ДИСЦИПЛИНА **Алгоритмы и структуры данных с использованием компилируемых языков**

(полное наименование дисциплины без сокращений)

ИНСТИТУТ **Искусственного интеллекта**

КАФЕДРА **Технологий Искусственного Интеллекта**

(полное наименование кафедры)

ВИД УЧЕБНОГО МАТЕРИАЛА **Материалы для практических/семинарских занятий**

(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ **Куликов Александр Анатольевич**

(фамилия, имя, отчество)

СЕМЕСТР **1, 2023-2024**

(указать семестр обучения, учебный год)

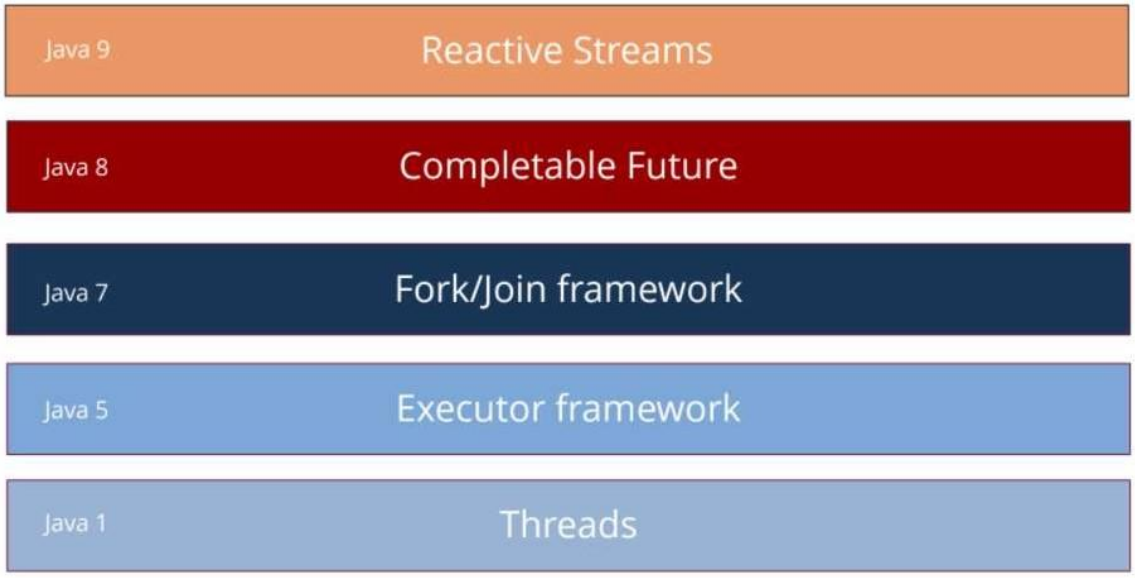
Практическая работа №3

Тема: Реактивность. RxJava

Теоретическое введение

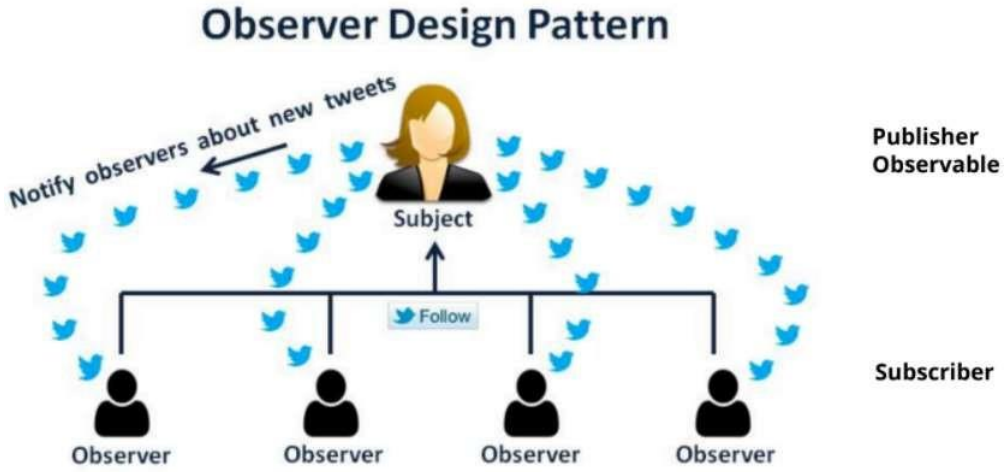
Реактивное программирование — это асинхронность, соединенная с потоковой обработкой данных. То есть если в асинхронной обработке нет блокировок потоков, но данные обрабатываются все равно порциями, то реактивность добавляет возможность обрабатывать данные потоком. Пример, когда начальник поручает задачу Васе, тот должен передать результат Диме, а Дима вернуть начальнику? Но задача — это некая порция, и пока она не будет сделана, дальше передать ее нельзя. Такой подход действительно разгружает начальника, но Дима и Вася периодически простаивают, ведь Диме надо дождаться результатов работы Васи, а Васе — дождаться нового задания.

А теперь представим, что задачу разбили на множество подзадач. И теперь они плывут непрерывным потоком небольших порции данных, конвейер с потоком данных, и каждый обработчик пропускает через себя эти данные, каким-то образом их преобразовывая. В качестве Васи и Димы выступают потоки выполнения (threads), обеспечивая, таким образом, многопоточную обработку данных.



На этой схеме показаны разные технологии распараллеливания, добавлявшиеся в Java в разных версиях. Как видим, спецификация Reactive Streams на вершине — она не заменяет всего, что было до нее, но добавляет самый высокий уровень абстракции, а значит ее использование просто и эффективно. Попробуем в этом разобраться.

Идея реактивности построена на паттерне проектирования Observer, пример с Twitter (запрещена на территории РФ) ниже.

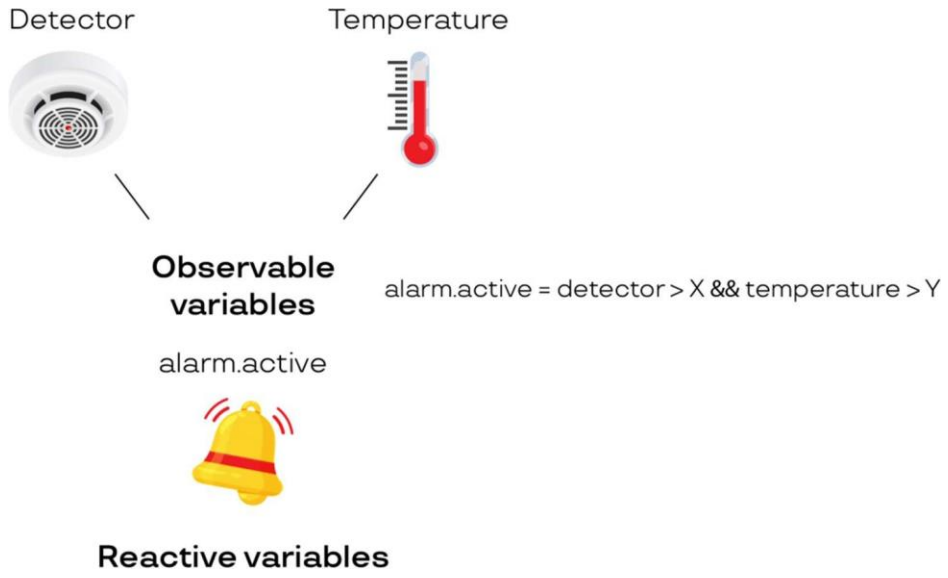


Есть подписчики и то, на что подписываемся. Мы можем подписаться на какое-то сообщество или человека, а потом получать обновления. После подписки, как только появляется новое сообщение, всем подписчикам приходит notify, то есть уведомление. Это базовый паттерн.

