

Chp7 面向对象三大特性

本章导读

面向对象三大特性指的是：封装、继承、多态。这三大特性支撑了整个面向对象的理论体系，是面向对象的核心。本章要介绍 Java 中这三大特性是如何体现的，以及与这三大特性相关的 Java 语法。

1 封装

有如下代码：

```
class CreditCard{
    String password = "123456";
}

public class TestCreditCard{
    public static void main(String args[]){
        CreditCard card = new CreditCard();
        System.out.println(card.password);
        card.password = "000000";
        System.out.println(card.password);
    }
}
```

上述代码，创建了一个信用卡对象，并且读取、修改了这个对象的 password 属性。

从 Java 基本语法上说，这并没有问题。但是对于生活来说，这就是一个大问题！对于信用卡对象而言，它的密码属性是不应该被随便访问和修改的。

面向对象中解决这个问题，可以采用封装的特性。封装指的是，任何对象都应该有一个明确的边界，这个边界对对象内部的属性和方法起到保护的作用。

1.1 属性的封装

为上述的 CreditCard 的 password 属性增加 private 关键字，如下：

```
class CreditCard{
    private String password = "123456";
}
```

则原有代码中会出现编译错误：

```
public class TestCreditCard{
    public static void main(String args[]){
        CreditCard card = new CreditCard();
        System.out.println(card.password); //编译错误
        card.password = "000000"; //编译错误
        System.out.println(card.password); //编译错误
    }
}
```

当为属性增加 `private` 之后，这个属性就成为了一个私有属性。所谓私有，指的是该属性只能在本类内部访问。例如，当我们把 `password` 属性设置为 `private`，对这个属性的访问就只能局限在 `CreditCard` 类的内部。现在我们试图在 `TestCreditCard` 类中访问这个属性，编译器就会报出编译错误。这就相当于，`card` 对象的边界对于 `password` 属性起到了保护的作用，任何试图越过边界，访问 `password` 属性的企图都会被阻止。

然而，对于用户而言，依然有可能要访问 `CreditCard` 的密码。例如，在生活中，如果忘了银行卡密码，我们可以凭借证件到银行去查询或重设密码。

对于这方面的需求，我们为 `CreditCard` 提供一对 `get/set` 方法。这两个方法的修饰符为“`public`”。用 `public` 修饰的属性和方法表示“公开的”，公开属性和方法不受对象边界的限制，在类的内部和外部都可以访问。代码如下：

```
class CreditCard{
    private String password = "123456";
    public void setPassword(String password){
        this.password = password;
    }
    public String getPassword(){
        return this.password;
    }
}
```

则 `TestCreditCard` 类可以改成：

```
public class TestCreditCard{
    public static void main(String args[]){
        CreditCard card = new CreditCard();
        System.out.println(card.getPassword() );
        card.setPassword("000000");
        System.out.println(card.getPassword());
    }
}
```

很显然，用户可以调用 `getPassword` 方法来获取 `password` 属性，调用 `setPassword` 方法来设置 `password` 属性。

下面是一个初学者常问的问题：既然提供了 `get/set` 方法就是为了访问属性，那又何必把属性作为私有？把属性做成公开的直接访问不行么？

把属性作为私有，并提供相应的 `get/set` 方法，最重要的概念在于：控制。由于不能直接访问属性，而必须通过 `get/set` 方法访问属性，因此可以在 `get/set` 方法上做手脚，来控制他人对对象属性的访问。

例如，希望 `password` 属性只能被获取，不能被改写。如果 `password` 属性用 `private` 修饰的话，可以只提供 `get` 方法而不提供 `set` 方法，这样 `password` 就成为了只读属性。而如果不把 `password` 做成私有，则无法达到“只读”的效果。

再例如，银行要求，信用卡的密码长度必须为 6，如果没有把 `password` 属性做成 `private` 的，那么下面的代码一定是正确的：

```
card.password="12345678";
```

我们就无法限制密码的长度了。而现在我们把 `password` 属性设置为 `private`，用户就只能通过 `setPassword` 方法来设置 `password` 属性：

```
card.setPassword("12345678");
```

我们就可以在 `setPassword` 方法中增加一个判断：密码长度为 6 才允许设置。

修改原有代码如下：

```
public void setPassword(String password) {
    if (password.length() != 6) return;
    this.password = password;
}
```

这样，如果密码长度不为 6 的话，设置密码就不会成功。

从上面一个例子中，我们可以看到，把属性设为 `private`，并提供相应的 `get/set` 方法，程序员才能够对属性的访问增加控制。因此，从现在开始，应当养成良好的习惯，把类的属性都做成 `private` 的，然后再提供相应的 `get/set` 方法。

1.2 方法的封装

除了属性之外，我们也可以将方法设置为 `private`。

在前面的章节中，读者往往会看到，一个方法会包含修饰符“`public`”，这表示该方法是公开的，不受对象边界的限制。而我们同样可以把方法修饰为“`private`”，与属性类似，一个被修饰为“`private`”的私有方法只能在类的内部访问。

我们在设计一个类的时候，会为此类设计很多方法。有些方法应该做成 `public` 方法，以供其他对象来调用，而有些方法只供自身调用，不作为对象对外暴露的功能，就应该做成 `private` 方法。例如，一个老师对象，拥有一个“讲课”方法，这个方法必须暴露出来，供学生对象来调用（老师从来不会讲课给自己听），因此这个方法应该是公开的。同时，老师作为一个人，还拥有“消化食物”方法，这个方法只供老师自己来调用，对别人是无益的，因此，应该是一个私有方法。

```
class MyClass{
    public void method1(){}
    private void method2(){}
}

public class TestPrivateMethod {
    public static void main(String[] args) {
        MyClass mc = new MyClass();
        mc.method1(); // 正确，method1方法为公开的，可以在类外面访问
        mc.method2(); // 编译失败，不能访问mc对象的私有方法
    }
}
```

如上述代码，`MyClass`类中具有两个方法，`method1`方法为`public`的，因此可以在类外面调用；`method2`方法为`private`的，不能在类的外部调用，否则会引发一个编译错误。

2 继承

2.1 继承的基本概念

继承是面向对象中另一个非常重要的概念。那继承的思想是怎么来的呢？我们首先从一个例子看起。

例如，我们要设计两个类：一个 `Dog` 类，一个 `Cat` 类，分别表示狗和猫。首先，我们来设计 `Dog` 类。首先考虑狗类的属性，也就是狗“有什么”。狗有年龄，有性别，因此狗就

有一个 `age` 属性和一个 `sex` 属性。其次，考虑狗类的方法，也就是狗“能干什么”。我们说，狗能够吃东西，狗也能够睡觉，狗还能够看家护院。因此，我们为狗设计三个方法：`eat()`、`sleep()`和 `lookAfterHouse()`。代码如下：

```
class Dog{
    int age;
    boolean sex; //true 表示雄性，false 表示雌性

    public void eat(){
        System.out.println("eat()");
    }

    public void sleep(){
        System.out.println("sleep()");
    }

    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
}
```

接下来，我们来设计 `Cat` 类。猫类有哪些属性呢？猫类有年龄，有性别，因此猫类有 `age` 和 `sex` 属性。那么猫有哪些方法呢？猫能够吃东西，猫能睡觉，猫还能抓老鼠。因此，我们为猫类设计三个方法：`eat()`、`sleep()`和 `catchMouse()`。代码如下：

```
class Cat{
    int age;
    boolean sex;

    public void eat(){
        System.out.println("eat()");
    }

    public void sleep(){
        System.out.println("sleep()");
    }

    public void catchMouse(){
        System.out.println("Cat can catch mouse");
    }
}
```

