AoA
Занятие 4: «f(x) или присутствие ипостеров в джаве»

Инфохимия, 1 курс, зима

Презентацию подготов… На
Эту. Часть.

А что от нас хотят?

9. Элементы функционального программирования в синтаксисе Java. Функциональные интерфейсы, лямбда-выражения. Ссылки на методы.

PLAN УРОКА

- Что такое функ. Прога?
- FunctiONAL Interfaces
- Lambda-выражения
- Ссылки (в описании) на методы

Что такое функц. Прога?

Определение 7.1.1 ($\varepsilon - \delta$ **определение предела функции**) *Пусть* $f : E \to \mathbb{R}$ *и* x_0 *– предельная точка для* E. *Число* A *называется пределом функции* $f(x)$ *в точке* x_0, *если*

$$\forall \varepsilon > 0 \; \exists \delta = \delta(\varepsilon) > 0 : \forall x \in E : 0 < |x - x_0| < \delta \Rightarrow |f(x) - A| < \varepsilon.$$

Что такое функц. Прога?

Что такое функциональное программирование? Если в двух словах, то функциональное программирование — это программирование, в котором функции являются объектами, и их можно присваивать переменным, передавать в качестве аргументов другим функциям, возвращать в качестве результата от функций и т. п.

Функциональные интерфейсы

Функциональным считается интерфейс с одним не реализованным (абстрактным) методом

```java
@FunctionalInterface
public interface Converter<T, N> {
   N convert(T t);
}
```

```
@FunctionalInterface
Public interface Convert_HUMAN_TO_DOG<Human, Dog>{
Dog convert(Human sharikoff);
}
```

```java
@FunctionalInterface
public interface Converter<T, N> {

   N convert(T t);

   static <T> boolean isNotNull(T t){
       return t != null;
   }
}
```

```java
@FunctionalInterface
public interface Converter<T, N> {

    N convert(T t);

    static <T> boolean isNotNull(T t){
        return t != null;
    }

    default void writeToConsole(T t) {
        System.out.println("Текущий объект - " + t.toString());
    }
}
```

Default-метод – метод, реализованный в интерфейсе с ключевым словом default

```java
@FunctionalInterface
public interface Converter<T, N> {

    N convert(T t);

    static <T> boolean isNotNull(T t){
        return t != null;
    }

    default void writeToConsole(T t) {
        System.out.println("Текущий объект - " + t.toString());
    }

    boolean equals(Object obj);
}
```

Базовые функциАНАЛЬНЫЕ интерфейсы, написанные в самой джаве

Predicate

`Predicate` — функциональный интерфейс для проверки соблюдения некоторого условия. Если условие соблюдается, возвращает `true`, иначе — `false`:

```
1  @FunctionalInterface
2  public interface Predicate<T> {
3     boolean test(T t);
4  }
```

В качестве примера рассмотрим создание `Predicate`, который будет проверять на чётность числа типа `Integer`:

```
1  public static void main(String[] args) {
2     Predicate<Integer> isEvenNumber = x -> x % 2==0;
3
4     System.out.println(isEvenNumber.test(4));
5     System.out.println(isEvenNumber.test(3));
6  }
```

Вывод в консоль:

```
true
false
```

Consumer

`Consumer` (с англ. — “потребитель”) — функциональный интерфейс, который принимает в качестве входного аргумента объект типа T, совершает некоторые действия, но при этом ничего не возвращает:

```
1  @FunctionalInterface
2  public interface Consumer<T> {
3     void accept(T t);
4  }
```

В качестве примера рассмотрим `Consumer`, задача которого — выводить в консоль приветствие с переданным строковым аргументом:

```
1  public static void main(String[] args) {
2     Consumer<String> greetings = x -> System.out.println("Hello " + x + " !!!");
3     greetings.accept("Elena");
4  }
```

Вывод в консоль:

```
Hello Elena !!!
```

Function

`Function` — этот функциональный интерфейс принимает аргумент T и приводит его к объекту типа R, который и возвращается как результат:

```
1  @FunctionalInterface
2  public interface Function<T, R> {
3     R apply(T t);
4  }
```

В качестве примера возьмём `Function`, который конвертирует числа из формата строк (`String`) в формат чисел (`Integer`):

```
1  public static void main(String[] args) {
2     Function<String, Integer> valueConverter = x -> Integer.valueOf(x);
3     System.out.println(valueConverter.apply("678"));
4  }
```

Запустив, получим вывод в консоль:

```
678
```

P.S.: если в строку мы передадим не только числа, но и другие символы, вылетит exception — `NumberFormatException`.

Почему ArrayList(), а не обычный массив?

