* **map —** преобразует элементы, испускаемые этим потоком, применяя синхронную функцию к каждому элементу.

Flux.just(1, 2, 3)  
 .map(s -> s + 1)  
 .subscribe(System.out::println);  
  
Output:  
2  
3  
4

* **flatMap** — асинхронно преобразует элементы, испускаемые этим потоком, в Publisher-ы, а затем объединяет эти внутренние Publisher-ы в единый поток посредством слияния

flatMap не гарантирует порядок элементов!

пример 1:

Flux.just("1,2,3", "4,5,6")  
 .flatMap(i -> Flux.fromIterable(Arrays.asList(i.split(","))))  
 .collect(Collectors.toList())  
 .subscribe(System.out::println);  
  
Output:  
[1, 2, 3, 4, 5, 6]

**concatMap** — работает почти так же, как flatMap, но сохраняет порядок элементов.

**switchMap —** он похож на flatMap, за исключением того, что он сохраняет результат только последней подписки, отбрасывая предыдущие.

**startWith —** добавляет элементы перед последовательностью.

Flux.range(1, 3)  
 .startWith(Flux.just(9, 8, 7))  
 .subscribe(System.out::println);  
  
Output:  
9  
8  
7

**collectList** -преобразование Flux в Mono (например Flux<T>.collectList() вернет Mono<List<T>>)

Flux.range(1, 3)  
 .collectList()  
 .subscribe(System.out::println);  
  
Output:  
[1, 2, 3]

* **flux** — преобразование Mono в Flux (например Mono<T>.flux() — вернет Flux<T>)

Mono.just(1)  
 .flux()  
 .subscribe(System.out::println);

* **collectSortedList** — преобразование потока в Mono (собирает все элементы, испускаемые этим потоком, пока эта последовательность не завершится, а затем сортирует их в естественном порядке в список)
* **collectList** -преобразование Flux в Mono (например Flux<T>.collectList() вернет Mono<List<T>>)
* **collectMap** — результатом операции collectMap является Mono, содержащий Map, отправленные входящим потоком.
* **collect** — соберает все элементы, испускаемые этим потоком, в контейнер, применяя Java 8 Stream API Collector. Собранный результат будет выпущен, когда эта последовательность завершится.
* **reduce** — позволяет конвертировать все элементы потока в один объект. Также позволяет выполнять агрегатные функции для всей коллекции (например, значения SUM, MIN и MAX)

Пример 1:

Flux.just(4, 5, 6)  
 .reduce(Integer::sum)  
 .subscribe(System.out::println);  
  
Output:  
15

* **runOn**

Reactor помогает распараллелить работу, предоставляя специальный тип *ParallelFlux*, который предоставляет операторы, оптимизированные для параллельной работы.

Чтобы получить ParallelFlux, вы можете использовать оператор *parallel()* на любом Flux. Сам по себе этот метод не распараллеливает работу. Скорее, он делит рабочую нагрузку на так называемые «rails» (по умолчанию столько, сколько имеется ядер в вашем ЦП).

Чтобы сообщить результирующему ParallelFlux, где запускать каждую направляющую (и, соответственно, запускать направляющие параллельно), вы должны использовать **runOn** (планировщик). Обратите внимание, что для параллельной работы рекомендуется использовать специальный планировщик Schedulers.parallel().

Если после параллельной обработки вашей последовательности вы захотите вернуться к «обычному» потоку и последовательно применить остальную часть цепочки операторов, вы можете использовать метод *sequential()* в ParallelFlux.

Пример 1:

Flux.range(1, 10)  
 .parallel(2)  
 .subscribe(i -> System.out.println(Thread.currentThread().getName() + " -> " + i));  
Output:  
main -> 1  
main -> 2  
main -> 3  
main -> 4  
main -> 5  
main -> 6  
main -> 7  
main -> 8  
main -> 9  
main -> 10

* **zip** — объединение данных из двух Publisher-ов

Flux.just("a", "b", "c")  
 .zipWith(Flux.just(1, 2, 3), (word, number) -> word + number)  
 .subscribe(System.out::println);  
Output:  
a1  
b2  
c3

**concat** — применяется если вам нужно объединить потоки так, чтобы сначала шли элементы из первого потока, а затем из второго, независимо от того, как они поступают в режиме реального времени.

concat() и concatWith() — эквивалентны.

