

程序员书库

初学者的入门宝典，程序员的百科全书



CD-ROM

10小时多媒体视频讲解

#### 本书特色

- ※ 起点低，即使没有任何编程经验，也能通过本书掌握Java
- ※ 避免大段理论讲解，而是通过大量实例进行讲解，有很强的实践性
- ※ 对代码进行了详细注释，阅读起来非常容易，没有任何障碍
- ※ 通过现实中的事物类比Java中的概念，使读者可以很容易理解
- ※ 重点讲解Java语言的基础知识和应用，并对一些设计模式也有所介绍
- ※ 全书提供190个实例和2个综合案例，非常实用

# Java

## 从入门到精通

高宏静 等编著



化学工业出版社

## 第6章 继 承

继承是面向对象编程的重要特征之一。顾名思义，继承就是在现有类的基础上构建新类以满足新的要求。在继承过程中，新的类继承原来的方法和实例变量，并且能添加自己的方法和实例变量。在本章中主要讲解的内容包括派生类的创建使用、方法覆写、抽象类的创建和使用、多态和动态绑定以及 `Object` 类。

### 6.1 派生类

面向对象语言的一个重要特性就是继承。继承是指声明一些类，可以再进行声明这些类的子类，而子类具有父类已经拥有的一些方法和属性，这跟现实中的父子关系是十分相似的，所以面向对象把这种机制称为继承，子类也称为派生类。

#### 6.1.1 继承的使用

继承是在已有类的基础上构建新的类。已有的类称为超类、父类或基类，产生的新类称为子类或派生类。在动物种类中可以包括老虎、大象和猴子等多种动物，这里通过这个为原型来学习继承。例如，可以构建一个 `Animal` 类，如下所示。

```
class Animal{
    String type;           //种类
    String name;           //名字
    int age;               //年龄
    int weight;            //体重
    void eat(){             //吃饭方法
        System.out.println("animal eat");
    }
    void breath(){          //呼吸方法
        System.out.println("animal breath");
    }
    void sleep(){           //睡觉方法
        System.out.println("animal sleep");
    }
}
```

在 `Animal` 类中，有种类、名字、年龄、体重这些实例变量描述动物，以及呼吸、吃饭、睡觉这些方法表示动物的动作。下面在 `Animal` 的基础上构建一个类 `Tiger` 来表示老虎。

```
//声明 Tiger 类继承 Animal 类
class Tiger extends Animal{
    String tigerType;
    String from;
    void tigerRun(){
        System.out.println("the tiger run");
    }
}
```

```

}
}

```

注意 Tiger 类的第一行 `class Tiger extends Animal`，表示 Tiger 类继承自 Animal 类。通过继承，新生成的老虎类不仅继承了 Animal 类的所有实例变量和方法，还有自己的独有的字段 `tigerType`、`from` 和方法 `tigerRun`。Tiger 类的使用如下所示。

```

public class TigerDemo {
    public static void main(String args[] ) {
        //构建一个 Tiger 对象
        Tiger tiger = new Tiger();
        //对 tiger 的属性进行赋值
        tiger.type = "Tiger";
        tiger.name="huhu";
        tiger.age=12;
        tiger.weight=120;
        tiger.tigerType="东北虎";
        tiger.from="长白山";
        //打印出属性值
        System.out.println("Animal 属性:种类="+tiger.type);
        System.out.println("Animal 属性:名字="+tiger.name);
        System.out.println("Animal 属性:年龄"+tiger.age);
        System.out.println("Animal 属性:体重"+tiger.weight);
        System.out.println("Tiger 属性: 老虎种类="+tiger.tigerType);
        System.out.println("Tiger 属性: 产地="+tiger.from);
        //使用 tiger 调用方法
        System.out.println("Animal 方法: 呼吸");
        tiger.breath();
        System.out.println("Animal 方法: 吃饭");
        tiger.eat();
        System.out.println("Animal 方法: 睡觉");
        tiger.sleep();
        System.out.println("Tiger 方法: 奔跑");
        tiger.tigerRun();
    }
}