В данной схеме есть:

- **Publisher — тот, кто публикует новые сообщения;**
- **Observer — тот, кто на них подписан. В реактивных потоках подписчик обычно называется Subscriber. Термины разные, но по сути это одно и то же. В большинстве сообществ более привычны термины Publisher/Subscriber.**

Это базовая идея, на которой все строится. Один из жизненных примеров реактивности — система оповещения при пожаре. Допустим, надо сделать систему, включающую тревогу в случае превышения задымленности и температуры.



Есть датчик дыма и градусник. Когда дыма становится много и/или температура растет, на соответствующих датчиках увеличивается значение. Когда значение и температура на датчике дыма оказываются выше пороговых, включается колокольчик и оповещает о тревоге.

Если бы был традиционный, не реактивный подход, мы бы писали код, который каждые пять минут опрашивает детектор дыма и датчик температуры, и включает или выключает колокольчик. Однако в реактивном подходе это делает реактивный фреймворк, а мы только прописываем условия: колокольчик активен, когда детектор больше X, а температура больше Y. Это происходит каждый раз, когда приходит новое событие.

От детектора дыма идет поток данных: например, значение 10, потом 12, и т.д. Температура тоже меняется, это другой поток данных — 20, 25, 15. Каждый раз, когда появляется новое значение, результат пересчитывается, что приводит к включению или выключению системы оповещения. Нам достаточно сформулировать условие, при котором колокольчик должен включиться.

Если вернуться к паттерну Observer, детектор дыма и термометр — это публикаторы сообщений, то есть источники данных (Publisher), а колокольчик на них подписан, то есть он Subscriber, или наблюдатель (Observer).

Операторы позволяют каким-либо образом трансформировать потоки данных, меняя данные и создавая новые потоки.

Для примера рассмотрим оператор **distinctUntilChanged**. Он убирает одинаковые значения, идущие друг за другом. Действительно, если значение на детекторе дыма не изменилось — зачем на него реагировать и что-то там пересчитывать:

Основы RxJava: наблюдатель и наблюдаемый (Observable и Observer)

В Java 9 нет реализации реактивных потоков — только спецификация. Но есть несколько библиотек — реализаций реактивного подхода.

RxJava - библиотека для обеспечения реактивного программирования в разработке, особенно на мобильные платформы.

Теперь, согласно определению, определим термины - асинхронный, основанный на событиях и т.д.

- *Асинхронный*

Это означает, что разные части программы выполняются одновременно.

- *Основанный на событиях*

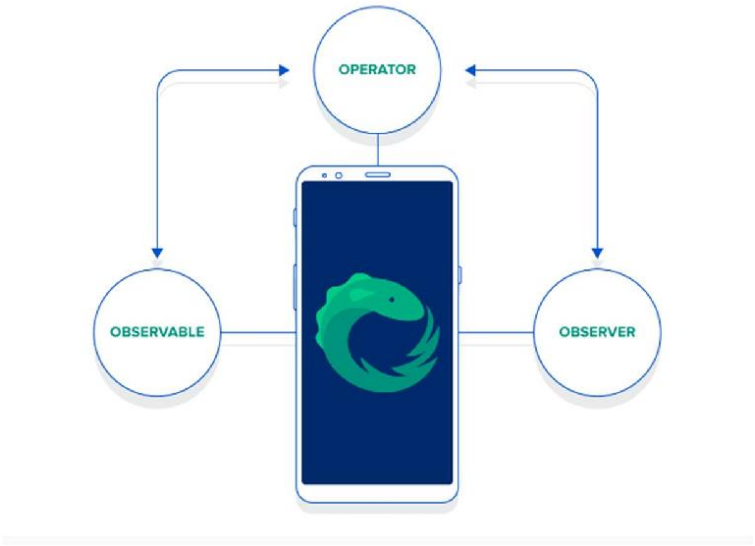
Программа выполняет код на основе событий, сгенерированных во время выполнения программы. Например, нажатие кнопки запускает событие, а затем обработчик событий программы получает это событие и выполняет соответствующую работу.

- *Наблюдаемые последовательности*

Observable и Flowable берут некоторые элементы и передают их подписчикам. Итак, эти элементы называются наблюдаемыми последовательностями или потоком данных.

- RxJava освобождает нас от ада обратного вызова, предоставляя стиль программирования. Мы можем подключать различные преобразования, которые напоминают функциональное программирование.

RxJava использует Observer и observable pattern, где субъект все время поддерживает своих наблюдателей, и если происходят какие-либо изменения, он уведомляет их, вызывая один из их методов.



Rx базируется на двух фундаментальных типах, в то время, как некоторые другие расширяют их функциональность. Этими базовыми типами являются **Observable** и **Observer**,

Rx построена на паттерне Observer. В этом нет ничего нового, обработчики событий уже существуют в Java (например, JavaFX EventHandler), однако они проигрывают в сравнении с Rx по следующим причинам:

- Обработку событий в них сложно компоновать
- Их вызов нельзя отложить
- Могут привести к утечке памяти
- Не существует простого способа сообщить об окончании потока событий
- Требуют ручного управления многопоточностью.

Observable

Observable – первый базовый тип. Этот класс содержит в себе основную часть реализации Rx, включая все базовые операторы. Рассмотрим их позже, а пока следует понять принцип работы метода **subscribe**. Вот ключевая перегрузка :

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

Метод **subscribe** используется для получения данных выдаваемых observable. Эти данные передаются наблюдателю, который предполагает их обработку в зависимости от требований потребителя. Наблюдатель в этом случае является реализацией интерфейса **Observer**. Observable сообщает три вида событий:

- Данные
- Сигнал о завершении последовательности (что означает, что новых данных больше не будет)
- Ошибку, если последовательность завершилась по причине исключительной ситуации (это событие так же предполагает завершение последовательности)

Observer

В Rx предусмотрена абстрактная реализация Observer, **Subscriber**. **Subscriber** реализует дополнительную функциональность и, как правило, именно его следует использовать для реализации **Observer**. Однако, для начала, рассмотрим только интерфейс:

```
interface Observer<T> {
   void onCompleted();
   void onError(java.lang.Throwable e);
   void onNext(T t);
}
```

Эти три метода являются поведением, которое описывает реакцию наблюдателя на сообщение от observable. **onNext** у наблюдателя будет вызван 0

или более раз, опционально сопровождаясь **onCompleted** или **onError**. После них вызовов больше не будет.

Реализация Observable и Observer

Можно в вручную реализовать **Observer** и **Observable**. В реальности в этом, как правило, нет необходимости: Rx предоставляет готовые решения, чтобы упростить разработку. Это также может быть не совсем безопасно, поскольку взаимодействие между частями библиотеки Rx включает в себя принципы и внутреннюю инфраструктуру, которые могут быть не очевидны новичку. В любом случае, будет проще для начала использовать множество инструментов уже предоставленных библиотекой для создания необходимого нам функционала. Чтобы подписаться на observable, совсем нет необходимости в реализации **Observer**. Существуют другие перегрузки метода **subscribe**, которые принимают в качестве аргументов соответствующие функции для **onNext**, **onError** и **onSubscribe**, инкапсулирующие создание экземпляра **Observer**.

Subject

Subject'ы являются расширением **Observable**, одновременно реализуя интерфейс Observer. Они могут принимать сообщения о событиях (как observer) и сообщать о них своим подписчикам (как observable). Это делает их идеальной отправной точкой для знакомства с Rx кодом: когда есть данные, поступающие извне, можно передать их в Subject, превращая их таким образом в observable. Существует несколько реализаций Subject. Рассмотрим самые важные из них.