这样，我们就完成了 `Dog` 类和 `Cat` 类的设计。

我们仔细分析一下 `Dog` 类和 `Cat` 类的代码，会发现这两个类中有大量相同的代码。例如，在这两个类中，都有 `age` 和 `sex` 属性；同样的，都有 `eat()`和 `sleep()`方法。为什么这两个类有这么多类似的代码呢？这是由这两个类的特点决定的。在生活中，狗和猫这两类事物有着很多的共性，例如能“吃”，能“睡”。而这些正是“动物”这个类的特点。也就是说，狗

和猫，都是特殊的动物。为了能够更好的表达这个概念，我们把 Cat 和 Dog 的共性抽象出来，写在另外的一个类 Animal 中。这个类包含了 Cat 类和 Dog 类中的共性。

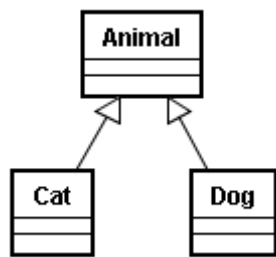
代码如下：

```
class Animal{
    int age;
    boolean sex;
    public void eat(){
        System.out.println("Animal Eat");
    }

    public void sleep(){
        System.out.println("sleep 8 hours");
    }
}
```

我们让 Animal 类的 eat 方法输出“Animal Eat”，让 Animal 的 sleep()方法输出：“sleep for 8 hours”（我们假设所有动物都是睡 8 个小时）。

然后，我们让 Cat 类和 Dog 类继承自 Animal 这个类。这样，Animal 类与 Cat 和 Dog 类之间，就形成了继承关系。而被继承的类 Animal，被称为“父类”；而 Cat 与 Dog 这两个类继承自 Animal，被称为“子类”。用图来表示如下：



我们如何来理解父类和子类呢？

在生活中，首先，我们见到了很多的猫对象、狗对象、猴子对象、大象对象等，从而形成了猫类、狗类、猴子类、大象类等这些类。这些都是具体的类，而当我们遇到大量的具体的类时，会经过总结和归纳，抽象出他们的共性，形成一个新的概念：动物。这样，我们在脑子中，就有了“动物”这个类的概念。这个类的出现并不是因为我们见到了“动物”对象，而是我们从大量具体的类中总结，归纳出来的。

上面我们讲述的过程，就是父类产生的过程：父类，是对子类共性的抽象。

我们也可以从另一角度来理解。从上面的代码中，我们可以看到，由于 Animal 中已经定义了 age、sex 属性以及 eat()、sleep()方法，因此在两个子类 Cat 和 Dog 中，这些共性不需要重复定义，只需要写出 Cat 和 Dog 类中的特性就可以了。因此，在父类中，往往可以定义一些比较一般的属性和方法，而在子类中，定义子类特有的属性或者方法。也就是说，父类和子类的关系，是由一般到特殊的关系。例如，Animal 是一般的类，而 Dog 和 Cat 是特殊的类，Dog 和 Cat 可以当做是特殊的 Animal。

由于父类是子类的共性的抽象，是一个一般的类，因此，我们在进行继承关系设计的时候，应当尽量把子类的共性放在父类，特性放在子类。

例如，有下面的继承关系：



这是一个非常典型的继承关系。生物，是一个很宽泛的概念，包括植物、动物等，都可以看做是生物。而动物，就是一种特殊的生物，因此，动物类是生物类的子类。同样的，哺乳动物是特殊的动物，而人又是特殊的哺乳动物。

现在，我们要设计一系列的方法：繁殖()，喂奶()，制造工具()。这三个方法应当分别写在哪一个类中呢？

繁殖是所有生物的共性，因此，繁殖()方法应当作为生物类的方法；喂奶是哺乳动物的共性，因此这个方法应当写在哺乳动物类中；而制造工具()是人类特有的方法，因此这个方法应当写在人这个类中。

上面的部分，我们简单介绍了一下继承的基本概念：父类是对子类共性的抽象，父类和子类的关系，是由一般到特殊的关系。在设计类的继承关系时，应当把共性放在父类，特性放在子类。

2.2 继承的基本语法

从语法上说，继承使用关键字：**extends**。在定义子类的时候，可以用 **extends** 关键字说明这个类的父类是哪一个类。代码如下：

```
class Animal{
    int age;
    boolean sex;
    public void eat(){
        System.out.println("Animal eat");
    }

    public void sleep(){
        System.out.println("sleep 8 hours");
    }
}

class Dog extends Animal{
    public void lookAfterHouse(){
        System.out.println("look after house");
    }
}
```

```

    }
}

class Cat extends Animal{
    public void catchMouse(){
        System.out.println("catch mouse");
    }
}

public class TestDog {
    public static void main(String args[]){
        Dog d = new Dog();
        d.sex = true;
        d.age = 3;
        d.eat();
        d.lookAfterHouse();
    }
}

```

我们可以看到,在代码中,使用 `extends` 关键字,来表明 `Cat` 类和 `Dog` 类继承自 `Animal`。由于共性在父类 `Animal` 中,因此,在 `Cat` 和 `Dog` 中, `Animal` 类中已经有的代码,没有必要进行重复。这样,程序中就少了很多重复和冗余的代码。与此同时,在主方法中,我们创建了一个 `Dog` 对象,并且,修改了这个 `Dog` 对象的 `sex` 属性和 `age` 属性,并调用了这个对象的 `eat()` 方法和 `lookAfterHouse()` 方法。

需要注意的是,在 `Dog` 类中,我们没有写代码来定义 `sex` 和 `age` 属性,也没有 `eat()` 方法的代码。这两个属性和一个方法,是 `Dog` 类从 `Animal` 类中继承而来的。也就是说,父类中的属性和方法,被子类继承之后,相当于子类中也有了相应的属性和方法。

这样,在父类中定义了属性和方法之后,子类中就能够直接继承,这样,就让父类中的代码得到了重用,从而提高了代码的可重用性。同时,子类也能够写一些子类的特性,这样,就在父类的基础上增加一些功能,体现了面向对象的可扩展性。

2.3 什么能被继承?

在上一节的例子中,我们在 `Animal` 类中定义了 `age` 和 `sex` 属性,这两个属性能够被子类 `Dog` 类和 `Cat` 类继承,并且我们在创建 `Dog` 对象之后,能够直接使用 `age` 和 `sex` 属性,也能够调用 `Animal` 类中定义的方法。

那是不是父类中所有的属性和方法,都能够被子类继承呢? 我们看下面的例子:

```

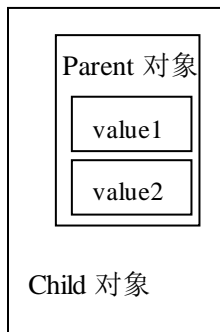
class Parent{
    int value1;
    private int value2;
}

class Child extends Parent{
    public void method(){
        value1 = 100; // 编译正确 Child 类从父类中继承到 value1 属性
        value2 = 200; // 编译错误, Child 没有继承到 value2 属性
    }
}

```

上面的例子中，Parent 类中的 value1 属性能够被 Child 类继承，所以在 Child 类的内部可以访问 value1 属性。但是，value2 属性被 private 关键字修饰，这就意味着，value2 只能在 Parent 类内部访问。对于 Parent 类来说，Child 类属于是 Parent 类的外部，因此不能够直接访问 value2 属性。

那么 Child 类中有没有 value2 属性呢？从空间上来说，在创建 Child 对象的时候，会在 Child 对象的内部，包含着一个 Parent 对象。在内存中的示意图如下：



可以看到，Child 对象的内部，包含了一个 Parent 对象。因此，在这个 Child 对象中，有两块数据区域，用来保存 parent 对象的 value1 属性和 value2 属性。但是，由于 value2 只能在 Parent 类内部访问，因此在 Child 对象中，无法访问 Parent 类中的 value2 属性。

