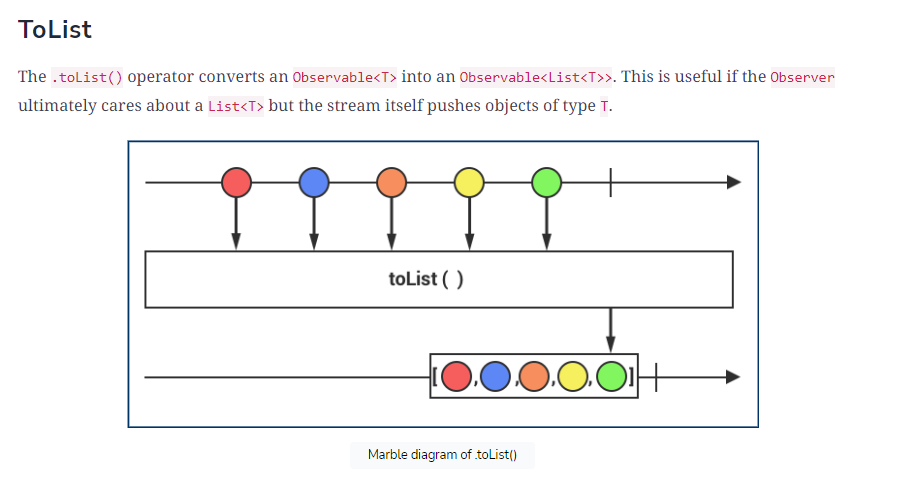
A diagram of a diagram with colored circles and arrows

Description automatically generatedA screenshot of a computer

Description automatically generated

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

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer program

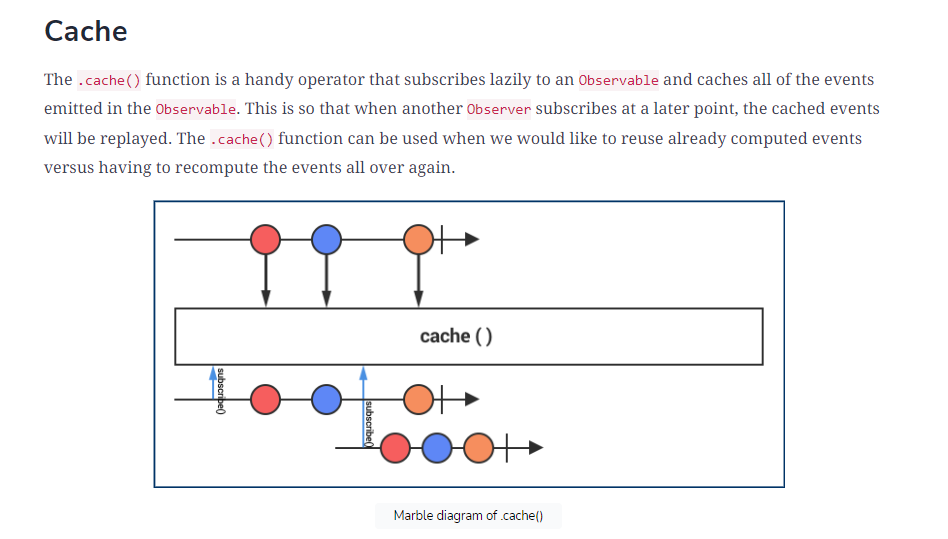
Description automatically generated

## Utility Operators: DoOnEach, and Cache

A screenshot of a computer program

Description automatically generated

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.