```

可以看到 Tiger 类继承了 Animal 类的所有属性和方法，并有自己特有的属性和方法。程序的运行结果如下。

```

Animal 属性:种类=Tiger
Animal 属性:名字=huhu
Animal 属性:年龄 12
Animal 属性:体重 120
Tiger 属性: 老虎种类=东北虎
Tiger 属性: 产地=长白山
Animal 方法: 呼吸
animal breath
Animal 方法: 吃饭
animal eat
Animal 方法: 睡觉
animal sleep
Tiger 方法: 奔跑
the tiger run

```

### 6.1.2 子类对象的构建

在上一章中已经学习了如何创建类的对象。继承也是对类进行操作。既然继承可以这么方便的使用，子类对象的创建过程是怎么样的呢？答案是从最顶层的基类开始往下一层层的调用默认构造函数。示例如下。

```
class A {
    A() {
        System.out.println("调用 A 的构造函数");
    }
}
//类 B 继承 A
class B extends A {
    B() {
        System.out.println("调用 B 的构造函数");
    }
}
//C 继承 B
class C extends B {
    C() {
        System.out.println("调用 C 的构造函数");
    }
}
//通过该类演示对象的构造过程
public class CallConstructor {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

程序的运行结果如下。

```
调用 A 的构造函数
调用 B 的构造函数
调用 C 的构造函数
```

在程序中定义了 3 个类 A、B、C，其中 B 继承自 A，C 继承自 B，当创建一个 C 类的对象时候，会自动调用父类的无参构造函数。如果想调用父类的有参构造函数，需要使用 `super` 关键字，调用父类的构造函数的语句要放在所在方法的第一个语句中。修改上面的程序如下。

```
class A {
    A() { //A 类的无参构造器
        System.out.println("调用 A 的构造函数");
    }
    A(int i) { //A 类的有参构造器
        System.out.println("调用 A 的有参构造函数");
    }
}
class B extends A { //让 B 类继承 A 类
    B() { //B 类的无参构造器
        System.out.println("调用 B 的构造函数");
    }
}
```

```

    B(int i){                //B 类的有参构造器
        super(5);           //调用父类的有参构造器
        System.out.println("调用 B 的有参构造函数");
    }
}

class C extends B {        //让 C 类继承 B 类
    C() {                  //C 类无参构造器
        System.out.println("调用 C 的构造函数");
    }
    C(int i){              //C 类的有参构造器
        super(5);          //调用父类也就是 B 类的有参构造器
        System.out.println("调用 C 的有参构造函数");
    }
}

public class CallConstructor2 {
    public static void main(String[] args) {
        C c = new C();      //创建 C 类对象
        C c0=new C(5);     //创建 C 类的具有参数对象
    }
}

```

给每个类都加上有参数的构造函数，在有参数的构造函数中，通过 `super` 关键字调用其父类构造函数。在 `CallConstructor` 中构建两个不同的对象，通过程序的输出可以看出这两个对象的构建过程。程序的执行结果如下。

```

调用 A 的构造函数
调用 B 的构造函数
调用 C 的构造函数
调用 A 的有参构造函数
调用 B 的有参构造函数
调用 C 的有参构造函数

```

在 C++ 中，一个类是可以有多个父类的，这样会使语言变的非常复杂，而且多重继承不是必须的。Java 改进了 C++ 的这一点，不支持多继承，一个类的直接父类只能有一个。

### 6.1.3 方法的覆写

方法覆写（`overload`）与方法的重载非常相似，它在 Java 的继承中也有很重要的应用。

写程序可能会碰到下面的情况，在父类中已经实现的方法可能不够精确，不能满足子类的需求。例如在前面的 `Animal` 类中，`breath` 方法就过于简单，对于鱼类动物是用腮呼吸的，而对于哺乳动物则是用肺呼吸的，如何实现呢，Java 提供的方法覆写就是解决这方面的问题。

在下面的程序中首先定义了一个父类 `Animal`，然后定义 `Animal` 的 3 个子类 `Tiger`、`Fish` 和 `Dog`，在父类中提供了 3 个方法 `eat`、`breath`、`sleep`，在两个子类 `Tiger` 和 `Fish` 中重新定义了 `breath` 方法，在 `Dog` 类中什么都没做。在 `OverloadDemo` 中，创建了一个 `Fish` 对象、一个 `Tiger` 对象和一个 `Dog` 对象，分别调用 `breath` 方法。

