Оглавление

[1. Функциональные интерфейсы, основные типы. 2](#_Toc60268245)

[2. Что такое лямбда выражение и ссылка на метод. 2](#_Toc60268246)

[3. Как они связаны с анонимным классом. 3](#_Toc60268247)

[4. Что такое Стрим. Когда нужно использовать? 3](#_Toc60268248)

[5. Какие бывают стримы 4](#_Toc60268249)

[6. Терминальные и промежуточные методы. 4](#_Toc60268250)

[7. Способы получения стрима. 7](#_Toc60268251)

[8. Что такое анонимный класс. 8](#_Toc60268252)

[9. Что такое функциональный интерфейс и для чего он нужен и зачем были добавлены? 8](#_Toc60268253)

[10. Какой аннотацией помечается функциональный интерфейс? 8](#_Toc60268254)

[11. Сколько дефолтных методов и статических методов, статических полей в интерфейсе? 9](#_Toc60268255)

[12. Где находятся функциональные интерфейсы? 9](#_Toc60268256)

[13. Перечислить основные семейства функциональных интерфейсов? Что они принимают и что возвращают? 9](#_Toc60268257)

[14. Какие есть способы инстанцировать функциональные интерфейсы? 10](#_Toc60268258)

[15. К каким переменным и как можно обращаться в теле лямда-выражениях? 10](#_Toc60268259)

[16. Чем является Stream в контексте Java? 11](#_Toc60268260)

[17. В каком пакете находится Stream? 11](#_Toc60268261)

[18. Чем Stream отличается от итератора? 11](#_Toc60268262)

[19. Сравнение стримов с коллекцией 12](#_Toc60268263)

[20. Из каких частей состоит использование стримов? 12](#_Toc60268264)

[21. В каком случае нужно закрывать стрим? 12](#_Toc60268265)

[22. Первый этап работы со стримом 12](#_Toc60268266)

[23. Откуда можно получить стрим? 12](#_Toc60268267)

[24. Разница методов .list() и walk() Что такое Supplier -поставщик? какой содержит метод? 13](#_Toc60268268)

[25. Как динамически получать стрим? 13](#_Toc60268269)

[26. Как получить стрим диапазона чисел? 13](#_Toc60268270)

[27. В чем разница методов range и rangeClosed? 13](#_Toc60268271)

[28. Можно ли конкатенировать стримы? если да то каким методом? 14](#_Toc60268272)

[29. Можно ли получить пустой стрим? 14](#_Toc60268273)

[30. Как получить стрим из массива? 14](#_Toc60268274)

[31. Какой второй этап работы со стримом? 14](#_Toc60268275)

[32. Перечислить основные промежуточные операции. Их предназначение. 14](#_Toc60268276)

[33. Для чего терминальные операции? Какие бывают и что делают? 16](#_Toc60268277)

[34. Что такое коллекторы? 18](#_Toc60268278)

[35. Сколько раз можно вызывать терминальную операцию? 18](#_Toc60268279)

[36. Что такое метод референс? 18](#_Toc60268280)

[37. Расскажите про Comparator и Comparable? 18](#_Toc60268281)

[38. В чем разница между foreach и foreachordered 19](#_Toc60268283)

[39. Зачем нужно функциональное программирование, где оно применяется и в чем его отличие от программирования в стиле ооп или в процедурном стиле? 20](#_Toc60268284)

[40. Что такое default методы в интерфейсе и для чего они были введены? 20](#_Toc60268285)

[41. Как создать экземпляр анонимного класса (подробно)? Где они применяются? 20](#_Toc60268286)

[42. Что такое ленивая инициализация стрима? 21](#_Toc60268287)

[43. Зачем нужен класс Optional? 21](#_Toc60268288)

[44. Все способы реализации функциональных интерфейсов. 21](#_Toc60268289)

[45. как взаимосвязаны лямбда и функциональный интерфейс? 21](#_Toc60268290)

[46. Любой анонимный класс можно заменить на лямбду? 21](#_Toc60268291)

[47. В чем разница map и flatMap? 22](#_Toc60268292)

[48. Зачем нужны default методы в функциональном интерфейсе - тут про обратную совместимость. 22](#_Toc60268293)

[49. ссылка на метод - ну и в каком виде передается: 22](#_Toc60268294)

[50. как получить стрим - через разные метод + бесконечные стримы ( через iterate и generate) 22](#_Toc60268295)

[51. что возвращают промежуточные операции: 22](#_Toc60268296)

[52. Методы peek и forEach - в чем разница: 22](#_Toc60268297)

[53. Любую ли лямбду можно свернуть в метод референс? 23](#_Toc60268298)

# Функциональные интерфейсы, основные типы.

Функциональный интерфейс в Java – это интерфейс, который содержит только 1 абстрактный метод. Основное назначение – использование в лямбда выражениях и method reference.

Наличие 1 абстрактного метода - это единственное условие, таким образом функциональный интерфейс может содержать так же default и static методы.

К функциональному интерфейсу можно добавить аннотацию @FunctionalInterface. Это не обязательно, но при наличии данной аннотации код не скомпилируется, если будет больше или меньше, чем 1 абстрактный метод.

Рекомендуется добавлять @FunctionalInterface. Это позволит использовать интерфейс в лямбда выражениях, не остерегаясь того, что кто-то добавит в интерфейс новый абстрактный метод и он перестанет быть функциональным.

В Java есть встроенные функциональные интерфейсы, размещенные в пакете java.util.function.

# Что такое лямбда выражение и ссылка на метод.

лямбда-выражение является реализацией абстрактного метода функционального интерфейса.

Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

# Как они связаны с анонимным классом.

Заменяют анонимные классы, синтаксический сахар, не нужно переопределять методы.

Лямбда-выражение имеет более легковесный синтаксис. Не нужно явно указывать тип функционального интерфейса, который лямбда реализует – он автоматически выведется из контекста. Лямбда-выражения добавлены в язык в первую очередь как синтаксический сахар.  
  
