

## Модуль 2. Объектно-ориентированное программирование

### Тема 2.7. Полиморфизм

2 часа

#### Оглавление

Тема 2.7. Полиморфизм.....	2
2.7.1 Понятие полиморфизма .....	2
2.7.2 Защита от переопределения методов.....	7
2.7.3 Понятие абстрактного метода.....	8
2.7.4 Понятие интерфейса .....	9
Задание 2.7.1.....	10
Благодарности .....	10

# Тема 2.7. Полиморфизм

## 2.7.1 Понятие полиморфизма

**Полиморфизм** (от греч. – «имеющий много форм») – одна из трех базовых парадигм ООП. Представляет собой способность изменения унаследованного функционала и обеспечивает обращение к методам объектов разных классов, используя единое имя.

Полиморфизм позволяет абстрагироваться от типа конкретного объекта. Суть абстрагирования заключается в том, чтобы определять метод в том месте, где есть наиболее полная информация о том, как он должен работать. Полиморфизм реализуется путем переопределения (не путать с перегрузкой!) методов.

**Переопределение метода** (англ. *Method overriding*) – возможность языка программирования, позволяющая производному классу обеспечивать специфическую реализацию метода, уже реализованного в базовом классе. Реализация метода в производном классе переопределяет (заменяет) его реализацию в базовом классе, описывая метод с тем же названием, что и у метода базового класса. Также у метода производного класса должны быть те же параметры и тип возвращаемого результата, что и у метода базового класса (иначе это будет совсем новый метод, не имеющий отношения к полиморфизму). Версия метода, которая будет исполнена, определяется типом объекта, вызывающего его.



Java позволяет защитить методы от переопределения при помощи ключевого слова *final*. Также методы, объявленные как *private* или *static* не могут быть переопределены, поскольку это соответствует неявному использованию *final*.



Производный класс содержащий метод, переопределяющий метод базового класса, может помимо своего вызывать и метод базового класса при помощи ключевого слова *super*.

Рассмотрим иерархию классов следующего вида (рисунок 1).

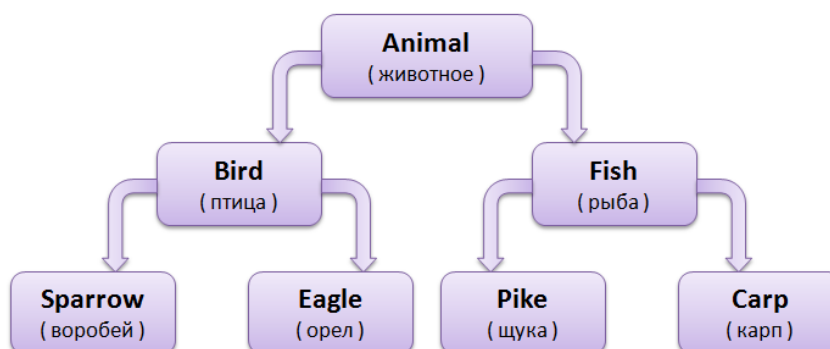


Рисунок 1 - Иерархия классов

Пусть каждый класс имеет специальный метод, позволяющий объекту называть имя класса, экземпляром которого он является. Такой функционал можно реализовать по-разному. Самый очевидный и понятный для нас способ - это реализовать в каждом классе метод со своим названием, чтобы мы и компилятор могли их различать. Зная тип объекта мы сможем вызывать соответствующий метод. На первый взгляд всё логично. Но такой способ содержит в себе ряд существенных недостатков:

1. схожие действия будут выполняться путем вызова разных методов и необходимо помнить какой метод в конкретном классе отвечает за эти действия;
2. при работе с совокупностью объектов разных классов необходимо написание кода, который будет определять тип объекта и вызывать соответствующий метод. При этом объем кода будет пропорционален количеству типов;
3. при изменении иерархии классов необходимо корректировать уже имеющийся код.

Применение полиморфизма решает все эти проблемы, он позволяет использовать единый интерфейс к независимо реализованным методам. Под единым интерфейсом понимается использование единого имени для реализации схожих действий.

```
package polimorph;

// класс Животное
class Animal {
    public String className() { return "Animal"; }
}
// класс Птица, потомок класса Животное
class Bird extends Animal {
    public String className() { return "Bird"; }
}
// класс Рыба, потомок класса Животное
class Fish extends Animal {
    public String className() { return "Fish"; }
}
// класс Воробей, потомок класса Птица
class Sparrow extends Bird {
    public String className() { return "Sparrow"; }
}
// класс Орел, потомок класса Птица
class Eagle extends Bird {
    public String className() { return "Eagle"; }
}
// класс Щука, потомок класса Рыба
class Pike extends Fish {
    public String className() { return "Pike"; }
}
// класс Карп, потомок класса Рыба
class Carp extends Fish {
    public String className() { return "Carp"; }
}

public class Math {
    ...
    public static final double PI = 3.14159265358979323846;
    ...
}

public class MyClass {
```

```
public static void main(String[] args) {
    Animal a = new Animal();
    Bird b = new Bird();
    Fish f = new Fish();
    Sparrow s = new Sparrow();
    Eagle e = new Eagle();
    Pike p = new Pike();
    Carp c = new Carp();
    // объявление массива типа Animal и инициализация
    // его элементов объектами производных классов
    Animal[] array = { a, b, f, s, e, p, c };
    // вывод типов элементов массива
    for (Animal anim : array)
        System.out.println(anim.className());
}
```

Приведенный пример показывает, что метод `className()` конкретизируется (переопределяется) в соответствии с положением класса-владельца метода в иерархии наследования классов.

Более того, мы вообще можем не знать объектами какого класса являются элементы массива. Но благодаря механизму позднего связывания (это понятие рассматривается ниже) будет вызываться реализация метода `className()`, соответствующая типу объекта. Фактический класс объекта определяется во время выполнения программы.

В предложенном примере изменим способ инициализации массива так, чтобы тип элемента определялся случайным образом. Для этого класс `MyClass` изменим следующим образом:

```
public class MyClass {
    static Animal randAnimal() {
        int k = (int) (Math.random() * 7);
        switch (k) {
            case 0: return new Animal();
            case 1: return new Bird();
            case 2: return new Fish();
            case 3: return new Sparrow();
            case 4: return new Eagle();
            case 5: return new Pike();
            case 6: return new Carp();
            default: return null;
        }
    }

    public static void main(String[] args) {

        Animal[] array = new Animal[10];
        // заполнение массива
        for (int i = 0; i < array.length; i++)
            array[i] = randAnimal();
        // вывод типов элементов массива
        for (Animal anim : array)
            System.out.println(anim.className());
    }
}
```

Используя ссылку на объект базового класса, можно использовать динамическое связывание для вызова полиморфных методов. Суть динамического связывания состоит в том, что решение на вызов переопределенного метода принимается во время выполнения, а не во время компиляции.

Явно это можно увидеть, изменив код класса MyClass следующим образом:

```
public class MyClass {  
  
    public static void main(String[] args) {  
        Animal a = new Animal(); // создание объекта типа Animal  
        Bird b = new Bird();      // создание объекта типа Bird  
        Eagle e = new Eagle();    // создание объекта типа Eagle  
        Animal temp; // объявление ссылки на объект типа Animal  
  
        temp = a;                // temp указывает на Animal-объект  
        // вызов Animal-версии метода className()  
        System.out.println(temp.className());  
        temp = b;                // temp указывает на Bird-объект  
        // вызов Bird-версии метода className()  
        System.out.println(temp.className());  
        temp = e;                // temp указывает на Eagle-объект  
        // вызов Eagle-версии метода className()  
        System.out.println(temp.className());  
    }  
}
```

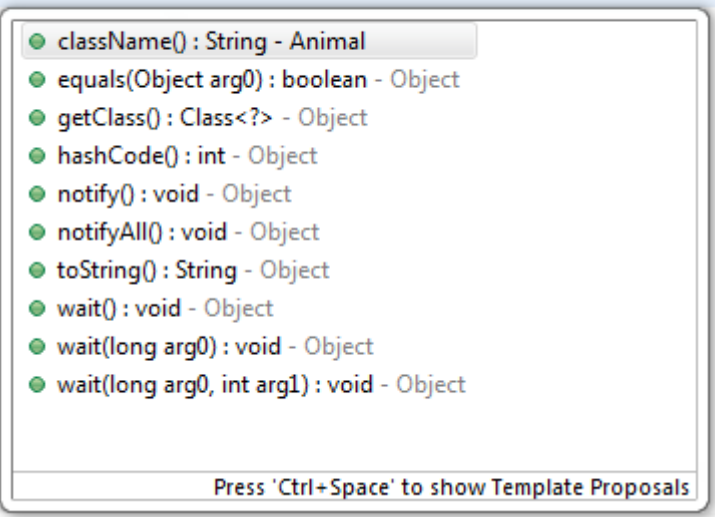
Полиморфизм позволяет при наследовании как полностью менять существующий функционал, так и расширять наследуемый. Чтобы обратиться к методу базового класса при переопределении необходимо использовать ключевое слово *super*. Модифицируем метод `className()` таким образом, чтобы он возвращал еще имя родительского класса. Например, описание класса `Bird` примет вид:


```
class Bird extends Animal {  
    public String className() {  
        return "Bird"+" -> "+super.className();  
    }  
}
```



Результат работы программы показывает, что в зависимости от положения объекта в иерархии классов (рисунок 1) метод `className()` сообщает различное количество своих предков (поколений). В классе на верхнем уровне иерархии (`Animal`) вызвать метод `className()` базового класса нельзя, поскольку его просто не существует.

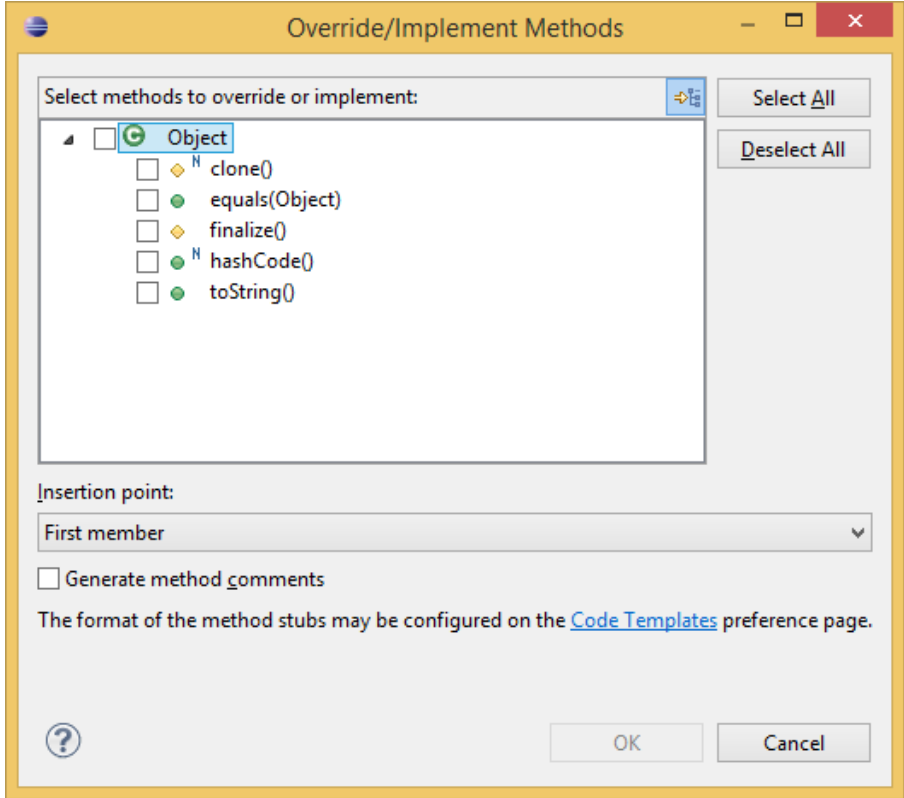


*По умолчанию все классы в Java являются потомками класса `Object`. Поэтому все описанные нами классы уже обладают атрибутами и методами класса `Object`.*

	<pre>Animal a; a.</pre>  <p>Определения класса приведенные ниже являются идентичными:</p> <pre>public class MyClass { ... } public class MyClass extends Object { ... }</pre>
--	---

	<p>Тип объекта можно узнать, не прибегая к написанию специального метода. Для этих целей в классе <i>Object</i> есть метод <i>getClass()</i>. Таким образом, узнать тип любого объекта можно, обратившись к нему так:</p> <pre>ob.getClass().getName();</pre>
--	---

 	<p>Возможности среды <i>Eclipse</i> позволяют в диалоговом режиме добавить переопределяемые методы в класс. Для этого необходимо находясь в теле класса выполнить команду «<i>Override/Implement Methods...</i>» из меню <i>Source</i>. В появившемся диалоговом окне отображаются доступные для переопределения в классе методы. Наиболее часто на практике переопределяются методы <i>equals(Object)</i> и <i>toString()</i>.</p>
---	---



В среде Android Studio аналогичные действия выполняются через меню *Code>Override Methods...*, которое вызывается сочетанием клавиш *Ctrl+O*

## 2.7.2 Защита от переопределения методов

Хотя переопределение методов — одно из наиболее мощных средств Java, бывают ситуации, когда его необходимо избежать. Чтобы запретить переопределение метода, в начале его объявления необходимо указать ключевое слово *final*.



Java позволяет защитить методы от переопределения при помощи ключевого слова *final*. Также методы, объявленные как *private* или *static* не могут быть переопределены, поскольку это соответствует неявному использованию *final*.

Применительно к нашему примеру рассмотрим ситуацию. Допустим требуется, чтобы потомки класса *Carp* не могли переопределить метод *className()*. Для этого в его объявление добавляем ключевое слово *final*. В этом случае попытка переопределить метод *className()* в производном классе *MirrorCarp* (зеркальный карп) приводит к ошибке «Cannot override the final method from *Carp*» (Невозможно переопределить финализированный метод класса *Carp*).

```
class Carp extends Fish {
    public final String className() {
        return "Carp" + " -> " + super.className();
    }
}

class MirrorCarp extends Carp {
    // попытка переопределить финализированный метод
    public String className() {
        return "Mirror Carp" + " -> " + super.className();
    }
}
```

### 2.7.3 Понятие абстрактного метода

Абстрактным называется метод, который не имеет реализации в классе. После круглых скобок, где перечислены его аргументы, следует не блок описания метода, а точка с запятой. То есть описание у абстрактного метода отсутствует. Перед именем метода указывается при этом ключевое слово *abstract*.

Абстрактный метод можно использовать только переопределив его в производном классе. Чтобы исключить возможность прямого использования абстрактного метода, в Java требуется, чтобы класс имеющий хотя бы один абстрактный метод был абстрактным. То есть наличие абстрактного метода автоматически делает класс абстрактным.