PublishSubject

PublishSubject – самая простая реализация **Subject**. Когда данные передаются в **PublishSubject**, он выдает их всем подписчикам, которые подписаны на него в данный момент.

```
public static void main(String[] args) {
    PublishSubject<Integer> subject = PublishSubject.create();
    subject.onNext(1);
    subject.subscribe(System.out::println);
    subject.onNext(2);
    subject.onNext(3);
    subject.onNext(4);
}
```

Вывод:

```
2
3
4
```

Как видим, **1** не была напечатана из-за того, что не были подписаны в момент когда она была передана. После того как подписались, начали получать все значения поступающие в subject.

Здесь впервые используем метод **subscribe**, так что стоит уделить этому внимание. В данном случае используем перегруженную версию, которая принимает один объект класса Function, отвечающий за **onNext**. Эта функция принимает значение типа Integer и ничего не возвращает. Функции, которые ничего не возвращают также называются actions.

Можно передать эту функцию следующими способами:

- Предоставить объект класса **Action1<Integer>**
- Неявно создать таковой используя лямбда-выражение
- Передать ссылку на существующий метод с соответствующей сигнатурой. В данном случае, **System.out::println** имеет перегруженную версию, которая принимает **Object**, поэтому мы передаем ссылку на него. Таким образом, подписка позволяет нам печатать в основной поток вывода все поступающие в **Subject** числа.

ReplaySubject

ReplaySubject имеет специальную возможность кэшировать все поступившие в него данные. Когда у него появляется новый подписчик, последовательность выдана ему начиная с начала. Все последующие поступившие данные будут выдаваться подписчикам как обычно.

```
ReplaySubject<Integer> s = ReplaySubject.create();
s.subscribe(v -> System.out.println("Early:" + v));
s.onNext(0);
s.onNext(1);
s.subscribe(v -> System.out.println("Late: " + v));
s.onNext(2);
```

Вывод

```
Early:0
Early:1
Late: 0
Late:  1
Early:2
Late: 2
```

Все значения были получены, не смотря на то, что один из подписчиков подписался позже другого. Следует обратить внимание, что до того как получить новое значение, подписчик получает все пропущенные. Таким образом, порядок последовательности для подписчика не нарушен.

Кэшировать всё подряд не всегда лучшая идея, так как последовательности могут быть длинными или даже бесконечными. Фабричный метод **ReplaySubject.createWithSize** ограничивает размер буфера, а **ReplaySubject.createWithTime** время, которое объекты будут оставаться в кеше.

```
ReplaySubject<Integer> s = ReplaySubject.createWithSize(2);
s.onNext(0);
s.onNext(1);
s.onNext(2);
s.subscribe(v -> System.out.println("Late: " + v));
s.onNext(3);
```

Вывод

```
Late: 1
Late: 2
Late: 3
```

Подписчик на этот раз пропустил первое значение, которое выпало из буфера размером 2.

Таким же образом со временем из буфера выпадают объекты у Subject созданного при помощи **createWithTime**.

```
ReplaySubject<Integer> s =ReplaySubject.createWithTime(150, TimeU-
nit.MILLISECONDS, Schedulers.immediate());
s.onNext(0);
Thread.sleep(100);
s.onNext(1);
Thread.sleep(100);
s.onNext(2);
s.subscribe(v -> System.out.println("Late: " + v));
s.onNext(3);
```

Вывод

```
Late: 1
Late: 2
Late: 3
```

Создание **ReplaySubject** с ограничением по времени требует объект планировщика (**Scheduler**), который является представлением времени в Rx.

ReplaySubject.createWithTimeAndSize ограничивает буфер по обоим параметрам.

BehaviorSubject

BehaviorSubject хранит только последнее значение. Это то же самое, что и **ReplaySubject**, но с буфером размером 1. Во время создания ему может быть присвоено начальное значение, таким образом гарантируя, что данные всегда будут доступны новым подписчикам.

```
BehaviorSubject<Integer> s = BehaviorSubject.create();
s.onNext(0);
s.onNext(1);
s.onNext(2);
s.subscribe(v -> System.out.println("Late: " + v));
s.onNext(3);
```

Вывод

```
Late: 2
Late: 3
```

Начальное значение предоставляется для того, чтобы быть доступным еще до поступления данных.

Так как роль **BehaviorSubject** – всегда иметь доступные данные, считается неправильным создавать его без начального значения, также как и завершать его.

AsyncSubject

AsyncSubject также хранит последнее значение. Разница в том, что он не выдает данных до тех пока не завершится последовательность. Его используют, когда нужно выдать единое значение и тут же завершиться.

```
AsyncSubject<Integer> s = AsyncSubject.create();
s.subscribe(v -> System.out.println(v));
s.onNext(0);
s.onNext(1);
s.onNext(2);
s.onCompleted();
```

Вывод

```
2
```

Обратите внимание, что если бы мы не вызвали **s.onCompleted()**, этот код ничего бы не напечатал.

Неявная инфраструктура

Существуют принципы, которые могут быть не очевидны в коде. Один из важнейших заключается в том, что ни одно событие не будет выдано после того, как последовательность завершена (**onError** или **onCompleted**). Реализация **subject**’ уважает эти принципы:

```
Subject<Integer, Integer> s = ReplaySubject.create();
s.subscribe(v -> System.out.println(v));
s.onNext(0);
s.onCompleted();
s.onNext(1);
s.onNext(2);
```

Вывод

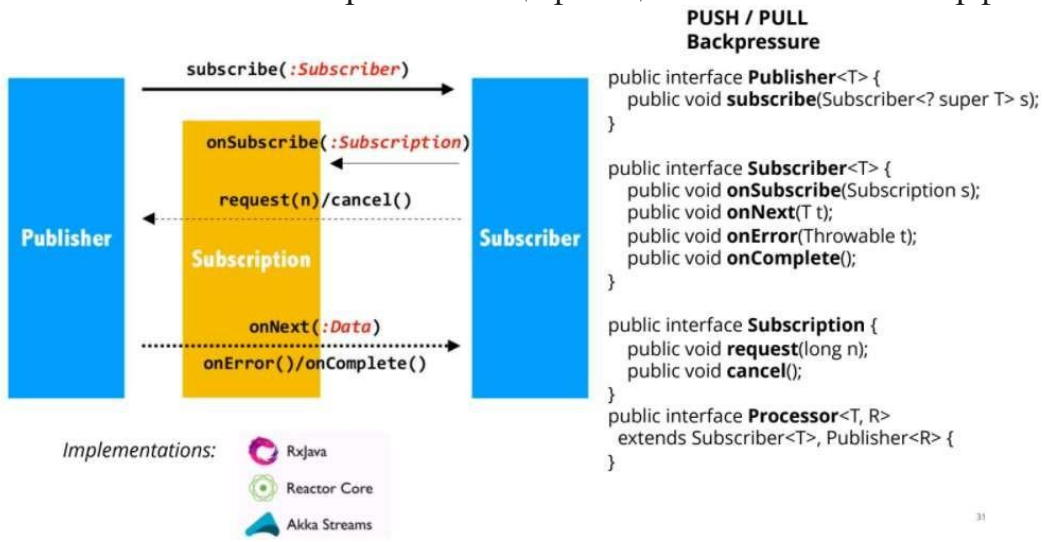
```
0
```

Безопасность не может быть гарантирована везде, где используется Rx, поэтому вам лучше быть осведомленным и не нарушать этот принцип, так как это может привести к неопределенным последствиям.

Реактивные потоки в Java 9 (Java Reactive Streams)

Реактивные потоки вошли в Java 9 как спецификация.

