

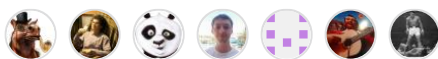


 enhorse / java-interview Public[Code](#) [Issues](#) 9 [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...

[java-interview](#) / java8.md

ScherbakovMike Update java8.md ...

 History 7 contributors 928 lines (710 sloc) | 69.3 KB

...

[Вопросы для собеседования](#)

Java 8

- [Какие нововведения, появились в Java 8 и JDK 8?](#)
- [Что такое «лямбда»? Какова структура и особенности использования лямбда-выражения?](#)
- [К каким переменным есть доступ у лямбда-выражений?](#)
- [Как отсортировать список строк с помощью лямбда-выражения?](#)
- [Что такое «ссылка на метод»?](#)
- [Какие виды ссылок на методы вы знаете?](#)
- [Объясните выражение `System.out::println`.](#)
- [Что такое «функциональные интерфейсы»?](#)
- [Для чего нужны функциональные интерфейсы `Function<T,R>`, `DoubleFunction<R>`, `IntFunction<R>` и `LongFunction<R>`?](#)
- [Для чего нужны функциональные интерфейсы `UnaryOperator<T>`, `DoubleUnaryOperator`, `IntUnaryOperator` и `LongUnaryOperator`?](#)
- [Для чего нужны функциональные интерфейсы `BinaryOperator<T>`, `DoubleBinaryOperator`, `IntBinaryOperator` и `LongBinaryOperator`?](#)

- Для чего нужны функциональные интерфейсы `Predicate<T>` , `DoublePredicate` , `IntPredicate` и `LongPredicate` ?
- Для чего нужны функциональные интерфейсы `Consumer<T>` , `DoubleConsumer` , `IntConsumer` и `LongConsumer` ?
- Для чего нужны функциональные интерфейсы `Supplier<T>` , `BooleanSupplier` , `DoubleSupplier` , `IntSupplier` и `LongSupplier` ?
- Для чего нужен функциональный интерфейс `BiConsumer<T,U>` ?
- Для чего нужен функциональный интерфейс `BiFunction<T,U,R>` ?
- Для чего нужен функциональный интерфейс `BiPredicate<T,U>` ?
- Для чего нужны функциональные интерфейсы вида `_To_Function` ?
- Для чего нужны функциональные интерфейсы `ToDoubleBiFunction<T,U>` , `ToIntBiFunction<T,U>` и `ToLongBiFunction<T,U>` ?
- Для чего нужны функциональные интерфейсы `ToDoubleFunction<T>` , `ToIntFunction<T>` и `ToLongFunction<T>` ?
- Для чего нужны функциональные интерфейсы `ObjDoubleConsumer<T>` , `ObjIntConsumer<T>` и `ObjLongConsumer<T>` ?
- Что такое `StringJoiner` ?
- Что такое `default` методы интерфейса?
- Как вызывать `default` метод интерфейса в реализующем этот интерфейс классе?
- Что такое `static` метод интерфейса?
- Как вызывать `static` метод интерфейса?
- Что такое `Optional` ?
- Что такое `Stream` ?
- Какие существуют способы создания стрима?
- В чем разница между `Collection` и `Stream` ?
- Для чего нужен метод `collect()` в стримах?
- Для чего в стримах применяются методы `forEach()` и `forEachOrdered()` ?
- Для чего в стримах предназначены методы `map()` и `mapToInt()` , `mapToDouble()` , `mapToLong()` ?
- Какова цель метода `filter()` в стримах?
- Для чего в стримах предназначен метод `limit()` ?
- Для чего в стримах предназначен метод `sorted()` ?
- Для чего в стримах предназначены методы `flatMap()` , `flatMapToInt()` , `flatMapToDouble()` , `flatMapToLong()` ?

- Расскажите о параллельной обработке в Java 8.
- Какие конечные методы работы со стримами вы знаете?
- Какие промежуточные методы работы со стримами вы знаете?
- Как вывести на экран 10 случайных чисел, используя `forEach()` ?
- Как можно вывести на экран уникальные квадраты чисел используя метод `map()` ?
- Как вывести на экран количество пустых строк с помощью метода `filter()` ?
- Как вывести на экран 10 случайных чисел в порядке возрастания?
- Как найти максимальное число в наборе?
- Как найти минимальное число в наборе?
- Как получить сумму всех чисел в наборе?
- Как получить среднее значение всех чисел?
- Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?
- Что такое `LocalDateTime` ?
- Что такое `ZonedDateTime` ?
- Как получить текущую дату с использованием Date Time API из Java 8?
- Как добавить 1 неделю, 1 месяц, 1 год, 10 лет к текущей дате с использованием Date Time API?
- Как получить следующий вторник используя Date Time API?
- Как получить вторую субботу текущего месяца используя Date Time API?
- Как получить текущее время с точностью до миллисекунд используя Date Time API?
- Как получить текущее время по местному времени с точностью до миллисекунд используя Date Time API?
- Как определить повторяемую аннотацию?
- Что такое `Nashorn` ?
- Что такое `jjs` ?
- Какой класс появился в Java 8 для кодирования/декодирования данных?
- Как создать Base64 кодировщик и декодировщик?