```

class Animal {
    String type;           //种类

```

```

String name;           //名称
int age;               //年龄
int weight;           //体重
void eat() {           //吃饭方法
    System.out.println("动物爱吃饭");
}
void breath() {        //呼吸方法
    System.out.println("动物呼吸");
}
void sleep() {         //睡觉方法
    System.out.println("动物在睡觉");
}
}
//Tiger 类继承 Animal 类
class Tiger extends Animal {
    String tigerType;   //老虎种类
    String from;        //定义老虎独有变量
//Tiger 自己的方法
    void tigerRun() {   //老虎的奔跑方法
        System.out.println("老虎在奔跑");
    }
    void breath(){      //继承来的呼吸方法
        System.out.println("老虎是用肺呼吸的");
    }
}
//Fish 继承 Animal 类
class Fish extends Animal{
    String fishType;
//Fish 自己的方法
    void swim(){
        System.out.println("鱼在游泳");
    }
    void breath(){
        System.out.println("鱼是用腮呼吸的");
    }
}
class Dog extends Animal{
}
public class OverloadDemo
{
    public static void main(String[ ] args) {
        //声明三个不同的对象
        Tiger tiger=new Tiger();
        Fish fish=new Fish();
        Dog dog=new Dog();
        //都调用 breath 方法
        tiger.breath();
        fish.breath();
        dog.breath();
    }
}

```

程序的运行结果如下。

老虎是用肺呼吸的

鱼是用腮呼吸的  
动物呼吸

方法被覆写后如果又需要调用，可以使用 `super` 关键字来实现，示例如下。

```
class Animal {
    String type;
    String name;
    int age;
    int weight;
    void eat() {
        System.out.println("动物爱吃饭");
    }
    void breath() {
        System.out.println("动物呼吸");
    }
}
class Tiger extends Animal {
    String tigerType;
    String from;
    void breath()
    {
        //通过 super 关键字调用父类的 breath 方法
        super.breath();           //调用动物类的呼吸方法
        System.out.println("老虎是用肺呼吸的");
    }
}
public class SuperDemo{
    public static void main(String args[] ){
        Tiger tiger=new Tiger();
        tiger.breath();
    }
}
```

在 `Animal` 的子类 `Tiger` 中，在 `breath` 方法中，使用语句 `super.breath()` 调用父类的 `breath` 方法。程序的运行结果如下。

动物呼吸  
老虎是用肺呼吸的

`super` 关键字主要有以下两个用途。

- ☐ 在子类构造函数中调用父类构造函数。
- ☐ 在子类中调用父类的方法。

#### 6.1.4 多态与动态绑定

多态是面向对象语言的又一重要特性。多态是指同一个方法根据上下文使用不同的定义的能力。从这一点看，上一节讲的方法覆写以及前面的方法重载都可被看做是多态。但是 `Java` 的多态更多的是跟动态绑定放在一起理解的。

动态绑定是一种机制，通过这种机制，对一个已经被重写的方法的调用将会发生在运行时，而不是在编译时解析。下面的程序演示了动态绑定，定义父类 `Animal` 和子类 `Tiger` 以及

Fish 如下。

```
class Animal {
    String type;
    String name;
    int age;
    int weight;
    void eat() {
        System.out.println("动物爱吃饭");
    }
    void breath() {
        System.out.println("动物呼吸");
    }
    void sleep() {
        System.out.println("动物在睡觉");
    }
}
class Tiger extends Animal {
    String tigerType;
    String from;
    void tigerRun() {
        System.out.println("老虎在奔跑");
    }
    void breath(){
        System.out.println("老虎是用肺呼吸的");
    }
}
class Fish extends Animal{
    String fishType;
    void swim(){
        System.out.println("鱼在游泳");
    }
    void breath(){
        System.out.println("鱼是用腮呼吸的");
    }
}
```

演示程序如下。