Если предыдущие технологии (Completable Future, Fork/Join framework) получили свою имплементацию в JDK, то реактивные потоки имплементации не имеют. Есть только очень короткая спецификация. Там всего 4 интерфейса:



Если рассматривать пример из картинки про Твиттер, можно сказать, что:

Publisher — тот, кто постит твиты;

Subscriber — подписчик. Определяет , что делать, если:

- Начали слушать поток (onSubscribe). Когда успешно подписались, вызовется эта функция;
- Появилось очередное значение в потоке (onNext);
- Появилось ошибочное значение (onError);
- Поток завершился (onComplete).

Subscription — есть подписка, которую можно отменить (cancel) или запросить определенное количество значений (request(long n)). Можно определить поведение при каждом следующем значении, а можем забирать значения вручную.

Processor — обработчик — это два в одном: он одновременно и Subscriber, и Publisher. Он принимает какие-то значения и куда-то их кладет.

Если надо на что-то подписаться, вызываем Subscribe, подписываемся, и потом каждый раз будем получать обновления. Можно запросить их вручную с помощью request. А можно определить поведение при приходе нового сообщения (onNext): что делать, если появилось новое сообщение, что делать, если пришла ошибка и что делать, если Publisher завершил поток. Мы можем определить эти callbacks, или отписаться (cancel).

PUSH / PULL модели

Существует две модели потоков:

- Push-модель — когда идет «проталкивание» значений.

Например, подписались на кого-то в Telegram или Instagram и получаете оповещения (они так и называются — push-сообщения, их не надо запрашиваеть, они приходят сами). Это может быть, например, всплывающее сообщение. Можно определить, как реагировать на каждое новое сообщение.

- Pull-модель — когда мы сами делаем запрос.

Например, мы не хотим подписываться, т.к. информации и так слишком много, а хотим сами заходить на сайт и узнавать новости.

Для Push-модели определяем callbacks, то есть функции, которые будут вызваны, когда придет очередное сообщение, а для Pull-модели можно воспользоваться методом request, когда мы захотим узнать, что новенького.

Pull-модель очень важна для Backpressure — «напирания» сзади. Что же это такое?

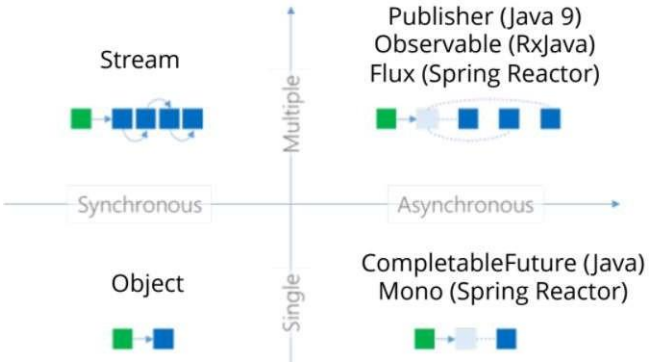
Вы можете быть просто заспамленными своими подписками. В этом случае прочитать их все нереально, и есть шанс потерять действительно важные данные — они просто утонут в этом потоке сообщений. Когда подписчик из-за большого потока информации не справляется со всем, что публикует Publisher, получается Backpressure. В этом случае можно использовать Pull-модель и делать request по одному сообщению, прежде всего из тех потоков данных, которые наиболее важны для вас.

Implementations

Посмотрим подробнее на Spring'овский Reactor.

Function may return…

Обобщим, что может возвращать функция:



- Single/Synchronous;

Обычная функция возвращает одно значение, и делает это синхронно.

- Multipple/Synchronous;

Если мы используем Java 8, можем возвращать из функции поток данных Stream. Когда вернулось много значений, их можно отправлять на обработку. Но

мы не можем отправить на обработку данные до того, как все они получены — ведь Stream работают только синхронно.

- Single/Asynchronous;

Здесь уже используется асинхронный подход, но функция возвращает только одно значение:

- либо CompletableFuture (Java), и через какое-то время приходит асинхронный ответ;
- либо Mono, возвращающая одно значение в библиотеке Spring Reactor.
- Multiple/Asynchronous.

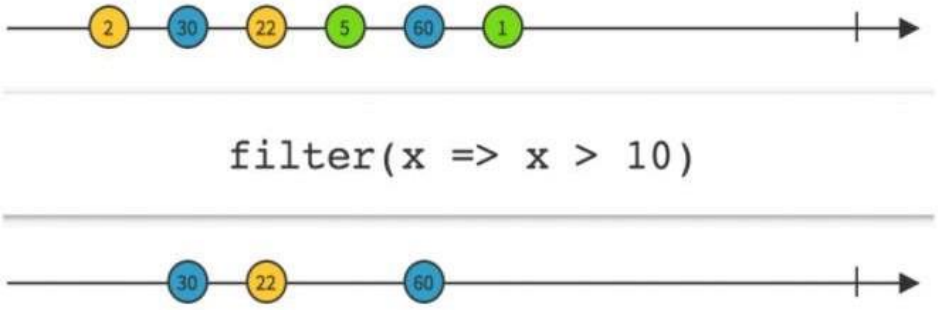
А вот тут как раз — реактивные потоки. Они асинхронные, то есть возвращают значение не сразу, а через какое-то время. И именно в этом варианте можно получить поток значений, причем эти значения будут растянуты во времени Таким образом, мы комбинируем преимущества потоков Stream, позволяющих вернуть цепочку значений, и асинхронности, позволяющей отложить возврат значения. Например, вы читаете файл, а он меняется. В случае Single/Asynchronous вы через какое-то время получаете целиком весь файл. В случае Multiple/Asynchronous вы получаете поток данных из файла, который сразу же можно начинать обрабатывать. То есть можно одновременно читать данные, обрабатывать их, и, возможно, куда-то записывать. . Реактивные асинхронные потоки называются:

- Publisher (в спецификации Java 9);
- Observable (в RxJava);
- Flux (в Spring Reactor).

Операторы

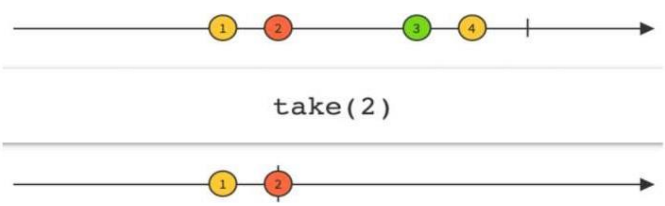
В реактивных потоках огромное количество операторов. Многие из них похожи на те, которые есть в обычных стримах Java. Мы рассмотрим только несколько самых распространенных операторов, которые понадобятся нам для практического примера применения реактивности.

Filter operator



По синтаксису этот фильтр точно такой же, как обычный. Но если в стриме Java 8 все данные есть сразу, здесь они могут появляться постепенно. Стрелки

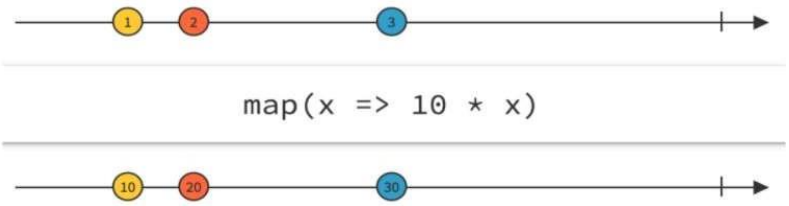
вправо — это временная шкала, а в кружочках находятся появляющиеся данные. Мы видим, что фильтр оставляет в итоговом потоке только значения, превышающие 10.



Take 2 означает, что нужно взять только первые два значения.