С другой стороны, у класса есть поля. Экземпляр анонимного класса сохраняет свое состояние между вызовами, и меняет его при необходимости. Для лямбды доступен лишь захват и effectively final использование внешних переменных.  
  
Лямбдой реализуется только функциональный интерфейс. Функциональный интерфейс – это тип с единственным абстрактным методом. Анонимным классом же можно расширить любой расширяемый класс или реализовать интерфейс с любым количеством абстрактных методов.  
  
Анонимный класс создает новый скоуп, лямбда работает в текущем. Это значит, что объявление переменной с именем, которое уже используется снаружи, в лямбде вызовет ошибку компиляции «variable is already defined», в анонимном классе скроет (shadowing) внешнюю переменную.  
  
С точки зрения реализации JVM, для лямбды не создается дополнительного .class файла, как это происходит для анонимного класса. Соответственно, не происходит и обычной загрузки и верификации класса. Вместо этого используется механизм invokedynamic и класс генерируется на лету с помощью LambdaMetafactory. Так что лямбда-выражения обычно работают быстрее.

# Что такое Стрим. Когда нужно использовать?

Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.

Ключевым понятием в Stream API является поток данных. Вообще сам термин "поток" довольно перегружен в программировании в целом и в Java в частности. В одной из предыдущих глав рассматривалась работа с символьными и байтовыми потоками при чтении-записи файлов. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы и коллекции.

Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.

# Какие бывают стримы

**(По разным критериям, например "конечные и бесконечные") знать 3 группы: 1) конечные и бесконечные, 2) последовательные и параллельные 3) объектные и примитивные + спрашивает про параллельные, промежуточные и терминальные (конечные и бесконечные это)**

1. **конечные и бесконечные**
2. **последовательные и параллельные**

Кроме последовательных потоков Stream API поддерживает параллельные потоки. Распараллеливание потоков позволяет задействовать несколько ядер процессора (если целевая машина многоядерная) и тем самым может повысить производительность и ускорить вычисления. В то же время говорить, что применение параллельных потоков на многоядерных машинах однозначно повысит производительность - не совсем корректно. В каждом конкретном случае надо проверять и тестировать.

Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод parallel. Кроме того, можно также использовать метод parallelStream() интерфейса Collection для создания параллельного потока из коллекции.

В то же время если рабочая машина не является многоядерной, то поток будет выполняться как последовательный.

1. **объектные и примитивные + спрашивает про параллельные, промежуточные и терминальные (конечные и бесконечные это)**

# Терминальные и промежуточные методы.

Промежуточные операции следует воспринимать как «отложенные», т.е. они не меняют сами данные, а только задают правила их изменения. А терминальные как раз инициируют всю цепочку преобразований, закрывают поток и возвращают модифицированные данные. Закрытый поток повторно использовать нельзя.

Примеры терминальных операций

**Преобразование в коллекцию**

Самая распространённая терминальная операция collect(). Результатом может быть, например, список.

List<String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        *// здесь могут быть ещё какие-то преобразования*  
        .collect(Collectors.toList());

А можно преобразовать стрим из строк в мапу, причём ключом сделать первую букву соответствующего слова:

Map<String, String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .collect(Collectors.toMap(e -> e.substring(0, 1), e -> e));  
        *// {a=apple, b=banana, l=lemon, o=orange}*

**Итерация по элементам**

Простой обход элементов стрима можно выполнить с помощью метода forEach(). Его смысл полностью аналогичен языковой конструкции for.

Stream.of(5, 3, 2, 10, 8).forEach(System.out::println);

Здесь мы просто выводим в консоль значение каждого элемента стрима.

**Подсчёт количества элементов**

Аналог метода size() в коллекциях и поля length в массивах. В стримах для этого используется метод count(). Обратите внимание, что метод возвращает не int, а long.

**long** count = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .count(); *// 4*

**Минимальное и максимальное значения**

Как нетрудно догадаться, min() возвращает минимальное значение, а max() - максимальное. В качестве параметра они принимают класс Comparator.

Также обратите внимание, как мы обошли «одноразовость» потока. Чтобы не писать два раза конструкцию Stream.of(...), мы использовали утилитарный класс Supplier<>. Вызов метода get() у него в действительности каждый раз создаёт новый поток с одними и теми же значениями.

Supplier<Stream<Integer>> streamSupplier = () -> Stream.of(5, 3, 2, 10, 8);  
Optional<Integer> min = streamSupplier.get().min(Comparator.naturalOrder()); *// Optional[2]*  
Optional<Integer> max = streamSupplier.get().max(Comparator.naturalOrder()); *// Optional[10]*

**Поиск первого подходящего элемента**

Метод findFirst() возвращает первый элемент стрима. Этот метод очень удобно использовать в паре с filter().

Optional<Integer> first = Stream.of(1, 2, 3, 4, 5)  
        .filter(e -> e % 2 == 0)  
        .findFirst(); *// Optional[2]*

В данном случае мы вернём первое чётное число, т.е. 2.

**Соответствие всех элементов**

Чтобы проверить все элементы стрима на соответствие некоторому условию, используйте метод allMatch(). Его результатом может быть только true или false.

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .allMatch(e -> e > 0); *// true*

В этом примере мы проверяем, что все элементы положительные.

**Полное несоответствие**

Если нужно проверить, что ни один элемент не удовлетворяет условию, воспользуйтесь методом noneMatch(). Перепишем предыдущий пример так:

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .noneMatch(e -> e < 0); *// true*

**Соответствие хотя бы одного элемента**

Чтобы проверить наличие хотя бы одного элемента, удовлетворяющего условию, используйте метод anyMatch().

**boolean** hasNegative = Stream.of(1, 2, 3, 4, 5)  
        .anyMatch(e -> e < 0); *// false*

***Промежуточные операции***

Рассмотрим промежуточные операции. Все промежуточные операции возвращают типизированный интерфейс Stream<>.

