



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 13

Функциональные интерфейсы

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021





Функциональный интерфейс

Что же такое *функциональный интерфейс*?

Все просто – это интерфейс, в котором объявлен *ровно один* метод:

```
interface MyFunctionalInterface {  
    String performTransform(int input);  
}
```

При этом, статические и default-методы интерфейса в счёт не идут:

```
interface MyFunctionalInterface {  
    double performTransform(double input);  
  
    default double performTransformTwice(double input) {  
        return performTransform(performTransform(input));  
    }  
  
    static void doSomething() { /* ... */ }  
}
```

Получается, функциональный интерфейс описывает функцию с некоторой сигнатурой

Аннотация `FunctionalInterface`

Функциональные интерфейсы принято обозначать аннотацией `FunctionalInterface`:

```
@FunctionalInterface
interface MyFunctionalInterface {
    double performTransform(double input);
}
```

Аннотация не является обязательной, чтобы функциональный интерфейс являлся таковым

По сути, у аннотации две функции:

1. Явно подчеркнуть, что интерфейс задумывался именно как функциональный
2. Компилятор будет явно требовать и проверять, что функциональный интерфейс объявлен корректно

```
@FunctionalInterface
interface TwoMethods {
    void meth1();
    void meth2();
}
```

```
java: Unexpected @FunctionalInterface annotation
ru.hse.lecture13.TwoMethods is not a functional interface
multiple non-overriding abstract methods found in interface ru.hse.lecture13.TwoMethods
```

```
@FunctionalInterface
interface NoMethods {
    static void staticDoesntCount() {}
}
```

```
java: Unexpected @FunctionalInterface annotation
ru.hse.lecture13.NoMethods is not a functional interface
no abstract method found in interface ru.hse.lecture13.NoMethods
```



Небольшой вопрос

Является ли этот интерфейс функциональным?

```
@FunctionalInterface
interface Mysterious {
    String toString();
}
```

Ответ: не является, поскольку `toString` – перегрузка существующего публичного метода класса `Object`, он не считается за отдельный метод

```
java: Unexpected @FunctionalInterface annotation
ru.hse.lecture13.Mysterious is not a functional interface
no abstract method found in interface ru.hse.lecture13.Mysterious
```

Интерфейс Runnable

Рассмотрим простой функциональный интерфейс – Runnable:

```
package java.lang;

@FunctionalInterface
public interface Runnable {
    void run();
}
```

Runnable описывает нечто, что можно просто запустить, вызвав метод run()

Но зачем вообще начали писать подобные интерфейсы?

Всё просто – иногда возникает необходимость использовать функцию как объект, чтобы её можно было передать в другую функцию или сохранить в переменную:

```
int makeUseOfRunnable(Runnable runnable) {
    runnable.run();
    // ...
}
```

```
public static void main(String[] args) {
    Runnable runnable = /* ... */;
    runnable.run();
    // ...
}
```



Пример с Runnable

Попробуем применить Runnable – реализуем класс Timer, который позволяет замерить время выполнения любой функции:

```
class Timer {  
    public void measureTime(Runnable runnable) {  
        long start_time = System.nanoTime();  
        runnable.run();  
        timeNanoseconds += System.nanoTime() - start_time;  
    }  
  
    public void reset() { timeNanoseconds = 0; }  
  
    public long getTimeMilliseconds() { return timeNanoseconds / 1_000_000; }  
  
    private long timeNanoseconds = 0;  
}
```

Теперь осталось понять как передать сюда саму функцию

Функция внутри класса

Теперь осталось понять как передать в Timer саму функцию

Давайте объявим функцию в классе, реализовав интерфейс, как мы умеем и знаем:

```
class StupidName implements Runnable {  
    public void run() {  
        long sum = 0;  
        for (int i = 1; i <= 1_000_000_000; ++i) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```

И теперь можем использовать наш таймер вот так:

```
public class Main {  
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        timer.measureTime(new StupidName());  
        System.out.println(timer.getTimeMilliseconds());  
    }  
}
```

Также, вспомним более магический способ – ссылка на метод:

```
class StupidName {  
    static public void stupidMethod() {  
        long sum = 0;  
        for (int i = 1; i <= 1_000_000_000; ++i) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Timer timer = new Timer();  
        timer.measureTime(StupidName::stupidMethod);  
        System.out.println(timer.getTimeMilliseconds());  
    }  
}
```

Магия в том, что ссылка на метод автоматически превращается в объект, соответствующий функциональному интерфейсу; не нужно руками ничего имплементировать и подгонять – язык сам всё сделает

Анонимная функция?

Ссылка на метод удобна, когда нужно использовать уже существующий метод, подходящий по сигнатуре под функциональный интерфейс

Но что, если функция нужна нам только в конкретном месте, и не хочется мусорить ей снаружи?

До появления Java 8 для этой цели использовались *анонимные классы*:

```
public static void main(String[] args) {
    Timer timer = new Timer();
    timer.measureTime(new Runnable() {
        @Override
        public void run() {
            long sum = 0;
            for (int i = 1; i <= 1_000_000_000; ++i) {
                sum += i;
            }
            System.out.println(sum);
        }
    });
    System.out.println(timer.getTimeMilliseconds());
}
```

Данный синтаксис позволяет объявить анонимный (безымянный) класс и создать объект прямо на месте в одной точке кода

Это позволяет объявить функцию специально для передачи в функ. интерфейс ровно там, где она нужна

У анонимных классов есть много ограничений, Подробности можно посмотреть [тут](#)

Есть недостаток, нужно писать кучу лишних строк с именем интерфейса и метода каждый раз

Лямбда-функция

Java 8 исправил проблему и добавил полноценные анонимные функции, именуемые лямбда-функциями:

```
public static void main(String[] args) {  
    Timer timer = new Timer();  
    timer.measureTime(() -> {  
        long sum = 0;  
        for (int i = 1; i <= 1_000_000_000; ++i) {  
            sum += i;  
        }  
        System.out.println(sum);  
    });  
    System.out.println(timer.getTimeMilliseconds());  
}
```

Конструкция `() -> {}` называется замыканием или лямбда-выражением и позволяет очень сжато объявить безымянную функцию

В круглых скобках указываются аргументы, но обо всём чуть попозже



Функциональные интерфейсы в библиотеке Java

Функциональные интерфейсы (и ссылки на методы + лямбда выражения) появились в Java 8

Интерфейсы для функций (например, `Comparator`) существовали в языке давно, но для использования требовали руками реализовать себя, хотя бы через анонимный класс

Старые интерфейсы без проблем заработали в Java 8, но также был добавлен целый отдельный пакет, в котором объявлено множество различных потенциально полезных функциональных интерфейсов

Давайте рассмотрим содержимое этого пакета `java.util.function`, разделив интерфейсы на группы

Групп будет пять: `Consumer`'ы, `Supplier`'ы, `Predicate`'ы, `Function`'ы, `Operator`'ы

Consumer'ы

Consumer – принимает значение, ничего не возвращает:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T var1);
}
```

Поскольку, дженерики не позволяют интерфейсу принимать примитивные типы, были созданы вариации:
IntConsumer, LongConsumer, DoubleConsumer


Также, есть вариант, принимающий два аргумента – BiConsumer:

```
public interface BiConsumer<T, U> {
    void accept(T var1, U var2);
}
```

И его вариации, принимающие вторым параметром примитивный тип:

```
public interface ObjDoubleConsumer<T> {
    void accept(T var1, double var2);
}
```

+ ObjIntConsumer и ObjLongConsumer



Supplier'ы

Supplier – ничего не принимает, возвращает значение:

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Есть вариации с примитивным типом:
BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier



Predicate'ы

Predicate – принимает значение, возвращает boolean:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T var1);
}
```

Есть специализации с примитивным принимаемым типом:
IntPredicate, LongPredicate, DoublePredicate

Ну и так же, BiPredicate – принимает два значения:

```
public interface BiPredicate<T, U> {
    boolean test(T var1, U var2);
}
```



Function'ы

Function – принимает значение, возвращает значение:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);
}
```

Специализаций для примитивных типов тут намного больше, например:

```
DoubleFunction: double -> T
LongToIntFunction: long -> int
ToIntFunction: T -> int
```

И, опять же, BiFunction – принимает два значения:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T var1, U var2);
}
```

И для него есть версии с примитивными возвращаемыми типами: ToIntBiFunction, ...



Operator'ы

Operator – частный случай функции, в котором совпадает принимаемый и возвращаемый тип:

```
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> { }
```

метод apply унаследован от Function

А также с двумя аргументами – BinaryOperator:

```
public interface BinaryOperator<T> extends BiFunction<T, T, T> {}
```

Ну и их версии для примитивных типов int, long и double:
IntUnaryOperator, LongBinaryOperator...

Весь набор можно посмотреть [тут](#)

Опять же, если какого-то интерфейса не хватает, его можно объявить самостоятельно



Инстанцирование функционального интерфейса

Есть три способа инстанцировать функциональный интерфейс

Первый способ – объявить именованный или анонимный класс, реализующий интерфейс:

```
class IntSquare implements IntUnaryOperator {  
    @Override  
    public int applyAsInt(int operand) {  
        return operand * operand;  
    }  
}
```

Так жили до Java 8, это громоздко и излишне, если класс не нужно переиспользовать

Ничего интересного тут нет, понятие функционального интерфейса по сути не используется

Лямбда-выражения

Второй способ – лямбда-выражения:

```
IntUnaryOperator square = (int x) -> {  
    return x * x;  
};
```

Компилятор умеет сам понять, в какой интерфейс присваивается лямбда, поэтому запись можно упростить:

```
IntUnaryOperator square = x -> x * x;
```

- Тип аргументов выводится автоматически
- Если аргумент один, то круглые скобки можно не писать
- Если выражение в теле функции одно, то можно не писать фигурные скобки и даже `return`

```
IntConsumer print = x -> System.out.print(x);
```

Захват переменных в лямбде

Так как лямбда может быть объявлена в теле функции, возникает вопрос: какие переменные доступны внутри лямбда-функции?

```
class Test {  
    private int counter;  
  
    public void test() {  
        IntUnaryOperator square = x -> x * x;  
  
        IntSupplier sequence = () -> counter++;  
  
        int bonus = 10;  
        IntUnaryOperator bonusAdder = (x) -> x + bonus;  
    }  
}
```

Очевидно, доступны параметры самой лямбды

Доступны поля класса, в котором объявили лямбду

И даже переменные из тела функции, но только если они по смыслу `final`, то есть присваиваются ровно один раз и дальше неизменны

Если хочется передать в лямбду изменяемую локальную переменную, то можно применить хак:
Хранить переменную в одноместном массиве:

```
int[] local_counter = new int[] {0};  
IntSupplier sequence = () -> local_counter[0]++;
```



Ссылка на метод

Третий способ – взятие ссылки на метод при помощи оператора ::

Можно взять ссылку на статический метод класса:

```
ToIntFunction<String> intParser = Integer::parseInt;
```

Или можно взять ссылку на нестатический метод конкретного объекта:

```
Consumer<Object> printer = System.out::println;
```

Ссылку на нестатический метод можно взять и просто от класса, тогда первым аргументом будет приниматься сам объект, у которого нужно вызвать метод:

```
Function<Object, String> objectToString = Object::toString;
```

Ссылку можно взять даже на конструктор, через слово **new**:

```
IntFunction<String[]> arrayCreator = String[]::new;
```



Полезные фишки

У стандартных функциональных интерфейсов есть различные полезные методы помимо главного (которые, очевидно, статические или дефолтные)

Например, у предикатов есть метод `negate()`, позволяющий инвертировать результат предиката:

```
IntPredicate isOdd = x -> x % 2 != 0;  
IntPredicate isEven = isOdd.negate();
```

Ещё, есть метод `and`, производящий конъюнкцию двух предикатов на одном аргументе:

```
IntPredicate p1 = /*...*/, p2 = /*...*/;  
IntPredicate p3 = p1.and(p2);
```

Аналогично, есть метод `or` для дизъюнкции



Полезные фишки

Метод `andThen` позволяет объединить два `Consumer`'а в один общий, который будет вызывать их по очереди:

```
Consumer<Object> printer = System.out::println;

List<Object> objects = new ArrayList<>();
Consumer<Object> collector = objects::add;

Consumer<Object> combinedConsumer = printer.andThen(collector);
```

Итоговый `Consumer` при вызове сначала напечатает свой аргумент на экран, а потом добавит его в массив

Полезные фишки

Function умеет делать композицию двух функций:

```
DoubleUnaryOperator square = x -> x * x;  
DoubleUnaryOperator sin = Math::sin;
```

```
DoubleUnaryOperator composition1 = sin.andThen(square);  
DoubleUnaryOperator composition2 = sin.compose(square);
```

→ $\sin^2(x)$

→ $\sin(x^2)$

Методы `andThen` и `compose` отличаются только порядком применения функций



Полезные фишки

Comparator имеет удобный метод для создания кастомных компараторов:

Вместо того, чтоб реализовывать сравнение по модулю вручную:

```
Comparator<Double> absoluteValueComparator =  
    (a, b) -> Double.compare(Math.abs(a), Math.abs(b));
```

Можно использовать статический метод `comparing`, который сначала применяет к элементам оператор из первого аргумента, а потом сравнивает их компаратором из второго:

```
Comparator<Double> absoluteValueComparator2 =  
    Comparator.comparing(Math::abs, Double::compare);
```

А можно вообще использовать `comparingDouble`, чтобы не передавать второй аргумент:

```
Comparator<Double> absoluteValueComparator3 =  
    Comparator.comparingDouble(Math::abs);
```