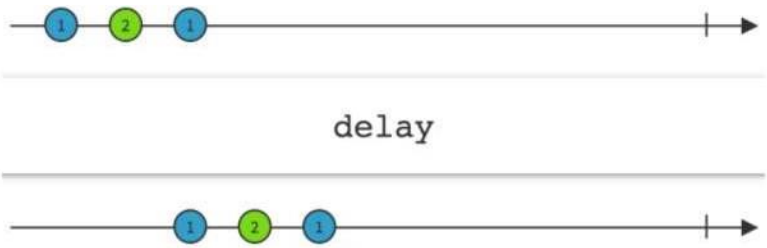
Map operator

Оператор Map тоже хорошо знаком:



Это действие, происходящее с каждым значением. Здесь — умножить на десять: было 3, стало 30; было 2, стало 20 и т.д.

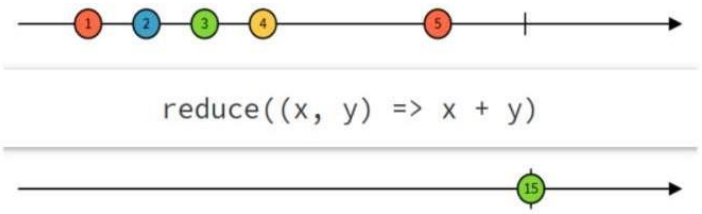
Delay operator



Задержка: все операции сдвигаются. Этот оператор может понадобиться, когда значения уже генерируются, но подготовительные процессы еще происходят, поэтому приходится отложить обработку данных из потока.

Reduce operator

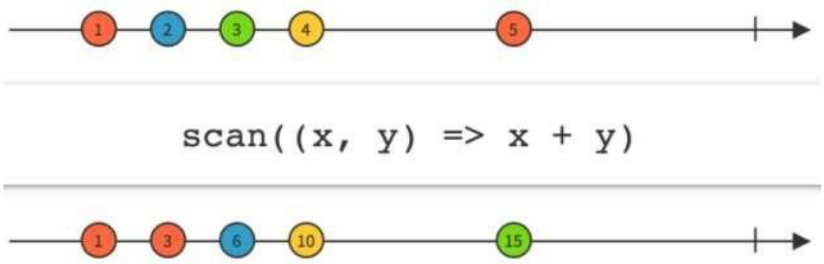
Еще один известный оператор:



Он дожидается конца работы потока (onComplete) — на схеме она представлена вертикальной чертой. После чего мы получаем результат — здесь это число 15. Оператор reduce сложил все значения, которые были в потоке.

Scan operator

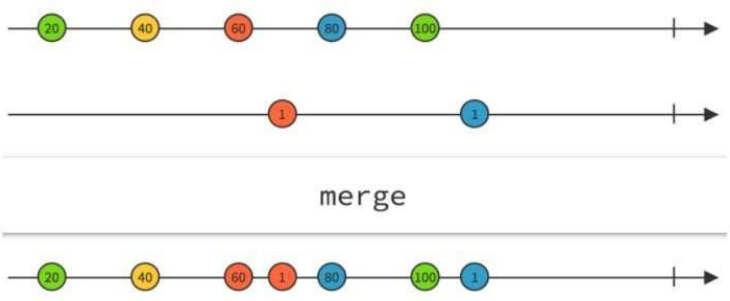
Этот оператор отличается от предыдущего тем, что не дожидается конца работы потока.



Оператор scan рассчитывает текущее значение нарастающим итогом: сначала был 1, потом прибавил к предыдущему значению 2, стало 3, потом прибавил 3, стало 6, еще 4, стало 10 и т.д. На выходе получили 15. Дальше мы видим вертикальную черту — onComplete. Но, может быть, его никогда не произойдет: некоторые потоки не завершаются. Например, у термометра или датчика дыма нет завершения, но scan поможет рассчитать текущее суммарное значение, а при некоторой комбинации операторов — текущее среднее значение всех данных в потоке.

Merge operator

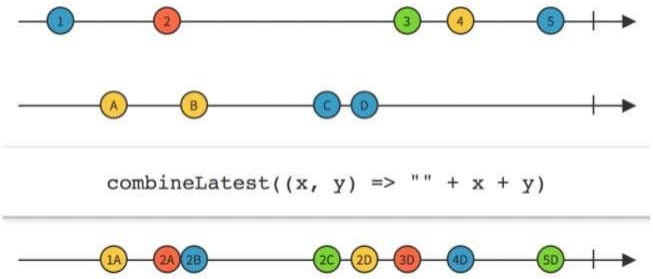
Объединяет значения двух потоков.



Например, есть два температурных датчика в разных местах, а нам нужно обрабатывать их единообразно, в общем потоке .

Combine latest

Получив новое значение, комбинирует его с последним значением из предыдущего потока.



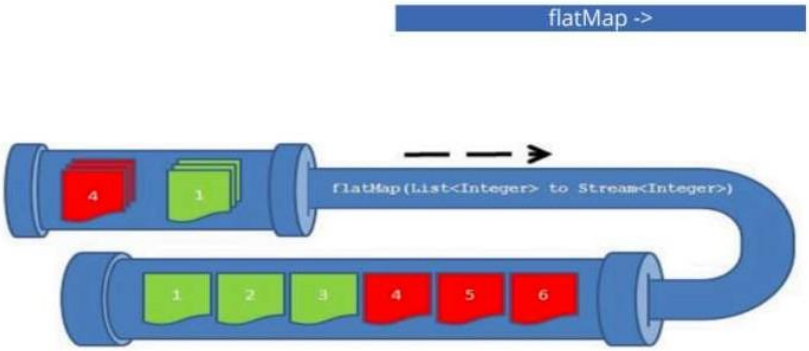
Если в потоке возникает новое событие, мы его комбинируем с последним полученным значением из другого потока. Скажем, таким образом мы можем комбинировать значения от датчика дыма и термометра: при появлении нового значения температуры в потоке temperatureStream оно будет комбинироваться с последним полученным значением задымленности из smokeStream. И мы будем получать пару значений. А уже по этой паре можно выполнить итоговый расчет:

temperatureStream.combineLatest(smokeStream).map((x, y) -> x > X && y > Y)

В итоге на выходе у нас получается поток значений true или false — включить или выключить колокольчик. Он будет пересчитываться каждый раз, когда будет появляться новое значение в temperatureStream или в smokeStream.

FlatMap operator

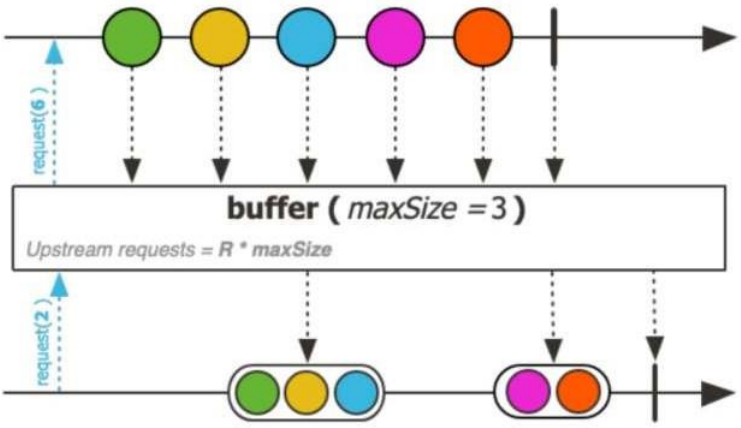
Этот оператор вам, скорее всего, знаком по стримам Java 8. Элементами потока в данном случае являются другие потоки. Получается поток потоков. Работать с ними неудобно, и в этих случаях нам может понадобиться «уплостить» поток.



Можно представить такой поток как конвейер, на который ставят коробки с запчастями. До того, как мы начнем их применять, запчасти нужно достать из коробок. Именно это делает оператор flatMap.

Flatmap часто используется при обработке потока данных, полученных с сервера. Т.к. сервер возвращает поток, чтобы мы смогли обрабатывать отдельные данные, этот поток сначала надо «развернуть». Это и делает flatMap.