也就是说，对于 value2 来说，在 Child 对象中有这个属性的空间，但是却无法访问。这种情况，我们就说 Parent 类中的 value2 属性无法被 Child 类继承。也就是说，父类中，无法被子类访问的属性和方法，不能被继承。

怎么来理解这一点呢？例如，一个百万富翁，死后留下了一大笔财产。有一部分财产，放在家里的抽屉中，他的儿子能够打开这个抽屉，取出这些钱。因此，这部分财产，他的儿子能够访问，因此上，可以认为他的儿子能够继承。

但是，如果还有一大笔钱，富翁存在了一个保险箱里面，这个保险箱的密码只有富翁本人知道。那么富翁死了之后，他的儿子算不算有这笔钱呢？从理论上说，他的儿子家里有一块空间用来存放这笔钱，他儿子也知道有这么一笔钱。但是，钱存在保险箱里面取不出来，那么这笔钱他儿子根本没法花，从实际效果来看，跟他儿子没有这笔钱一样。也就是说，这部分财产他的儿子无法访问，因此我们可以认为，他的儿子没有继承到这部分财产。

也就是说，只有子类能够访问的属性和方法，才能够被子类继承。

2.4 访问权限修饰符

上一节我们说到，只有能被子类访问的属性和方法，才能被子类继承。那么，怎么判断哪些属性和方法能够被访问，哪些不能被访问呢？这需要看属性或者方法的访问权限修饰符。

在 Java 中，总共有四种访问修饰符。除了我们之前介绍的 private 和 public 两个修饰符之外，还有两个跟访问权限相关的修饰符：default 以及 protected。要注意的是，default 修饰符指的是：如果在属性或方法前面，不加任何的访问修饰符（即不加 private、public、protected），则访问权限是 default 权限。例如：

```
class Student {
    int age;
}
```

在这个 Student 类中，其 age 属性没有加上任何访问修饰符，因此 age 属性的权限就是 default 权限。要注意的是，要把一个属性设为 default 时，千万不能写上“default”这个单

词。例如下面的代码就是错误的：

```
class Student{
    default int age; //编译错误!
}
```

private 和 **public** 这两个修饰符不再赘述。我们详细介绍一下 **default** 和 **protected** 修饰符。

访问权限为 **default** 的属性或者方法，只能被本类或者同包的其他类访问。例如，有下面的代码：

```
package p1; //MyClassA 类处于 p1 包下
public class MyClassA{
    int value = 10; //value 属性的访问权限是 default
    public void m(){
        System.out.println(value); //value 属性能够在本类内部访问
    }
}
```

//MyClassB 与 MyClassA 处于同一个包下

```
package p1;
public class MyClassB{
    public void m2(){
        MyClassA mca = new MyClassA();

        //由于在同一个包下
        //所以 MyClassB 中能够访问 MyClassA 对象的 value 属性
        mca.value = 100;
        System.out.println(mca.value);
    }
}
```

package p2; //MyClassC 与 MyClassA 不在同一包下

```
public class MyClassC{
    public void m3(){
        p1.MyClassA mca = new p1.MyClassA();

        //由于不在同一个包下
        //所以 MyClassB 中不能访问 MyClassA 对象的 value 属性
        mca.value = 200; // 编译错误!
    }
}
```

对于 **default** 的属性和方法而言，只有同包的类才能够访问。因此，只有那些和父类在相同包下的子类，才能够继承 **default** 修饰的属性和方法。例如下面的代码：

```
package p1;
public class Parent{
    int value = 20;
```

```

}

package p1; //与 Parent 同一个包
public class Child1 extends Parent{
    public void m1(){
        System.out.println(value); //继承了 Parent 的 value 属性
    }
}

package p2; //与 Parent 不同包
public class Child2 extends p1.Parent{
    public void m2(){
        //编译错误! Child2 类和 Parent 不同包, 没有继承 value 属性
        System.out.println(value);
    }
}

```

用 **protected** 修饰符修饰的属性和方法, 能够被本类内部、同包的类以及非同包的子类访问。首先, **protected** 修饰符的访问权限, 要比 **default** 的访问权限大。其次, 由于 **protected** 的属性和方法, 能够被同包的类访问, 因此, 必定能被同包的子类所继承。另一方面, 这样的属性和方法也能被非同包的子类访问, 因此, 用 **protected** 修饰的属性和方法, 能够被同包的子类和非同包的子类访问, 也就是能被所有的子类继承。换句话说, 用 **protected** 修饰的属性和方法一定能够被子类继承。

但是, 如果是非同包的子类, 则无法访问 **protected** 属性或方法。例如下面的例子:

```

package p1;
public class Parent{
    protected int value;
}

package p1;
public class SamePackage{
    public void m1(){
        Parent p = new Parent();
        System.out.println(p.value); //同包的类可以访问
    }
}

package p2;
public class Child extends p1.Parent{
    public void m2(){
        //非同包的子类可以访问父类的 value 属性
        System.out.println(value);
    }
}

```

```

package p2;
public class Other {
    public void m2() {
        p1.Parent p = new p1.Parent();
        //编译错误! Other 类和 Parent 类并不同包, 也没有继承关系, 不能访问
        System.out.println(p.value);
    }
}

```

上面就是四种访问修饰符的介绍, 我们对这四种访问权限修饰符总结如下:

修饰符	访问范围	是否能被子类继承
private	本类内部	不能被继承
(default)	本类内部+同包的其他类	能被同包的子类继承
protected	本类内部+同包的其他类+非同包的子类	能被继承
public	公开, 能被所有类访问	能被继承

这张表所列的四种访问权限修饰符, 按照“private→default→protected→public”的顺序, 访问权限依次变宽。

2.5 方法覆盖

之前的代码中, 我们为 `Animal` 写了一个 `sleep()` 方法。由于 `Dog` 类继承自 `Animal` 类, 因此 `Dog` 类中也有一个 `sleep()` 方法。调用这个 `sleep()` 方法时, 实际上调用的是 `Animal` 类中的 `sleep()` 方法。代码如下:

```

class Animal{
    public void eat(){
        System.out.println("Animal eat");
    }

    public void sleep(){
        System.out.println("sleep 8 hours");
    }
}
class Dog extends Animal{
    public void lookAfterHouse(){
        System.out.println("Dog can look after house");
    }
}
public class TestDog {
    public static void main(String args[]){
        Dog d = new Dog();
        d.sleep();
    }
}

```

```
    }  
}
```

运行结果如下：

```
sleep 8 hours
```

这样，在调用 `Dog` 类的 `sleep` 方法时，实际上调用的是 `Dog` 类从 `Animal` 类中继承来的 `sleep` 方法。

现在，我们有了新的需求。假设说，`Dog` 类与 `Animal` 这个类相比，有自己的特点。狗与大多数动物不同，因为它要看家护院，所以它的睡眠时间，比一般动物要短。假设说，狗每天睡觉都是睡 6 个小时。

从概念上说，`Animal` 类中有 `sleep` 方法，而 `Dog` 类中也有 `sleep` 方法。在上一章我们提到过，方法分为两个部分：方法的声明和方法的实现。其中，方法声明表示一个类“能做什么”，而方法实现表示“怎么做”。`Animal` 和 `Dog` 类中都有 `sleep` 方法，就可以理解为，`Animal` 能睡觉，而 `Dog` 也能睡觉。也就是说，父类和子类的 `sleep` 方法，在方法声明上是一致的。