Какие нововведения, появились в Java 8 и JDK 8?

- Методы интерфейсов по умолчанию;

- Лямбда-выражения;
- Функциональные интерфейсы;
- Ссылки на методы и конструкторы;
- Повторяемые аннотации;
- Аннотации на типы данных;
- Рефлексия для параметров методов;
- *Stream API* для работы с коллекциями;
- Параллельная сортировка массивов;
- Новое API для работы с датами и временем;
- Новый движок JavaScript *Nashorn*;
- Добавлено несколько новых классов для потокобезопасной работы;
- Добавлен новый API для `Calendar` и `Locale` ;
- Добавлена поддержка *Unicode 6.2.0*;
- Добавлен стандартный класс для работы с *Base64*;
- Добавлена поддержка беззнаковой арифметики;
- Улучшена производительность конструктора `java.lang.String(byte[], *)` и метода `java.lang.String.getBytes()` ;
- Новая реализация `AccessController.doPrivileged` , позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;
- *Password-based* алгоритмы стали более устойчивыми;
- Добавлена поддержка *SSL/TLS Server Name Indication (NSI)* в *JSSE Server*;
- Улучшено хранилище ключей (`KeyStore`);
- Добавлен алгоритм *SHA-224*;
- Удален мост *JDBC - ODBC*;
- Удален *PermGen*, изменен способ хранения мета-данных классов;
- Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;
- Инструментарий
 - Добавлена утилита `jjs` для использования *JavaScript Nashorn*;
 - Команда `java` может запускать *JavaFX* приложения;
 - Добавлена утилита `jdeps` для анализа *.class*-файлов.

[к оглавлению](#)

Что такое «лямбда»? Какова структура и особенности использования лямбда-выражения?

Лямбда представляет собой 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.

```
interface Operationable {
    int calculate(int x, int y);
}

public static void main(String[] args) {
    Operationable operation = (x, y) -> x + y;
    int result = operation.calculate(10, 20);
    System.out.println(result); //30
}
```

По факту **лямбда-выражения** являются в некотором роде **сокращенной формой внутренних анонимных классов**, которые ранее применялись в Java.

- *Отложенное выполнение (deferred execution) лямбда-выражения*- определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.
- *Параметры лямбда-выражения* должны соответствовать по типу параметрам метода функционального интерфейса:

```
operation = (int x, int y) -> x + y;
//При написании самого лямбда-выражения тип параметров разрешается не указывать:
(x, y) -> x + y;
//Если метод не принимает никаких параметров, то пишутся пустые скобки, например,
() -> 30 + 20;
```

```
//Если метод принимает только один параметр, то скобки можно опустить:  
n -> n * n;
```

- *Конечные лямбда-выражения* не обязаны возвращать какое-либо значение.

```
interface Printable {  
    void print(String s);  
}  
  
public static void main(String[] args) {  
    Printable printer = s -> System.out.println(s);  
    printer.print("Hello, world");  
}
```

- *Блочные лямбда-выражения* обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции `if`, `switch`, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор `return`:

```
Operationable operation = (int x, int y) -> {  
    if (y == 0) {  
        return 0;  
    }  
    else {  
        return x / y;  
    }  
};
```

- *Передача лямбда-выражения в качестве параметра метода:*

```
interface Condition {  
    boolean isAppropriate(int n);  
}  
  
private static int sum(int[] numbers, Condition condition) {  
    int result = 0;  
    for (int i : numbers) {  
        if (condition.isAppropriate(i)) {  
            result += i;  
        }  
    }  
    return result;  
}
```

```
}

public static void main(String[] args) {
    System.out.println(sum(new int[] {0, 1, 0, 3, 0, 5, 0, 7, 0, 9}, (n) -> n != 0));
}
```

[к оглавлению](#)

К каким переменным есть доступ у лямбда-выражений?

Доступ к переменным внешней области действия из лямбда-выражения очень схож к доступу из анонимных объектов. Можно сослаться на:

- неизменяемые (*effectively final* - не обязательно помеченные как `final`) локальные переменные;
- поля класса;
- статические переменные.

К методам по умолчанию реализуемого функционального интерфейса обращаться внутри лямбда-выражения запрещено.

[к оглавлению](#)

Как отсортировать список строк с помощью лямбда-выражения?

```
public static List<String> sort(List<String> list){
    Collections.sort(list, (a, b) -> a.compareTo(b));
    return list;
}
```

[к оглавлению](#)

Что такое «ссылка на метод»?

Если существующий в классе метод уже делает все, что необходимо, то можно воспользоваться механизмом **method reference (ссылка на метод)** для непосредственной передачи этого метода. Такая ссылка передается в виде:

- `имя_класса::имя_статического_метода` для статического метода;
- `объект_класса::имя_метода` для метода экземпляра;
- `название_класса::new` для конструктора.