```
public class DynamicMethodDemo
{
    public static void main(String args[ ])
    {
        Animal [ ]animal=new Animal[3];
        //创建不同的对象，但是都存入 Animal 的引用中
        animal[0]=new Animal();
        animal[1]=new Tiger();
        animal[2]=new Fish();
        animal[0].breath();
        animal[1].breath();
        animal[2].breath();
    }
}
```

程序定义一个存放 Animal 对象的数组，animal 数组的 3 个元素分别存放一个 Animal 对象、一个 Tiger 对象、一个 Fish 对象，然后对这 3 个对象调用 breath 方法。程序的执行结果如下。



动物呼吸  
老虎是用肺呼吸的  
鱼是用腮呼吸的

在 Java 中，对象是多态的，定义一个 `Animal` 对象，它既可以存放 `Animal` 对象，也可以存放 `Animal` 的子类 `Tiger`、`Fish` 的对象。

存放在 `Animal` 数组中的 `Tiger` 对象和 `Fish` 对象在执行 `breath` 方法时会自动地调用原来对象的方法而不是 `Animal` 的 `breath` 方法，这就是动态绑定。

需要注意一点的是，通过数组元素访问方法的时候只能访问在 `Animal` 中定义的方法，对于 `Tiger` 类和 `Fish` 中定义的方法时却不能调用，例如语句 `animal[2].swim();` 就是不正确的。当需要访问这些方法时需要用到类型转换，演示程序如下。

```
public class DynamicMethodDemo2{
    public static void main(String args[ ]){
        Animal [ ]animal=new Animal[3];
        animal[0]=new Animal();
        animal[1]=new Tiger();
        animal[2]=new Fish();
        DynamicMethodDemo2 dm=new DynamicMethodDemo2();
        dm.move(animal[0]);
        dm.move(animal[1]);
        dm.move(animal[2]);
    }
    void move(Animal animal){
        //进行对象类型的判断
        if(animal instanceof Tiger)
            ((Tiger)animal).tigerRun();
        else if(animal instanceof Fish)
            ((Fish)animal).swim();
        else animal.sleep();
    }
}
```

主要看 `move` 方法，`move` 方法首先判断 `animal` 对象是哪个类的对象，由判断执行不同的方法。在判断过程使用了 `instanceof` 运算符，它是用来判断对象类型的一个运算符。当判断出它的类型之后，再对其进行类型转换，得到原始类型后就可以调用它的类所特有的方法了。程序的运行结果如下。

动物在睡觉  
老虎在奔跑  
鱼在游泳

### 6.1.5 final 关键字

编写程序时可能需要把类定义为不能继承的，即最终类，或者是有的方法不希望被子类继承，这时候就需要使用 `final` 关键字来声明。把类或方法声明为 `final` 类或 `final` 方法的方法很简单，在类前面加上 `final` 关键字即可。

```
final class 类名 extends 父类{
    //类体
}
```

方法也可以被声明为 `final` 的，形式如下。

```
修饰符 final 返回值类型 方法名(){
//方法体
}
```

例如：

```
public final void run(){
//方法体
}
```

需要注意的是，实例变量也可以被定义为 `final`，被定义为 `final` 的变量不能被修改。被声明为 `final` 的类的方法自动地被声明为 `final`，但是它的实例变量并不是 `final`。

## 6.2 抽象类

抽象类是指在类中定义方法，但是并不去实现它，而在它的子类中去具体的实现。定义的抽象方法不过是一个方法占位符。继承抽象类的子类必须实现父类的抽象方法，除非子类也被定义成一个抽象类。

### 6.2.1 抽象类的定义

对抽象类有了基本了解后，就来看一下如何定义抽象类。定义抽象类是通过 `abstract` 关键字实现的。抽象类的一般形式如下。