但是，`Animal` 类中的 `sleep` 方法，输出“sleep 8 hours”，而我们希望 `Dog` 类中的 `sleep` 方法输出“sleep 6 hours”。这样，在方法声明相同的基础上，我们希望 `Dog` 类有一个和父类不同的特殊实现。为此，我们可以在 `Dog` 类中，再次定义 `sleep` 方法，将父类继承下来的 `sleep` 方法重新实现。代码如下：

```
class Dog extends Animal{  
    public void lookAfterHouse(){  
        System.out.println("Dog can look after house");  
    }  
    public void sleep(){  
        System.out.println("sleep 6 hours");  
    }  
}  
public class TestDog {  
    public static void main(String args[]){  
        Dog d = new Dog();  
        d.sleep();  
    }  
}
```

运行结果如下：

```
sleep 6 hours
```

在上面的代码中，我们在 `Dog` 类中，为 `sleep` 方法给出了一个新的实现，来替换从 `Animal` 类中继承的 `sleep` 方法的实现。像这样，子类中用一个特殊实现，来替换从父类中继承到的一般实现，这种语法叫做“方法覆盖”。

这样，运行代码时，程序输出：sleep 6 hours。说明程序运行时，对 `Dog` 对象调用 `sleep` 方法时，真正调用的是 `Dog` 类覆盖以后的方法。

从语法上说，方法覆盖对方法声明的五个部分都有要求。具体来说，

1. 访问修饰符相同或更宽

例如，父类的方法如果是 `protected` 方法，子类如果想要覆盖这个方法，则修饰符至少是 `protected`，也可以是 `public`，但是不能是 `default` 或者 `private` 的。

例如下面的例子：

```
class Super{
    protected void m() {}
}
class Child extends Super{
    void m() {} //编译错误!
}
```

由于子类中试图用一个 default 权限的 m 方法覆盖父类中 protected 权限的 m 方法,会让上面的代码会产生一个编译错误。错误如下：

```
D:\Book\chp7>javac TestOverride.java
TestOverride.java:5: m() in Child cannot override m() in Super; attempting to as
sign weaker access privileges; was protected
    void m(){}
        ^
1 error
D:\Book\chp7>
```

2. 返回值类型相同。

如果返回值类型不同,则会产生一个编译错误。例如下面的代码

```
class Parent{
    public void m() {}
}

class Child extends Parent{
    public int m(){
        return 0;
    }
}
```

在这段代码中,子类的覆盖方法返回值为 int 类型,而由于父类方法的返回值为 void 类型,子类和父类方法的返回值不同,因此子类方法无法覆盖父类方法,所以会产生一个编译错误。错误如下。

```
D:\Book\chp7>javac ErrorOverride.java
ErrorOverride.java:6: m() in Child cannot override m() in Parent; attempting to
use incompatible return type
found   : int
required: void
    public int m(){
            ^
1 error
D:\Book\chp7>
```

3. 方法名相同。这是必然的,如果方法名不同,则谈不到覆盖。

4. 参数表相同。

要注意的,如果不满足参数表不同,编译不出错,但是不构成方法覆盖。例如

```
class Super{
    void m(){System.out.println("m in super");}
}
```

```
class Sub extends Super{
    void m(int n){System.out.println("m(int) in sub");}
}
```

这个代码能够编译通过。但是，虽然 **Sub** 中定义了一个 **m** 方法，但是这个方法并没有替换父类中的 **m** 方法的实现。相反，由于 **Sub** 类继承自 **Super** 类，因此 **Sub** 类中会继承到一个无参的 **m()** 方法；而此外，**Sub** 类中还定义了另一个带 **int** 参数的 **m(int n)** 方法。这样，**Sub** 类中就有了两个 **m** 方法，这两个方法方法名相同，参数表不同，因此，这样两个方法构成重载。所以，这样的代码编译不会出错，但是 **Sub** 类中的 **m** 方法并没有覆盖 **Super** 类中的 **m** 方法。

要注意的是，方法覆盖并不仅仅包含这些条件。方法覆盖还对包括 **static** 修饰符的要求、对抛出的异常的要求，等等。相关内容会在后续的课程中陆续进行介绍。

2.6 对象创建的过程

在有了继承关系之后，对象创建过程就变的相对复杂一些了。由于子类对象中包含一个父类的对象(虽然，父类对象中的有些属性无法访问，但是还是会创建一个完整的父类对象)，因此创建子类对象时必然要先创建父类对象。

在有了继承关系之后，对象创建过程如下：

1. 分配空间。要注意的是，分配空间不光是指分配子类的空间，子类对象中包含的父类对象所需要的空间，一样在这一步统一分配。在分配空间的时候，会把所有的属性值都设为默认值。
2. 递归的构造父类对象。这一过程我们会在下面进一步介绍
3. 初始化本类属性。
4. 调用本类的构造方法。

我们下面结合一个具体的例子来介绍对象创建的过程。

假设有如下代码：

```
class A{
    int valueA = 100;
    public A(){ valueA=150; }
}
class B extends A{
    int valueB = 200;
    public B(){ valueB=250; }
}
class C extends B{
    int valueC = 300;
    public C(){ valueC=350; }
}
public class TestInherit{
    public static void main(String args[]){
        C c = new C();
    }
}
```

```
}  
}
```

我们在主方法中创建了一个 C 对象，则创建时的过程如下。

1. 分配空间。在分配空间时，会把 C、B、A 这三个对象的空间一次性都分配完毕，然后把这三个对象的属性都设为默认值。这样，value1，value2，value3 这三个属性都被设置为 0
2. 递归构造 C 对象的父类对象。在这里，要 C 对象的父类对象，就是 B 对象。因此，在这一步需要创建一个 B 对象。
3. 初始化 C 的属性，即把 valueC 赋值为 300
4. 调用 C 的构造方法。

其中，第 2 步，C 对象的父类为 B 对象，因此必须要先创建一个 B 对象。创建 B 对象不用重新分配空间，需要以下几步：

- 2.1 递归的构造 B 对象的父类对象
- 2.2 初始化 B 属性：把 valueB 赋值为 200
- 2.3 调用 B 的构造方法。

在 2.1 这个步骤中，递归的创建 B 对象的父类对象，也就是创建 A 对象。创建 A 对象不需要分配空间，因此，A 对象的创建有这样几步：

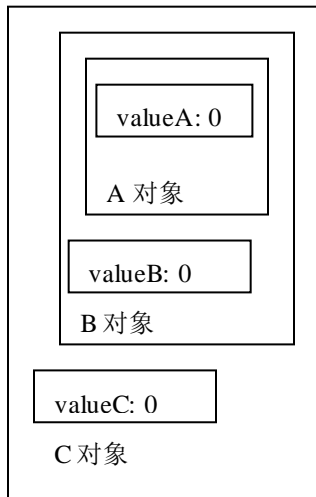
- 2.1.1 创建 A 对象的父类对象。这一步在运行时，没有任何的输出。
- 2.1.2 初始化 A 的属性，把 valueA 赋值为 100
- 2.1.3 调用 A 的构造方法。

总结一下：创建 C 对象的步骤一共有 7 步：

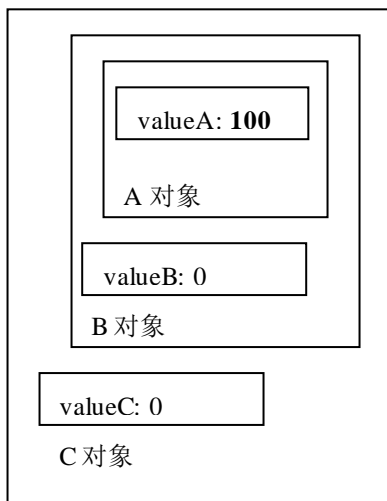
1. 分配空间
2. 初始化 A 类的属性
3. 调用 A 类的构造方法
4. 初始化 B 类的属性
5. 调用 B 类的构造方法
6. 初始化 C 类的属性
7. 调用 C 类的构造方法