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

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

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? A screenshot of a computer program

Description automatically generated

## Adding Asynchronicity

A screen shot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

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

A screenshot of a computer

Description automatically generated

### 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

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

## 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

A close up of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

# Reactive Modeling on Android

## Bridging the Non-reactive and Reactive Worlds

A screenshot of a computer

Description automatically generated

## RxJava's Consequences: Method Signature

A screenshot of a computer program

Description automatically generated

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

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

## Reactive Everything: Replacing Callbacks

A close-up of a text

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

## Reactive Everything: Multicasting

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Reactive Everything: View Events

A screenshot of a computer program

Description automatically generated

A text on a white background

Description automatically generated

## 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().

A computer screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

# 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().

A diagram of a sample

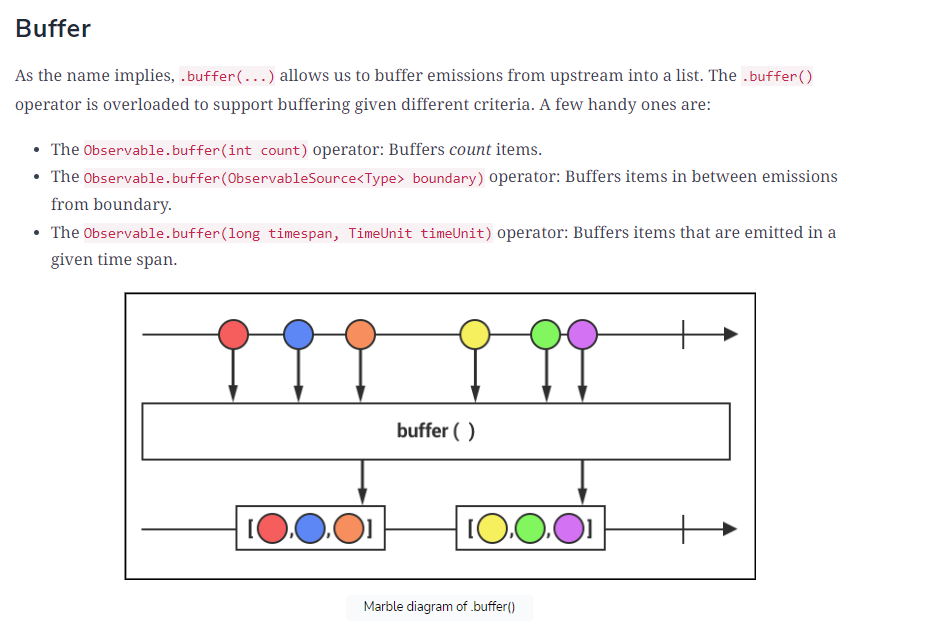
Description automatically generated

A screenshot of a computer program

Description automatically generated

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().

A diagram of a window

Description automatically generated

# Error handling

## 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: A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

## Error-handling Operators

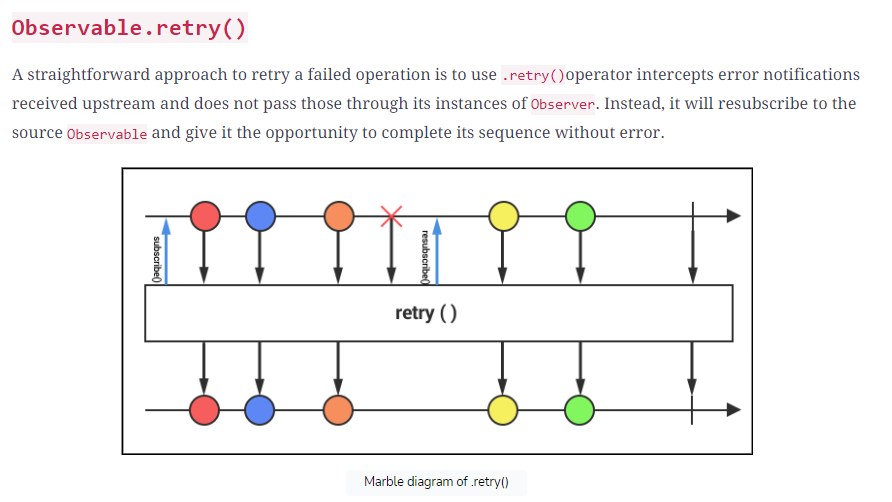
A screenshot of a computer error

Description automatically generatedA screen shot of a computer error

Description automatically generatedA diagram of a error

Description automatically generated

## Retry Operators

A diagram of a network

Description automatically generated

## 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.