Buffer operator



Это оператор, который помогает группировать данные. На выходе Buffer получается поток, элементами которого являются списки (List в Java). Он может пригодиться, когда мы хотим отправлять данные не по одному, а порциями.

Реактивные потоки позволяют разбить задачу на подзадачи, и обрабатывать их маленькими порциями. Но иногда лучше наоборот, собрать много маленьких частей в блоки. Скажем, продолжая пример с конвейером и запчастями, нам может понадобиться отправлять запчасти на другой завод (другой сервер). Но каждую отдельную запчасть отправлять неэффективно. Лучше их собрать в коробки, скажем по 100 штук, и отправлять более крупными партиями.

На схеме выше мы группируем отдельные значения по три элемента (так как всего их было пять, получилась «коробка» из трех, а потом из двух значений). То есть если flatMap распаковывает данные из коробок, buffer, наоборот, упаковывает их.

Всего существует более сотни операторов реактивного программирования. Здесь разобрана только небольшая часть.

Методы для побочных эффектов (side effects)

Методы побочных эффектов не влияют на поток сам по себе. Вместо этого они вызываются, когда происходят определенные события, чтобы позволить реагировать на эти события. Например: если заинтересованы в том, чтобы делать что-то вне ваших Subscriber обратных вызовов при возникновении какой-либо ошибки, вы должны использовать doOnError() метод и передавать ему функциональный интерфейс, который будет использоваться при возникновении ошибки:

```
someObservable
    .doOnError(new Action1() {
      @Override
      public void call(Throwable t) {
        // use this callback to clean up resources,
        // log the event or or report the
        // problem to the user
      }
    })
    //…
```

Наиболее важной частью является call() метод. Код этого метода будет выполнен до Subscriberвызова onError() метода.

В дополнение к исключениям RxJava предлагает еще много событий, на которые вы можете реагировать:

События и соответствующие им побочные эффекты		
метод	**Функциональный интерфейс**	**Мероприятие**

doOnSubscribe ()	Action0	Подписчик подписывается на Observable
doOnUnsubscribe ()	Action0	Подписчик отписывается от подписки
doOnNext ()	Action1 <T>	Следующий элемент выбрасывается
doOnCompleted ()	Action0	The Observable больше не будет излучать предметы
doOnError ()	Action1 <T>	Произошла ошибка
doOnTerminate ()	Action0	Либо произошла ошибка, либо Observable больше не будет излучать элементы
doOnEach ()	Action1 <Уведомление <T >>	Либо элемент был выпущен, наблюдаемое завершено, либо произошла ошибка. Объект Notification содержит информацию о типе события

doOnRequest ()	Action1 <Long>	Нижестоящий оператор запрашивает выделение большего количества элементов

<T> относится либо к типу испускаемого предмета, либо, в случае onError() метода, к типу броска Throwable. Все функциональные интерфейсы имеют тип Action0 или Action1 . Это означает, что отдельные методы этих интерфейсов ничего не возвращают и принимают либо нулевые аргументы, либо один аргумент, в зависимости от конкретного события. Поскольку эти методы ничего не возвращают, они не могут использоваться для изменения испускаемых элементов и, таким образом, никоим образом не изменяют поток элементов. Вместо этого эти методы предназначены для создания побочных эффектов, таких как запись чего-либо на диск, очистка состояния или чего-либо еще, что манипулирует состоянием самой системы вместо потока событий.

Для чего они полезны?

Теперь, поскольку они не меняют поток предметов, для них должно быть другое использование. Три примера того, чего можно достичь с помощью этих методов:

- Использовать doOnNext() для отладки
- Использовать doOnError() внутри flatMap() для обработки ошибок
- Используйте doOnNext() для сохранения / кэширования результатов сети

Итак, давайте посмотрим на эти примеры в деталях.

Использование doOnNext () для отладки

С RxJava Observable иногда работает не так, как ожидалось. Особенно, когда только начинаете его использовать. Поскольку используется свободный API для преобразования какого-либо источника во что-то, на что вы хотите подписаться, вы увидите только то, что получите в конце этого конвейера преобразования. Есть «плавный» API для перемещения из одного потока одного типа в другой поток другого типа. Но что, если это не сработает, как ожидалось? С Java 8 Streams есть peek() метод. Можно использовать doOnNext() метод в

любом месте конвейера обработки, чтобы увидеть, что происходит и каков промежуточный результат.

Вот пример этого:

```
Observable someObservable = Observable
        .from(Arrays.asList(new Integer[]{2, 3, 5, 7, 11}))
        .doOnNext(System.out::println)
        .filter(prime -> prime % 2 == 0)
        .doOnNext(System.out::println)
        .count()
        .doOnNext(System.out::println)
        .map(number -> String.format("Contains %d elements", number));
Subscription subscription = o.subscribe(
        System.out::println,
        System.out::println,
        () -> System.out.println("Completed!"));
```

И вот вывод этого кода:

```
2
3
3
5
5
7
7
11
11
4
Contains 4 elements
Completed!
```

Таким образом, можно получить ценную информацию о том, что происходит, когда ваш Observable ведет себя не так, как вы ожидали.

Эти `doOnError()` и `doOnCompleted()` методы могут быть также полезны для отладки состояния

Использование doOnError() внутри flatMap()

Допустим, вы используете Retrofit(*Retrofit — это REST клиент для Java и Android. Он позволяет легко получить и загрузить JSON (или другие структурированные данные) через веб-сервис на основе REST.*) для доступа к какому-то ресурсу по сети. Поскольку Retrofit поддерживает наблюдаемые объекты, можно легко использовать эти вызовы в своей цепочке обработки с

помощью flatMap(). Сетевые вызовы могут пойти не так во многих отношениях, особенно на мобильных устройствах. В flatMap() методе есть Observable. Таким образом, можно использовать doOnError() метод, чтобы каким-то образом изменить пользовательский интерфейс, но при этом иметь рабочий поток Observable для будущих событий.

Итак, как это выглядит:

```
flatMap(id -> service.getPost()
      .doOnError(t -> {
        // report problem to UI
      })
      .onErrorResumeNext(Observable.empty())
)
```

Этот метод особенно полезен, если вы запрашиваете удаленный ресурс в результате потенциально повторяющихся событий пользовательского интерфейса.

Использование doOnNext() для сохранения/кеширования сетевых результатов

Если в какой-то момент вашей цепочки вы совершаете сетевые вызовы, вы можете использовать doOnNext() для сохранения входящих результатов в вашей локальной базе данных или поместить их в какой-то кеш. Пример на слайде:

```
// getOrderById is getting a fresh order
// from the net and returns an observable of orders
// Observable<Order> getOrderById(long id) {…}

Observable.from(aListWithIds)
      .flatMap(id -> getOrderById(id)
                  .doOnNext(order -> cacheOrder(order))
      // carry on with more processing
```

.

Обработка ошибок в RxJAVA

В роли базового обработчика ошибок в RxJava используется RxJavaPlugins.onError. Он обрабатывает все ошибки, которые не удается доставить до подписчика. По умолчанию, все ошибки отправляются именно в него, поэтому могут возникать критические сбои приложения.

Если у RxJava нет базового обработчика ошибок — подобные ошибки будут скрыты от нас и разработчики будут находится в неведении относительно потенциальных проблем в коде.

Начиная с версии 2.0.6, RxJavaPlugins.onError разделяет ошибки библиотеки/реализации и ситуации когда ошибку доставить невозможно. Ошибки, отнесенные к категории «багов» вызываются как есть, остальные же оборачиваются в UndeliverableException и после вызываются.

Одна из основных ошибок, с которыми сталкиваются в RxJava — OnErrorNotImplementedException. Эта ошибка возникает, если observable вызывает ошибку, а в подписчике не реализован метод onError. Данная ошибка — пример ошибки, которая для базового обработчика ошибок RxJava является «багом» и не оборачивается в UndeliverableException.