每个步骤的内存示意图如下：

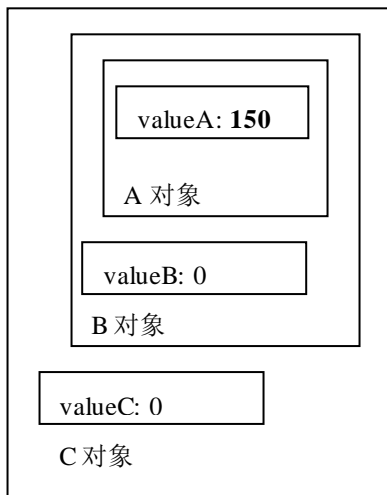
1. 分配空间，如之前所述，在一次分配空间时，会把整个继承关系中涉及到的类所需要的空间，都分配完毕，并把所有属性都设为默认值 0。如下图所示：



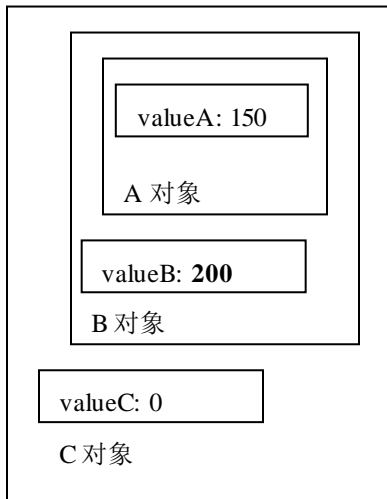
2. 初始化 A 的属性，把 valueA 赋值为 100。如下：



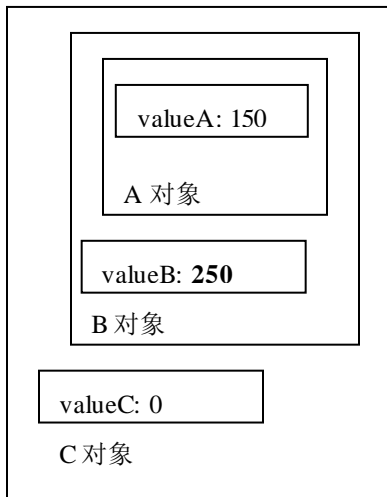
3. 调用 A 的构造方法，此时，会把 valueA 的值设为 150。如下图：



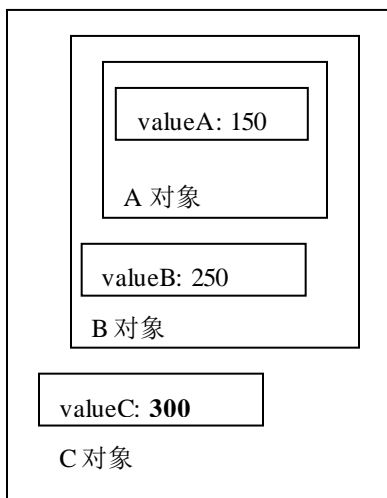
4. 初始化 B 属性：把 vauleB 赋值为 200。如下图：



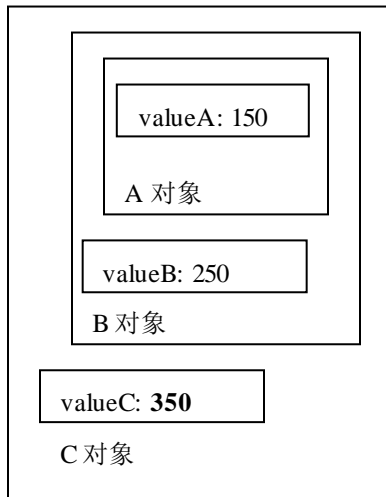
5. 调用 B 的构造方法，此时，会把 valueB 的值设为 250。如下图



6. 初始化 C 的属性，即把 valueC 赋值为 300。如下图：



7. 调用 C 的构造方法，此时，会把 valueC 的值设为 350。如下图



这就是一个完整的创建对象的过程。

2.7 super 关键字

在介绍对象创建的过程时，我们介绍了在创建对象时，会创建该对象的父类对象。而在创建父类对象的时候，很显然会调用父类的构造方法。但是，如果父类有多个构造方法，会调用哪一个呢？

例如下面的代码：

```
class Parent{
    public Parent(){
        System.out.println("Parent()");
    }

    public Parent(String str){
        System.out.println("Parent(String)");
    }
}

class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
    }
}

public class TestInheritConstructor{
    public static void main(String args[]){
        Child c = new Child("Hello");
    }
}
```

在上面的代码中，我们创建了一个 **Child** 类型的对象。在创建这个对象的时候，会首先创建一个 **Parent** 对象，并且调用 **Parent** 类中某一个构造方法。**Parent** 类中定义了一个无参的构造方法，也定义了一个带字符串参数的构造方法。那么，我们在调用父类的构造方法时，

会调用哪一个构造方法呢？

运行上面的代码，得到结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java

D:\Book\chp7>java TestInheritConstructor
Parent<>
Child(String)

D:\Book\chp7>_
```

可以看到，在创建 Child 对象时，首先会创建一个 Parent 对象，并且调用 Parent 对象的无参构造方法。也就是说，在默认情况下，创建子类对象时，都会调用父类的无参构造方法。

但是，在父类中我们定义好了带字符串参数的构造方法，能不能要求 Java 在创建 Parent 对象时，调用父类带字符串参数的构造方法呢？

为了解决这个问题，我们为大家介绍 super 关键字。super 关键字有两种不同的用法。

super 关键字用法一：super 用在构造方法上

super 关键字的第一种用法，就是可以指定在递归构造父类对象的时候，调用父类的哪一个构造方法。

例如，我们要指定，在创建 Child 对象时，会自动创建 Child 的父类对象：Parent 对象，在这时，我们希望能够调用 Parent 类中带字符串参数的构造方法。这时，我们就可以在 Child 类的构造方法中，加上一个语句：super(str)。修改后的 Child 类代码如下：

```
class Child extends Parent{
    public Child(String str){
        super(str);
        System.out.println("Child(String)");
    }
}
```

可以看到，在 Child 的构造方法中，我们用 super(str)，来指定要调用父类哪一个构造方法：我们在 super 后面的圆括号中传入了一个字符串参数，这就意味着我们希望调用父类中带字符串参数的构造方法。

修改后的代码运行结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java

D:\Book\chp7>java TestInheritConstructor
Parent(String)
Child(String)

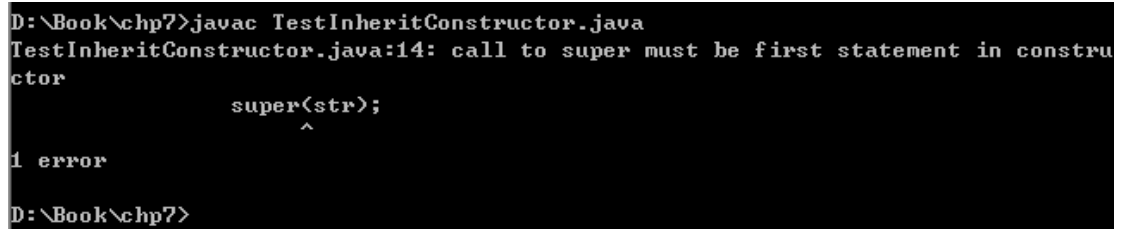
D:\Book\chp7>_
```

