# ООП

## Наследование и полиморфизм

*Что такое наследование?*

При наследовании вы можете создать новый класс не с нуля, а на основе существующего класса. При этом новый класс получает те же свойства и методы, которые были объявлены в старом классе, кроме приватных. Новый класс при этом называют дочерним классом, а старый – родительским. В дочернем классе обычно описываются новые свойства и методы, которые необходимо добавить к тому, что есть в родительском. Кроме того, в ряде случаев дочерний класс может заменять полученные от родительского класса методы своими вариантами.

*По умолчанию любой класс наследуется от класса Object.*

Чтобы сделать класс дочерним от другого класса, следует при его создании после его имени указать слово extends и имя родительского класса.

Например, возьмем класс:

```
public class Point {
        public int x;
        public int y;
}
```

И нам надо добавить цвет:

```
public class ColorPoint extends Point {
        private int color;
}
```

Таким образом, класс **ColorPoint** получает все свойства и методы, которые были в **Point**, и при этом добавляет к ним собственное свойство color.

Надо учитывать, что, если какие-то свойства или методы в предке имели модификатор private, дочерний класс напрямую видеть их не может. То есть его методы не могут обращаться к закрытым свойствам и методам, унаследованным от родителя.

Если вы хотите в классе закрыть свойства или методы от посторонних, но при этом дать доступ дочерним классам, следует использовать модификатор не private, а protected. Тогда ваш дочерний класс сможет работать с ними напрямую.

При этом, если необходимо запретить наследование от создаваемого класса, это можно сделать, указав при его создании модификатор final.

Если заголовок класса **Point** выглядел как:

```
public final class Point {

}
```

Попытка создать класс **ColorPoint** вызовет ошибку.

Последовательность вызовов конструкторов родительского и дочернего классов следующая: сначала вызывается конструктор родительского класса, а затем дочернего.

Если возникает необходимость вызвать конструктор родительского класса, он вызывается с помощью ключевого слова super и входных параметров.

```
super (входные параметры);
```

Если родительский класс не имеет конструктора без параметров, явный вызов родительского конструктора с помощью super является обязательным, причем вызов super должен стоять в дочернем конструкторе самым первым.

```
class A {
        private int y;
        A (int x) {
                this.y = x;
        }
}
class B extends A {
        private int x; B () {
                super(1);
                this.x = 0;
        }
}
```

Следует также отметить, что в Java отсутствует множественное наследование классов.

## Переопределение методов

Вы можете в дочернем классе создавать методы с таким же именем, типом возвращаемого значения и набором параметров, как и в родительском классе. Тем самым вы переопределяете метод. Этот процесс называется по-английски overriding.

Любой класс, создаваемый в вашей программе, даже если вы явно не использовали наследование будет иметь предка: класс Object. И даже пустой класс будет содержать методы, унаследованные от него. Один из самых полезных методов, наследуемых из этого класса toString. Данный метод автоматически вызывается, когда надо преобразовать ваш класс в строку. Например, если вы написали:

```
Point p = new Point();

System.out.println(p);
```

В этом случае будет вызван toString (из класса Object, который наследуют все классы) и распечатан результат его работы. Вы можете переопределить toString, чтобы при распечатке выводилась нужная вам информация.

```
public String toString() {
        return "А это мой класс!";
}
```

## Перегрузка методов

Вы можете в одном классе создавать методы с таким же именем, типом возвращаемого значения, но разным набором параметров. Тем самым вы перегружаете метод, этот процесс называется по-английски overload.

Например, если вы написали:

```
public class A {
        int average(int x, int y) {
                return (x + y)/2;
        }
        int average(int x, int y, int z) {
                return (x + y + z)/3;
        }
}
```

В этом случае метод **average** будет перегружен и при его вызове в зависимости от количества входных параметров JVM будет вызван соответствующий метод. На перегрузку влияет как количество, так и типы параметров. Если JVM не сможет понять, какой конкретно метод нужно вызвать, то программа не скомпилируется.

# Пакеты

Когда в программе классов становится много, то их имеет смысл сгруппировать. Для этого используются пакеты. Физически **пакет** – каталог на диске, в котором лежат исходные и скомпилированные файлы класса.

Если класс не лежит в корневом каталоге, а положен в пакет, то в самом начале исходного текста следует указать имя пакета при помощи ключевого слова package.

**package** *javaguru*;

Если это записано первой строкой исходника, значит он должен лежать не в корневом каталоге проекта, а в каталоге с именем *javaguru*.

Пакеты, как и каталоги могут вкладываться друга. В этом случае записывается вся цепочка пакетов, разделяемая точками.

**package** by.javaguru;

Это означает, что класс лежит в каталоге **javaguru**, который лежит в каталоге **by**.

