Хекслет

Лямбда-выражения в Java

Наставник: Теплинская Мария Георгиевна
Группа: java-167
Дата: 10.04.2022

Функциональный интерфейс

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

Метод является абстрактным, если у него отсутствует реализация по умолчанию.

Функцирональный интерфейс еще называют **SAM**-типом, где SAM означает Single Abstract Method.

Анонимные классы

Анонимные классы - это классы, что не имеют имени и их создание происходит в момент инициализации объекта.

```
DoubleNumber number = new DoubleNumber() {
    1 usage
    @Override
    public double getValue() {
        return 3.14;
    }
};
System.out.println(number.getValue());
```

Лямбда-выражения

Лямбда-выражение, по существу, является анонимной (т.е. безымянной) функцией. Но она не выполняется самостоятельно, а служит для реализации метода, определяемого в функциональном интерфейсе.

У этой функции нет имени, но есть список параметров, тело, возвращаемый тип и, возможно, список исключений, которые из нее могут быть выброшены.

Структура лямбда-выражения

```
DoubleNumber number = () -> 3.14;

IntFilter isEven = (n) -> (n % 2) == 0;

IntFilter inRange = (int n) -> n >= 2 && n < 4;

IntFilter isPositive = n -> n > 0;

Comparator<Developer> comparator =
        (Developer o1, Developer o2) -> o1.getAge().compareTo(o2.getAge());
```

Блочное лямбда-выражение

```
StringProcessor reverse = (str) -> {
    StringBuilder sb = new StringBuilder();
    for (int i = str.length() - 1; i >= 0; i--) {
        sb.append(str.charAt(i));
    }
    return sb.toString();
};
System.out.println(reverse.apply( s: "Hexlet"));
```

Захват переменных

В лямбда-выражении есть возможность использовать переменные, которые находятся вне этого выражения. Это может быть переменная экземпляра класса, статическая переменная или локальная переменная метода.

Если захватывается локальная переменная, то есть ограничение. Она должна быть действительно конечной. Это значит, что она не должна изменяться после инициализации нигде, ни до, ни после, ни внутри лямбда-выражения.

Параметризация поведения

Лямбда-выражения можно передавать в качестве аргументов функции.

Параметром функции должен выступать соответствующий функциональный интерфейс.

Таким образом мы имеем возможность передавать некую функцию в качестве параметра и применять разные функции внутри одного и того же метода. Такой подход называется параметризацией поведения.

Предопределенные ФИ

В Java есть предопределенные функциональные интерфейсы, которые лежат в java.util.function.

Самые распространенные:

Predicate - в нем находится единственный абстрактный метод test, который принимаете T и возвращает boolean.

Consumer - определяет метод apply, который принимает T и возвращает void.

Function - определяет метод apply, который принимает в себя объект класса T, а возвращает объект класса R.

Таблица функциональных интерфейсов

Функциональный интерфейс	Функциональный Дескриптор	Специфические классы
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, LongFunction<R>, DoubleToIntFunction, ToIntFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier

UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

Объединение предикатов

У предикатов есть 3 дополнительных метода: negate, and и or.

negate дает возможность получить отрицание предиката.

and и or логически склеивают предикаты.

```
Predicate<String> empty = s -> s.isEmpty();
Predicate<String> nonEmpty = empty.negate();

Predicate<String> containsA = s -> s.contains("a");
Predicate<String> nonEmptyWithA = nonEmpty.and(containsA);

Predicate<String> sLength8 = s -> s.length() == 8;
Predicate<String> hasAOrL8 = containsA.or(sLength8);
```

Объединение функций

Для объединения функций существуют дополнительные функции andThen и compose.

```
Function<Integer, Integer> add = i -> i + 1;
Function<Integer, Integer> mult = i -> i * 2;
Function<Integer, Integer> ops = add.andThen(mult);
System.out.println(ops.apply( t: 1));
Function<Integer, Integer> opsCompose = add.compose(mult);
System.out.println(opsCompose.apply( t: 1));
```

Ссылки на методы

Ссылки на методы - это более короткая версия лямбда-выражений, в теле которых используется только один метод. В некоторых случаях это удобнее, потому что это описывает “что делать”, не вдаваясь в подробности как это будет сделано.

Тем не менее основная идея ссылок на методы - удобочитаемость!

```
Predicate<String> empty = s -> s.isEmpty();

Predicate<String> emptyMethodRef = String::isEmpty;
```

Ссылки на методы

Есть три способа превратить лямбду в ссылку на метод

```
Function<String, Integer> parseFunc = Integer::parseInt;
System.out.println(parseFunc.apply( t: "123"));

Consumer<Integer> printNum = System.out::println;
printNum.accept(Integer.valueOf( i: 1));

Developer dev = new Developer( id: 1,  name: "Sasha",  age: 27,  salary: 85000);
Supplier<String> level = dev::getLevelBySalary;
System.out.println(level.get());
```

Ссылки на конструкторы

```
Supplier<Developer> devMethodRef = Developer::new;

Supplier<Developer> suppDev = () -> new Developer();
System.out.println(suppDev.get());

Function<String, Developer> devCreationFunc = Developer::new;
System.out.println(devCreationFunc.apply( t: "Evgeny"));
```

Полезные ссылки

- Java. Полное руководство - Герберт Шилдт (Глава “Лямбда-выражения”)
- Modern Java in Action (Passing Code with behavior Parameterization + Lambda Expressions).
  Русская версия: Современный язык Java
- Лекции Тагира Валеева: https://youtu.be/iuw30Oa9IYs?t=3177
- Мини-туториал от JetBrains: https://youtu.be/WmMavkXMXDg

Спасибо за внимание!