需要注意的是，虽然 super() 是写在子类的构造方法中，但是这并不意味着能够在子类中调用父类的构造方法。我们在这里写 super()，是提示 Java，在创建父类对象时调用哪一个构造方法。编译器在进行编译的时候，就会记住我们的这个提示。这样在创建一个子类对象的时候，当执行到“调用父类构造方法”那一个步骤时，就会根据我们之前写的 super，来选择调用父类中某一个构造方法。

要格外注意的是，`super` 用在构造方法中时，只能作为构造方法的第一句。例如，上面的 `Child` 代码，如果不把 `super` 作为构造方法的第一个语句时，就会产生一个编译错误。代码如下：

```
class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
        super(str);
    }
}
```

编译时的出错信息如下：



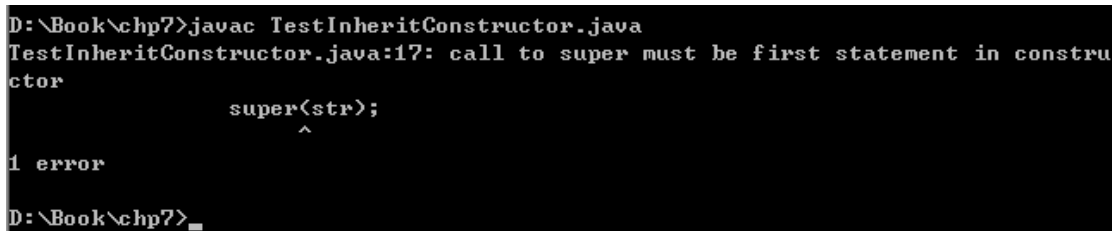
```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:14: call to super must be first statement in constructor
        super(str);
        ^
1 error
D:\Book\chp7>
```

然而，我们曾经介绍过，`this` 关键字可以在构造方法中，指明调用本类的其他构造方法。并且，对 `this()` 来说，这个语句也只能作为构造方法的第一个语句。

这样，在构造方法中，就不能够既使用 `this()`，又使用 `super()`。例如下面的代码：

```
class Child extends Parent{
    public Child(int n){
        System.out.println("Child(int)");
    }
    public Child(String str){
        this(10);
        super(str);
        System.out.println("Child(String)");
    }
}
```

这段代码编译的结果如下：



```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:17: call to super must be first statement in constructor
        super(str);
        ^
1 error
D:\Book\chp7>
```

编译器提示，对 `super` 的调用必须是构造方法中的第一个语句。那如果把 `super` 放到 `this()` 的前面呢？例如下面的代码：

```
public Child(String str){
    super(str);
    this(10);
    System.out.println("Child(String)");
}
```

此时编译代码，结果如下：

```
D:\Book\chp7>javac TestInheritConstructor.java
TestInheritConstructor.java:17: call to this must be first statement in constructor
        this(10);
        ^
1 error
D:\Book\chp7>_
```

编译器提示，对 `this` 的调用必须是构造方法中的第一个语句。

通过上面的例子说明，`this()`和 `super()`，在构造方法中不能同时使用。

这样，我们构造方法的第一个语句，就有了三种可能

1. `super(参数)` 指明调用父类哪个构造方法
2. `this(参数)` 指明调用本类哪个构造方法
3. 既不是 `this(参数)`又不是 `super(参数)`。

在第 3 种情况下，编译器就会自动加上一句“`super()`”，即调用父类的无参构造方法。

例如，在一开始的例子中，`Child` 类代码如下：

```
class Child extends Parent{
    public Child(String str){
        System.out.println("Child(String)");
    }
}
```

此时，在 `Child` 类的构造方法中，第一句既不是 `this(参数)`，也不是 `super(参数)`，因此，编译器会自动在这个构造方法中增加一个语句：“`super()`”，代码如下：

```
class Child extends Parent{
    public Child(String str){
        super(); //这句代码是系统默认添加的
        System.out.println("Child(String)");
    }
}
```

这也就解释了，为什么在默认情况下，创建父类对象时会调用父类的无参构造方法。

掌握了在构造方法中如何使用 `super` 关键字之后，我们可以看下面这个例子。

有如下代码：

```
class Super{
}

class Sub extends Super{
    public Sub(){}
    public Sub(String str){
        super(str);
    }
}
```

上面的代码，应当如何修改才能编译通过？

首先，我们可以编译上述代码。出错信息如下：

```
D:\Book\chp7>javac TestSuper.java
TestSuper.java:7: cannot find symbol
symbol   : constructor Super<java.lang.String>
location: class Super
super(str);
^
1 error
D:\Book\chp7>
```

编译器提示，找不到一个 Super 类中，带字符串参数的构造方法。我们在 Sub 类中的构造方法里，使用 super(str)这个语句来调用了 Super 类中带字符串参数的构造方法。而 Super 类中显然缺少这个构造方法。

接下来，我们为 Super 类增加一个构造方法，代码如下：

```
class Super{
    public Super(String str){}
}
```

```
class Sub extends Super{
    public Sub(){}
    public Sub(String str){
        super(str);
    }
}
```

此时，再编译代码，依然出错。出错信息如下：

```
D:\Book\chp7>javac TestSuper.java
TestSuper.java:6: cannot find symbol
symbol   : constructor Super<>
location: class Super
public Sub(){}
         ^
1 error
D:\Book\chp7>
```

此时，编译器显示，找不到 Super 类的无参构造方法。

我们在什么地方调用了 Super 类的无参构造方法呢？注意，在 Sub 类的构造方法中：

```
public Sub(){}

```

这个构造方法中，没有写任何语句，因此，第一句既不是 this 又不是 super，此时编译器会自动加上一句：“super();”。因此，在 Sub 类的这个构造方法中，调用了 Super 类的无参构造方法。

既然在这儿调用了 Super 类的无参构造方法，那为什么第一次编译没有提示缺少无参构造方法呢？

要注意，当我们把 Super 类写成如下形式时：

```
class Super{}
```

此时，在 Super 类中没有定义任何构造方法，因此，编译器就会自动生成一个公开的、无参的空构造方法。因此，在 Sub 类中调用父类的无参构造方法时，能够找到这个无参的

构造方法，编译能够通过。

而当我们为 **Super** 类增加了一个字符串参数的构造方法后，编译器就不会自动生成这个无参构造方法，因此就会产生找不到 **Super** 类无参构造方法的错误。

因此，如果想要让代码编译通过，必须为 **Super** 类增加两个构造方法。代码如下：

```
class Super{
    public Super(String str){}
    public Super(){}
}

class Sub extends Super{
    public Sub(){}
    public Sub(String str){
        super(str);
    }
}
```

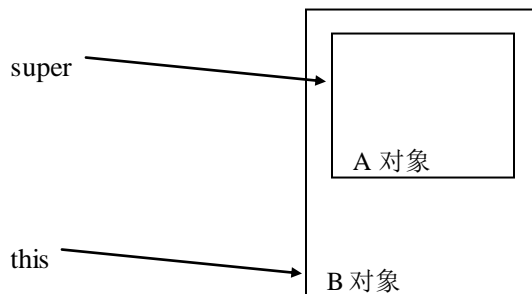
super 关键字用法二：super 用作引用

super 关键字的第二种用法，就是把 **super** 当做是一个引用，这个引用指向父类对象。例如，有如下代码：

```
class A{
}

class B extends A{
}
```

则创建一个 **B** 对象时，**B** 对象内部，会包含一个父类对象：**A** 对象。同时，**B** 对象内部会有两个引用：**this** 和 **super**。其中 **this** 引用指向当前的 **B** 对象，而 **super**，则指向 **B** 对象内部的 **A** 对象。内存中示意图如下：