Flux<Integer> oddFlux = Flux.just(1, 3);  
Flux<Integer> evenFlux = Flux.just(2, 4);  
  
Flux.concat(evenFlux, oddFlux)  
 .subscribe(value -> System.out.println("Outer: " + value));  
  
Output:  
Outer: 2  
Outer: 4  
Outer: 1  
Outer: 3

* **merge** — объединение элементов двух Publisher-ов в один

merge() и mergeWith() — эквивалентны

* **transform** — позволяет вам инкапсулировать часть оператора в функцию. Эта функция применяется к исходной цепочке операторов во время сборки, чтобы дополнить ее некоторыми инкапсулированными операторами.

Те. мы берем отдельный метод фильтра и map в отдельную функцию, а затем инкапсулируем ее в нашу логику.

Function<Flux<String>, Flux<String>> filterAndMap =  
 f -> f.filter(color -> !color.equals("orange"))  
 .map(String::toUpperCase);  
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))  
 .transform(filterAndMap)  
 .subscribe(d -> System.out.println("MapAndFilter for: " + d));

* **hasElements** — выдает логическое значение true, если эта последовательность Mono/Flux имеет хотя бы один элемент в последовательности (аналог any в Stream API)

Flux<Integer> oddFlux = Flux.just(1, 3);  
  
oddFlux.hasElements()  
 .subscribe(value -> System.out.println("true/false: " + value));  
  
  
Output:  
true/false: true

* **hasElement** — выдает логическое значение true, если какой-либо из элементов этой последовательности Mono/Flux равен предоставленному значению

Flux<Integer> oddFlux = Flux.just(1, 3);  
  
oddFlux.hasElement(1)  
 .subscribe(value -> System.out.println("true/false: " + value));  
  
  
Output:  
true/false: true

* **groupBy** — разделяет последовательность на динамически создаваемый поток (или группы) для каждого уникального ключа этой последовательности

Пример 1:

Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)  
 .groupBy(i -> i % 2 == 0 ? "even:" : "odd:")  
 .concatMap(Flux::collectList)  
 .subscribe(System.out::println);  
Output:  
[1, 3, 5, 11, 13]  
[2, 4, 6, 12]

* **repeat/repeatWhen** — подписка на источник на неопределенный срок после завершения предыдущей подписки.

Обратите внимание, что repeat и repeatWhen работают только до тех пор, пока метод не выдаст ошибку. Если вы хотите повторить попытку при возникновении ошибки, вы можете использовать retry или retryWhen

public class TestWork {  
  
 private int testNumbers(int value) {  
 if (value > 4) {  
 System.out.println("Test checked");  
 }  
 return value;  
 }  
 private Flux<Integer> checkOnErrorMethodRetry() {  
 Flux.range(1, 5)  
 .map(this::testNumbers)  
 .**repeat**(2)  
 .subscribe(value -> System.out.println("Value: " + value),  
 error -> System.out.println("Error: " + error.getMessage()));  
  
 return null; }  
 public static void main(String[] args) {  
 TestWork testWork = new TestWork();  
 System.out.println(testWork.checkOnErrorMethodRetry());}}  
  
Output:Value: 1  
Value: 2  
Test checked  
Value: 3  
Value: 1  
Value: 2  
Test checked  
Value: 3  
Value: 1  
Value: 2  
Test checked  
Value: 3

* **retry/retryWhen** — аналог repeat, но используется если вы хотите повторить выполнение при возникновении ошибки.
* Flux<String> flux = Flux  
   .<String>error(new IllegalArgumentException())  
   .doOnError(System.out::println)  
   .**retryWhen**(companion -> companion.take(3));  
  Output:  
  java.lang.IllegalArgumentException  
  java.lang.IllegalArgumentException  
  java.lang.IllegalArgumentException  
  java.lang.IllegalArgumentException
* **defaultIfEmpty** — вернет значение по умолчанию, если последовательность пуста.