Если в вашей программе используются классы из других пакетов, вы обязаны их подключить с помощью команды import.

import by.javaguru.MyClass;

Это означает, что мы будем использовать класс MyClass в пакете by.javaguru. Для множественного импортирования классов используется:

import by.javaguru.*;

Все классы из стандартной библиотеки java разложены по пакетам и большинство из них требуют подключения с помощью import. Исключением из этого правила являются классы, расположенные в пакете java.lang. Классы типа String, System, Math являются исключениями.

При объявлении переменных и методов класса может не быть никакой тип доступа, это значит, что доступ для этого класса и других классов в этом же пакете.

Доступ **protected** также дает доступ классам из этого же пакета.

Для импортирования статических констант используется следующее выражение:

import static java.lang.Math.abs;

# Абстрактные классы

Java позволяет создавать классы, предназначенные только для наследования, то есть классы, которые не позволяют создавать объекты, а только позволяют

их унаследовать при создании других классов. Такие классы называются абстрактными. Для того, чтобы указать, что класс будет абстрактным, в заголовке класса указывается ключевое слово abstract. Абстрактный класс может иметь и реализованные методы, и абстрактные методы, у которых отсутствует тело, и которые должны быть реализованы в дочерних классах. Такие методы должны быть описаны с ключевым словом abstract, а вместо тела метода ставится знак точки с запятой. Например, описание абстрактного класса геометрической фигуры:

```
abstract class Figure {
	protected float x;
	protected float y;
	float getX() {
		return x;
	}
	float getY() {
		return y;
	}
	void move(float dx, float dy) {
		x += dx;
		y += dy;
	}
	abstract float calcArea();
	abstract void getType();
}


public class Rectangle extends Figure {
	private float h;
	private float w;
	public Rectangle() {
		x = 1;
		y = 1;
		w = 1;
		h = 1;
	}
```

```
        public float calcArea() {
                return h*w;
        }
        public String getType() {
                return "Прямоугольник";
        }
}


public class MyFirst {
        public static void main(String[] args) {
                Rectangle r1 = new Rectangle();
                System.out.println(r1.getType());
                System.out.println("Area = " + r1.calcArea());
        }
}
```

В данном примере создан абстрактный класс **Figure**, в котором имеются абстрактные методы **calcArea** и **getType**, а также обычные свойства и методы **x, y, move, getX, getY**. Затем был создан дочерний класс **Rectangle**, описывающий прямоугольник. В нем были добавлены свойства **w** и **h**, а также перегружены абстрактные методы родительского класса.

Следует обратить внимание, что для создания обычного класса на основе абстрактного следует перегрузить все абстрактные методы. В противном случае компилятор выдаст ошибку, сообщающую, какие методы еще не перегружены.

## Интерфейсы

Существует еще один особый тип, который является еще более абстрактным, чем абстрактные классы. Это интерфейсы. Если абстрактный класс может хотя бы частично содержать реализацию класса (в нем могут присутствовать обычные свойства и методы), то интерфейс описывает только то, как класс будет взаимодействовать с другими классами. Поэтому в нем могут содержаться абстрактные методы, а также **default** и **private** методы. Описание обычных методов в интерфейсе не допустимо.

Общий вид описания интерфейса аналогичен описанию класса. В начале указывается ключевое слово **interface** (вместо class), затем имя интерфейса и

в фигурных скобках его содержимое. Например, интерфейс, описывающий лампочку:

```
interface Lamp {
        void on();
        void off();
        void printState();
        public default void light() {
                light1();
                light2();
                System.out.println("default method");
        }
        private void light1(){
                System.out.println("private method");
        }


        private static void light2(){
                System.out.println("private static method");
        }
}
```

Несмотря на то, что все методы являются абстрактными, ключевое слово **abstract** здесь может не использоваться. Оно используется по умолчанию, также следует помнить, что так как интерфейс описывает взаимодействие с другими классами, то методы по умолчанию имеют тип доступа не пакетный, а **public**.

Интерфейсы могут содержать не только методы, но и поля. Но при этом на поля налагаются существенные ограничения. В частности, свойства могут быть только статическими и константными. Даже если это не указано, свойства являются **static** и **final**. При объявлении свойства ему обязательно присвоить начальное значение. Например, если все типы лампочек имеют одинаковый срок годности, то в интерфейс можно добавить свойство следующего вида:

```
int lifeTime = 12;
```

Данное свойство будет константным, то есть изменить значение, заданное в интерфейсе, невозможно. И будет доступно через сам интерфейс, без создания дочерних классов и их объектов.

```
System.out.println(Lamp.lifeTime);
```

Это позволяет иногда использовать интерфейсы, как контейнеры для набора связанных между собой констант.