这样，**super** 引用就可以使用 **A** 父类对象中的属性和方法。最典型的用途是，使用 **super** 在子类中，调用父类被覆盖的方法。例如：

```
class Parent{
    public void m(){
        System.out.println("m in Parent");
    }
}

class Child extends Parent{
    public void m () {
```

```

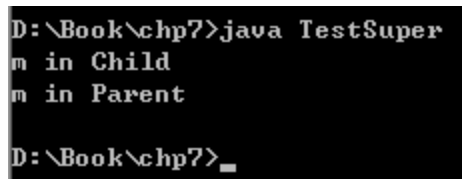
        System.out.println("m in Child");
    }
    public void m1 () {
        this.m();
    }
    public void m2 () {
        super.m();
    }
}

public class TestSuper {
    public static void main (String args []) {
        Child c = new Child();
        c.m1();
        c.m2();
    }
}

```

子类 **Child** 覆盖了父类 **Parent** 中的 **m** 方法。在子类的 **m1** 方法中，调用了 **this.m()** 方法。由于 **this** 指向当前对象，因此这样，调用的是 **Child** 类中定义的 **m** 方法。当然，如果直接写 **m()** 而省略 **this** 的话，调用的同样是当前对象的 **m** 方法。

而 **Child** 类中 **m2** 方法，则利用 **super** 关键字，调用了 **Parent** 类中，被覆盖之前的 **m()** 方法。运行结果如下：



```

D:\Book\chp7>java TestSuper
m in Child
m in Parent
D:\Book\chp7>_

```

super 可以调用父类被覆盖的方法，这种特性在实际编程中有着广泛的应用。例如，有一个 **Server** 类，这个类代表一个服务器。其中，有一个 **startService()** 方法，这个方法的代码非常非常复杂，并且写的也相对比较成熟。遗憾的是，这个代码有一个缺点：无法记录日志文件。代码大概是这个样子：

```

class Server {
    public void startService () {
        //很长的代码
        //很复杂的逻辑
        //但是没有保存日志
    }
}

```

如果我们直接在 **Server** 类上进行修改的话，就会在 **startService** 这个复杂的方法中增加更多的复杂性，也降低了代码的可读性以及可重用性。

为此，我们可以创建一个新的类：**MyServer**，让这个类继承自 **Server** 类，并且在 **MyServer** 中，覆盖 **Server** 类的 **startService()** 方法。

但是，如果让 **MyServer** 从头再把 **startService** 方法写一遍，则完全没必要。我们可以在 **MyServer** 的 **startService** 方法内部，利用 **super** 关键字，调用父类中的 **startService()** 方法。然

后，在根据需求，添加上保存日志的逻辑。代码示意如下：

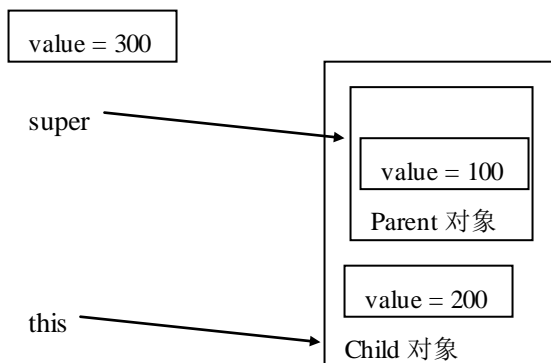
```
class MyServer extends Server{
    public void startService(){
        super.startService();
        //保存日志的功能
    }
}
```

因此，在实际开发中，如果要增强一个类中某一个方法的功能，可以继承这个类，然后覆盖这个方法。在覆盖时，可以利用 `super` 关键字调用父类中的实现，然后在父类实现的基础上，增加子类特有的新功能。

`super` 关键字除了可以用来调用方法之外，也可以用来指向属性。例如下面的代码：

```
class Parent{
    int value = 100;
}
class Child extends Parent{
    int value = 200;
    public void print(){
        int value = 300;
        //...
    }
}
```

在这个代码中，我们定义了两个类：一个类是 `Parent`，一个类是 `Child`。这两个类中，都包含一个 `value` 属性，一个值为 100，另一个值为 200。此外，在 `print` 方法中，还定义了一个局部变量 `value`，值为 300。内存中的示意图如下：



此时，在内存中有三个 `value` 变量。如果在 `print` 方法中直接打印 `value` 变量，则由于局部变量优先，会打印 300 的值。如果想要打印 `Child` 中的 `value` 属性，输出 200，则应当使用 `this.value`。类似的，如果想要打印 `Parent` 中的 `value` 属性，则应当使用 `super.value`。

完整代码如下：

```
class Parent{
    int value = 100;
}
class Child extends Parent{
    int value = 200;
```

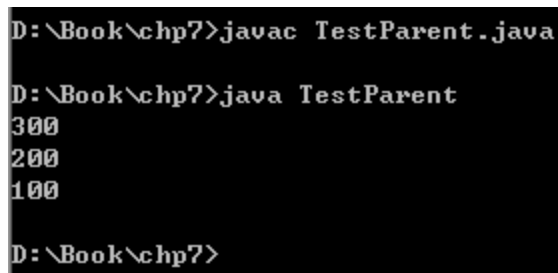
```

        public void print(){
            int value = 300;
            System.out.println(value);
            System.out.println(this.value);
            System.out.println(super.value);
        }
    }

    public class TestParent{
        public static void main(String args[]){
            Child c = new Child();
            c.print();
        }
    }
}

```

输出结果如下：



```

D:\Book\chp7>javac TestParent.java

D:\Book\chp7>java TestParent
300
200
100

D:\Book\chp7>

```

2.8 单继承

Java 语言规定，每一个类只能有一个直接父类。这就是 Java 语言的“单继承”规则。也就是说，我们不能试图让一个类去同时继承两个以上的类。如下面的代码是错误的：

```

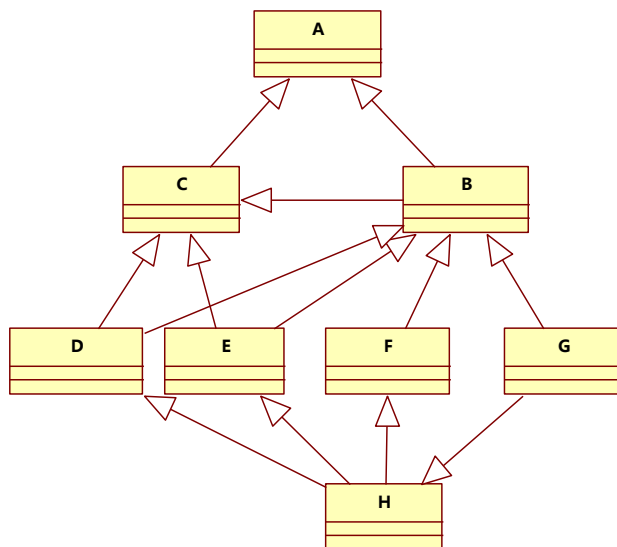
class A{ }
class B{ }
class C extends A,B{ } // 编译出错，一个类不能同时有两个直接父类

```

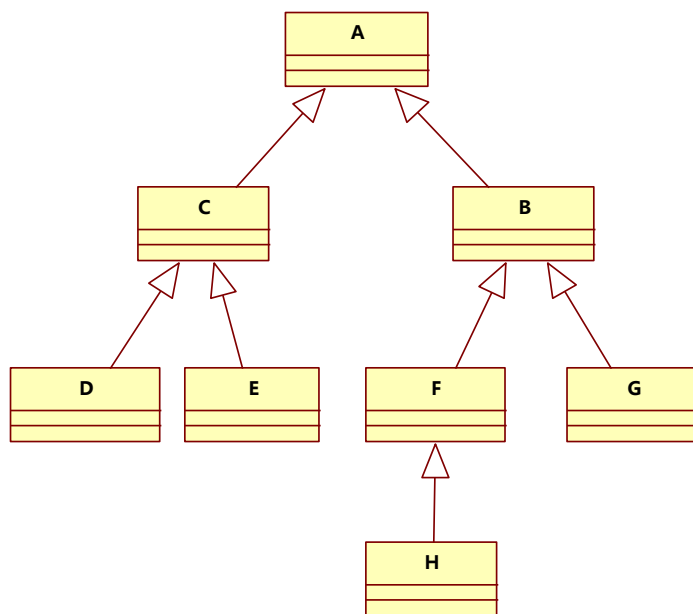
对于 C 类来说，它要么继承 A 类，要么继承 B 类，不能同时继承这两个类。这就是 Java 中的“单继承”。

