

Занятие № 8.

Абстракция,
наследование,
полиморфизм

Наследование
Полиморфизм

Интерфейс
Абстрактный класс

Класс Object и его методы

Наследование

Наследование является неотъемлемой частью Java. При использовании наследования вы говорите: Этот новый класс похож на тот старый класс. В коде это пишется как `extends`, после которого указываете имя базового класса. Тем самым вы получаете доступ ко всем полям и методам базового класса. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем вы можете наследоваться от него и создать новый класс, который будет иметь свои уникальные характеристики. Главный наследуемый класс в Java называют суперклассом. Наследующий класс называют подклассом. Получается, что подкласс - это специализированная версия суперкласса, которая наследует все члены суперкласса и добавляет свои собственные уникальные элементы. К примеру, в Android есть класс `View` и подкласс `TextView`.

Чтобы наследовать класс, достаточно вставить имя наследуемого класса с использованием ключевого слова `extends`:

```
public class MainActivity extends Activity {  
  
}
```

В этом коде мы наследуемся от класса `Activity` и добавляем свой код, который будет отвечать за наше приложение.

Подкласс в свою очередь может быть суперклассом другого подкласса. Так например, упоминавший ранее класс `TextView` является суперклассом для `EditText`.

В производный класс можно добавлять новые методы.

Для каждого создаваемого подкласса можно указывать только один суперкласс. При этом никакой класс не может быть собственным суперклассом.

Хотя подкласс включает в себя все члены своего суперкласса, он не может получить доступ к тем членам суперкласса, которые объявлены как `private`.

Помните, мы создавали класс `Box` для коробки кота. Давайте наследуемся от этого класса и создадим новый класс, который будет иметь не только размеры коробки, но и вес.

В том же файле `Box.java` после последней закрывающей скобки добавьте новый код:

```
class HeavyBox extends Box {  
    int weight; // вес коробки  
  
    // конструктор  
    HeavyBox(int w, int h, int d, int m) {  
        width = w;  
        height = h;  
        depth = d;  
        weight = m; // масса  
    }  
}
```

Возвращаемся в главную активность и пишем код:

```
HeavyBox box = new HeavyBox(15, 10, 20, 5);  
  
int vol = box.getVolume();  
  
textInfo.setText("Объём коробки: " + vol + " Вес коробки: " + box.weight);
```

Обратите внимание, что мы вызываем метод `getVolume()`, который не прописывали в классе `HeavyBox`. Однако мы можем его использовать, так как мы наследовались от класса `Box` и нам доступны все открытые поля и методы. Заодно мы вычисляем вес коробки с помощью новой переменной, которую добавили в подкласс.

Теперь у нас появилась возможность складывать в коробку различные вещи. В хозяйстве всё пригодится.

При желании вы можете создать множество разных классов на основе одного суперкласса. Например, мы можем создать цветную коробку.

```
class ColorBox extends Box {  
    int color; // цвет коробки  
  
    // конструктор  
    ColorBox(int w, int h, int d, int c) {  
        width = w;  
        height = h;  
        depth = d;  
        color = c; // цвет  
    }  
}
```

Полиморфизм

В ООП применяется понятие полиморфизм .

Полиморфизм в ООП означает возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Понятие полиморфизма, в свою очередь, опирается на два других понятия: совместное использование (`overloading`) и переопределение (`overriding`).

Рассмотрим их подробнее.

Термин `overloading` можно перевести как перегрузку, доопределение, совместное использование. Мы будем использовать перевод совместное использование . Под совместным использованием понимают использование одноименных методов с различным набором параметров. При вызове метода в зависимости от набора параметров выбирается требуемый метод. При этом одноименные методы могут быть как в составе одного класса, так и в разных классах, связанных отношением наследования. Это статический полиморфизм методов классов. Примеры совместного использования мы уже встречали ранее. Приведем еще несколько примеров.

```
class X {  
    int f() {  
        . . .  
    }  
  
    void f(int k) {  
        . . .  
    }  
    . . .  
}
```


В классе X есть два метода f(...), но с разными типами возвращаемого значения и разными наборами параметров. Тип возвращаемого значения не является определяющим фактором при совместном использовании — при вызове метода транслятору нужно определить, какой из одноименных методов вызывать, а тип возвращаемого значения, в общем случае, не позволяет сделать это однозначно. Поэтому нельзя описать в рамках одного класса два метода с одинаковым набором параметров и разными типами возвращаемых значений.

```
class Base {  
    int f(int k) {  
        . . .  
    }  
    . . .  
}  
  
class Derived extends Base {  
    int f(String s, int k) {  
        . . .  
    }  
    . . .  
}
```

В данном примере представлено совместное использование при наследовании. Класс `Derived` имеет два метода `f(...)`. Один он наследует от класса `Base`, другой описан в самом классе `Derived`.

Понятие **overloading** нужно отличать от понятия **overriding** (задавливание, подавление, переопределение). При переопределении (**overriding**) методов речь идет только о паре классов — базовом и порожденном. В порожденном классе определяется метод, полностью идентичный как по имени, так и по набору параметров тому, что есть в базовом.

Пример

```
class A {
    int x;
    int f(int a) {
        return a+x;
    }
    . . .
}

class B extends A {
    int y;

    int f(int s) {
        return s*x;
    }
    . . .
}

B b = new B();
A a = b;          // здесь происходит формальное преобразование типа: B => A
int c = a.f(10);  // ??? какой из f(...) будет вызван ???
```

Здесь самым интересным моментом является последняя строка. В ней "a" формально имеет тип A, но фактически ссылается на объект класса B. Возникает вопрос, какой из двух совершенно одинаково описанных методов f() будет вызван. Ответ на этот вопрос — B.f().

В Java (как и в других объектно-ориентированных языках) выполняется вызов метода данного объекта с учетом того, что объект может быть не того же класса, что и ссылка, указывающая на него. Т.е. выполняется вызов метода того класса, к которому реально относится объект.

Это — динамический полиморфизм методов. Он называется поздним связыванием (dynamic binding, late binding, run-time binding). В C++ соответствующий механизм называется механизмом виртуальных функций.

Рассмотрим содержательный пример использования возможностей, которые дает переопределение методов и позднее связывание. Реализуем классы Issue и Book иначе.

```
public class Issue {
    String name;
    public Issue(String name) {
        this.name = name;
    }
    public void print(PrintStream out) {
        out.println("Наименование:");
        out.println(name);
    }
    . . .
}

public class Book extends Issue {
    String authors;
    public Book(String name, String authors) {
        super(name);
        this.authors = authors;
    }

    public void print(PrintStream out) {
        out.println("Авторы:");
        out.println(authors);
        super.print(out);      // явный вызов метода базового класса
    }
    . . .
}
```


и переделаем фрагмент, обеспечивающий печать нашего каталога.

```
Issue[] catalog = new Issue[] {  
    new Journal("Play Boy"),  
    new Newspaper("Спид Инфо"),  
    new Book("Война и мир", "Л.Толстой"), };  
  
...  
for(int i = 0; i < catalog.length; i++) {  
    catalog[i].print(System.out);  
}
```

В классах Issue и Book вместо двух методов `printName(...)` и `printAuthors(...)` теперь один метод `print(..)`. В классе Book метод `print(...)` переопределяет одноименный метод класса Issue.

При написании метода `print(...)` в Book для сокращения кода использован прием явного вызова метода базового класса с использованием ключевого слова **`super`** .

Теперь при печати каталога мы можем не делать специальную проверку для Book. Нужный метод `print(...)` класса Book будет вызван автоматически благодаря механизму позднего связывания.



```
1 package edu.javacourse.robot;
2
3 public class RobotManager
4 {
5     public static void main(String[] args) {
6         // Первое проявление полиморфизма - ссылке на класс-предок
7         // можно присвоить класс-потомок
8         Robot robot = new RobotTotal(0, 0);
9
10        robot.forward(20);
11        robot.setCourse(90);
12        robot.forward(20);
13        robot.setCourse(90);
14        robot.forward(50);
15        // Напечатать координаты
16        robot.printCoordinates();
17        // Напечатать общую дистанцию уже не получится
18        // компилятор выдает ошибку
19        //System.out.println(robot.getTotalDistance());
20    }
21 }
```