**Преобразование**

Любое изменение исходного элемента можно делать с помощью метода map(). В качестве параметра метод принимает лямбда-выражение.

Stream.of(1, 2, 3, 4, 5, 6)  
        .map(n -> n \* 10) *// умножает каждый элемент на 10*

Stream.of(**"apple"**, **"orange"**)  
        .map(String::toUpperCase) *// преобразует буквы в каждом слове в верхний регистр*

Во втором случае мы воспользовались краткой записью лямбда-выражения через method reference.

**Фильтрация**

Отсеивание части объектов можно сделать с помощью метода filter().

Stream.of(1, 2, 3, 4, 5, 6)  
        .filter(n -> n > 3) *// оставляет только те числа, которые больше 3*

**Доступ к элементу**

Если требуется получить доступ к элементу, никак его при этом не меняя, нам поможет метод peek(). Например, в целях логирования мы хотим вывести значение элемента в консоль.

Stream.of(1, 2, 3, 4, 5, 6)  
        .peek(System.out::println)

**Удаление дублей**

Если в стриме несколько элементов имеют одинаковые значения, а вы хотите получить только уникальные, используйте метод distinct().

Stream.of(2, 2, 3, 2, 3, 3, 2, 2)  
        .distinct() *// в результате останется только два элемента: [2, 3]*

**Сортировка**

Для сортировки используйте метод sorted(). По умолчанию он использует «натуральный порядок», т.е. сортировку по возрастанию.

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted() *// получим [1, 2, 3, 4, 5, 6]*

Перегруженная версия этого метода принимает интерфейс Comparator, поэтому вы можете задать свои собственные правила сортировки или воспользоваться готовыми. Приведённый выше пример равносилен следующему:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.naturalOrder())) *// получим [1, 2, 3, 4, 5, 6]*

Нетрудно догадаться, что обратную сортировку можно задать таким образом:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.reverseOrder()) *// получим [6, 5, 4, 3, 2, 1]*

**Ограничение по количеству элементов**

Для получения первых N элементов используйте метод limit(). В данном случае мы берём первые 3 элемента:

Stream.of(1, 2, 3, 4, 5, 6)  
        .limit(3) *// [1, 2, 3]*

**Пропуск первых элементов**

Чтобы пропустить нужное количество первых элементов, используйте метод skip(). Здесь мы пропускаем первые 4 элемента и получаем оставшиеся два:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(4) *// [5, 6]*

Обратите внимание, что комбинация методов limit() и skip() позволяет организовать постраничный вывод информации. Например, в пользовательском интерфейсе мы отображаем по две записи на странице. Тогда, если у нас запросят вторую страницу, то выражение будет выглядеть так:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(2)  
        .limit(2) *// [3, 4]*

**Разворачивание многомерных структур**

Предположим, у вас есть многомерная структура (двумерный массив элементов типа Integer) и вы хотите его развернуть (сделать «плоским»), просто помещая каждый подмассив в конец результирующего. В этом нам поможет метод flatMap():

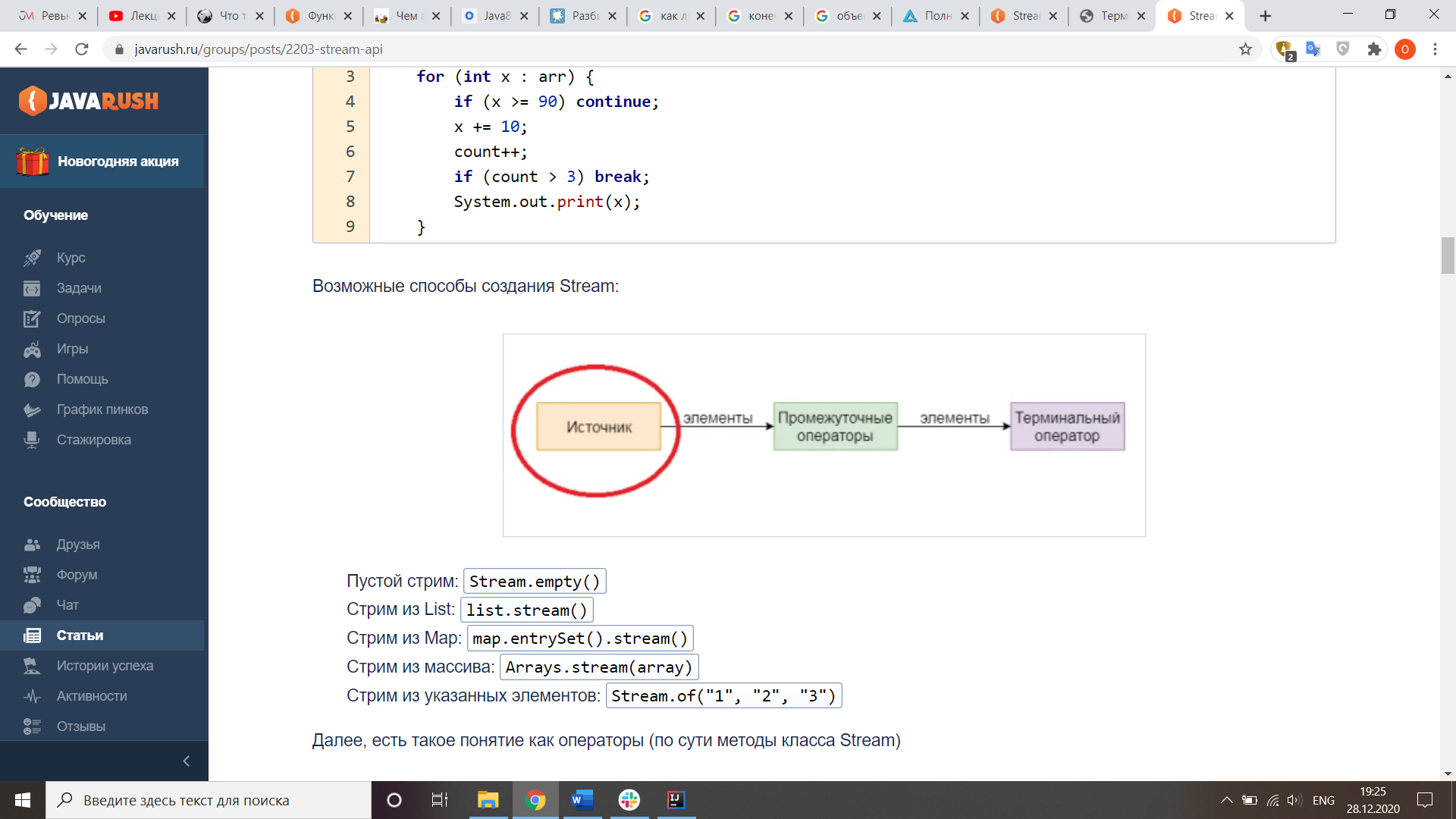
Integer[][] array2d = **new** Integer[][]{{1, 2}, {3, 4}};  
Stream<Integer> stream =  
        Arrays.stream(array2d).flatMap(Arrays::stream); *// [1, 2, 3, 4]*