Результат будет в точности таким же, как в случае определения **лямбда-выражения**, которое вызывает этот метод.

```
private interface Measurable {  
    public int length(String string);  
}  
  
public static void main(String[] args) {  
    Measurable a = String::length;  
    System.out.println(a.length("abc"));  
}
```

Ссылки на методы потенциально более эффективны, чем использование **лямбда-выражений**. Кроме того, они предоставляют компилятору более качественную информацию о типе и **при возможности выбора** между использованием ссылки на существующий метод и использованием лямбда-выражения, **следует всегда предпочитать** использование **ссылки на метод**.

[к оглавлению](#)

Какие виды ссылок на методы вы знаете?

- на статический метод;
- на метод экземпляра;
- на конструкторе.

[к оглавлению](#)

Объясните выражение `System.out::println`.

Данное выражение иллюстрирует механизм **instance method reference**: передачи ссылки на метод `println()` статического поля `out` класса `System`.

[к оглавлению](#)

Что такое «функциональные интерфейсы»?

Функциональный интерфейс - это интерфейс, который определяет только один абстрактный метод.

Чтобы точно определить интерфейс как **функциональный**, добавлена аннотация `@FunctionalInterface`, работающая по принципу `@Override`. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Интерфейс может включать сколько угодно **default** методов и при этом оставаться **функциональным**, потому что **default** методы - не абстрактные.

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `Function<T,R>`, `DoubleFunction<R>`, `IntFunction<R>` и `LongFunction<R>`?

`Function<T, R>` - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса `T` и возвращающая на выходе экземпляр класса `R`.

Методы по умолчанию могут использоваться для построения цепочек вызовов (`compose`, `andThen`).

```
Function<String, Integer> toInteger = Integer::valueOf;  
Function<String, String> backToString = toInteger.andThen(String::valueOf);  
backToString.apply("123");    // "123"
```

- `DoubleFunction<R>` - функция, получающая на вход `Double` и возвращающая на выходе экземпляр класса `R`;
- `IntFunction<R>` - функция, получающая на вход `Integer` и возвращающая на выходе экземпляр класса `R`;
- `LongFunction<R>` - функция, получающая на вход `Long` и возвращающая на выходе экземпляр класса `R`.

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `UnaryOperator<T>`, `DoubleUnaryOperator`, `IntUnaryOperator` и `LongUnaryOperator`?

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

```
UnaryOperator<Integer> operator = x -> x * x;  
System.out.println(operator.apply(5)); // 25
```

- `DoubleUnaryOperator` - унарный оператор, получающий на вход `Double`;
- `IntUnaryOperator` - унарный оператор, получающий на вход `Integer`;
- `LongUnaryOperator` - унарный оператор, получающий на вход `Long`.

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `BinaryOperator<T>`, `DoubleBinaryOperator`, `IntBinaryOperator` и `LongBinaryOperator`?

`BinaryOperator<T>` (бинарный оператор) - интерфейс, с помощью которого реализуется функция, получающая на вход два экземпляра класса `T` и возвращающая на выходе экземпляр класса `T`.

```
BinaryOperator<Integer> operator = (a, b) -> a + b;  
System.out.println(operator.apply(1, 2)); // 3
```

- `DoubleBinaryOperator` - бинарный оператор, получающий на вход `Double`;
- `IntBinaryOperator` - бинарный оператор, получающий на вход `Integer`;
- `LongBinaryOperator` - бинарный оператор, получающий на вход `Long`.

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `Predicate<T>`, `DoublePredicate`, `IntPredicate` и

LongPredicate ?

Predicate<T> (предикат) - интерфейс, с помощью которого реализуется функция, получающая на вход экземпляр класса **T** и возвращающая на выходе значение типа **boolean**.

Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (**and**, **or**, **negate**).

```
Predicate<String> predicate = (s) -> s.length() > 0;  
predicate.test("foo"); // true  
predicate.negate().test("foo"); // false
```

- **DoublePredicate** - предикат, получающий на вход **Double** ;
- **IntPredicate** - предикат, получающий на вход **Integer** ;
- **LongPredicate** - предикат, получающий на вход **Long** .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы **Consumer<T>**, **DoubleConsumer**, **IntConsumer** и **LongConsumer** ?

Consumer<T> (потребитель) - интерфейс, с помощью которого реализуется функция, которая получает на вход экземпляр класса **T**, производит с ним некоторое действие и ничего не возвращает.

```
Consumer<String> hello = (name) -> System.out.println("Hello, " + name);  
hello.accept("world");
```

- **DoubleConsumer** - потребитель, получающий на вход **Double** ;
- **IntConsumer** - потребитель, получающий на вход **Integer** ;
- **LongConsumer** - потребитель, получающий на вход **Long** .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы

Supplier<T>, BooleanSupplier, DoubleSupplier, IntSupplier и LongSupplier ?