Для интерфейсов возможно использование наследования. Наследование производится аналогично наследованию между классами.

```
interface VolumeLamp extends Lamp {
    void setVolume();
}
```

То есть интерфейс **VolumeLamp** будет описывать те же свойства и методы, что и **Lamp**, но при этом добавит свой метод **setVolume**.

Все отличия интерфейсов от абстрактных классов, рассмотренные до этого, не являются существенными. Главным отличием интерфейсов от классов является возможность множественного наследования. Допустим, что кроме описанного выше интерфейса **Lamp**, имеется интерфейс **Color**:

```
interface Color {
    void setColor();
}
```

На основе этих интерфейсов можно создать интерфейс **ColorLamp**, следующим образом:

```
interface ColorLamp extends Lamp, Color {
}
```

То есть, чтобы итоговый интерфейс содержал все свойства и методы из нескольких родительских, ему следует указать имена этих интерфейсов через запятую. В данном примере интерфейс не имеет собственных свойств и методов, а только унаследованные, но собственные могут быть также добавлены обычным образом.

После создания интерфейса на его основе может быть создан класс. Создание класса из интерфейса аналогично созданию обычного дочернего класса, но вместо слова **extends** используется слово **implements**.

```
public class OrdLamp implements Lamp {
    protected boolean state = false;
    public void on() {
        this.state = true;
    }
    public void off(){
        this.state = false;
```

```
    }
    public void printState() {
        if (state) {
            System.out.println("Лампа включена!");
        } else {
            System.out.println("Лампа выключена!");
        }
    }
}
```

Следует обратить внимание, что при переопределении методов, описанных в интерфейсе, в классе обязательно явным образом указывать правильный тип доступа (в данном случае public), хотя в интерфейсе он использовался по умолчанию. При создании класса может одновременно применяться наследование от другого класса и интерфейсов:

```
public class ColorLamp extends OrdLamp implements Color {
    protected int color;
    public void setColor() {
        this.color = 1;
    }
}
```

## Внутренние классы

Если объекты класса могут быть использованы только внутри другого класса, иногда имеет смысл создавать сам класс внутри другого.

В Java существуют 4 типа вложенных (**nested**) классов:

- статические вложенные классы;
- внутренние классы;
- локальные классы;
- анонимные (безымянные) классы.

Как пример рассмотрим внутренний класс:

```
class ClassA {
```

```
        class ClassB {
                int x = 1;
        }
        void printData() {
                ClassB b = new ClassB();
                System.out.println("inner" + b.x);
        }
}
```

В данном примере создан класс **ClassA**, внутри которого создан класс **ClassB**. Другие классы не могут, минуя **ClassA** создавать объекты класса **ClassB**. Чтобы внешний класс мог создать объект внутреннего класса, он должен создать объект основного класса. Например:

```
ClassA a = new ClassA();
ClassA.ClassB b = a.new ClassB();
```

Следует обратить внимание, что оператор **new** вызывается как метод объекта основного класса.

Такая запись возможна, только если внутренний класс имеет тип доступа, открытый или пакетный. Если внутренний класс создан с ключевым словом **private**, создать объект внутреннего класса таким образом невозможно.

Другой класс может получить доступ к внутреннему классу с типом доступа **private**, только если внутренний класс был создан на основе интерфейса, и в основном классе есть метод, возвращающий объект внутреннего класса.

```
interface IntB {
        int getX();
}
class ClassA {
        private class ClassB implements IntB {
                int x=1;
                public int getX() {
                        return x;
                }
        }
        void printData() {
                ClassB b = new ClassB();
```

```
            System.out.println("inner " + b.x);
        }
        IntB getB() {
            return new ClassB();
        }
}
```

```
ClassA a = new ClassA();
IntB b = a.getB();
System.out.println(b.getX());
```

Как видно, внутренний класс является закрытым, но тем не менее имеется возможность через интерфейс получить доступ к его интерфейсным методам. Подобный подход использован в итераторах классов-коллекций.

## Анонимные классы

В случае, если объект данного класса создается исключительно в одном месте, можно не создавать отдельный внутренний класс, а сделать так называемый анонимный класс, который описывается прямо на месте создания объекта.

Анонимный класс на основе интерфейса или другого класса создается следующим образом:

```
Интерфейс имя_переменной = new Интерфейс(){
    Здесь код…
}
```

Таким образом, если рассматривать приведенный ранее пример, для итератора можно сделать анонимный класс. В этом случае метод интерфейс будет иметь вид:

```
interface IntB {
    int getX();
}
...
IntB intB = new IntB {
    int getX() {
```

```
            return 5;
        }
}
```

И специально объявлять класс, имплементирующий интерфейс **IntB**, уже нет необходимости. Анонимные классы достаточно широко используются в программах с графическим интерфейсом пользователя.