Основы программирования на языке Java. Уровень 1. Занятие 6

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

26 октября 2016 года

Программа (агенда) занятия 6

- Четыре кита ООП
- Инкапсуляция (модификаторы доступа)
- Хороший тон разработки на Java
- Геттеры и сеттеры
- Ключевое слово this
- Конструкторы
- Наследование в Java и его применение
- Класс Object
- Преимущества полиморфизма в ООП языках
- Переопределение и перегрузка
- Абстракция при построении архитектуры приложения

Объектно-ориентированное программирование

Способ программирования, в котором используются **объекты** и **классы**.
Абстрагирование — выделение набора важных с точки зрения решаемой задачи характеристик объекта, исключая из рассмотрения неважные.
Абстракция — набор всех важных характеристик объекта, реализуется в ООП-языках при помощи Класса.
Пример: решаем задачу с точками на плоскости, для каждой точки важны её координаты $(x; y)$.
Класс — создаваемый программистом новый тип данных, модель ещё не существующей сущности (объекта).

```
class Point { double x,y; }
```

Объект — конкретная «точка», экземпляр класса.

```
Point A = new Point(2.0, 1.5), B = new Point(-2, 1);
```

Прототип — объект-образец, по образу и подобию которого создаются другие объекты путём копирования и изменения различных свойств.

Принципы ООП

Инкапсуляция — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя.
Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется **базовым**, **родительским** или **суперклассом**. Новый класс — **потомком**, **наследником**, **дочерним** или **производным** классом.
Полиморфизм — свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. При использовании термина «полиморфизм» в сообществе ООП подразумевается полиморфизм подтипов; а использование параметрического полиморфизма называют обобщённым программированием.

Класс «Точка» — не используя ООП

Если нам нужно хранить 100 точек, то мы можем создать 2 независимых массива:

```
double x[100], y[100];
```

И синхронно их использовать:

```
  x[0] = 1; y[0] = 2;
```

Или создать класс точка:

```
public class Point {
  public double x, y;
};
```

Чтобы потом создать массив из точек:

```
Point[] p = new Point[100];
```

И обращаться к нему:

```
  p[0].x = 1; p[0].y = 2;
  Point p1 = new Point();
  p1.x = 2;
```

Объявление и использование класса

Объявление класса «Point2D»:

```
class Point2D {
  public double x, y; // Поля (данные) класса
  // Метод класса «Передвинуть точку»
  public void move(double dx, double dy) { x += dx; y += dy; }
  // Повернуть точку относительно начала координат
  public void rotate(double angle) { ... }
};
```

Использование класса:

```
Point2D[] p = new Point2D[100]; // Массив
p[10].x = 10.1; p[10].y = 10.3;
p[0].move(1, 2);
```

Конструкторы и деструкторы

Конструктор (construct - создавать) — специальный метод класса, который выполняется после создания объекта и предназначен для инициализации полей класса некоторыми начальными значениями.
Несколько конструкторов должны отличаться типами передаваемых значений.

Абстракция I

В системе может быть 2 класса «Самолёт» не имеющие общих данных вообще:

```
package abstraction.flysupport;

// Самолёт с точки зрения пилота (полёта)
public class Airplane {
    double maxSpeed; // Максимальная скорость км/ч
    double weight; // Масса, кг.
}
```

```
package abstraction.sales;

// Самолёт с точки зрения продажи билетов
public class Airplane {
    Place[] business = new Place[10]; // Места в бизнес-классе
    Place[] econom = new Place[10]; // Места в эконом-классе
}
```

Абстракция II

```
package abstraction.sales;

// Место в самолёте
public class Place {
    int num; // Номер места
    double price; // Цена билета
}
```

Инкапсуляция I

Латынь: in capsula — размещение в оболочке, изоляция, закрытие чего-либо инородного с целью исключения влияния на окружающее.
Например: поместить радиоактивные отходы в капсулу, закрыть кожухом механизм, убрать мешающее в ящик или шкаф (чёрный ящик).

```
package encapsula;

// Квадрат: сторона side, прощадь area
// Они связаны: side^2 = area
interface Square {
    // Получить длину стороны
    double getSide();

    // Изменить длину стороны: value - новая длина
    void setSide(double value);

    // Получить площадь
    double getArea();

    // Изменяем площадь квадрата: value - новая площадь
    void setArea(double value);
}
```

Инкапсуляция II

```
package encapsula;

// Квадрат, который хранит площадь
public class SquareArea implements Square {
    double area;

    public void setSide(double value) {
        area = Math.pow(value, 2);
    }

    public void setArea(double value) {
        area = value;
    }

    public double getSide() {
        return Math.sqrt(area);
    }

    public double getArea() {
        return area;
    }
}
```

Инкапсуляция III

```
package encapsula;

// Квадрат, который хранит сторону
public class SquareSide implements Square {
    double side;

    public void setSide(double value) {
        side = value;
    }

    public void setArea(double value) {
        side = Math.sqrt(value);
    }

    public double getSide() {
        return side;
    }

    public double getArea() {
        return Math.pow(side, 2);
    }
}
```

Наследование I

Inheritance — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.
Наследование обеспечивает создание иерархических классификаций. Используя наследование, можно создать общий класс, определяющий характеристики, которые будут общими для множества родственных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять собственные уникальные характеристики.
Чтобы наследовать класс, надо добавить его имя в объявление другого класса с помощью **extends** — т.е. подкласс расширяет суперкласс, дополняя его собственными членами (полями и методами).
Для каждого подкласса можно указать только один суперкласс (множественное наследование не поддерживается).