Пример 1:

Mono.empty()  
 .defaultIfEmpty("No such elements!")  
 .subscribe(System.out::println);  
Output:  
No such elements!

* **switchIfEmpty** — переключает на альтернативного Publisher-а, если эта последовательность завершена без каких-либо данных.

Flux.empty()  
 .switchIfEmpty(Flux.range(1,3))  
 .subscribe(System.out::println);  
  
Output:  
1  
2  
3

* **ignoreElements**

*Для Flux*: игнорирует сигналы onNext (отбрасывает их) и распространяет только события завершения.

Для Mono: создает новый Mono, который игнорирует элементы из источника (отбрасывая их), но завершается, когда источник завершает работу.

private int testNumbers(int value) {  
 if (value > 2) {  
 System.out.println("Failed");  
 throw new IllegalArgumentException("value is too high!");  
 }  
 return value;  
 }  
 private Flux<Integer> checkOnErrorMethodRetry() {  
 Flux.range(1, 3)  
 .map(this::testNumbers)  
 .doOnError(  
 error -> {  
 System.err.println("Statement failed: " + error);  
 })  
 .ignoreElements()  
 .subscribe(System.out::println);  
  
 return null;  
  
 }

* **doOnNext** — следующий элемент в последовательности (без изменения последовательности), то есть выполнить дополнительное действие. Если у нас есть, например, 5 элементов в потоке, то когда doOnNext вызывается один раз, он будет проходить через все 5 элементов.

***doOnEach*** и ***doOnNext*** действительно довольно близки. Основное различие между этими методами заключается в том, что в doOnEach вы также получаете уведомления об ошибках и событиях завершения.

**doOnComplete** — сделать что-то если операция завершилась успешно (для **Flux**).

**doOnSuccess** — добавление поведения срабатывает при успешном завершении **Mono** — результатом является T или null.

**doOnCancel** — добавление поведения (побочный эффект) срабатывает при отмене Flux/Mono (аналог для вызова Subscription.cancel()).

Disposable result = Flux.just(5, 6, 7)  
 .doOnCancel(() -> System.out.println("Cancel!"))  
 .doOnComplete(() -> System.out.println("Yes!"))  
 .subscribe(System.out::println,  
 e -> {},  
 () -> {},  
 Subscription::cancel);  
  
Output:  
Cancel!

**first** — вернет тот Mono, который выдаст значение быстрее

Mono<String> mono1 = Mono.just("Test");  
Mono<String> mono2 = Mono.just("Test2");  
  
firstMono(mono1, mono2);static Mono<String> firstMono(Mono<String> mono1, Mono<String> mono2) {  
 return Mono.first(mono1, mono2);  
 }

* **justOrEmpty** — напечатает Mono, если элемент не нулевой, в противном случае сигнал завершения.

String string = new String("Peter");  
justOrEmptyMethod(string);static Mono<String> justOrEmptyMethod(String string) {  
 return Mono.justOrEmpty(string);  
 }

**doFinally** — аналог try-catch-finally

private Flux<Integer> checkOnErrorMethodTryCatchFinally() {  
 Flux.just(1, 2, 3)  
 .map(this::testNumbers)  
 .doFinally(signalType -> {  
 if (signalType == SignalType.ON\_ERROR) {  
 System.out.println("Error signal");  
 } else if (signalType == SignalType.CANCEL) {  
 System.out.println("Cancel signal");  
 }  
 })  
 .subscribe(value -> System.out.println("Value: " + value),  
 error -> System.out.println("Error: " +  
 error.getMessage()));  
 return null;  
 }

* **using** — аналог ***try-with-resource***

Flux<String> flux =  
 Flux.using(  
 () -> disposableInstance,  
 disposable -> Flux.just(disposable.toString()),  
 Disposable::dispose  
 );  
flux.subscribe(System.out::println);

* **log** — для вывода логов в консоль

Flux<Integer> reactiveStream = Flux.range(1, 5).log();

* **filter** — для фильтрации элементов в потоке.

Мы фильтруем и оставляем пользователя с именем «name1»

Flux.just(new User("Name1", 20), new User("name2", 25), new User("name3", 30))  
 .filter(s -> s.getName().equals("name1"))  
 .subscribe(s -> System.out.println("Value: " + s.getName()));

* **dispose** — может использоваться для отмены подписки после завершения Flux или ошибок или очистки.

Disposable disposable = Flux.just(1, 2, 3, 4, 5, 6, 7, 8)  
 .delayElements(Duration.ofSeconds(3))  
 .subscribe(value -> System.out.println("Value: " + value));  
  
Thread.sleep(7000);  
disposable.dispose();  
System.out.println("Cancelling subscription");  
  
  
Output:  
Value: 1  
Value: 2  
Cancelling subscription

* **doAfterTerminate** — добавление поведения (побочный эффект) срабатывает после завершения Mono, либо путем успешного завершения нисходящего потока, либо с ошибкой.

Аналог Subscriber.onComplete() или Subscriber.onError(java.lang.Throwable)

Flux<Integer> stream = Flux.just(4, 5, 6);  
  
stream.doAfterTerminate(() -> System.out.println("Closed!")).subscribe(System.out::println);  
Output:4  
5  
6  
Closed!

**then/thenMany** — подписаться на второго Publisher-а после окончания первого Publisher-а.

***Для Mono:***

Mono<Integer> mono1 = Mono.just(1);  
Mono<Integer> mono2 = Mono.just(2);  
  
mono1.then(mono2)  
 .subscribe(value -> System.out.println("Value: " + value));  
Output:  
Value: 2

***Для Flux:***

Flux<Integer> oddFlux = Flux.just(1, 3, 5);  
Flux<Integer> evenFlux = Flux.just(2, 4, 6);  
  
oddFlux.thenMany(evenFlux)  
 .subscribe(value -> System.out.println("Value: " + value));

* **take** — берет первые N элементов из потока (если они доступны).

Flux.range(1, 10)  
 .take(3)  
 .subscribe(System.out::println);  
  
  
Output:  
1  
2  
3

**takeLast** — берет последний N элементов из потока (если они доступны).

**takeWhile** — берет значения из потока пока не будет выполнено некоторое условие.

**takeUntilOther** — публикация значений из потока до тех пор, пока конкретный Publisher не выдаст свои значения.

Flux.range(1, 5)  
 .delayElements(Duration.ofSeconds(1))   
 .**takeUntilOther**(Mono  
 .just(10) // hot publisher  
  
 // задержка '10' на 3 секунды  
 .delayElement(Duration.ofSeconds(3))  
 )  
 .subscribe(System.out::print);

**last** — последний элемент в потоке или ошибка NoSuchElementException если его нет.

**skip** — пропускает указанное количество элементов с начала потока, а затем испускает оставшиеся элементыв потоке.

**skipLast** — пропускает указанное количество элементов с конца потока и печатает те, что были ДО!

**skipUntil** — пропускает значения из этого потока, пока предикат не вернет значение true.

**error** — Mono/Flux заканчивается указанной ошибкой сразу после подписки.

Flux.error(new RuntimeException())  
 .subscribe(System.out::println);  
  
  
Output:  
Exception in thread "main" reactor.core.Exceptions$ErrorCallbackNotImplemented:

**onErrorReturn** — вы можете вернуть значение по умолчанию в случае ошибки.

.map(this::testNumbers)  
 .**onErrorReturn**(100)

**onErorResume** — в случае ошибки вы можете восстановить и продолжить работу (выполнить метод резервного копирования.

.map(this::testNumbers)  
 .**onErrorResume**(error -> Flux.just(4, 5, 6))

**doOnError** — оператор doOnError, как и все операторы с префиксом doOn, иногда называют «побочным эффектом». Они позволяют вам заглянуть внутрь последовательности событий, не меняя их. Мы знаем, что метод потерпит неудачу, но мы все еще можем обещать (зарегистрировать) эту ошибку или сделать что-либо по нашему усмотрению.

