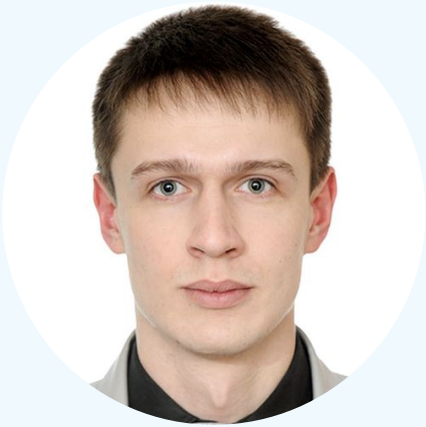
нетология

Лямбда-выражения и функциональные интерфейсы

Григорий
Вахмистров



Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. Лямбда-выражения
2. Функциональный интерфейс
3. Ссылки на метод
4. Итоги
5. Домашнее задание

Лямбда-выражения

Определение

Лямбда-выражение представляет собой набор инструкций, которые можно:

- сохранить в ссылочную переменную
- передать в любой метод в качестве аргумента
- вызвать в любой момент
- исполнить один или несколько раз

Введение

Лямбда-выражение в Java — упрощённая запись анонимного класса, реализующего функциональный интерфейс.

```
@FunctionalInterface
interface ISum {
   public int sum(int a, int b);
}
```

```
class Calc implements ISum {
   @Override
   public int sum(int a, int b) {
       return a + b;
   }
}
```

```
Calc calc = new Calc();
int sum = calc.sum(1, 2);
```

Введение

Лямбда-выражение в Java — упрощённая запись анонимного класса, реализующего функциональный интерфейс.

```
@FunctionalInterface
interface ISum {
    public int sum(int a, int b);
}
```

```
class Calc implements ISum {
    @Override
    public int sum(int a, int b) {
        return a + b;
    }
}
```

```
Calc calc = new Calc();
int sum = calc.sum(1, 2);
```

```
ISum summer = new ISum() {
    @Override
    public int sum(int a, int b) {
        return a + b;
    }
};

int sum = summer.sum(1, 2);
```

Введение

Лямбда-выражение в Java — упрощённая запись анонимного класса, реализующего функциональный интерфейс.

```
@FunctionalInterface
interface ISum {
   public int sum(int a, int b);
}
```

```
ISum summer = new ISum() {
   @Override
   public int sum(int a, int b) {
       return a + b;
   }
};

int sum = summer.sum(1, 2);
```

```
ISum summer = (a, b) -> a + b;
int sum = summer.sum(1, 2);
```

Синтаксис

Основу лямбда-выражения составляет **лямбда-оператор**, который обозначается символом «**->**». Этот оператор разделяет лямбда-выражение на две части:

- левая часть - параметры выражения;
- правая - тело лямбда-выражения, где выполняется действие.

Синтаксис:

```
(параметры) -> {тело метода}
```

Структура

- Имеют от 0 и более входных параметров.
- Параметры указываются либо явно, либо могут быть получены из контекста.
  *Например **(int a)** можно записать как **(a)***
- Параметры заключаются в круглые скобки и разделяются запятыми.
  *Например **(a, b)** или **(int a, int b)** или **(String a, int b, float c)***
- Без параметров необходимо использовать пустые круглые скобки.
  *Например **() -> 42***
- Для одного параметра без указания типа скобки можно опустить.
  *Например **a -> return a*a***
- Тело может содержать от 0 и более операторов.
- Если тело состоит из одного оператора, его можно не заключать в фигурные скобки, а возвращаемое значение можно указывать без ключевого слова **return**.

Примеры лямбда-выражений:

```
(int a, int b) -> {  return a + b; }
() -> System.out.println("Hello World");
(String s) -> { System.out.println(s); }
() -> 42
() -> { return 3.1415 }
```

Блоки кода в лямбда-выражениях

Существуют два типа лямбда-выражений:

- однострочное выражение;
- блок кода.

Пример однострочного выражения:

```
s -> System.out.println(s);
```

Блочные выражения обрамляются фигурными скобками. Здесь можно использовать вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д.

```
s -> {
    if (s.isEmpty()) {
        s = "Hello World";
    }
    System.out.println(s);
};
```

Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор **return**.

```
s -> {
    boolean b = s.isEmpty();
    return b;
}
```

Отложенное выполнение

Одним из ключевых моментов в использовании лямбда-выражений является отложенное выполнение. Иными словами, определенное в одном месте программы лямбда-выражение можем вызвать при необходимости неопределенное количество раз в различных частях программы.

Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

- Выполнение кода в отдельном потоке
- Выполнение одного и того же кода несколько раз
- Выполнение кода в результате какого-то события
- Выполнение кода только в том случае, когда он действительно необходим

Пример

Лямбда-выражения были включены в Java с выходом JDK 8, что позволило писать быстрее более лаконичный код.

Давайте убедимся в этом на примере:

Отсортируем массив строк.

```
List<String> cities = Arrays.asList("Мадрид", "Париж", "Москва", "Токио");
Collections.sort(cities, new Comparator<String>() {
   @Override
   public int compare(String a, String b) {
       return b.compareTo(a);
   }
});
```

> Статический метод **sort()** принимает список **cities** и анонимный экземпляр компаратора **Comparator**, который используется для сортировки списка.