- Ограниченный размер. Нужно уже на этапе создания массива знать, сколько ячеек он должен содержать. Недооценишь нужное количество — места не хватит. Переоценишь — массив останется полупустым, и это еще полбеды. Ведь получается, ты еще и выделишь под него больший объем памяти, чем нужно.
- У массива нет методов для добавления элементов. Всегда приходится явно указывать индекс ячейки, куда нужно добавить элемент. Если нечаянно указать уже занятую ячейку с каким-то нужным значением, оно перезапишется.
- Нет методов для удаления элемента. Значение можно только “обнулить”.

```
ArrayList<Items> things = new ArrayList<>();

public void Buy(Items what){
//System.out.print(Scene.FirstWord(this.name) + " bought " + what.name + ".");
things.add(what);
}
public void Sell (Items what){
System.out.print(Scene.FirstWord(this.name) + " sold " + what.name + ".");
things.remove(what);
}
public void Own(){
System.out.println(Scene.FirstWord(this.name) + " owns:");
for(int i = 0; i < things.size(); i++){
if (i == things.size() - 1)
System.out.print(", " + things.get(i) + ".");
else
System.out.print(", " + things.get(i));
}
}
```

Supplier

`Supplier` (с англ. — поставщик) — функциональный интерфейс, который не принимает никаких аргументов, но возвращает некоторый объект типа T:

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

В качестве примера рассмотрим `Supplier`, который будет выдавать рандомные имена из списка:

```
public static void main(String[] args) {
   ArrayList<String> nameList = new ArrayList<>();
   nameList .add("Elena");
   nameList .add("John");
   nameList .add("Alex");
   nameList .add("Jim");
   nameList .add("Sara");

   Supplier<String> randomName = () -> {
       int value = (int)(Math.random() * nameList.size());
       return nameList.get(value);
   };

   System.out.println(randomName.get());
}
```

И если мы это запустим, то увидим в консоли случайные результаты из списка имен.

UnaryOperator

`UnaryOperator` — функциональный интерфейс, принимает в качестве параметра объект типа T, выполняет над ним некоторые операции и возвращает результат операций в виде объекта того же типа T:

```
1  @FunctionalInterface
2  public interface UnaryOperator<T> {
3     T apply(T t);
4  }
```

`UnaryOperator`, который своим методом `apply` возводит число в квадрат:

```
1  public static void main(String[] args) {
2     UnaryOperator<Integer> squareValue = x -> x * x;
3     System.out.println(squareValue.apply(9));
4  }
```

Вывод в консоль:

```
81
```

Мы рассмотрели пять функциональных интерфейсов. Это не все, что доступно нам начиная с Java 8 — это основные интерфейсы. Остальные из доступных — это их усложненные аналоги. Полный список можно посмотреть в официальной документации Oracle.

Ля…ля…ЛЯМБДА!
Выражения

Lambda-выражения – это анонимные функции (может и не 100% верное определение для Java, но зато привносит некоторую ясность). Проще говоря, это метод без объявления, т.е. без модификаторов доступа, возвращающие значение и имя.

Lambda-выражения в Java обычно имеют следующий синтаксис `(аргументы) -> (тело)`. Например:

```
1  (арг1, арг2...) -> { тело }
2
3  (тип1 арг1, тип2 арг2...) -> { тело }
```

Далее идет несколько примеров настоящих Lambda-выражений:

```
1  (int a, int b) -> {  return a + b; }
2
3  () -> System.out.println("Hello World");
4
5  (String s) -> { System.out.println(s); }
6
7  () -> 42
8
9  () -> { return 3.1415 };
```



```
public int Sum(int a, int b){
return a + b;
}
```

Особенности

- Lambda-выражения могут иметь от 0 и более входных параметров.
- Тип параметров можно указывать явно либо может быть получен из контекста. Например (`int a`) можно записать и так (`a`)
- Параметры заключаются в круглые скобки и разделяются запятыми. Например (`a, b`) или (`int a, int b`) или (`String a`, `int b`, `float c`)
- Если параметров нет, то нужно использовать пустые круглые скобки. Например `() -> 42`
- Когда параметр один, если тип не указывается явно, скобки можно опустить. Пример: `a -> return a*a`
- Тело Lambda-выражения может содержать от 0 и более выражений.
- Если тело состоит из одного оператора, его можно не заключать в фигурные скобки, а возвращаемое значение можно указывать без ключевого слова `return`.
- В противном случае фигурные скобки обязательны (блок кода), а в конце надо указывать возвращаемое значение с использованием ключевого слова `return` (в противном случае типом возвращаемого значения будет `void`).

Ссылки на методы

Ссылки на методы-это, по сути, сокращенные лямбда-выражения, используемые для вызова методов.

Они состоят из двух частей:

```
Class::method;
```

```
public class Main {

    public static void main(String... arguments) {
        Consumer<String> printer = System.out::println;
        printer.accept( t: "Hello World!");
    }
}
```

СПАСИБО ЗА ВНИМАНИЕ!!!