UndeliverableException

Поскольку ошибки относящиеся к «багам» легко исправить — не будем на них останавливаться.Ошибки,которые RxJava оборачивает в UndeliverableException, интереснее, так как не всегда может быть очевидно почему же ошибка не может быть доставлена до onError. Случаи, в которых это может произойти, зависят от того, что конкретно делают источники и подписчики. Примеры рассмотрим ниже, но в общем можно сказать, что такая ошибка возникает, если нет активного подписчика, которому может быть доставлена ошибка.

Пример с zipWith()

Первый вариант, в котором можно вызвать `UndeliverableException` — оператор `zipWith`.

```
val observable1 = Observable.error<Int>(Exception())
val observable2 = Observable.error<Int>(Exception())
val zipper = BiFunction<Int, Int, String> { one, two -> "$one - $two" }
observable1.zipWith(observable2, zipper)
        .subscribe(
                { System.out.println(it) },
```

```
            { it.printStackTrace() }
    )
```

Можно предположить, что onError будет вызван дважды, но это противоречит спецификации Reactive streams. После единственного вызова терминального события (onError, onCompelete) требуется, чтобы никаких вызовов больше не осуществлялось Получается, что при единственном вызове onError повторный вызов уже невозможен. Что произойдёт при возникновении в источнике второй ошибки? Она будет доставлена в RxJavaPlugins.onError. Простой способ попасть в подобную ситуацию — использовать zip для объединения сетевых вызовов (например, два вызова Retrofit, возвращающие Observable). Если в обоих вызовах возникает ошибка (например, нет интернет соединения) — оба источника вызовут ошибки, первая из которых попадёт в реализацию onError, а вторая будет доставлена базовому обработчику ошибок (RxJavaPlugins.onError).

Пример с ConnectableObservable без подписчиков

ConnectableObservable также может вызвать UndeliverableException. Стоит напомнить, что ConnectableObservable вызывает события независимо от наличия активных подписчиков, достаточно вызвать метод connect(). Если при отсутствии подписчиков в ConnectableObservable возникнет ошибка — она будет доставлена базовому обработчику ошибок.
пример, который может вызвать такую ошибку:

```
someApi.retrofitCall() // Сетевой вызов с использованием Retrofit
    .publish()
    .connect()
```

Если someApi.retrofitCall() вызовет ошибку (например, нет подключения к интернету) — приложение упадет, так как сетевая ошибка будет доставлена базовому обработчику ошибок RxJava. Этот пример кажется выдуманным, но очень легко попасть в ситуацию, когда ConnectableObservable все еще соединен(connected), но подписчиков у него нет.

Обрабатываем ошибки

Первый шаг — посмотреть на возникающие ошибки и попытаться определить что их вызывает. Идеально, если вам удастся исправить проблему у её источника, чтобы предотвратить передачу ошибки в RxJavaPlugins.onError.

Решение для примера с zipWith — взять один или оба источника и реализовать в них один из методов для перехвата ошибок. Например, можно использовать onErrorReturn для передачи вместо ошибки значения по умолчанию. Пример с ConnectableObservable исправить проще — просто надо убедиться в отсоединении Observable в момент, когда последний подписчик отписывается. autoConnect(), к примеру, имеет перегруженную реализацию, которая принимает функцию, отлавливающую момент соединения.

Другой путь решения проблемы — подменить базовый обработчик ошибок своим собственным. Метод RxJavaPlugins.setErrorHandler(Consumer<Throwable>) поможет в этом. Если это подходящее для решение — можно перехватывать все ошибки отправленные в RxJavaPlugins.onError и обрабатывать их по своему усмотрению. Это решение может оказаться довольно сложным — помните, что RxJavaPlugins.onError получает ошибки от всех потоков (streams) RxJava в приложении.

Если вручную создаются Observable, то можно вместо emitter.onError() вызывать emitter.tryOnError(). Этот метод передает ошибку только если поток (stream) не уничтожен (terminated) и имеет подписчиков. Надо помнить, что данный метод экспериментальный.

Горячие и холодные потоки (hot/cold)

В RxJava есть два вида Observable: Hot и Cold.

Cold Observable:

• Не рассылает объекты, пока на него не подписался хотя бы один подписчик;

• Если observable имеет несколько подписчиков, то он будет рассылать всю последовательность объектов каждому подписчику.

Пример cold observable – методы ретрофит-интерфейса. Каждый раз когда вызывается метод subscribe(), выполняется соответствующий запрос на бэкенд и подписчик получает объект-респонс.

Hot Observable:

• Рассылает объекты, когда они появляются, независимо от того есть ли подписчики;

• Каждый новый подписчик получает только новые объекты, а не всю последовательность.

Рассмотрим cold Observables.

Cold Observables

Cold Observables очень похожа на музыкальный компакт-диск, который может быть воспроизведен каждым слушателем, и каждый может слушать эту музыку в любое время.
Точно так же холодные Observables могут воспроизводить свои Наблюдения для каждого Наблюдателя, гарантируя, что все Наблюдатели получают все данные. Большинство управляемых данными Observables холодные, включая фабрики Observable.just () и Observable.fromIterable ().

Пример: у нас есть два наблюдателя, подписанных на Observable. Observable сначала испускает все выбросы первому Observer, а затем вызывает onComplete (). Затем он снова отправляет все выбросы второму Обозревателю и затем вызывает onComplete (). Через два отдельных потока они оба получают одинаковые данные. Это типичное поведение cold Observables:

```
import io.reactivex.Observable;
public class Launcher {
   public static void main(String[] args) {
      Observable<String> source =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsi-
lon");
```

```
            //first observer
            source.subscribe(s -> System.out.println("Observer 1 Received: " +
s));
            //second observer
            source.subscribe(s -> System.out.println("Observer 2 Received: " +
s));
        }
    }
```

Даже после того, как второй наблюдатель преобразовал выбросы, он все равно получает поток собственных выбросов. Использование map() и filter() может по-прежнему создавать cold Observables:

```
    import io.reactivex.Observable;
    public class Launcher {
        public static void main(String[] args) {
            Observable<String> source =
                    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsi-
lon");
            //first observer
            source.subscribe(s -> System.out.println("Observer 1 Received: " +
s));
            //second observer
            source.map(String::length).filter(i -> i >= 5)
                    .subscribe(s -> System.out.println("Observer 2 Received: " + s));
        }
    }
```

Его вывод

```
    Observer 1 Received: Alpha
    Observer 1 Received: Beta
    Observer 1 Received: Gamma
    Observer 1 Received: Delta
    Observer 1 Received: Epsilon
    Observer 2 Received: 5
    Observer 2 Received: 5
    Observer 2 Received: 5
    Observer 2 Received: 7
```

Если надо сделать запрос к базе данных SQLite, можно включить драйвер JDBC SQLite и библиотеку RxJava-JDBC =.Затем можно запросить таблицу базы данных реактивно, например так:

```
import com.github.davidmoten.rx.jdbc.ConnectionProviderFromUrl;
import com.github.davidmoten.rx.jdbc.Database;
import rx.Observable;
import java.sql.Connection;
public class Launcher {
   public static void main(String[] args) {
      Connection conn = new ConnectionProvider-
FromUrl("jdbc:sqlite:/home/thomas/rexon_metals.db").get();
      Database db = Database.from(conn);
      Observable<String> customerNames =
         db.select("SELECT NAME FROM CUS-
TOMER").getAs(String.class);
      customerNames.subscribe(s -> System.out.println(s));
   }
}
```

Observable на основе SQL - это cold. Многие Observables передают данные из ограниченных источников данных (таких как базы данных, текстовые файлы или JSON), и все они холодные. RxJava-JDBC выполняет запросы для каждого наблюдателя. Это означает, что если данные изменяются между двумя подписками, второй наблюдатель может получать выбросы, отличные от первой. Cold Наблюдаемые будут восстанавливать эмиссию для каждого Наблюдателя.