.map(this::testNumbers)  
 .**doOnError**(e -> {  
 // to do something  
 System.out.println("Log error");  
 })

**block** — блокирует текущий поток, пока мы не получим объект (для Mono)

**blockLast** — блокирует весь поток, пока он полностью не будет завершен (для Flux)

* **subscribeOn** — вы можете изменить поток выполнения (принимает Scheduler в качестве аргумента и позволяет изменить контекст выполнения на указанный Scheduler). Он работает сразу во всей цепочке подписчиков, то есть все остальные подписчики будут выполняться в контексте нового указанного Scheduler-а.
* **publishOn** — отличие от subscribeOn в том, что его можно использовать в середине цепочки вызовов, и все остальные подписчики будут выполняться в контексте указанного Scheduler-а (в оригинале, который был указан)
* Flux.range(1, 5)  
   .doOnNext(consumer)  
   .map(i -> {  
   System.out.println("Inside map the thread is " + Thread.currentThread().getName());  
   return i \* 10;  
   })  
     
   .publishOn(Schedulers.newElastic("First\_PublishOn()\_thread"))  
   .doOnNext(consumer)  
   .subscribeOn(Schedulers.newElastic("subscribeOn\_thread"))  
   .subscribe();

***connect()*** — запускает подписку вручную когда подписалось N-количество подписчиков

ConnectableFlux<Integer> source = Flux.*range*(1, 3)  
 .doOnSubscribe(s -> System.*out*.println("Subscribed"))  
 .publish();source.subscribe(System.*out*::println); // 1-ый подписчик  
source.subscribe(x -> System.*out*.println(x + "\_new")); // 2-ой подписчикSystem.*out*.println("Ready");  
Thread.*sleep*(500);  
System.*out*.println("Connection");source.connect(); // стартуемВывод:Ready  
Connection  
Subscribed  
1  
1\_new  
2  
2\_new  
3  
3\_new

**- autoConnect(n)** — запускает подписку автоматически после N-количества подписчиков

Flux<Integer> autoConnect = source2.publish().autoConnect(2); // как только подписалось 2 подписчика - стартуем !

**Обработка ошибок**

**onErrorReturn**

Вы можете вернуть значение по умолчанию в случае ошибки выполнения метода

## **В случае ошибки вы можете выбросить собственное исключение**

.**onErrorMap**(error -> new SQLException("this is SQLException"))

**HOT COLD PUBLISHER**

Метод *share()* немедленно запускает источник и заставляет поздних подписчиков видеть более поздние элементы.

private Stream<String> getAudioPodcast(){  
 System.out.println("Start streaming...");  
 return Stream.of(  
 "thread 1",  
 "thread 2",  
 "thread 3",  
 "thread 4",  
 "thread 5"  
 );  
 }  
  
Flux<String> radioStation = Flux.fromStream(() -> getAudioPodcast())  
 .delayElements(Duration.ofSeconds(2)).**share()**;  
  
// Первый человек слушает радио подкаст  
radioStation.subscribe(scene -> System.out.println("First person is listening " + scene));  
  
// Второй человек начал слушать радио подкаст спустя 5 секунд  
Thread.sleep(5000);  
radioStation.subscribe(scene -> System.out.println("Second person is listening " + scene));Output:  
Start streaming...  
First perosn is watching scene 1  
First perosn is watching scene 2  
First perosn is watching scene 3  
Second perosn is watching scene 3  
First perosn is watching scene 4  
Second perosn is watching scene 4  
First perosn is watching scene 5  
Second perosn is watching scene 5

**Context API**

Существует такое понятие как **Context API**.

**Context** - это интерфейс, напоминающий **Map**: он хранит пары ключ-значение и позволяет получить значение, которое вы сохранили по ключу.  
Ключом и значением может быть любой объект.  
Метод *put()* — чтобы положить ключ/значение.  
Метод ***hasKey()*** — позволяет проверить наличие ключа.  
Метод ***getOrDefault()*** — получить значение по умолчанию, если у контекста нет этого ключа.  
Метод ***getOrEmpty()*** — получить пустое значение, если у контекста нет этого ключа.  
Метод ***delete()*** — удаляет значение, связанное с ключом.

Вы можете создать пустой контекст с помощью ***Context.empty()***.

Прочитать данные из контекста можно с помощью ***Mono.subscriberContext().***

А теперь более подробно разберемся зачем он нужен.

Например вместо него можно использовать ThreadLocal. Но если мы захотим получить информацию в этом ThreadLocal из другого потока — у нас ничего не выйдет!

Но зато при помощи Context мы сможем это сделать: сможем получить информацию из первого потока находясь в другом потоке (например в первом потоке в map в ThreadLocal добавляем какое-то значение, затем меняем поток выполнения и пытаемся получить из map из ThreadLocal эти значения).

Экземпляр Context доступен через оператора ***Mono.subscriberContext*** и может добавляться в поток данных с использованием оператора ***subscriberContext(Context)***.

Context изначально проектировался как неизменяемый объект и после добавления в него нового элемента мы получим новый экземпляр Context. Это единственный способ передать Context с потоком данных и иметь возможность динамически добавлять в него информацию, которая должна быть доступна в течение всего этапа сборки или подписки.

Именно поэтому нежелательно просто взять и добавить в сам Context что-либо. Для этого лучше внутри Context использовать map и уже в эту map добавить значения.

Например у нас етсь поток который содержит дополнительные методы обработки (т.е. получается несколько издателей Publisher “друг в друге”). На этапе подписки подписчик “как-бы идет” снизу вверх по цепочке издателей и на каждом этапе обертывается в некое локальное представление Subscriber, которое предлагает дополнительную логику обработки. Reactor использует расширенную версию интерфейса Subscriber – интерфейс **CoreSubscriber**. Он передает Context как поле. В этом интерфейсе есть метод *currentContext()*, который обеспечивает доступ к текущему объекту Context.

Оператор, с помощью которого можно изменить текущий объект Context, — это ***subscriberContext***, возвращающий реализацию CoreSubscriber, которая хранит контекст исходящего потока и передается как параметр.

# Schedulers

С помощью **Scheduler** можно приказать потоку всегда выполняться в одном и том же треде или выделять новый тред при каждой подписке, или использовать собственный пул потоков и т.д.

**subscribeOn** — Чтобы изменить тред, в котором будет происходить выполнение реактивного потока. В случае с **subscribeOn** оператор «глобальный», срабатывает сразу на всю цепочку Subscriber. После вызова subscribe() контекстом выполнения будет указанный Scheduler. Далее контекст может изменяться с помощью оператора publishOn. Последующие вызовы subscribeOn игнорируются.

**publishOn** — если мы хотим например часть операторов выполнить в одном треде а часть в другом. В случае с **publishOn** этот оператор применяется так же, как и любой другой, посреди цепочки вызовов. Все последующие Subscriber будут выполняться в контексте указанного Scheduler.

## Виды планировщиков:

**-Schedulers.single()** — выполнение в выделенном потоке. Он поддерживает планирование с учетом времени, поэтому может использоваться для планирования периодичес- ких событий с задержкой.

-**Schedulers.immediate()** — выполнение будет происходить в текущем потоке.  
-**Schedulers.parallel()** —работает с пулом рабочих потоков фиксированного размера (по умолчанию размер пула ограничивается числом ядер процессора). Хорошо подходит для вычислительных задач.

-**Schedulers.elastic()** —динамически создает рабочие потоки и кеширует пулы потоков выполнения. Максимальное число создаваемых пулов потоков выполнения не ограничивается, поэтому этот планировщик можно использовать для организации выполнения задач, связанных с вводом/выводом. Здесь при использовании — **Schedulers.newElastic()** — можно указать количество **workers** **(ExecutorService).**

Некоторые операторы из Flux и Mono запускаются сразу на конкретном Scheduler (но можно передать и свой).

Например **Flux.interval()** по умолчанию запускается на Schedulers.parallel(), но мы можем передать свой.

Flux.interval(Duration.ofMillis(300), Schedulers.newSingle("test"))