НАСЛЕДОВАНИЕ

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, имеется следующий класс Person, описывающий отдельного человека:

```
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
```

И, возможно, впоследствии мы захотим добавить еще один класс, который описывает сотрудника предприятия - класс Employee. Так как этот класс реализует тот же функционал, что и класс Person, поскольку сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (наследником, подклассом) от класса Person, который, в свою очередь, называется базовым классом, родителем или суперклассом:

```
class Employee extends Person{
    public Employee(String name){
        super(name);    // если базовый класс определяет конструктор
                        //  то производный класс должен его вызвать
    }
}
```

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова **super**. Например, класс Person имеет

конструктор, который принимает один параметр. Поэтому в классе Employee в конструкторе нужно вызвать конструктор класса Person. То есть вызов super(name) будет представлять вызов конструктора класса Person.

При вызове конструктора после слова super в скобках идет перечисление передаваемых аргументов. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса. Таким образом, установка имени сотрудника делегируется конструктору базового класса.

Причем даже если производный класс никакой другой работы не производит в конструкторе, как в примере выше, все равно необходимо вызвать конструктор базового класса.

Использование классов:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
        tom.display();
        Employee sam = new Employee("Sam");
        sam.display();
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    public  Employee(String name){
        super(name);    // если базовый класс определяет конструктор
                        //  то производный класс должен его вызвать
```

```
    }
}
```

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором **private**. При этом производный класс также может добавлять свои поля и методы:

```
public class Program{

    public static void main(String[] args) {

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
        sam.work();     // Sam works in Microsoft
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    public void work(){
        System.out.printf("%s works in %s \n", getName(), company);
    }
}
```

В данном случае класс Employee добавляет поле company, которое хранит место работы сотрудника, а также метод work.

Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Например, переопределим в классе Employee метод display:

```
public class Program{

    public static void main(String[] args) {

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
                        // Works in Microsoft
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){
```

```
        System.out.printf("Name: %s \n", getName());
        System.out.printf("Works in %s \n", company);
    }
}
```

Перед переопределяемым методом указывается аннотация **@Override**. Данная аннотация в принципе необязательна.

При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом класса. Например, если в базовом классе метод имеет модификатор public, то и в производном классе метод должен иметь модификатор public.

Однако в данном случае мы видим, что часть метода display в Employee повторяет действия из метода display базового класса. Поэтому мы можем сократить класс Employee:

```
class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){

        super.display();
        System.out.printf("Works in %s \n", company);
    }
}
```

С помощью ключевого слова super мы также можем обратиться к реализации методов базового класса.

Запрет наследования

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова **final**. Например:

```
public final class Person {
}
```

Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

```
class Employee extends Person{ {
}
```

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод display(), запретим его переопределение:

```
public class Person {

    //.....................

    public final void display(){

        System.out.println("Имя: " + name);
    }
}
```

В этом случае класс Employee не сможет переопределить метод display.

Динамическая диспетчеризация методов

Наследование и возможность переопределения методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса:

```
Person sam = new Employee("Sam", "Oracle");
```

Так как Employee наследуется от Person, то объект Employee является в то же время и объектом Person. Грубо говоря, любой работник предприятия одновременно является человеком.

Однако несмотря на то, что переменная представляет объект Person, виртуальная машина видит, что в реальности она указывает на объект Employee. Поэтому при вызове методов у этого объекта будет вызываться та версия метода, которая определена в классе Employee, а не в Person. Например:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
```

```
        tom.display();
        Person sam = new Employee("Sam", "Oracle");
        sam.display();
    }
}
class Person {

    String name;

    public String getName() { return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.printf("Person %s \n", name);
    }
}

class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company = company;
    }
    @Override
    public void display(){

        System.out.printf("Employee %s works in %s \n", super.getName(), company);
    }
}
```

Консольный вывод данной программы:

```
Person Tom
Employee Sam works in Oracle
```

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в

подклассе. Данный процесс еще называется **dynamic method lookup** или динамический поиск метода или динамическая диспетчеризация методов.