Supplier<T> (поставщик) - интерфейс, с помощью которого реализуется функция, ничего не принимающая на вход, но возвращающая на выход результат класса `T` ;

```
Supplier<LocalDateTime> now = LocalDateTime::now;  
now.get();
```

- `DoubleSupplier` - поставщик, возвращающий `Double` ;
- `IntSupplier` - поставщик, возвращающий `Integer` ;
- `LongSupplier` - поставщик, возвращающий `Long` .

[к оглавлению](#)

Для чего нужен функциональный интерфейс BiConsumer<T,U> ?

BiConsumer<T,U> представляет собой операцию, которая принимает два аргумента классов `T` и `U` производит с ними некоторое действие и ничего не возвращает.

[к оглавлению](#)

Для чего нужен функциональный интерфейс BiFunction<T,U,R> ?

BiFunction<T,U,R> представляет собой операцию, которая принимает два аргумента классов `T` и `U` и возвращающая результат класса `R` .

[к оглавлению](#)

Для чего нужен функциональный интерфейс BiPredicate<T,U> ?

BiPredicate<T,U> представляет собой операцию, которая принимает два аргумента классов `T` и `U` и возвращающая результат типа `boolean` .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы вида `_To_Function` ?

- `DoubleToIntFunction` - операция, принимающая аргумент класса `Double` и возвращающая результат типа `Integer` ;
- `DoubleToLongFunction` - операция, принимающая аргумент класса `Double` и возвращающая результат типа `Long` ;
- `IntToDoubleFunction` - операция, принимающая аргумент класса `Integer` и возвращающая результат типа `Double` ;
- `IntToLongFunction` - операция, принимающая аргумент класса `Integer` и возвращающая результат типа `Long` ;
- `LongToDoubleFunction` - операция, принимающая аргумент класса `Long` и возвращающая результат типа `Double` ;
- `LongToIntFunction` - операция, принимающая аргумент класса `Long` и возвращающая результат типа `Integer` .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `ToDoubleBiFunction<T,U>` , `ToIntBiFunction<T,U>` и `ToLongBiFunction<T,U>` ?

- `ToDoubleBiFunction<T,U>` - операция принимающая два аргумента классов `T` и `U` и возвращающая результат типа `Double` ;
- `ToLongBiFunction<T,U>` - операция принимающая два аргумента классов `T` и `U` и возвращающая результат типа `Long` ;
- `ToIntBiFunction<T,U>` - операция принимающая два аргумента классов `T` и `U` и возвращающая результат типа `Integer` .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `ToDoubleFunction<T>` , `ToIntFunction<T>` и `ToLongFunction<T>` ?

- `ToDoubleFunction<T>` - операция, принимающая аргумент класса `T` и

возвращающая результат типа `Double` ;

- `ToLongFunction<T>` - операция, принимающая аргумент класса `T` и возвращающая результат типа `Long` ;
- `ToIntFunction<T>` - операция, принимающая аргумент класса `T` и возвращающая результат типа `Integer` .

[к оглавлению](#)

Для чего нужны функциональные интерфейсы `ObjDoubleConsumer<T>`, `ObjIntConsumer<T>` и `ObjLongConsumer<T>` ?

- `ObjDoubleConsumer<T>` - операция, которая принимает два аргумента классов `T` и `Double` , производит с ними некоторое действие и ничего не возвращает;
- `ObjLongConsumer<T>` - операция, которая принимает два аргумента классов `T` и `Long` , производит с ними некоторое действие и ничего не возвращает;
- `ObjIntConsumer<T>` - операция, которая принимает два аргумента классов `T` и `Integer` , производит с ними некоторое действие и ничего не возвращает.

[к оглавлению](#)

Что такое `StringJoiner` ?

Класс `StringJoiner` используется, чтобы создать последовательность строк, разделенных разделителем с возможностью присоединить к полученной строке префикс и суффикс:

```
StringJoiner joiner = new StringJoiner(".", "prefix-", "-suffix");
for (String s : "Hello the brave world".split(" ")) {
    joiner.add(s);
}
System.out.println(joiner); //prefix-Hello.the.brave.world-suffix
```

[к оглавлению](#)

Что такое `default` методы интерфейса?

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`:

```
interface Example {  
    int process(int a);  
    default void show() {  
        System.out.println("default show()");  
    }  
}
```

- Если класс реализует интерфейс, он может, но не обязан, реализовать методы по-умолчанию уже реализованные в интерфейсе. Класс наследует реализацию по умолчанию.
- Если некий класс реализует несколько интерфейсов, которые имеют одинаковый метод по умолчанию, то класс должен реализовать метод с совпадающей сигнатурой самостоятельно. Ситуация аналогична, если один интерфейс имеет метод по умолчанию, а в другом этот же метод является абстрактным - никакой реализации по умолчанию классом не наследуется.
- Метод по умолчанию не может переопределить метод класса `java.lang.Object`.
- Помогают реализовывать интерфейсы без страха нарушить работу других классов.
- Позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.
- Дают свободу классам выбрать метод, который нужно переопределить.
- Одной из основных причин внедрения методов по умолчанию является возможность коллекций в Java 8 использовать лямбда-выражения.