Используем лямбда-выражение

Лямбда-выражения предоставляет гораздо более короткий синтаксис для создания анонимных объектов. Заменим создание экземпляра анонимного класса **Comparator** лямбда-выражением:

```
Collections.sort(names, (String a, String b) -> {
   return b.compareTo(a);
});
```

Для однострочных методов скобки **{}** и ключевое слово **return** опускаются:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

Компилятору известны типы параметров, поэтому их можно тоже опустить:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Давайте сравним

Сравним две реализации сортировки.

Первый вариант без использования лямбда-выражений:

```
List<String> cities = Arrays.asList("Мадрид", "Париж", "Москва", "Токио");
Collections.sort(cities, new Comparator<String>() {
   @Override
   public int compare(String a, String b) {
       return b.compareTo(a);
   }
});
```

Второй вариант с использованием лямбда-выражения:

```
List<String> cities = Arrays.asList("Мадрид", "Париж", "Москва", "Токио");
Collections.sort(cities, (a, b) -> b.compareTo(a));
```

> Выглядит непривычно, необходимо время, чтобы привыкнуть к такому синтаксису. Но при этом мы сэкономили 5 строчек кода при реализации такой простой операции.

Как это работает?

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе*.

Чтобы объявить и использовать лямбда-выражение, необходимо пройти ряд этапов:

1) Определение ссылки на функциональный интерфейс:

```
Comparator<String> comparator;
```

2) Создание лямбда-выражения:

```
comparator = (a, b) -> b.compareTo(a);
```

3) Использование лямбда-выражения в виде вызова метода интерфейса:

```
Collections.sort(cities, comparator);
```

* более подробно о функциональных интерфейсах вы узнаете в следующих слайдах

Обратите внимание

- Параметры лямбда-выражения соответствуют параметрам единственного метода **compare()** интерфейса **Comparator**, а результат соответствует возвращаемому результату метода интерфейса.

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T var1, T var2);
}
```

- В качестве тела метода **compare()** интерфейса **Comparator** нами реализован вызов у второго параметра **b** типа **String** метода **compareTo()**, который сравнивает объекты **a** и **b** и возвращает **int**:

```
b.compareTo(a)
```

* более подробно о функциональных интерфейсах вы узнаете в следующих слайдах

Лямбда-выражение как аргумент

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

Рассмотрим небольшой пример:

Определим лямбда-выражение в качестве переменной и передадим ее как аргумент функции**.*

```
public static void main(String[] args) {
    List<String> cities = Arrays.asList("Мадрид", "Париж", "Москва", "Токио");

    Comparator<String> comparator1 = (a, b) -> b.compareTo(a);
    Comparator<String> comparator2 = (a, b) -> a.compareTo(b);

    sortList(cities, comparator2);
}

public static <T> void sortList(List<T> list, Comparator<T> comparator) {
    list.sort(comparator);
    list.forEach(System.out::println);
}
```

*переменная - это поименованная, либо адресуемая иным способом область памяти, адрес которой можно использовать для осуществления доступа к данным

**аргумент функции - это переменная, которая передается методу при его вызове

Контрольные вопросы

- Что такое лямбда-выражение?
- Как выглядит лямбда-оператор?
- Сколько может быть аргументов у лямбда-выражения?
- В чем отличие между однострочным и блочным выражением?
- Что Вы понимаете под отложенным выполнением лямбда-выражения?

Функциональный интерфейс

Функциональный интерфейс

Функциональный интерфейс в Java – это интерфейс, который содержит только один абстрактный метод. Основное назначение – использование в лямбда выражениях и ссылках на методы.

Пример функционального интерфейса:

```
@FunctionalInterface
interface MyFunctionalInterface<T> {
   boolean test(T value);
}
```

Функциональный интерфейс **MyFunctionalInterface** имеет всего один метод без реализации **test(),** который принимает один аргумент типа **T**. Результатом выполнения метода **test()** является **boolean**.

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

Встроенные функциональные интерфейсы

В JDK 8 было добавлено несколько встроенных функциональных интерфейсов, широко применяемых в Stream API.

Рассмотрим основные из этих интерфейсов:

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

Predicate

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);
}
```

Пример:

```
Predicate<Integer> isPositive = x -> x >= 0;
System.out.println(isPositive.test(5)); // true
```

Function

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);
}
```

Например:

```
Function<Integer, String> convert = x -> x + " долларов";
System.out.println(convert.apply(5)); // 5 долларов
```

Consumer

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

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

Например:

```
Consumer<Integer> printer = x -> System.out.printf("%d долларов \n", x);
printer.accept(600); // 600 долларов
```

Supplier

Supplier<T> не принимает никаких аргументов, но возвращает объект типа **T**:

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

Например:

```
Supplier<String> stringFactory = () -> "new";
String str = stringFactory.get();
System.out.println(str); // new
```

UnaryOperator

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);
}
```

Например:

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

BinaryOperator

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

```
public interface BinaryOperator<T> {
   T apply(T t1, T t2);
}
```

Например:

```
BinaryOperator<Integer> multiply = (x, y) -> x * y;
System.out.println(multiply.apply(3, 5)); // 15
```

Свой функциональный интерфейс

Если ни один из встроенных интерфейсов не подходит для решения поставленной задачи, то можно создать собственный функциональный интерфейс.