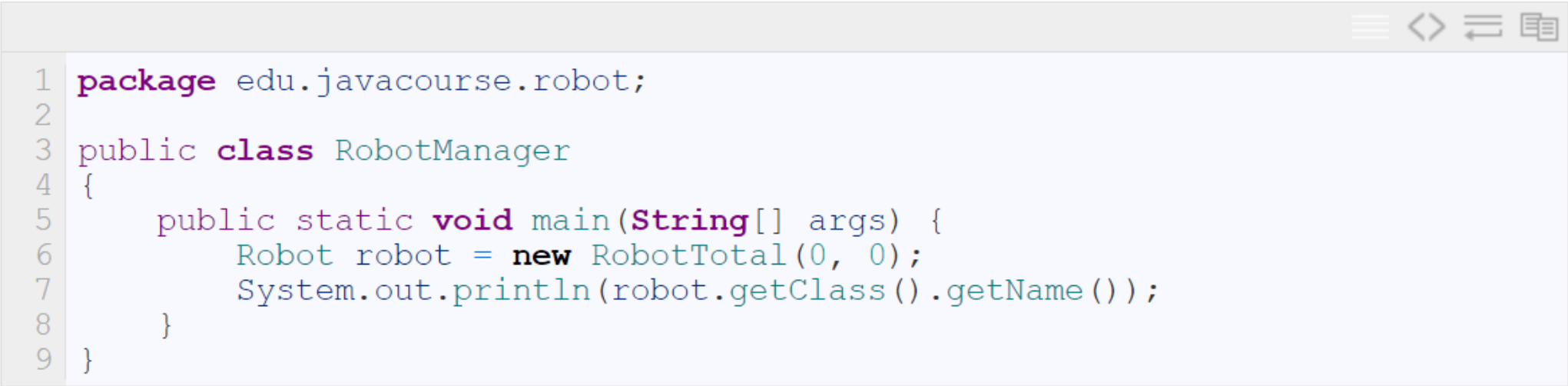

Как видим, создавая объект RobotTotal мы его «сужаем» до объекта класса Robot. С одной стороны это выглядит достаточно логично и непротиворечиво — RobotTotal является роботом. С другой стороны возникает вопрос — а метод forward будет вызываться какой ? Класса RobotTotal или Robot ? Думаю, что для вас ответ «Как RobotTotal» выглядит предпочтительнее — и это совершенно правильный ответ. Можете в этом убедиться сами, добавив в метод forward какую-либо поясняющую печать. Например так:

```
1      @Override
2      public void forward(int distance) {
3          super.forward(distance);
4          totalDistance += distance;
5          System.out.println("RobotTotal");
6      }
```

В этом можно убедиться еще более удобным и практичным образом — спросить у объекта его класс и у класса спросить его имя. Этот механизм называется Reflection — он позволяет получить информацию об объекте прямо в момент выполнения программы. Вот как может выглядеть такой вызов:

```
System.out.println(robot.getClass().getName());
```

Расшифровывается это приблизительно так: сначала у объекта `robot` получаем его класс методом `getClass()`. Возвращается объект типа `Class` (есть такой стандартный класс — `Class`). И у этого класса есть метод, который возвращает имя — `getName()`. Вот возможный полный код:

A screenshot of a code editor window with a light blue background. The code is written in Java and is color-coded: keywords like 'package', 'public', 'class', 'void', 'new', and 'String' are in purple; identifiers like 'edu', 'javacourse', 'robot', 'RobotManager', 'RobotTotal', and 'System' are in blue; and literals like '0' are in green. The code is as follows:

```
1 package edu.javacourse.robot;
2
3 public class RobotManager
4 {
5     public static void main(String[] args) {
6         Robot robot = new RobotTotal(0, 0);
7         System.out.println(robot.getClass().getName());
8     }
9 }
```

Как видим мы можем присвоить ссылке на объект класса-предка объект-потомок — и это работает. С одной стороны, мы работаем как-будто с классом Robot, с другой стороны — поведение нашего объекта соответствует классу RobotTotal. **ВАЖНО !!!** А вот в обратную сторону присваивание НЕ работает. На запись вот такого вида

```
RobotTotal robot = new Robot(0, 0);
```

компилятор будет выдавать ошибку.

В программировании полиморфизм используется для того, чтобы сделать приложения более модульными и расширяемыми. Вместо беспорядочных условных предложений, описывающих различные направления действия, вы создаете взаимозаменяемые объекты, которые подбираете согласно своим нуждам. Это основная задача полиморфизма.

Абстрактные классы

Абстрактным называется класс, на основе которого не могут создаваться объекты. При этом наследники класса могут быть не абстрактными, на их основе объекты создавать, соответственно, можно.