[к оглавлению](#)

Как вызывать default метод интерфейса в реализующем этот интерфейс классе?

Используя ключевое слово `super` вместе с именем интерфейса:

```
interface Paper {  
    default void show() {  
        System.out.println("default show()");  
    }  
}
```

```
}  
  
class Licence implements Paper {  
    public void show() {  
        Paper.super.show();  
    }  
}
```

[к оглавлению](#)

Что такое static метод интерфейса?

Статические методы интерфейса похожи на методы по умолчанию, за исключением того, что для них отсутствует возможность переопределения в классах, реализующих интерфейс.

- Статические методы в интерфейсе являются частью интерфейса без возможности переопределить их для объектов класса реализации;
- Методы класса `java.lang.Object` нельзя переопределить как статические;
- Статические методы в интерфейсе используются для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.

[к оглавлению](#)

Как вызывать static метод интерфейса?

Используя имя интерфейса:

```
interface Paper {  
    static void show() {  
        System.out.println("static show()");  
    }  
}  
  
class Licence {  
    public void showPaper() {  
        Paper.show();  
    }  
}
```


[к оглавлению](#)

Что такое Optional ?

Опциональное значение `Optional` — это контейнер для объекта, который может содержать или не содержать значение `null`. Такая обёртка является удобным средством предотвращения `NullPointerException`, т.к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся `if null/notNull` проверок:

```
Optional<String> optional = Optional.of("hello");

optional.isPresent(); // true
optional.ifPresent(s -> System.out.println(s.length())); // 5
optional.get(); // "hello"
optional.orElse("ops..."); // "hello"
```

[к оглавлению](#)

Что такое Stream ?

Интерфейс `java.util.Stream` представляет собой последовательность элементов, над которой можно производить различные операции.

Операции над стримами бывают или *промежуточными (intermediate)* или *конечными (terminal)*. Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же стрим. Таким образом вы можете строить цепочки из несколько операций над одним и тем же стримом.

У стрима может быть сколько угодно вызовов промежуточных операций и последним вызов конечной операции. При этом все промежуточные операции выполняются лениво и пока не будет вызвана конечная операция никаких действий на самом деле не происходит (похоже на создание объекта `Thread` или `Runnable`, без вызова `start()`).

Стримы создаются на основе каких-либо источников, например классов из `java.util.Collection`.

Ассоциативные массивы (maps), например, `HashMap`, не поддерживаются.

Операции над стримами могут выполняться как последовательно, так и параллельно.

Потоки не могут быть использованы повторно. Как только была вызвана какая-нибудь конечная операция, поток закрывается.

Кроме универсальных объектных **существуют особые виды стримов для работы с примитивными типами** данных `int`, `long` и `double`: `IntStream`, `LongStream` и `DoubleStream`. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:

- используют специализированные лямбда-выражения, например, `IntFunction` или `IntPredicate` ВМЕСТО `Function` и `Predicate`;
- поддерживают дополнительные конечные операции `sum()`, `average()`, `mapToObj()`.

[к оглавлению](#)

Какие существуют способы создания стрима?

1. Из коллекции:

```
Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();
```

2. Из набора значений:

```
Stream<String> fromValues = Stream.of("x", "y", "z");
```

3. Из массива:

```
Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});
```

4. Из файла (каждая строка в файле будет отдельным элементом в стриме):

```
Stream<String> fromFile = Files.lines(Paths.get("input.txt"));
```

5. Из строки:

```
IntStream fromString = "0123456789".chars();
```

6. С помощью `Stream.builder()` :

```
Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();
```

7. С помощью `Stream.iterate()` (бесконечный):

```
Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);
```

8. С помощью `Stream.generate()` (бесконечный):

```
Stream<String> fromGenerate = Stream.generate(() -> "0");
```

[к оглавлению](#)

В чем разница между Collection и Stream?

Коллекции позволяют работать с элементами по-отдельности, тогда как стримы так делать не позволяют, но вместо этого предоставляют возможность выполнять функции над данными как над одним целым.

Также стоит отметить важность самой концепции сущностей: `Collection` - это прежде всего воплощение *Структуры Данных*. Например, `Set` не просто хранит в себе элементы, он реализует идею множества с уникальными элементами, тогда как `Stream`, это прежде всего абстракция необходимая для реализации *конвейера вычислений*, собственно, поэтому, результатом работы конвейера являются те или иные *Структуры Данных* или же результаты проверок/поиска и т.п.

[к оглавлению](#)

Для чего нужен метод collect() в стримах?

Метод `collect()` является *конечной операцией*, которая используется для представление результата в виде *коллекции* или какой-либо другой структуры данных.

`collect()` принимает на вход `Collector<Тип_источника, Тип_аккумулятора, Тип_результата>`, который содержит четыре этапа: *supplier* - инициализация аккумулятора, *accumulator* - обработка каждого элемента, *combiner* - соединение двух аккумуляторов при параллельном выполнении, *[finisher]* - необязательный метод последней обработки аккумулятора. В Java 8 в классе `Collectors` реализовано несколько распространённых коллекторов:

- `toList()`, `toCollection()`, `toSet()` - представляют стрим в виде списка, коллекции или множества;
- `toConcurrentMap()`, `toMap()` - позволяют преобразовать стрим в `Map`;
- `averagingInt()`, `averagingDouble()`, `averagingLong()` - возвращают среднее значение;
- `summingInt()`, `summingDouble()`, `summingLong()` - возвращает сумму;
- `summarizingInt()`, `summarizingDouble()`, `summarizingLong()` - возвращают `SummaryStatistics` с разными агрегатными значениями;
- `partitioningBy()` - разделяет коллекцию на две части по соответствию условию и возвращает их как `Map<Boolean, List>`;
- `groupingBy()` - разделяет коллекцию на несколько частей и возвращает `Map<N, List<T>>`;
- `mapping()` - дополнительные преобразования значений для сложных `Collector` -ОВ.

Так же существует возможность создания собственного коллектора через `Collector.of()`:

```
Collector<String, List<String>, List<String>> toList = Collector.of(
    ArrayList::new,
    List::add,
    (l1, l2) -> { l1.addAll(l2); return l1; }
);
```

[к оглавлению](#)

Для чего в стримах применяются методы `forEach()` и `forEachOrdered()`?

- `forEach()` применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется;

- `forEachOrdered()` применяет функцию к каждому объекту стрима с сохранением порядка элементов.

[к оглавлению](#)

Для чего в стримах предназначены методы `map()` и `mapToInt()`, `mapToDouble()`, `mapToLong()` ?

Метод `map()` является промежуточной операцией, которая заданным образом преобразует каждый элемент стрима.

`mapToInt()`, `mapToDouble()`, `mapToLong()` - аналоги `map()`, возвращающие соответствующий числовой стрим (то есть стрим из числовых примитивов):

```
Stream
    .of("12", "22", "4", "444", "123")
    .mapToInt(Integer::parseInt)
    .toArray(); //[12, 22, 4, 444, 123]
```

[к оглавлению](#)

Какова цель метода `filter()` в стримах?

Метод `filter()` является промежуточной операцией, принимающей предикат, который фильтрует все элементы, возвращая только те, что соответствуют условию.

[к оглавлению](#)

Для чего в стримах предназначен метод `limit()` ?

Метод `limit()` является промежуточной операцией, которая позволяет ограничить выборку определенным количеством первых элементов.

[к оглавлению](#)

Для чего в стримах предназначен метод `sorted()` ?

Метод `sorted()` является промежуточной операцией, которая позволяет сортировать значения либо в натуральном порядке, либо задавая `Comparator`.

Порядок элементов в исходной коллекции остается нетронутым - `sorted()` всего лишь создает его отсортированное представление.

[к оглавлению](#)

Для чего в стримах предназначены методы `flatMap()`, `flatMapToInt()`, `flatMapToDouble()`, `flatMapToLong()`?

Метод `flatMap()` похож на `map`, но может создавать из одного элемента несколько. Таким образом, каждый объект будет преобразован в ноль, один или несколько других объектов, поддерживаемых потоком. Наиболее очевидный способ применения этой операции — преобразование элементов контейнера при помощи функций, которые возвращают контейнеры.

```
Stream
    .of("H e l l o", "w o r l d !")
    .flatMap((p) -> Arrays.stream(p.split(" ")))
    .toArray(String[]::new); //[ "H", "e", "l", "l", "o", "w", "o", "r", "l", "d",
```

`flatMapToInt()`, `flatMapToDouble()`, `flatMapToLong()` - это аналоги `flatMap()`, возвращающие соответствующий числовой стрим.

[к оглавлению](#)

Расскажите о параллельной обработке в Java 8.

Стримы могут быть последовательными и параллельными. Операции над последовательными стримами выполняются в одном потоке процессора, над параллельными — используя несколько потоков процессора. Параллельные стримы используют общий `ForkJoinPool` доступный через статический `ForkJoinPool.commonPool()` метод. При этом, если окружение не является многоядерным, то поток будет выполняться как последовательный. Фактически применение параллельных стримов сводится к тому, что данные в стримах будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются конечные операции.

Для создания параллельного потока из коллекции можно также использовать метод `parallelStream()` интерфейса `Collection`.

Чтобы сделать обычный последовательный стрим параллельным, надо вызвать у объекта `Stream` метод `parallel()`. Метод `isParallel()` позволяет узнать является ли стрим параллельным.

С помощью методов `parallel()` и `sequential()` можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот:

```
collection
    .stream()
    .peek(...) // операция последовательна
    .parallel()
    .map(...) // операция может выполняться параллельно,
    .sequential()
    .reduce(...) // операция снова последовательна
```

Как правило, элементы передаются в стрим в том же порядке, в котором они определены в источнике данных. При работе с параллельными стримами система сохраняет порядок следования элементов. Исключение составляет метод `forEach()`, который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод `forEachOrdered()`.