Зачастую, это используется при создании так называемых **колбэков** - объектов, которые определяются в классе **А** и передаются в качестве аргумента в класс **Б**.

```
@FunctionalInterface
public interface OnDoneListener<T> {
   void onDone(T v);
}
```

> Если необходимо вернуть результат работы класса **Б** в класс **А**, то у переданного ему **колбэка** необходимо вызвать метод функционального интерфейса.

Экземпляр функционального интерфейса

Создадим экземпляр определенного ранее функционального интерфейса **OnDoneListener**. Классом **A** для нас станет класс **Main**.

```
public class Main {
    public Main() {
        OnDoneListener<String> listener = v -> System.out.println(v);
        Test test = new Test(listener);
        test.execute();
    }

    public static void main(String[] args) {
        new Main();
    }
}
```

> Объект **listener** класса **OnDoneListener** будет ждать аргумент **v** типа **String** и передавать его на печать в консоль. Объект **listener** передаем в качестве аргумента в конструктор класса **Test**, который будет являться классом **Б**.

Экземпляр функционального интерфейса

В классе **Test** определяем функцию **execute()**, в которой будет содержаться длительная операция, по завершении которой будет вызван метод **onDone()** колбэка.

```
public class Test {
    private OnDoneListener<String> callback;

    public Test(OnDoneListener<String> callback) {
        this.callback = callback;
    }

    public void execute() {
        // какая-либо длительная операция
        callback.onDone("complete");
    }
}
```

> После выполнения длительной операции в **execute()**, будет вызван метод **onDone()** и в качестве аргумента передана строка, которая будет выведена на экран.

Контрольные вопросы

- Что такое функциональный интерфейс?
- Для каких целей используется аннотация **@FunctionalInterface?**
- Перечислите основные встроенные функциональные интерфейсы.
- Как создать свой функциональный интерфейс?

Ссылки на метод

Ссылки на метод

В Java 8 добавилась новая функциональность, названная **ссылкой на метод**.

Если лямбда-выражение используется для вызова метода **функционального интерфейса**, можно его заменить ссылкой на метод. Это компактная и простая форма лямбда-выражения.

Есть несколько типов ссылок на метод:

- ссылка на статический метод;
- ссылка на метод экземпляра класса;
- ссылка на конструктор.

Ссылка на статический метод

Можно ссылаться на статический метод класса.

Синтаксис:

```
ContainingClass::staticMethodName
```

Создадим статический метод в классе **Main** и сделаем на него ссылку при реализации лямбда-выражения:

```
public static void main(String[] args) {
    Consumer<String> sayable = Main::saySomething;
    sayable.accept("Hello, this is static method");
}

private static void saySomething(String s) {
    System.out.println(s);
}
```

> В роли функционального интерфейса выступает интерфейс **Consumer** с одним методом без реализации **accept()**. В роли статического метода выступает метод **saySomething(),** который выводит в консоль строку.

Ссылка на метод объекта

Можно ссылаться на метод экземпляра класса.

Синтаксис:

```
containingObject::instanceMethodName
```

Создадим метод в классе **Main** и сделаем на него ссылку при реализации лямбда-выражения:

```
public static void main(String[] args) {
   Main main = new Main();
   Consumer<String> sayable = main::saySomething;
   sayable.accept("Hello, this is non-static method");
}

private void saySomething(String s) {
   System.out.println(s);
}
```

> В роли функционального интерфейса выступает интерфейс **Consumer** с одним методом без реализации **accept()**. В роли метода выступает метод экземпляра класса **saySomething(),** который выводит в консоль строку.

Ссылка на конструктор класса

Можно ссылаться на конструктор класса.

Синтаксис:

```
ClassName::new
```

Определим функциональный интерфейс **Consumer** и класс **Main** с конструктором:

```
public Main(String string) {
    System.out.println(string);
}

public static void main(String[] args) {
    Consumer<String> sayable = Main:: new;
    sayable.accept("Hello, this is Main class");
}
```

> В роли функционального интерфейса выступает интерфейс **Consume** с одним методом без реализации **accept()**. В качестве класса выступает **Main**, в конструктор которого передается строка и выводится в консоль.

Итоги

Итоги

- Лямбда-выражения позволяют писать быстрее и делать код более ясным.
- Лямбда-выражения можно выделить в отдельную переменную и передать в качестве аргумента в функцию.
- Функциональный интерфейс - это интерфейс с одним абстрактным методом.
- Лямбда-выражение реализует метод, определенный в функциональном интерфейсе.
- Лямбда-выражение можно его заменить ссылкой на метод.
- Ссылки могут быть как на статические методы, так на методы объекта и на конструкторы объектов.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров