Доступ к полям и методам

Модификатор	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Переопределение методов (Override, не путать с Overload!)

В классе-потомке можно переопределить метод базового класса, но при этом должны сохраниться:

- Имя метода
- Тип возвращаемого значения
- Список аргументов
- Модификатор доступа – такой же или мягче

Начиная с Java5, тип возвращаемого значения может быть наследником, возвращаемого в базовом классе типа.

Пример 1. Переопределение методов (override)

```
public class Employee {
    protected String name;
    protected double salary;
    public String toString() {
        return "Name: " + name + "\n" +
        "Salary: " + salary;
    }
}
```

```
public class Manager extends Employee {
    protected String department;
    public String toString() {
        return "Name: " + name + "\n" +
        "Salary: " + salary + "\n" +
        "Manager of: " + department;
    }
}
```

Пример 2. Переопределение методов (override)

```
public class Employee {
    private String name;
    private double salary;
    public String toString() {
        return "Name: " + name + "\nSalary: " + salary;
    }
}
```

```
public class Manager extends Employee {
    private String department;
    public String toString() {
        // call parent method
        return super.toString()
            + "\nDepartment: " + department;
    }
}
```

Полиморфные (гетерогенные) массивы

- Коллекции объектов одного типа называются *гомогенные*

```
MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

- Коллекции объектов разного типа называются *гетерогенные*

```
Employee[] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

Полиморфные аргументы

Поскольку `Manager` является `Employee`, можно написать:

```
public class TaxService {
    public TaxRate findTaxRate(Employee e) {
        // calculate the employee's tax rate
    }
}

// ...

TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

Оператор **instanceof**

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee

//...

public void doSomething(Employee e) {
    if ( e instanceof Manager ) {
        // Process a Manager
    } else if ( e instanceof Engineer ) {
        // Process an Engineer
    } else {
        // Process any other type of Employee
    }
}
```

Преобразование объектных типов (Casting)

```
public void doSomething(Employee e) {
    if ( e instanceof Manager ) {
        Manager m = (Manager) e;
        System.out.println("This is the manager of "
            + m.getDepartment());
    }
    // rest of operation
}
```

Полиморфизм реализуется с помощью методов

Не реализуется с помощью полей!

(см. пример на след. слайде)

```
class A{
    String name = "Class A";
    String getName() {
        return name;
    }
}

class B extends A{
    String name = "Class B"; // Никогда так не делайте (не переопределяйте поля) !!!
    String getName() {
        return name;
    }
}

public class AB{
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        A ab = new B();

        System.out.println("a : " + a.name + " " + a.getName());
        System.out.println("b : " + b.name + " " + b.getName());
        System.out.println("ab: " + ab.name + " " + ab.getName());
    }
}
```

Класс **Object**

- Класс **Object** class является предком всех классов в Java.
- Декларация класса без слова **extends** эквивалентна **extends Object**.

```
public class Employee {
...
}
```

```
public class Employee extends Object {
...
}
```

Класс **Object**

Доступ	Тип	Имя
protected	Object	**clone**()
public	boolean	**equals**(Object obj)
protected	void	**finalize**()
public	Class<?>	**getClass**()
public	int	**hashCode**()
public	void	**notify**()
public	void	**notifyAll**()
public	String	**toString**()
public	void	**wait**()
public	void	**wait**(long timeout)
public	void	**wait**(long timeout, int nanos)

```
public class EquTest {
   public static void main(String[] args) {
      String a = "a";
      String b = "b";
      String ab1 = "ab";
      String ab2 = a+b;
      System.out.println(ab1);
      System.out.println(ab2);
      System.out.println(ab1 == ab2);
      System.out.println(ab1.equals(ab2));
   }
}
```

static

Используется при объявлении

- полей
- методов
- блоков статической инициализации
- импорта классов (1.5 и выше)
- вложенных классов

static-поля

- принадлежат всему классу, а не какому-то конкретному объекту
- можно использовать без создания объекта
- для обращения можно использовать имя класса или объекта
- инициализация происходит при загрузке класса

```
class Count {
    private int serialNumber;
    public static int counter = 0;

    public Count() {
        counter++;
        serialNumber = counter;
    }

    public int getSerialNumber() {
        return serialNumber;
    }
}

public class StaticTest {
    public static void main(String[] args) {
         Count c1 = new Count();
         System.out.println(c1.getSerialNumber());
         Count c2 = new Count();
         System.out.println(c2.getSerialNumber());
         System.out.println(c1.counter);
         System.out.println(Count.counter);
    }
}
```