Наследование II

```java
package inheritance;

// Наследование: класс-предок
class SuperClass {
    public SuperClass() { // Конструктор Суперкласса
        System.out.println("SuperClass Constructor");
    }
    SuperClass(int i) { // Другой конструктор суперкласса
        System.out.println("Another constructor");
    }
}
```

```java
package inheritance;

public class SubClass extends SuperClass {

    public SubClass() { // Конструктор Подкласса
        super(10); // Вызываем конструктор суперкласса
        System.out.println("SubClassConstructor");
    }

    public static void main(String[] args) {
        SubClass subClass = new SubClass();
    }
}
```

Наследование III

Пример наследования от одного класса и двух интерфейсов:

```
class A { }
interface I1 { }
interface I2 { }
class B extends A implements I1, I2 { }
```

Модификаторы доступа I

Модификаторы доступа:

- private — «для себя», только внутри того же класса
- protected — «и для наследников»
- Без модификатора: package local
- public — «для всех»

```
package n_public_private_protected;

public class A {
    // Только внутри класса A
    private int onlyInA = 2;

    // Без модификатора: Поле доступно в текущем пакете
    int packageLocal = 1;

    // Внутри класса A и всех наследников A + внутри пакета
    protected int withSubclasses = 3;

    // Доступно всем
    public int forAll = 4;
}
```

Модификаторы доступа II

```
package n_public_private_protected;

// Класс B - наследник класса A (в том же пакете)
public class B extends A {
    int bField = 100; // Поле класса B

    void methodB() {
        packageLocal = 10; // Видна везде внутри пакета
        withSubclasses = 20; // Видна так как B - потомок A
        forAll = 30; // Видна всем
        // onlyInA = 10; // Не видна, т.к. с модификатором private в A
    }
}
```

Модификаторы доступа III

```
package n_public_private_protected;

// Пользовательский класс внутри того же пакета
public class UserClassInSamePackage {

    public static void main(String[] args) {
        A a = new A();
        a.forAll = 10;
        a.withSubclasses = 10;
        System.out.println("a.withSubclasses = " + a.withSubclasses);
        a.packageLocal = 11;
    }
}
```

Модификаторы доступа IV

```java
package n_public_private_protected_user;

import n_public_private_protected.A;

// Класс B - наследник класса A (в том же пакете)
public class C extends A {
    int bField = 100; // Поле класса B

    void methodB() {
        // packageLocal = 10; // Недоступна, т.к. в другом пакете
        withSubclasses = 20; // Видна так как B - потомок A
        forAll = 30; // Видна всем
        // onlyInA = 10; // Не видна, т.к. с модификатором private в A
    }
}
```

ООП: Геометрические фигуры I

```
package shapes;

// Фигура (абстрактный класс)
abstract class Shape {
    String name; // Имя фигуры

    abstract public String show(); // Вывести информацию о фигуре

    abstract public double area(); // Площадь фигуры
}
```

ООП: Геометрические фигуры II

```
package shapes;

import static java.lang.Math.PI;
import static java.lang.Math.pow;

public class Circle extends Shape { // Круг
    private double r; // Радиус круга

    // Конструктор: r - радиус круга
    public Circle(double r) {
        this.r = r;
    }

    @Override
    public String show() { // Показываем информацию
        return "Circle: " + r + " " + area();
    }

    @Override
    public double area() {
        return PI * pow(r, 2);
    }
}
```

ООП: Геометрические фигуры III

```
package shapes;

import static java.lang.Math.*;

public class Square extends Shape { // Квадрат
    private final double side;

    public Square(double side) {
        this.side = side;
    }

    @Override
    public String show() {
        return String.format("Square: side %s area %s", side, area());
    }

    @Override
    public double area() {
        return pow(side, 2);
    }
}
```

ООП: Геометрические фигуры IV

```
package shapes;

public class Rectangle extends Shape { // Прямоугольник
    private final double a, b; // Стороны прямоугольника

    // Констуктор: a Первая сторона b Вторая сторона
    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public String show() {
        return "a = " + a + " b = " + b + " area = " + area();
    }

    @Override
    public double area() {
        return a * b;
    }
}
```

ООП: Геометрические фигуры V

```
package shapes;

import static java.lang.Math.pow;
import static java.lang.Math.sqrt;

public class Point2D extends Shape { // Класс: Точка
    static int count = 0;
    public int x, y;
    Coordinates coordinates;


    public Point2D(int x, int y) { // Конструктор
        this.x = x;
        this.y = y;
        count++;
    }

    public void move(int dx, int dy) {
        x += dx;
        y += dy;
    }

    @Override
    public String show() {
        return "(" + x + ";" + y + ")";
```

ООП: Геометрические фигуры VI

```
    }

    @Override
    public double area() {
        return 0.0;
    }

    // Расстояние до другой точки: sqrt((x - p.x)^2 + (x - p.x)^2)
    public double distance(Point2D p) {
        return sqrt(pow(x - p.x, 2) + pow(x - p.x, 2));
    }

    // Point2D.Coordinates
    public static class Coordinates {
        public int x, y;
    }
}
```

Exceptions: checked / unchecked

Исключения делятся на несколько классов, но все имеют общего предка — класс **Throwable**.

- Exception — контролируемые исключения (checked)
- Неконтролируемые исключения (unchecked)
  - RuntimeException
  - Errors —

Обработка исключений

```
try {
  throw new RuntimeException();
} catch (Exception e) {
}
```