Для того, чтобы превратить класс в абстрактный перед его именем надо указать модификатор `abstract`.

Рассмотрим пример абстрактного класса A:

```
abstract class A {
    int p1;
    A() {
        p1 = 1;
    }
    void print() {
        System.out.println(p1);
    }
}
class B extends A {
}
public class Main {
    public static void main(String[] args) {
        A ob1;
        // ошибка: ob1 = new A();
        B ob2 = new B(); // будет вызван конструктор по умолчанию из A
        ob2.print();
    }
}
```

Java разрешит описать конструкторы в классе A, но не разрешит ими воспользоваться (потому что запрещено создавать объекты абстрактного класса).

Обратите внимание на то, что объявление переменной ob1 как ссылки, на объект класса A тоже не запрещается.

Абстрактные методы

Абстрактным называется метод, который не имеет реализации в данном классе. После круглых скобок, где перечислены его аргументы, ставится не открывающая фигурная скобка, чтобы начать блок описания метода, а точка с запятой. То есть описание у абстрактного метода отсутствует. Перед именем метода указывается при этом модификатор `abstract`.

Какой смысл в создании метода без реализации? Ведь его нельзя будет использовать. Для объектов того класса, где метод описан – конечно же использовать нельзя, но вот если унаследовать класс и в потомках переопределить метод, задав там его описание, то для объектов классов потомков метод можно будет вызывать (и работать будут описанные в классах потомков реализации).

Чтобы исключить возможность использования абстрактного метода, в Java введено следующее требование: класс имеющий хоть один абстрактный метод обязан быть абстрактным классом.

Когда же уместно использовать абстрактные методы и классы? Сначала рассмотрим пример иерархии классов домашних животных, где нет ни абстрактных классов, ни абстрактных методов.

```
class Pet {  
    String name;  
    int age;  
    boolean hungry;  
    void voice() {  
    }  
    void food() {  
        hungry = false;  
    }  
}  
class Snake extends Pet {  
    double length;  
    void voice() {  
        System.out.println("Шшш-ш-ш");  
    }  
}
```

```
class Dog extends Pet {
    void voice() {
        System.out.println("Гав-гав");
    }
}
class PatrolDog extends Dog {
    void voice() {
        System.out.println("Ppp-p-p");
    }
}
class Cat extends Pet {
    void voice() {
        System.out.println("Мяу-мяу");
    }
}
class Fish extends Pet {
}
public class Main {
    public static void main(String[] args) {
        Pet zorka = new Pet();
        zorka.food();
        Fish nemo = new Fish();
        nemo.voice();
    }
}
```


Поскольку нет какого-то общего звука, который издавали бы все домашние животные, то мы в классе `Pet` не стали задавать какую-то реализацию методу `voice()`, внутри метода не делается совсем ничего, но тем не менее у него есть тело, обособленное блоком из фигурных скобок. Метод `voice()` хороший претендент на то, чтобы стать абстрактным.

Кроме того, вряд ли можно завести себе домашнее животное неопределенного вида, то есть у вас дома вполне могли бы жить `Cat`, `Dog` или даже `Snake`, но вряд ли вы бы смогли завести животное `Pet`, являющееся непонятно кем.

Соответственно, в рамках реальной задачи вряд ли потребуется создавать объекты класса `Pet`, а значит его можно сделать абстрактным (после чего, правда, мы даже при делании не сможем создать объекты на его основе).

Рассмотрим пример с участием абстрактного класса и абстрактного метода:

```
abstract class Pet {
    String name;
    int age;
    boolean hungry;
    abstract void voice();
    void food() {
        hungry = false;
    }
}
class Snake extends Pet {
    double length;
    void voice() {
        System.out.println("Шшш-ш-ш");
    }
}
class Dog extends Pet {
    void voice() {
        System.out.println("Гав-гав");
    }
}
class PatrolDog extends Dog {
    void voice() {
        System.out.println("Ppp-p-p");
    }
}
```

```
class Cat extends Pet {
    void voice() {
        System.out.println("Мяу-мяу");
    }
}
class Fish extends Pet {
    void voice() {
    }
}
public class Main {
    public static void main(String[] args) {
        // ошибка: Pet zorka = new Pet();
        Fish nemo = new Fish();
        nemo.voice();
    }
}
```

Обратите внимание, что теперь, во-первых, мы не можем создавать объекты абстрактного класса Pet, а, во-вторых, реализация метода voice() должна иметься во всех его потомках (хотя бы пустая реализация), не являющихся абстрактными классами.

Хотя, мы могли бы создать абстрактного потомка:

```
abstract class Fish extends Pet {  
}
```

Но не могли бы создавать объектов класса Fish, нам пришлось бы расширять класс, чтоб в итоге получить не абстрактный и создавать на его основе объекты. Например:

```
class GoldenFish extends Fish {  
    void voice() {  
    }  
}
```

Интерфейсы

Интерфейс - это конструкция языка программирования Java, в рамках которой могут описываться только абстрактные публичные (`abstract public`) методы и статические константные свойства (`final static`). То есть также, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.

Один интерфейс может быть наследником другого интерфейса.

Классы могут реализовывать интерфейсы (т. е. получать от интерфейса список методов и описывать реализацию каждого из них), притом, что особенно важно, один класс может реализовывать сразу несколько интерфейсов.

Перед описанием интерфейса указывается ключевое слово `interface`. Когда 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.

Пример:

```
interface Instruments {  
    final static String key = "До мажор";  
    public void play();  
}  
class Drum implements Instruments {  
    public void play() {  
        System.out.println("бум бац бац бум бац бац");  
    }  
}  
class Guitar implements Instruments {  
    public void play() {  
        System.out.println("до ми соль до ре до");  
    }  
}
```

Поскольку все свойства интерфейса должны быть константными и статическими, а все методы общедоступными, то соответствующие модификаторы перед свойствами и методами разрешается не указывать. То есть интерфейс можно было описать так:

```
interface Instruments {  
    static public String key = "До мажор";  
    void play();  
}
```

Но когда метод play() будет описываться в реализующем интерфейс классе, перед ним всё равно необходимо будет явно указать модификатор public.

Java не поддерживает множественное наследование классов. Вместо множественного наследования классов в Java введено множественное наследование интерфейсов.

Класс Object

В Java есть специальный суперкласс Object и все классы являются его подклассами.

Поэтому ссылочная переменная класса Object может ссылаться на объект любого другого класса.

Так как массивы являются тоже классами, то переменная класса Object может ссылаться и на любой массив.

У класса есть важные методы.

- `Object clone()` - создаёт новый объект, не отличающийся от клонируемого.
- `boolean equals(Object object)` - определяет, равен ли один объект другому.
- `void finalize()` - вызывается перед удалением неиспользуемого объекта.
- `Class<?> getClass()` - получает класс объекта во время выполнения.
- `int hashCode()` - возвращает хэш-код, связанный с вызывающим объектом.
- `void notify()` - возобновляет выполнение потока, который ожидает вызывающего объекта
- `void notifyAll()` - возобновляет выполнение всех потоков, которые ожидают вызывающего объекта
- `String toString()` - возвращает строку, описывающий объект
- `void wait()` - ожидает другого потока выполнения
- `void wait(long ms)` - ожидает другого потока выполнения
- `void wait(long ms, int nano)` - ожидает другого потока выполнения

Методы **getClass()**, **notify()**, **notifyAll()**, **wait()** являются **final** методами и их нельзя переопределять.

Каждый класс реализует метод **toString()**, так как он определён в классе **Object**. Но использовать метод по умолчанию не слишком удобно, так как не содержит полезной информации.

Разработчики предпочитают переопределять данный метод под свои нужды. Сам метод имеет форму:

```
String toString()
```

Вам остаётся вернуть объект класса **String**, который будет содержать полезную информацию о вашем классе. Давайте возьмём для примера класс **Box**.

```
class Box {  
    int width; // ширина коробки  
    int height; // высота коробки  
    int depth; // глубина коробки  
  
    Box(int width, int height, int depth){  
        this.width = width;  
        this.height = height;  
        this.depth = depth;  
    }  
  
    // вычисляем объём коробки  
    String getVolume() {  
        return "Объём коробки: " + (width * height * depth);  
    }  
  
    public String toString() {  
        return "Коробочка для кота размером " + width + "x" + height + "x" + depth;  
    }  
}
```

Теперь можете узнать о классе **Box**:

```
TextView tvInfo = (TextView)findViewById(R.id.textView1);  
Box box = new Box(4, 5, 6);  
tvInfo.setText(box.toString());
```

Метод очень часто используется при создании собственных классов.