**Выводы**

В этой статье мы рассмотрели на примерах промежуточные методы Stream API, а во второй части рассмотрим уже терминальные (конечные) методы.

# Способы получения стрима.

Возможные способы создания Stream:



* Пустой стрим: Stream.empty()
* Стрим из List: list.stream()
* Стрим из Map: map.entrySet().stream()
* Стрим из массива: Arrays.stream(array)
* Стрим из указанных элементов: Stream.of("1", "2", "3")

# Что такое анонимный класс.

Это вложенный локальный(inner class) класс, без имени, который разрешено декларировать в любом месте обрамляющего класса, разрешающем размещение выражений.

- Нет конструктора, только тело

- Создание экземпляра анонимного класса происходит одновременно с его объявлением. В зависимости от местоположения анонимный класс ведет себя как статический, либо как нестатический вложенный класс - в нестатическом контексте появляется окружающий его экземпляр.

Анонимные классы имеют несколько ограничений:

Их использование разрешено только в одном месте программы - месте его создания;

Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;

Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

создания объекта функции (function object), например, реализация интерфейса Comparator;

создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных;

в статическом методе генерации;

инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

Анонимные классы выполняют две функции,

* Описание класса
* Создание экземпляра(объекта) этого класса

# Что такое функциональный интерфейс и для чего он нужен и зачем были добавлены?

Основное назначение – использование в лямбда выражениях и method reference.

# Какой аннотацией помечается функциональный интерфейс?

К функциональному интерфейсу можно добавить аннотацию @FunctionalInterface. Это не обязательно, но при наличии данной аннотации код не скомпилируется, если будет больше или меньше, чем 1 абстрактный метод.

Рекомендуется добавлять @FunctionalInterface. Это позволит использовать интерфейс в лямбда выражениях, не остерегаясь того, что кто-то добавит в интерфейс новый абстрактный метод и он перестанет быть функциональным.

# Сколько дефолтных методов и статических методов, статических полей в интерфейсе?

Интерфейс может включать сколько угодно default методов и статик методов.

# Где находятся функциональные интерфейсы?

java.util.function

# Перечислить основные семейства функциональных интерфейсов? Что они принимают и что возвращают?

Рассмотрим основные из этих интерфейсов:

* **Predicate<T>**
* **Consumer<T>**
* **Function<T,R>**
* **Supplier<T>**
* **UnaryOperator<T>**
* **BinaryOperator<T>**

**Predicate<T>**

Функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T:

public interface Predicate<T> {

    boolean test(T t);

}

**BinaryOperator<T>**

BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T:

public interface BinaryOperator<T> {

    T apply(T t1, T t2);

}

**UnaryOperator<T>**

UnaryOperator<T> принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T:

public interface UnaryOperator<T> {

    T apply(T t);

}

**Function<T,R>**

Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R:

public interface Function<T, R> {

    R apply(T t);

}

**Consumer<T>**

Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

|  |  |
| --- | --- |
|  | public interface Consumer<T> {      void accept(T t);  } |

**Supplier<T>**

Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T:

|  |  |
| --- | --- |
|  | public interface Supplier<T> {      T get();  } |

# Какие есть способы инстанцировать функциональные интерфейсы?

есть следующие способы его инстанцировать:

* создать экземпляр анонимного или именованного класса, реализующего интерфейс Function;
* воспользоваться ссылкой на метод;
* написать лямбда-выражение

# К каким переменным и как можно обращаться в теле лямда-выражениях?

- можно обращаться к параметрам л-выражения, а также свободно объявлять и использовать внутри тела л-выражения любые переменные **IntUnaryOperator sq = x -> x \* x ;**

- можно обращаться к полям того класса внутри которого объявлена лямбда, причем их можно как читать, так и писать;

**public class A {**

**private int counter;**

**……**

**IntSupplier sequence = () -> counter++;**

**-** переменные внутри метода (**эффективно-финальные**), где создана лямбда, но значения должны быть инициализированы до создания лямбды и присвоены единожды: т.е. из лямбды нельзя присвоить новые значения переменным содержащего ее метода, т.к. будет ругаться IDEa.

Либо явно финализирована, либо не переписана.

**final int bonus = 10;**

**IntUnaryOperator bonusAdder = (x) -> x + bonus;**

Для обхода этого ограничения используют массив единичной длины

**int[] counter = new int[1];**

**IntSupplier sequence1 = () -> counter[0]++;**

static переменные класса, переменные интерфейса (с которым лямбда работает)???