Критерии, которые могут повлиять на производительность в параллельных стримах:

- Размер данных - чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.
- Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.
- Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из `ArrayList` легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа `LinkedList` - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.
- Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов.

- Крайне не рекомендуется использовать параллельные стримы для скольких-нибудь долгих операций (например, сетевых соединений), так как все параллельные стримы работают с одним ForkJoinPool, то такие долгие операции могут остановить работу всех параллельных стримов в JVM из-за отсутствия доступных потоков в пуле, т.е. параллельные стримы стоит использовать лишь для коротких операций, где счет идет на миллисекунды, но не для тех где счет может идти на секунды и минуты;
- Сохранение порядка в параллельных стримах увеличивает издержки при выполнении и если порядок не важен, то имеется возможность отключить его сохранение и тем самым увеличить производительность, использовав промежуточную операцию `unordered()` :

```
collection.parallelStream()  
    .sorted()  
    .unordered()  
    .collect(Collectors.toList());
```

[к оглавлению](#)

Какие конечные методы работы со стримами вы знаете?

- `findFirst()` возвращает первый элемент;
- `findAny()` возвращает любой подходящий элемент;
- `collect()` представление результатов в виде коллекций и других структур данных;
- `count()` возвращает количество элементов;
- `anyMatch()` возвращает `true` , если условие выполняется хотя бы для одного элемента;
- `noneMatch()` возвращает `true` , если условие не выполняется ни для одного элемента;
- `allMatch()` возвращает `true` , если условие выполняется для всех элементов;
- `min()` возвращает минимальный элемент, используя в качестве условия `Comparator` ;
- `max()` возвращает максимальный элемент, используя в качестве условия `Comparator` ;
- `forEach()` применяет функцию к каждому объекту (порядок при

параллельном выполнении не гарантируется);

- `forEachOrdered()` применяет функцию к каждому объекту с сохранением порядка элементов;
- `toArray()` возвращает массив значений;
- `reduce()` позволяет выполнять агрегатные функции и возвращать один результат.

Для числовых стримов дополнительно доступны:

- `sum()` возвращает сумму всех чисел;
- `average()` возвращает среднее арифметическое всех чисел.

[к оглавлению](#)

Какие промежуточные методы работы со стримами вы знаете?

- `filter()` отфильтровывает записи, возвращая только записи, соответствующие условию;
- `skip()` позволяет пропустить определённое количество элементов в начале;
- `distinct()` возвращает стрим без дубликатов (для метода `equals()`);
- `map()` преобразует каждый элемент;
- `peek()` возвращает тот же стрим, применяя к каждому элементу функцию;
- `limit()` позволяет ограничить выборку определенным количеством первых элементов;
- `sorted()` позволяет сортировать значения либо в натуральном порядке, либо задавая `Comparator`;
- `mapToInt()`, `mapToDouble()`, `mapToLong()` - аналоги `map()` возвращающие стрим числовых примитивов;
- `flatMap()`, `flatMapToInt()`, `flatMapToDouble()`, `flatMapToLong()` - похожи на `map()`, но могут создавать из одного элемента несколько.

Для числовых стримов дополнительно доступен метод `mapToObj()`, который преобразует числовой стрим обратно в объектный.

[к оглавлению](#)

Как вывести на экран 10 случайных чисел, используя

forEach() ?

```
(new Random())  
    .ints()  
    .limit(10)  
    .forEach(System.out::println);
```

[к оглавлению](#)

Как можно вывести на экран уникальные квадраты чисел используя метод map() ?

```
Stream  
    .of(1, 2, 3, 2, 1)  
    .map(s -> s * s)  
    .distinct()  
    .forEach(System.out::println);
```

[к оглавлению](#)

Как вывести на экран количество пустых строк с помощью метода filter() ?

```
System.out.println(  
    Stream  
        .of("Hello", "", " ", " ", "world", "!")  
        .filter(String::isEmpty)  
        .count());
```

[к оглавлению](#)

Как вывести на экран 10 случайных чисел в порядке возрастания?

```
(new Random())  
    .ints()  
    .limit(10)
```

```
.sorted()  
.forEach(System.out::println);
```

[к оглавлению](#)

Как найти максимальное число в наборе?

```
Stream  
    .of(5, 3, 4, 55, 2)  
    .mapToInt(a -> a)  
    .max()  
    .getAsInt(); //55
```

[к оглавлению](#)

Как найти минимальное число в наборе?

```
Stream  
    .of(5, 3, 4, 55, 2)  
    .mapToInt(a -> a)  
    .min()  
    .getAsInt(); //2
```

[к оглавлению](#)

Как получить сумму всех чисел в наборе?

```
Stream  
    .of(5, 3, 4, 55, 2)  
    .mapToInt()  
    .sum(); //69
```

[к оглавлению](#)

Как получить среднее значение всех чисел?

```
Stream  
    .of(5, 3, 4, 55, 2)
```

```
.mapToInt(a -> a)
.average()
.getAsDouble(); //13.8
```

[к оглавлению](#)

Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?