Покажем на примере, когда целесообразно использование абстрактных методов. Добавим дополнительный функционал в классы: объекты должны сообщать способ своего передвижения (метод `move()`). Мы точно знаем, что объекты класса `Fish` и его потомков плавают, а объекты класса `Bird` и его потомков - летают. Поскольку нет какого-то общего способа передвижения для всех животных, то мы в классе `Animal` не можем задать конкретный функционал в методе `move()`, но тем не менее он должен иметь тело (хотя бы пустое). В этом случае метод `move()` является претендентом на то, чтобы стать абстрактным.

```
// класс Животное (абстрактный)
abstract class Animal {
    public String className() { return "Animal"; }

    public abstract String move(); // абстрактный метод
}

// класс Птица, потомок класса Животное
class Bird extends Animal {
    public String className() { return "Bird"; }

    public String move() { return "Flying"; }
}

// класс Рыба, потомок класса Животное
class Fish extends Animal {
    public String className() { return "Fish"; }
}
```



```
    public String move() { return "Swimming"; }  
}
```

Тем более, не понятно, зачем надо создавать экземпляр животного неопределенного вида. Поэтому в рамках реальной задачи вряд ли потребуется создавать объекты класса `Animal`, а значит требование сделать класс абстрактным является логичным.

Класс-наследник абстрактного класса обязан переопределить все абстрактные методы базового класса.

## 2.7.4 Понятие интерфейса

Интерфейс - это конструкция языка программирования, подобная классу, в рамках которой могут описываться только абстрактные публичные (`abstract public`) методы и статические константные свойства (`final static`). Другими словами, интерфейс - это полностью абстрактный класс. И на него распространяются те же требования, что и на абстрактные классы.



*Поскольку все свойства интерфейса должны быть константными и статическими, а все методы общедоступными, то соответствующие модификаторы перед свойствами и методами разрешается не указывать.*

Классы могут *реализовывать* (*implements*) интерфейсы (получать от интерфейса список методов и описывать реализацию каждого из них). В отличие от наследования классов, возможна множественная реализация интерфейсов.

Перед описанием интерфейса указывается ключевое слово *interface*. Если класс реализует интерфейс, то после его имени указывается ключевое слово *implements* и далее через запятую перечисляются имена тех интерфейсов, методы которых будут полностью описаны в классе.

Реализуем интерфейс `Liveable` (способный жить), включив в него методы, соответствующие обязательным процессам жизнедеятельности любого животного, например, двигаться, питаться, дышать. Описание интерфейса `Liveable` и его реализация классами `Bird` и `Fish` имеют вид:

```
// интерфейс "Способный жить"  
interface Liveable  
{  
    boolean living = true;  
  
    String feed();           // чем питается  
    String move();          // как передвигается  
    String breathe();       // чем дышит  
}  
  
// класс Птица, реализует интерфейс "Способный жить"  
class Bird implements Liveable {  
    public String className() { return "Bird"; }  
}
```

```
    public String feed() { return "Insect"; }
    public String move() { return "Flying"; }
    public String breathe() { return "Free air"; }
}

// класс Рыба, реализует интерфейс "Способный жить"
class Fish implements Liveable {
    public String className() { return "Fish"; }

    public String feed() { return "Plankton"; }
    public String move() { return "Swimming"; }
    public String breathe() { return "Dissolved oxygen"; }
}
```

Часто всю открытую часть класса (общедоступные методы) определяют в интерфейсе. Тогда, взглянув на интерфейс, можно понять какие методы могут использоваться для взаимодействия с объектами данного класса. То есть интерфейсы полностью соответствуют принципам инкапсуляции и полиморфизма. В разных классах метод некоторого интерфейса может быть реализован по-разному, хотя и с одним и тем же именем.

Такая программа легко расширяема, поскольку можно добавлять новые возможности через наследование новых типов данных от общего базового класса. Методы манипулирующие интерфейсом базового класса не нуждаются в изменении в новых классах.

## Задание 2.7.1

1. Реализовать иерархию объектов «Геометрические фигуры».
2. Реализовать в каждом классе полиморфный метод.
3. Один из методов базового класса сделать абстрактным.
4. Переопределить метод toString() класса Object.
5. Реализовать интерфейс Moveable (способный передвигаться) с методами Сдвиг и Поворот. Реализовать методы интерфейса в классах, которые его реализуют.

## Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю ИТ ШКОЛЫ SAMSUNG Томилову Ивану Николаевичу.