Пример 2.

```
public class StaticTest2 {
    static int myStatic = returnIntSayHello();

    static int returnIntSayHello() {
        System.out.println("Hello");
        return 1;
    }

   public static void main(String[] args) {
        System.out.println("Entry point");
   }
}
```

static-методы

- вызываются для целого класса, а не для конкретного объекта
- можно использовать без создания объекта
- для обращения можно использовать имя класса или объекта
- могут обращаться только к статическим полям и методам
- отсутствует “**this**”

```
class Test3 {
    static int myStatic;
    int myNonStatic;

    static Test3 staticMethod() {
        myStatic = 1; // OK
//      myNonStatic = 2; // ERROR !
//      return this;  //ERROR !
        return new Test3(); // OK
    }
}
```

Блоки статической инициализации

- выполняется один раз при загрузке класса
- как правило, используется для инициализации статических полей, когда простой инициализации с помощью одного оператора присваивания недостаточно

Пример 1

```
class A{
    static int[] a = new int[5];
    static {
        for(int i=0; i<a.length; i++){
            a[i] = i*i;
        }
        System.out.println("Init done!");
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println("Begin");
        A a = new A();
        A b = new A();
    }
}
```

Пример 2 (Часть 1/2)

```
class A {
    static int a1=printStringReturnInt("a1");

    static {
        printStringReturnInt("static block");
    }

    A() {
        printStringReturnInt("Constructor");
    }

    static int a2 = printStringReturnInt("a2");

    public static int printStringReturnInt(String s) {
        System.out.println(s);
        return 0;
    }
}
```

Пример 2 (Часть 2/2)

```
public class Main {
    static {
        System.out.println("Before main");
    }

    public static void main(String[] args) {
        System.out.println("Begin of main");
        A a = new A();
        System.out.println("End of main");
    }

    static {
        System.out.println("After main");
    }
}
```

Статический импорт

- позволяет вызывать статические методы класса без необходимости написания имени класса
- Java 1.5 и выше

Пример. Сравните:

```
public class Test {
   public static void main(String[] args) {
      double x = 0.5;
      double y = Math.sin(x)*Math.sin(x) +
         Math.cos(x)*Math.cos(x);
      System.out.println(y);
   }
}
```

и

```
import static java.lang.Math.*;
public class Test {
   public static void main(String[] args) {
      double x = 0.5;
      double y = sin(x)*sin(x) +
         cos(x)*cos(x);
      System.out.println(y);
   }
}
```

final

Используется при объявлении

- полей
- методов
- классов

final-поля

- после инициализации значения изменить нельзя
- используйте **public static final** для описания констант в классе

```
public class Test {
    public static final MY_CONST=1;
    public static void main(String[] args) {
        final int f=42;
        f=43; // ERROR!

        final int f2;
        f2 = 4242;  // OK

        final int a;
        if (f < 0) {
            a = 2;  // OK
        } else {
            a = 3;  // OK
        }
        a = 4;          // ERROR!
    }
}
```

final-методы

- невозможно переопределить при наследовании

```
class Base {
   public final void myMethod(){}
}

class Extended extends Base {
   public void myMethod(){} // Error!
}
```

final-классы

- невозможно создать наследника

```
class MySuperMegaString extends String { //ERROR!

}
```

abstract

Используется при объявлении

- методов
- классов

- Если в классе есть хоть один абстрактный метод, класс также должен быть объявлен как абстрактный
- Невозможно создать экземпляр абстрактного класса
- Возможно обращение к конкретному объекту-потомку используя ссылку на абстрактного предка

Пример (Часть 1/2)

```
abstract class Parent {
   abstract void sayHello();
   void sayBye() {
      System.out.println("Bye!");
   }
}

class Child1 extends Parent {
   void sayHello() {
      System.out.println("Hello, I am Child1");
   }
}

class Child2 extends Parent {
   void sayHello() {
      System.out.println("Hello, I am Child2");
   }
}
```

Пример (Часть 2/2)

```
public class Test {
   public static void main(String[] args) {
      //Parent p = new Parent(); // ERROR!!!
      Parent c1 = new Child1();
      Parent c2 = new Child2();
      c1.sayHello();
      c2.sayHello();
      c1.sayBye();
      c2.sayBye();
   }
}
```

Задания

Создать консольное приложение, удовлетворяющее следующим требованиям:

1. Использовать возможности ООП: классы, наследование, полиморфизм, инкапсуляция.
2. Каждый класс должен иметь исчерпывающее смысл название и информативный состав.
3. Наследование должно применяться только тогда, когда это имеет смысл.
4. Работа с консолью или консольное меню должно быть минимальным.
5. Иерархия классов должна быть 3 уровней.
6. В программе посоздавать экземпляры классов.

1. **Цветочница.** Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей. Создать букет из цветов трех видов, где выбор каждого цветка букета происходит случайно. Первый выбирается с вероятностью 30 %, второй – 5 %, третий – 65 %.

2. **Домашние электроприборы.** Определить иерархию электроприборов. Включить некоторые в розетку. Посчитать потребляемую мощность Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.