- `putIfAbsent()` добавляет пару «ключ-значение», только если ключ отсутствовал:

```
map.putIfAbsent("a", "Aa");
```

- `forEach()` принимает функцию, которая производит операцию над каждым элементом:

```
map.forEach((k, v) -> System.out.println(v));
```

- `compute()` создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //[ "a", "aAa" ]
```

- `computeIfPresent()` если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.computeIfPresent("a", (k, v) -> k.concat(v));
```

- `computeIfAbsent()` если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):

```
map.computeIfAbsent("a", k -> "A".concat(k)); //[ "a", "Aa" ]
```

- `getOrDefault()` в случае отсутствия ключа, возвращает переданное значение по-умолчанию:

```
map.getOrDefault("a", "not found");
```

- `merge()` принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключем значение

отсутствует, то записывает туда передаваемое значение.

```
map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //[ "a", "Aaz" ]
```

[к оглавлению](#)

Что такое LocalDateTime ?

`LocalDateTime` объединяет вместе `LocalDate` и `LocalTime`, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу. Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как `plusMinutes`, `plusHours`, `isAfter`, `toSecondOfDay` и т.д.

[к оглавлению](#)

Что такое ZonedDateTime ?

`java.time.ZonedDateTime` — аналог `java.util.Calendar`, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601. Включает временную зону, поэтому все операции с временными сдвигами этот класс проводит с её учётом.

[к оглавлению](#)

Как получить текущую дату с использованием Date Time API из Java 8?

```
LocalDate.now();
```

[к оглавлению](#)

Как добавить 1 неделю, 1 месяц, 1 год, 10 лет к текущей дате с использованием Date Time API?

```
LocalDate.now().plusWeeks(1);  
LocalDate.now().plusMonths(1);  
LocalDate.now().plusYears(1);  
LocalDate.now().plus(1, ChronoUnit.DECADES);
```

[к оглавлению](#)

Как получить следующий вторник используя Date Time API?

```
LocalDate.now().with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
```

[к оглавлению](#)

Как получить вторую субботу текущего месяца используя Date Time API?

```
LocalDate
    .of(LocalDate.now().getYear(), LocalDate.now().getMonth(), 1)
    .with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY))
    .with(TemporalAdjusters.next(DayOfWeek.SATURDAY));
```

[к оглавлению](#)

Как получить текущее время с точностью до миллисекунд используя Date Time API?

```
new Date().toInstant();
```

[к оглавлению](#)

Как получить текущее время по местному времени с точностью до миллисекунд используя Date Time API?

```
LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault());
```

[к оглавлению](#)

Как определить повторяемую аннотацию?

Чтобы определить повторяемую аннотацию, необходимо создать аннотацию-контейнер для списка повторяемых аннотаций и обозначить повторяемую мета-аннотацией `@Repeatable` :

```
@interface Schedulers
{
    Scheduler[] value();
}

@Repeatable(Schedulers.class)
@interface Scheduler
{
    String birthday() default "Jan 8 1935";
}
```

[к оглавлению](#)

Что такое Nashorn ?

Nashorn - это движок JavaScript, разрабатываемый на Java компанией Oracle. Призван дать возможность встраивать код JavaScript в приложения Java. В сравнении с *Rhino*, который поддерживается Mozilla Foundation, Nashorn обеспечивает от 2 до 10 раз более высокую производительность, так как он компилирует код и передает байт-код виртуальной машине Java непосредственно в память. Nashorn умеет компилировать код JavaScript и генерировать классы Java, которые загружаются специальным загрузчиком. Так же возможен вызов кода Java прямо из JavaScript.

[к оглавлению](#)

Что такое jjs ?

jjs это утилита командной строки, которая позволяет исполнять программы на языке JavaScript прямо в консоли.

[к оглавлению](#)

Какой класс появился в Java 8 для кодирования/декодирования данных?

Base64 - потокобезопасный класс, который реализует кодировщик и декодировщик данных, используя схему кодирования base64 согласно *RFC 4648* и *RFC 2045*.

Base64 содержит 6 основных методов:

`getEncoder()` / `getDecoder()` - возвращает кодировщик/декодировщик base64, соответствующий стандарту *RFC 4648*; `getUrlEncoder()` / `getUrlDecoder()` - возвращает URL-safe кодировщик/декодировщик base64, соответствующий стандарту *RFC 4648*; `getMimeEncoder()` / `getMimeDecoder()` - возвращает MIME кодировщик/декодировщик, соответствующий стандарту *RFC 2045*.

[к оглавлению](#)

Как создать Base64 кодировщик и декодировщик?

```
// Encode
String b64 = Base64.getEncoder().encodeToString("input".getBytes("utf-8")); //aW5
// Decode
new String(Base64.getDecoder().decode("aW5wdXQ=="), "utf-8"); //input
```

[к оглавлению](#)

Источники

- [Хабрахабр - Новое в Java 8](#)
- [Хабрахабр - Шпаргалка Java программиста 4. Java Stream API](#)
- [METANIT.COM](#)
- [javadevblog.com](#)

[Вопросы для собеседования](#)