```
修饰符 abstract 类名{
//类体
}
```

抽象方法的定义形式如下。

```
修饰符 abstract 返回值类型 方法名();
```

注意：在抽象类中的方法不一定是抽象方法，但是含有抽象方法的类必须被定义成抽象类。这里利用抽象类的方法对前面的 `Animal`、`Tiger`、`Fish` 类重新定义。

```
//抽象类的声明
abstract class Animal {
    String type;
    String name;
    int age;
    int weight;
    void eat() {
        System.out.println("动物爱吃饭");
    }
    abstract void breath();
    void sleep() {
        System.out.println("动物在睡觉");
    }
}
```

```
//Tiger 继承抽象类 Animal
class Tiger extends Animal {
    String tigerType;
    String from;
    void tigerRun() {
        System.out.println("老虎在奔跑");
    }
    void breath() {
        System.out.println("老虎是用肺呼吸的");
    }
}
class Fish extends Animal {
    String fishType;
    void swim() {
        System.out.println("鱼在游泳");
    }
    void breath() {
        System.out.println("鱼是用腮呼吸的");
    }
}
```

程序把 `Animal` 定义为抽象类，里面的 `breath` 方法被定义为抽象方法，而后面定义的 `Animal` 的子类 `Tiger` 和 `Fish` 都实现了 `breath` 方法。

### 6.2.2 抽象类的使用

定义完抽象类后，就可以使用它。但是抽象类和普通类不同，抽象类不可以实例化，如语句 `Animal animal = new Animal();` 是无法通过编译的，但是可以创建抽象类的对象变量，只是这个变量只能用来指向它的非抽象子类对象。示例如下。

```
public class UseAbstract
{
    public static void main(String[ ] args)
    {
        Animal animal1=new Fish();
        animal1.breath();
        Animal animal2=new Tiger();
        animal2.breath();
    }
}
```

程序中定义了两个 `Animal` 对象变量，一个存放 `Fish` 对象，另一个存放 `Tiger` 对象，分别调用这两个对象的 `breath` 方法。由于根本不可能构建出 `Animal` 对象，所以存放的对象仍然是 `Tiger` 对象，它会动态绑定正确的方法进行调用。

需要注意的是，尽管 `animal` 存放的是 `Tiger` 对象或是 `Fish` 对象，但是不能直接调用这些子类的方法，语句 `animal.swim();` 和 `animal2.tigerRun();` 都是不正确的。调用这项方法的时候仍然需要进行类型转换，正确的使用方法如下。

```
((Fish) animal1).swim();
((Tiger) animal2).tigerRun();
```

## 6.3 Object 类

Java 中存在一个非常特殊的类——Object 类，它是所有类的祖先类。在 Java 中如果定义了一个类并没有继承任何类，那么它默认继承 Object 类。而如果它继承了一个类，则它的父类，甚至父类的父类必然是继承自 Object 类，所以说任何类都是 Object 类的子类。

### 6.3.1 Object 对象

由于 Object 类的特殊性，所以在实际开发中，经常会使用到它。在本节中就简单地介绍一下如何使用 Object 类和如何使用类中的两个重要方法。定义一个 Object 对象，根据前面继承的知识，它可以存放任何类型，示例如下。

```
public class Test
{
    public static void main(String[ ] args)
    {
        //创建一个存放 Object 数据的数组
        Object [ ]object=new Object[3];
        Animal animal1 = new Fish();
        Animal animal2 = new Tiger();
        //将上边创建的对象存数 Object 数组
        object[0]=animal1;
        object[1]=animal2;
        object[2]=new String("String");
        //取出对象后需要进行类型转换才能调用相应类型的方法
        ((Fish) object[0]).swim();
    }
}
```

示例中用到的 Animal 类、Fish 类和 Tiger 类都在上一节中定义过了。可以把 3 个不同的对象放进 Object 数组中，但是放进去后对象的类型被丢弃了，取出后要进行类型转换。当然类型转换不像程序中这么简单，可能要使用到类型判断，即用 instanceof 运算符实现。

### 6.3.2 equals 方法和 toString 方法

在 Object 类中也定义了一系列的方法，读者可以阅读 API 自己查看。其中比较重要的两个方法就是 equals()方法和 toString()方法。在 Object 类中 equals 方法的定义形式如下。

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

当且仅当两个对象指向同一个对象的时候才会返回真值。程序要想进行详细的判断，必须自己进行 equals 方法的覆写。在前面的内容中，介绍了 String 类 equals 方法的实现。

