

Основы программирования на языке Java. Уровень 1.

Занятие 8

Степулёнок Денис Олегович — denis.stepulenok@oracle.com
Солодкая Анастасия Сергеевна — a.s.solodkaya@gmail.com

8 ноября 2016 года

Параметризация. Лямбда-выражения

- Динамическая типизация в Java
- Создание класса с Generic (параметризированным) полем
- Лямбда-выражения, как альтернатива анонимным классам функциональных интерфейсов
- Применение лямбда-выражений
- Интерфейс Comparator
- Многоуровневая сортировка объектов

Создание класса с Generic (параметризированным) полем I

Обобщённое программирование — подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания. В Java, начиная с версии J2SE 5.0, добавлены средства обобщённого программирования, синтаксически основанные на C++ — Generics.

Generics I

- Позволяют использовать различные типы данных
- Позволяют вводить ограничения на различные типы данных
- По сути - инструкции для компилятора. Только compile-time. JVM ничего не знает о них.

Объявление Generic классов и интерфейсов I

```
class A<T1, T2, ...> { // Класс  
}
```

```
interface I<T1, T2, ...> { // Интерфейс  
}
```

```
class A<T extend Number> { // Ограничения на параметр  
}
```

```
class A<T extends Appendable & Serializable> { // 2 ограничения  
}
```

Использование Generics I

Стек (stack — стопка; читается стэк) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (last in — first out, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.



Собственная реализация стека:

Использование Generics II

```
package generic;

// Стек
// LIFO: Last In First Out
// Generic (параметризованный класс)
// T - тип данных для стека
// При использовании будем указывать конкретный тип
class MyStack<T> {
    // Вершина стека
    private Element top = null;

    // Поместить значение на вершину стека
    void push(T value) {
        // Создаем новый элемент
        Element e = new Element();
        e.value = value;
        e.next = top; // Весь прошлый список прикрепляем
        // Сохраняем этот элемент как первый
        top = e;
    }

    // Забрать значение с вершины стека
    T pop() {
        // Берём значение с вершины стека
        T value = top.value;
```

Использование Generics III

```
// Перемещаем начало на следующий элемент
top = top.next;
return value; // Возвращаем
}
```

```
private class Element {
    T value;
    Element next;
}
}
```

Пример использования:

```
package generic;

public class Main {
    public static void main(String[] args) {
        MyStack<String> stack = new MyStack<>();
        stack.push("Hello");
        System.out.println(stack.pop());
    }
}
```


Лямбда-выражения, как альтернатива анонимным классам функциональных интерфейсов I

Лямбда-исчисление (λ -исчисление) — формальная система, разработанная американским математиком Алонзо Чёрчем, для формализации и анализа понятия вычислимости.

λ -исчисление реализовано Джоном Маккарти в языке LISP.

LISP (LISt Processing language, «язык обработки списков») — семейство языков программирования, программы и данные в которых представляются системами линейных списков символов.

Джон Маккарти занимался исследованиями в области искусственного интеллекта (ИИ) и созданный им язык до сих пор является одним из основных средств моделирования различных аспектов ИИ.

Лямбда-выражения в Java — «синтаксический сахар» (это не настоящие лямбда-выражения). Они просто напоминают лямбда-выражения (введены в Java 8).

Интерфейс для лямбда-выражения:

Лямбда-выражения, как альтернатива анонимным классам функциональных интерфейсов II

```
package lambda;
```

```
public interface Operation {  
    int apply(int a, int b);  
}
```

Применение лямбда-выражений:

```
package lambda;
```

```
public class Main {  
    public static void main(String[] args) {  
        int[] array = {2, 4, 5, 1};  
        System.out.println(forEach(array, new Operation() {  
            @Override  
            public int apply(int a, int b) {  
                return a + b;  
            }  
        }));  
        System.out.println("+ " + forEach(array, (a, b) -> a + b));  
        System.out.println("+ " + forEach(array, Main::sum));  
    }  
}
```

Лямбда-выражения, как альтернатива анонимным классам функциональных интерфейсов III

```
        System.out.println("* " + forEach(array, (a, b) -> a * b));
        System.out.println("min " + forEach(array, Math::min));
        System.out.println("max " + forEach(array, Math::max));
    }

    private static int sum(int a, int b) {
        return a + b;
    }

    private static int forEach(int[] array, Operation operation) {
        int result = array[0];
        for (int i = 1; i < array.length; i++) {
            result = operation.apply(result, array[i]);
        }
        return result;
    }
}
```

Интерфейс Comparator<T> I

Многоуровневая сортировка объектов:

```
import java.util.Arrays;
import java.util.Comparator;

public class ComparatorDemo {
    public static void main(String[] args) {
        Point[] points = {new Point(1, 2), new Point(4, 5), new Point(5, 3)};
        Arrays.sort(points, new Comparator<Point>() {
            @Override
            public int compare(Point p1, Point p2) {
                if (p1.x > p2.x) return +1;
                if (p1.x < p2.x) return -1;
                if (p1.y > p2.y) return +1;
                if (p1.y < p2.y) return -1;
                return 0;
            }
        });
    }

    private static class Point {
        private final int x, y;

        public Point(int x, int y) {
            this.x = x;
        }
    }
}
```

Интерфейс Comparator<T> II

```
        this.y = y;  
    }  
}
```