# Чем является Stream в контексте Java?

Stream представляет собой последовательность элементов, на которых могут выполняться последовательные и параллельные операции агрегации.

# В каком пакете находится Stream?

java.util.stream.

# Чем Stream отличается от итератора?

Действие, указанное в параметре Stream.forEach , должно быть невмешательским

Iterable.forEach потенциально имеет меньше ограничений.

Для простых случаев, таких как иллюстрированный, они в основном одинаковы. Тем не менее, есть ряд тонких различий, которые могут быть значительными.

Одна проблема с заказом. При Stream.forEach этом порядок не определен . Это вряд ли произойдет с последовательными потоками, тем не менее, оно находится в пределах спецификации для Stream.forEach выполнения в произвольном порядке. Это часто происходит в параллельных потоках. Напротив, Iterable.forEach всегда выполняется в порядке итерации Iterable, если он указан.

Другая проблема связана с побочными эффектами. Действие , указанное в Stream.forEach обязан быть без вмешательства . (См. Документ пакета java.util.stream .) Iterable.forEach Потенциально имеет меньше ограничений. Для коллекций java.util, Iterable.forEach как правило , будут использоваться эти коллекции Iterator, большинство из которых предназначены для быстрого отказа и которые будут выбрасываться, ConcurrentModificationException если коллекция будет структурно изменена во время итерации. Однако модификации, которые не являются структурными , допускаются во время итерации. Например, документация класса ArrayList гласит: «Простая установка значения элемента не является структурной модификацией». Таким образом, действие дляArrayList.forEach разрешено устанавливать значения в базовом ArrayList без проблем.

Параллельные коллекции еще раз отличаются. Вместо быстрого отказа они разработаны как слабо последовательные . Полное определение по этой ссылке. Кратко, однако, рассмотрим ConcurrentLinkedDeque. Действие, переданное его forEachметоду, может изменять базовую деку, даже структурно, и ConcurrentModificationExceptionникогда не генерируется. Тем не менее, изменения, которые происходят, могут или не могут быть видны в этой итерации. (Отсюда и «слабая» последовательность.)

Еще одно отличие видно, если Iterable.forEachперебирает синхронизированную коллекцию. В такой коллекции один раз Iterable.forEach принимает блокировку коллекции и удерживает ее во всех вызовах метода действия. Stream.forEachВызов использует spliterator своей коллекции, которая не замок, и которая опирается на преобладающих правило невмешательства. Коллекция, поддерживающая поток, может быть изменена во время итерации, и, если это так, это ConcurrentModificationExceptionможет привести к несовместимому поведению.

# Сравнение стримов с коллекцией

Разница между коллекцией(Collection) данных и потоком(Stream) из новой JDK8 в том что коллекции позволяют работать с элементами по-отдельности, тогда как поток(Stream) не позволяет. Например, с использованием коллекций, вы можете добавлять элементы, удалять, и вставлять в середину. Поток(Stream) не позволяет манипулировать отдельными элементами из набора данных, но вместо этого позволяет выполнять функции над данными как одним целом.

# Из каких частей состоит использование стримов?

Получение стрима.  
Промежуточные операции возвращают трансформированный поток. К возвращенному потоку также можно применить ряд промежуточных операций.

Конечные или терминальные операции возвращают конкретный результат. После этого никаких промежуточных операций естественно применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция.

# В каком случае нужно закрывать стрим?

Когда открывается, файловая операция, сетевые операции,

# Первый этап работы со стримом

Наверно получение стрима

# Откуда можно получить стрим?

Классический: Создание стрима из коллекции

**Collection collection = Arrays.asList("a1", "a2", "a3"); Stream streamFromCollection = collection.stream();**

Создание стрима из значений

**Stream streamFromValues = Stream.of("a1", "a2", "a3");**

Создание стрима из массива

**String[] array = {"a1","a2","a3"}; Stream streamFromArrays = Arrays.stream(array);**

Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме)

**Stream streamFromFiles = Files.lines(Paths.get("file.txt"))**

Создание стрима из строки

**IntStream streamFromString = "123".chars()**

С помощью Stream.builder

**Stream.builder().add("a1").add("a2").add("a3").build()**

Создание параллельного стрима

**Stream stream = collection.parallelStream();**

Создание бесконечных стрима с помощью Stream.iterate

**Stream streamFromIterate = Stream.iterate(1, n -> n + 1)**

Создание бесконечных стрима с помощью

**Stream streamFromGenerate = Stream.generate(() -> "a1")**

# Разница методов .list() и walk() Что такое Supplier -поставщик? какой содержит метод?

list (директории) дает поток файлов в каталоге директории, тогда как метод walk проходит по поддереву своего аргумента, включая корень поддерева – сам каталог.

Возвращает значение, одно и тоже или разные:

* Интерфейс Supplier используется тогда, когда на вход не передаются значения, но необходимо вернуть результат.
* Имеет метод T get();
* Функциональный дескриптор интерфейса:
* () -> T

# Как динамически получать стрим?

С помощью Stream.generate он вызывает Supplier поставщик.

# Как получить стрим диапазона чисел?

С помощью методов **range** и **rangeClosed**.

# В чем разница методов range и rangeClosed?

**IntStream.range​(int startInclusive, int endExclusive)  
LongStream.range​(long startInclusive, long endExclusive)**

Создаёт стрим из числового промежутка [start..end), то есть от start (включительно) по end.

**IntStream.range(0, 10).forEach(System.out::println); // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

**LongStream.range(-10L, -5L).forEach(System.out::println); // -10, -9, -8, -7, -6**

**IntStream.rangeClosed​(int startInclusive, int endInclusive)  
LongStream.range​Closed(long startInclusive, long endInclusive)**

Создаёт стрим из числового промежутка [start..end], то есть от start (вкл) по end (вкл).