```
public boolean equals(Object anObject) {
```

```

    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n-- != 0) {
                if (v1[i++] != v2[j++])
                    return false;
            }
            return true;
        }
    }
    return false;
}

```

程序首先判断两个对象是否指向同一个对象，如果是的话当然返回 `true`，如果不是继续判断提供的对象是否是 `String` 类型，如果不是的话返回 `false`，是的话再进一步判断；取出两个 `String` 类的每一个位置的字符进行判断，如果有一个不等则返回 `false`。

其实这是 `equals` 方法，这里模仿 `String` 的 `equals` 方法的实现提供一个 `Animal` 的 `equals` 方法，其实这是一种比较标准的 `equals` 方法编写方式。

```

public boolean equals(Object ob) {
    boolean bool = false;
    if (this == ob)
        bool = true;
    if (ob == null)
        bool = false;
    if (ob instanceof Animal) {
        bool = ((Animal) ob).age == this.age
            && ((Animal) ob).name == this.name
            && ((Animal) ob).type == this.type
            && ((Animal) ob).weight == this.weight;
    }
    return bool;
}
}

```

编写测试类如下。

```

public class Testequals {
    public static void main(String[] args) {
        //声明三个对象
        Animal animal1 = new Fish();
        Animal animal2 = new Tiger();
        Animal animal3 = new Fish();
        //对对象属性赋值
        animal1.age = 12;
        animal1.name = "dingding";
        animal1.type = "dog";
        animal1.weight = 12;
        animal2.age = 12;
    }
}

```

```

        animal2.name="dingding";
        animal2.type="dog";
        animal2.weight=12;
        animal3.age=12;
        animal3.name="dongdogn";
        animal3.type="dog";
        animal3.weight=12;
        //进行比较, 并打印出结果
        System.out.println(animal1.equals(animal2));
        System.out.println(animal1.equals(animal3));
    }
}

```

程序的执行结果如下。

```

true
false

```

测试成功。其实 Java 的源代码都是经过仔细斟酌的, 读者有时间可以多学习一下 Java 源码。对于 toString() 方法, Object 类是这样实现的。

```

public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

```

其中 getClass 方法是 Object 类提供的一个方法。

```

public final native Class<?> getClass();

```

它返回一个 Class 类型, 然后调用 Class 类的 getName 方法, 请读者查看 API 中 getName 的实现。下面编写一个 Animal 的 toString 方法。

```

public String toString() {
    String returnString = null;
    returnString = "名字: " + this.name + "\n" + "种类: " + this.type + "\n"
        + "年龄: " + this.age + "\n" + "体重: " + this.weight;
    return returnString;
}

```

把这段代码加入 Animal 类中, 测试代码如下。

```

public class TesttoString
{
    public static void main(String[ ] args)
    {
        Animal animal1 = new Fish();
        animal1.age = 12;
        animal1.name = "dingding";
        animal1.type = "dog";
        animal1.weight = 12;
        System.out.println(animal1.toString());
    }
}

```

程序的运行结果如下。

```

名字: dingding
种类: dog

```

年龄: 12  
体重: 12

## 6.4 小结

本章首先讲解了 Java 继承的实现及方法的覆写，读者要注意跟方法重载的区别；然后讲解了多态和动态绑定以及 `final` 关键字的使用；接着讲解了抽象类的使用；最后讲解了 `Object` 类，其中 `equals` 方法借鉴了 `String` 类中 `equals` 方法的实现。