Hot Observables

Hot Observable больше похожа на радиостанцию. В то же время он передает одинаковые выбросы всем наблюдателям. Если наблюдатель подписывается на hot **Observable**, получает те же выбросы, а затем приходит к другому наблюдателю, второй наблюдатель пропустит эти выбросы. Также как радиостанция, если вы включите ее поздно, вы не услышите эту песню. hot Observables обычно представляют события, а не ограниченные наборы данных. События могут нести данные, но они являются чувствительными ко времени компонентами, и более поздние наблюдатели будут пропускать предыдущие данные. Например, событие пользовательского интерфейса JavaFX или Android может быть представлено как Hot Observable. В JavaFX можно использовать метод selectedProperty() объекта ToggleButton Добавить наблюдателя. Затем надо преобразовать логическое излучение в строку, указывающую состояние кнопки (ВВЕРХ или ВНИЗ). Observer используется для отображения в метке:

```
import io.reactivex.Observable;
```

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
public class MyJavaFxApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        ToggleButton toggleButton = new ToggleButton("TOGGLE ME");
        Label label = new Label();
        Observable<Boolean> selectedStates = valuesOf(toggleButton.select-
edProperty());
        selectedStates.map(selected -> selected ? "DOWN" : "UP")
            .subscribe(label::setText);
        VBox vBox = new VBox(toggleButton, label);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
    private static <T> Observable<T> valuesOf(final ObservableValue<T>
fxObservable) {
        return Observable.create(observableEmitter -> {
            //emit initial state
            observableEmitter.onNext(fxObservable.getValue());
            //emit value changes uses a listener
            final ChangeListener<T> listener = (observableValue, prev, cur-
rent) -> observableEmitter.onNext(current);
            fxObservable.addListener(listener);
        });
    }
}
```

JavaFX ObservableValue не имеет ничего общего с RxJava Observable. При каждом нажатии кнопки ToggleButton Observable будет выдавать соответствующее значение true или false в зависимости от состояния. События пользовательского интерфейса JavaFX и Android в основном являются Hot Observable. Можно использовать Hot Observable для отражения запросов сервера. Если добавить Observable Твиты для темы в Твиттере, которая также популярна в

Observable. Hot Observable не должна быть неограниченной, пока она делится выбросами со всеми наблюдателями, Выбросы, не пропущенные при воспроизведении, горячие.

ConnectableObservable

ConnectableObservable является полезным для hot Observable. Это может быть любая Наблюдаемая (в том числе cold), пусть она станет hot, так что все выбросы выдаются Наблюдателям только один раз. Чтобы сделать этот переход, надо вызвать publish() для любого Observable, и генерируется ConnectableObservable. Тем не менее, подписка не может начать выбросы. Нужно вызвать его метод connect(), чтобы начать эмиссию выбросов. Таким образом, можно предварительно настроить своих наблюдателей:

```
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;
public class Launcher {
   public static void main(String[] args) {
      ConnectableObservable<String> source =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsi-
lon").publish();
      //Set up observer 1
      source.subscribe(s -> System.out.println("Observer 1: " + s));
      //Set up observer 2
      source.map(String::length)
            .subscribe(i -> System.out.println("Observer 2: " + i));
      //Fire!
      source.connect();
   }
}
```

Его вывод

```
Observer 1: Alpha
Observer 2: 5
Observer 1: Beta
Observer 2: 4
Observer 1: Gamma
Observer 2: 5
Observer 1: Delta
Observer 2: 5
Observer 1: Epsilon
```

```
Observer 2: 7
```

Обратите внимание, что один наблюдатель получает строку, а другой - длину. Оба чередуются. Подписки предварительно настроены, а затем вызовите connect(), чтобы начать передачу. Каждое излучение отправляется каждому наблюдателю одновременно. Используйте ConnectableObservable, чтобы заставить каждую эмиссию отправляться всем наблюдателям одновременно, это называется многоадресной рассылкой, которая будет подробно обсуждаться позже. ConnectableObservable полезен для предотвращения воспроизведения данных для каждого наблюдателя.

Задание на практическую работу №3

Задание 1

Реализовать следующую систему:

Датчик температуры. Каждую секунду публикует значение температуры (случайное значение от 15 до 30). Датчик CO2. Каждую секунду публикует значение содержания CO2 в воздухе. (Случайное значение от 30 до 100). Сигнализация. Получает значения от датчиков. Если один из показателей превышает норму, выводит предупреждение об этом. Если норму превышаю оба показателя выводит сообщение «ALARM!!!».

Норма показателей: Температура — 25. CO2 — 70.

Обязательно использование классов Observer и Observable из библиотеки RxJava.

Задание 2

2.1.1 Преобразовать поток из 1000 случайных чисел от 0 до 1000 в поток, содержащий квадраты данных чисел.

2.1.2 Преобразовать поток из 1000 случайных чисел от 0 до 1000 в поток, содержащий только числа больше 500.

2.1.3 Преобразовать поток из случайного количества (от 0 до 1000) случайных чисел в поток, содержащий количество чисел.

2.2.1. Даны два потока по 1000 элементов: первый содержит случайную букву, второй — случайную цифру. Сформировать поток, каждый элемент которого объединяет элементы из обоих потоков. Например, при входных потоках (A, B, C) и (1, 2, 3) выходной поток — (A1, B2, B3).

2.2.2. Даны два потока по 1000 элементов. Каждый содержит случайные цифры. Сформировать поток, обрабатывающий оба потока последовательно. Например, при входных потоках (1, 2, 3) и (4, 5, 6) выходной поток — (1, 2, 3, 4, 5, 6).

2.2.3. Даны два потока по 1000 элементов. Каждый содержит случайные цифры. Сформировать поток, обрабатывающий оба потока параллельно. Например, при входных потоках (1, 2, 3) и (4, 5, 6) выходной поток — (1, 4, 2, 5, 3, 6).

2.3.1. Дан поток из 10 случайных чисел. Сформировать поток, содержащий все числа, кроме первых трех.

2.3.2. Дан поток из 10 случайных чисел. Сформировать поток, содержащий только первые 5 чисел.

2.3.3. Дан поток из случайного количества случайных чисел. Сформировать поток, содержащий только последнее число.

Каждый студент выполняет задания 2.1.X, 2.2.X, 2.3.X, где X = номер по журналу%3+1.

Должно быть реализовано при помощи инструментов RxJava.

Задание 3

Реализовать класс UserFriend. Поля — int userId, friendId. Заполнить массив объектов UserFriend случайными данными.

Реализовать функцию: Observable<UserFriend> getFriends(int userId), возвращающую поток объектов UserFriend, по заданному userId. (Для формирования потока из массива возможно использование функции Observable.fromArray(T[] arr)).

Дан массив из случайных userId. Сформировать поток из этого массива. Преобразовать данный поток в поток объектов UserFriend. Обязательно получение UserFriend через функцию getFriends.

Задание 4

Реализовать следующую систему.

Файл. Имеет следующие характеристики:

0. Тип файла (например XML, JSON, XLS)

1. Размер файла — целочисленное значение от 10 до 100.

Генератор файлов — генерирует файлы с задержкой от 100 до 1000 мс.

Очередь — получает файлы из генератора. Вместимость очереди — 5 файлов.

Обработчик файлов — получает файл из очереди. Каждый обработчик имеет параметр — тип файла, который он может обработать. Время обработки файла: «Размер файла*7мс». Система должна быть реализована при помощи инструментов RxJava.