**IntStream.rangeClosed(0, 5).forEach(System.out::println); // 0, 1, 2, 3, 4, 5**

**LongStream.range(-8L, -5L).forEach(System.out::println); // -8, -7, -6, -5**

# Можно ли конкатенировать стримы? если да то каким методом?

Объединяет два стрима так, что вначале идут элементы стрима A, а по его окончанию последуют элементы стрима B.

**Stream.concat(**

**Stream.of(1, 2, 3),**

**Stream.of(4, 5, 6)).forEach(System.out::println); // 1, 2, 3, 4, 5, 6**

Да можно, **Stream.empty().forEach(System.out::println); // Вывода нет**

# Можно ли получить пустой стрим?

ofNullable(T t)

Появился в Java 9. Возвращает пустой стрим, если в качестве аргумента передан null, в противном случае, возвращает стрим из одного элемента.

**String str = Math.random() > 0.5 ? "I'm feeling lucky" : null;**

**Stream.ofNullable(str)**

**.forEach(System.out::println);**

# Как получить стрим из массива?

String[] array = {"a1","a2","a3"}; Stream streamFromArrays = Arrays.stream(array);

# Какой второй этап работы со стримом?

Наверное, запуск промежуточного оператора

# Перечислить основные промежуточные операции. Их предназначение.

Все методы Stream API можно разделить на две группы: промежуточные и терминальные (конечные). Промежуточные операции следует воспринимать как «отложенные», т.е. они не меняют сами данные, а только задают правила их изменения. А терминальные как раз инициируют всю цепочку преобразований и возвращают модифицированные данные.

Рассмотрим промежуточные операции. Все промежуточные операции возвращают типизированный интерфейс Stream<>.

**Преобразование**

Любое изменение исходного элемента можно делать с помощью метода map(). В качестве параметра метод принимает лямбда-выражение.

Stream.of(1, 2, 3, 4, 5, 6)  
        .map(n -> n \* 10) *// умножает каждый элемент на 10*

Stream.of(**"apple"**, **"orange"**)  
        .map(String::toUpperCase) *// преобразует буквы в каждом слове в верхний регистр*

Во втором случае мы воспользовались краткой записью лямбда-выражения через method reference.

**Фильтрация**

Отсеивание части объектов можно сделать с помощью метода filter().

Stream.of(1, 2, 3, 4, 5, 6)  
        .filter(n -> n > 3) *// оставляет только те числа, которые больше 3*

**Доступ к элементу**

Если требуется получить доступ к элементу, никак его при этом не меняя, нам поможет метод peek(). Например, в целях логирования мы хотим вывести значение элемента в консоль.

Stream.of(1, 2, 3, 4, 5, 6)  
        .peek(System.out::println)

**Удаление дублей**

Если в стриме несколько элементов имеют одинаковые значения, а вы хотите получить только уникальные, используйте метод distinct().

Stream.of(2, 2, 3, 2, 3, 3, 2, 2)  
        .distinct() *// в результате останется только два элемента: [2, 3]*

**Сортировка**

Для сортировки используйте метод sorted(). По умолчанию он использует «натуральный порядок», т.е. сортировку по возрастанию.

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted() *// получим [1, 2, 3, 4, 5, 6]*

Перегруженная версия этого метода принимает интерфейс Comparator, поэтому вы можете задать свои собственные правила сортировки или воспользоваться готовыми. Приведённый выше пример равносилен следующему:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.naturalOrder())) *// получим [1, 2, 3, 4, 5, 6]*

Нетрудно догадаться, что обратную сортировку можно задать таким образом:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.reverseOrder()) *// получим [6, 5, 4, 3, 2, 1]*

**Ограничение по количеству элементов**

Для получения первых N элементов используйте метод limit(). В данном случае мы берём первые 3 элемента:

Stream.of(1, 2, 3, 4, 5, 6)  
        .limit(3) *// [1, 2, 3]*

**Пропуск первых элементов**

Чтобы пропустить нужное количество первых элементов, используйте метод skip(). Здесь мы пропускаем первые 4 элемента и получаем оставшиеся два:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(4) *// [5, 6]*

Обратите внимание, что комбинация методов limit() и skip() позволяет организовать постраничный вывод информации. Например, в пользовательском интерфейсе мы отображаем по две записи на странице. Тогда, если у нас запросят вторую страницу, то выражение будет выглядеть так:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(2)  
        .limit(2) *// [3, 4]*

**Разворачивание многомерных структур**

Предположим, у вас есть многомерная структура (двумерный массив элементов типа Integer) и вы хотите его развернуть (сделать «плоским»), просто помещая каждый подмассив в конец результирующего. В этом нам поможет метод flatMap():

Integer[][] array2d = **new** Integer[][]{{1, 2}, {3, 4}};  
Stream<Integer> stream =  
        Arrays.stream(array2d).flatMap(Arrays::stream); *// [1, 2, 3, 4]*

**Выводы**

В этой статье мы рассмотрели на примерах промежуточные методы Stream API, а во второй части рассмотрим уже терминальные (конечные) методы.

# Для чего терминальные операции? Какие бывают и что делают?

Промежуточные операции следует воспринимать как «отложенные», т.е. они не меняют сами данные, а только задают правила их изменения. А терминальные как раз инициируют всю цепочку преобразований, закрывают поток и возвращают модифицированные данные.

**Преобразование в коллекцию**

Самая распространённая терминальная операция collect(). Результатом может быть, например, список.

List<String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        *// здесь могут быть ещё какие-то преобразования*  
        .collect(Collectors.toList());

А можно преобразовать стрим из строк в мапу, причём ключом сделать первую букву соответствующего слова:

Map<String, String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .collect(Collectors.toMap(e -> e.substring(0, 1), e -> e));  
        *// {a=apple, b=banana, l=lemon, o=orange}*

**Итерация по элементам**

Простой обход элементов стрима можно выполнить с помощью метода forEach(). Его смысл полностью аналогичен языковой конструкции for.

Stream.of(5, 3, 2, 10, 8).forEach(System.out::println);

Здесь мы просто выводим в консоль значение каждого элемента стрима.

**Подсчёт количества элементов**

Аналог метода size() в коллекциях и поля length в массивах. В стримах для этого используется метод count(). Обратите внимание, что метод возвращает не int, а long.

**long** count = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .count(); *// 4*

**Минимальное и максимальное значения**

Как нетрудно догадаться, min() возвращает минимальное значение, а max() - максимальное. В качестве параметра они принимают класс Comparator.

Также обратите внимание, как мы обошли «одноразовость» потока. Чтобы не писать два раза конструкцию Stream.of(...), мы использовали утилитарный класс Supplier<>. Вызов метода get() у него в действительности каждый раз создаёт новый поток с одними и теми же значениями.

Supplier<Stream<Integer>> streamSupplier = () -> Stream.of(5, 3, 2, 10, 8);  
Optional<Integer> min = streamSupplier.get().min(Comparator.naturalOrder()); *// Optional[2]*  
Optional<Integer> max = streamSupplier.get().max(Comparator.naturalOrder()); *// Optional[10]*

**Поиск первого подходящего элемента**

Метод findFirst() возвращает первый элемент стрима. Этот метод очень удобно использовать в паре с filter().

Optional<Integer> first = Stream.of(1, 2, 3, 4, 5)  
        .filter(e -> e % 2 == 0)  
        .findFirst(); *// Optional[2]*

В данном случае мы вернём первое чётное число, т.е. 2.

**Соответствие всех элементов**

Чтобы проверить все элементы стрима на соответствие некоторому условию, используйте метод allMatch(). Его результатом может быть только true или false.

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .allMatch(e -> e > 0); *// true*

В этом примере мы проверяем, что все элементы положительные.

**Полное несоответствие**

Если нужно проверить, что ни один элемент не удовлетворяет условию, воспользуйтесь методом noneMatch(). Перепишем предыдущий пример так:

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .noneMatch(e -> e < 0); *// true*

**Соответствие хотя бы одного элемента**

Чтобы проверить наличие хотя бы одного элемента, удовлетворяющего условию, используйте метод anyMatch().

**boolean** hasNegative = Stream.of(1, 2, 3, 4, 5)  
        .anyMatch(e -> e < 0); *// false*

# Что такое коллекторы?

Большинство операций класса Stream, которые модифицируют набор данных, возвращают этот набор в виде потока. Однако бывают ситуации, когда хотелось бы получить данные не в виде потока, а в виде обычной коллекции, например, ArrayList или HashSet. И для этого у класса Stream определен метод **collect**.

Эта функция представляет объект Collector, который определен в пакете java.util.stream. Мы можем написать свою реализацию функции, однако Java уже предоставляет ряд встроенных функций, определенных в классе Collectors:

toList(): преобразование к типу List

toSet(): преобразование к типу Set

toMap(): преобразование к типу Map

# Сколько раз можно вызывать терминальную операцию?

один

# Что такое метод референс?

Это компактные лямбда выражения для методов у которых уже есть имя.

Ссылка на статический метод

Ссылка на нестатический метод конкретного объекта

Ссылка на нестатический метод любого объекта конкретного типа

Ссылка на конструктор

Через обычную имплементацию в классе

Создать экземпляр анонимного или именованного класса, реализующего функциональный интерфейс.

# Расскажите про Comparator и Comparable?

В интерфейсе **Comparator(**функциональный**)** объявлен метод **compare** (Object obj1, Object obj2), который позволяет сравнивать между собой два объекта. На выходе метод возвращает значение 0, если объекты равны, положительное значение или отрицательное значение, если объекты не тождественны.

В интерфейсе **Comparable** объявлен только один метод **compareTo** (Object obj), предназначенный для упорядочивания объектов класса. Данный метод удобно использовать для сортировки списков или массивов объектов.

Comparable и Comparator - это общие интерфейсы в Java, используемые для сравнения элементов данных объектов. Интерфейс Comparable присутствует в пакете java.lang, а интерфейс Comparator присутствует в пакете java.util. Основное различие между интерфейсами Comparable и Comparator состоит в том, что интерфейс Comparable обеспечивает единую последовательность сортировки, тогда как интерфейс Comparator обеспечивает несколько последовательностей сортировки. Есть несколько других отличий между интерфейсом Comparable и Comparator, которые мы изучим в Сравнительной таблице.

### Сравнительная таблица

|  |  |  |
| --- | --- | --- |
| **Основа для сравнения** | **сравнимый** | **компаратор** |
| основной | Сопоставимый интерфейс допускает только одну последовательность сортировки. | Интерфейс Comparator позволяет несколько последовательностей сортировки. |
| пакеты | Сопоставимый интерфейс присутствует в пакете java.lang. | Интерфейс Comparator присутствует в пакете java.util. |
| методы | Сопоставимый интерфейс содержит только один метод public int compareTo (Object obj); | Интерфейс Comparator содержит два метода public int Compare (Объект obj1, Объект obj2) логическое равенство (Object obj) |
| Реализация | Сопоставимый интерфейс реализуется классом, чьи объекты должны сравниваться. | Интерфейс компаратора реализуется отдельным классом вместо класса, чьи объекты должны сравниваться. |
| сравнение | Метод compareTo (Object obj) сравнивает объект, который используется для вызова метода, с указанным объектом, переданным методу. | Метод сравнения (Object obj1, Object obj2) сравнивает оба указанных объекта, которые передаются в метод. |
| Список / Массив | Когда необходимо сравнить список объектов типа Comparable, класс Collection предоставляет метод ie Collections.sort (List lst). | Когда необходимо сравнить список объектов типа Comparable, класс Collection предоставляет метод, т.е. Collections.sort (список, компаратор). |

# В чем разница между foreach и foreachordered

Разница между forEach и forEachOrdered заключается в том, что forEach позволяет обрабатывать любой элемент параллельного потока в любом порядке, в то время как forEachOdered всегда обрабатывает элементы параллельного потока в порядке их появления в исходном потоке

# Зачем нужно функциональное программирование, где оно применяется и в чем его отличие от программирования в стиле ооп или в процедурном стиле?

В отличие от ООП, функциональное программирование характеризуется слабой связью функции с данными, которыми она оперирует. Это позволяет избежать побочных эффектов при выполнении функций — например чтения и изменения глобальных переменных, операций ввода-вывода и так далее. Детерминированные функции ФП возвращают один и тот же результат для одних и тех же аргументов.

# Что такое default методы в интерфейсе и для чего они были введены?

Метод, реализованный в интерфейсе, называется методом по умолчанию и обозначается ключевым словом default.

Класс наследует реализацию по умолчанию. Вот почему не обязательно модифицировать классы при изменении интерфейса, который они реализуют.

Если методы объявлены более чем в одном интерфейсе, то никакой реализации по умолчанию классом не наследуется - вы получите ошибку компиляции.

Переменные интерфейса являются public static final по умолчанию и эти модификаторы необязательны при их объявлении.

# Как создать экземпляр анонимного класса (подробно)? Где они применяются?

1. Создаем локальный класс без имени,

2. Можно объявить такой класс, который может расширить (extends) другой класс или реализовать (implements) интерфейс. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора new и экземпляр некоторого класса.

3. Не должно быть конструктора,

**UnaryOperator<Integer> myOperatorWithoutName = new UnaryOperator<Integer>() {**

**@Override**

**public Integer apply(Integer x) {**

**return x \* x;**

**}**

**};**

Любые аргументы, которые вы укажете в круглых скобках, стоящих за именем родительского класса в определении анонимного класса, неявно передаются конструктору родительского класса.

Где они применяются?

Чаще всего анонимные классы применяются для расширения родительских классов простыми классами, которые не требуют аргументов конструктора, поэтому скобки в определении анонимного класса зачастую пусты.

Чтобы не создавать отдельный класс, если нужно встроить некоторую логику в существующий класс или связать с др.классом,

# Что такое ленивая инициализация стрима?

Ленивая инициализация - это оптимизация производительности, при которой вы откладываете (потенциально дорогостоящее) создание объекта до тех пор, пока оно вам действительно не понадобится.

Один из хороших примеров - это не создавать подключение к базе данных заранее, а только непосредственно перед тем, как вам нужно будет получить данные из базы данных.

Основная причина этого заключается в том, что (часто) вы можете полностью избежать создания объекта, если он вам никогда не понадобится.

# Зачем нужен класс Optional?

Optional - это класс оболочка, которая внутри себя содержит некоторое значение которое может быть NULL и если  это значение является NULL'ом, может предпринять какие-то действия, например бросить исключение или подставить значение по-умолчанию.

Целями которого являются повышение null-safety и читаемости кода при работе со ссылками, значения которых могут быть null. Главная же цель Optional — замена null-значений, благодаря чему должна повышаться безопасность и читаемость кода.

# Все способы реализации функциональных интерфейсов.

4 штуки: ссылка на метод, лямбда, через анонимный класс, через обычную имплементацию в классе.

# как взаимосвязаны лямбда и функциональный интерфейс?

Заменяют анонимные классы, синтаксический сахар, не нужно переопределять методы.

# Любой анонимный класс можно заменить на лямбду?

Нет. Вы можете создать анонимный класс для не финальных классов и интерфейсов. (до Java 8 каждый интерфейсный метод был абстрактным, но поскольку интерфейсы Java 8 также могут иметь стандартные и статические методы, которые не являются абстрактными, поскольку они имеют реализацию ).

Только анонимные классы, которые являются реализациями интерфейса (Single Abstract Method interfaces) SAM (например, Runnable, ActionListener, Comparator, Predicate), могут быть заменены лямбда-выражением. DefaultConsumer не может быть лямбда-мишенью, потому что даже не является интерфейсом.

Несмотря на то, что Consumer является интерфейсом, он не является интерфейсом SAM, поскольку имеет более одного абстрактного метода, поэтому он также не может быть лямбда-мишенью.

# В чем разница map и flatMap?

И map и flatMap могут быть применены к стриму Stream<T> и оба возвращают стрим Stream<R>. Разница заключается в том, что операция map создает одно выходное значение для каждого входного значения, тогда как операция flatMap создает произвольное число(ноль или больше) значений для каждого входного значения

# Зачем нужны default методы в функциональном интерфейсе - тут про обратную совместимость.

Чтобы не испортить код для других разработчиков.

# ссылка на метод - ну и в каком виде передается:

имя класса:: имя стат метода (для статического метода); объект класса:: имя метода (для метода экземпляра); название класса:: new(для конструктора)

# как получить стрим - через разные метод + бесконечные стримы ( через iterate и generate)

Поток 0 - 9

//Stream.iterate(initial value, next value)

Stream.iterate(0, n -> n + 1)

.limit(10)

.forEach(x -> System.out.println(x));

Выход

0

1

2

3

4

5

6

7

8

9

Stream<String> streamFromGenerate = Stream.generate(() -> "a1")

Stream.generate служит для генерации константных и случайных значений, он просто выдает значения, соответствующие выражению, в данном примере, он будет выдавать бесконечное количество значений «a1».

# что возвращают промежуточные операции:

stream

# Методы peek и forEach - в чем разница:

первый промежуточный, второй - терминальный, ну и функционал какой у каждого

peek - это просто способ заглянуть внутрь потока, он не меняет его, но функция в peek получит все элементы, которые находятся в потоке.

forEach - это терминальная операция, которая перебирает все данные в потоке и возвращает значение void.

# Любую ли лямбду можно свернуть в метод референс?

Если лямбда выражение вызывают только один метод