我们知道，Java 语言的设计极大程度上借鉴了 C++ 语言。但是这两种语言在继承关系上有着重大的区别。C++ 语言是允许“多继承”的，也就是说，完全可以让一个类同时去继承多个类。

对于多继承来说，一个典型的类继承关系如下图：



而对于单继承来说，类继承关系如下图所示：



很显然，Java 中的类关系比较简单。这是因为，在多继承关系下，类之间会形成错综复杂的网状结构。而相比之下，由于单继承的原因，Java 中的类关系只会形成树状结构。树状结构自然会比网状结构要简单的多。

因此，Java 语言中单继承的特性，被认为是 Java 语言相对于 C++ 语言，“简单性”的一个重要体现。

3 多态

多态是面向对象三大特性中，最为重要也是最为灵活的一个特性。多态部分的讨论，是基于以下的代码：

```
class Animal{
```

```

        public void eat(){
            System.out.println("Animal eat");
        }
        public void sleep(){
            System.out.println("sleep 8 hours");
        }
    }
    class Dog extends Animal{
        public void sleep(){
            System.out.println("sleep 6 hours");
        }
        public void lookAfterHouse(){
            System.out.println("Dog can look after house");
        }
    }
    class Cat extends Animal{
        public void catchMouse(){
            System.out.println("Cat can catch mouse");
        }
    }
}

```

这段代码在“继承”部分中曾经分析过，Animal 类代表一般的动物，具有“eat”和“sleep”两种行为，而 Dog 类代表狗，作为特殊的动物继承 Animal 类，除了添加自身特有的“lookAfterHouse”方法之外，狗类还覆盖了“sleep”方法。Cat 类代表猫，也是 Animal 类的子类。除了从 Animal 类中继承两个方法之外，猫类还添加了“catchMouse”方法。

3.1 引用类型和对象类型

我们来看下面的代码：

```

01 : Animal a ;
02: a = new Animal();

```

在这两行代码中，分别出现了“Animal”字样。但是含义却不相同：

第一行代码中，我们定义了一个引用 a，而约束这个引用的类型为 Animal。我们知道，Java 语言是一个强类型的语言，在定义变量的时候，必须指定变量的类型。变量类型和变量中存放的数据类型必须匹配。这样，a 引用中只能存放 Animal 类型的对象。在这里，我们称“a 引用的引用类型为 Animal”。

第二行代码中，我们创建了一个 Animal 类型的对象，将这个对象的地址赋给 a 引用。每当我们创建对象时，总要指定这个对象的类型。对象的类型我们会写在“new”关键字的后面。在这里，我们称“a 引用所指向的对象类型为 Animal”。

也就是说，我们在定义引用的时候，为引用指定“引用类型”。而将对象放入引用的时候，引用中的对象还有一个“对象类型”。

以我们目前的知识程度，我们可以认为：引用类型和对象类型必须是一致的，这是 Java 语言强类型的约束。例如，下面的代码是正确的：

```

Dog d = new Dog();
d 引用的引用类型为 Dog，就必须存放对象类型为 Dog 类的对象。

```

而下面的代码是错误的：

```
Dog d = new Cat();
```

d 引用中只能存放 **Dog** 类的对象，而不能存放 **Cat** 类的对象。

而利用多态，下面的代码也是正确的：

```
Animal d = new Dog();
```

这是为什么呢？我们知道，**Dog** 类是 **Animal** 的子类。也就是说，**Dog** 类是特殊的 **Animal** 类。进而我们可以认为，一个 **Dog** 类的对象就是一个特殊的 **Animal** 类的对象。只要两个类之间存在继承关系，子类的对象就一定可以看作是特殊的父类对象。例如：汽车对象是特殊的交通工具对象；桌子对象是特殊的家具对象等等。

在上述代码中，**d** 引用是 **Animal** 类型的，只能存放 **Animal** 对象。而 **Dog** 对象也可以看作是特殊的 **Animal** 对象。那么一个 **Dog** 对象，当然就可以放入 **Animal** 类型的引用中。这并不违反 Java 语言强类型的限制。

因此我们可以得出结论：子类的对象可以放入父类的引用中！

也就是说，一个引用的引用类型和对象类型未必完全一致，对象类型可以是引用类型的子类。当我们看到一个引用时，一方面要考察这个引用的引用类型，另一方面还要考察这个引用中所存储的对象的类型，这两个类型可能是不同的。

3.2 多态的语法特性

在上一章节中，我们介绍了，子类的对象可以存放在父类的引用中，这是多态语法的根本出发点。

我们设想这样一个场景：小强是一个 2 岁的小朋友，在他幼小的意识中，从来没有见过“狗”这种东西。但是，他非常明确什么是“动物”。也知道动物会“吃”，会“睡”。因此，当他第一次见到一只狗的时候，他奶声奶气的问妈妈“这是什么动物？”。也就是说，小强认为，出现在他眼前的是一个动物类的对象。

我们以前提到过，类是人对客观对象的认识。很显然，在小强的头脑中，已经建立起了“动物”类的概念，但是却没有建立起“狗”类的概念。因此，当他面对一个狗类的对象时，他把它当做是一个动物类的对象。这并没有错，因为狗对象确实可以看作是动物对象。

我们可以认为，小强的这种思维过程可以用下面的代码来描述：

```
Animal a = new Dog();
```

我们可以这样理解，引用类型代表着一种“主观类型”。把狗类的对象放入动物类型的引用中，就意味着，主观上，把一个狗对象看作是一个动物对象。而这事实上就是多态的典型应用。

我们结合这个例子来具体分析多态的语法。总结起来，多态的语法主要体现在以下几点：

1. 对象类型永远不变

一个对象在创建的时候，其对象类型就已经决定了。直到这个对象消亡，它的对象类型永远不会改变。在多态的语法中，一个对象可能存放在不同类型的引用中，但是请注意，对象自身的类型从来也不会发生变化。例如在小强的眼中，这个狗对象被当作了一个动物对象，但这个对象实际上还是一只狗，这是不会变化的。

这一点很好理解，一个对象，它是狗就是狗，是猫就是猫，如果你认为它是狗它就是狗，你认为它是猫它就是猫，这显然是不合理，不“唯物”的。

4 只能对一个引用调用其引用类型中定义的方法

在小强的眼中，这个对象是一个动物对象，而他知道，动物会“吃饭”，会“睡觉”。也

就是说，在动物类中定义了 `eat` 方法和 `sleep` 方法。那么小强自然可以对这个对象调用 `eat` 和 `sleep` 方法。

但是，出现在小强眼前的对象实际上是一个狗对象，狗类中不仅有 `eat` 方法和 `sleep` 方法，还有一个 `lookAfterHouse` 方法。也就是说，狗还会看家护院。那么小强能否调用这个方法呢？显然不行，尽管这个对象确实具有这个方法，但是小强并不清楚这一点。

在实际生活中，我们能够对一个对象调用什么方法，这并不取决于这个对象具有什么方法，而主要取决于，我们“知道”这个对象具有什么方法。例如，同样一个手机对象，在有些人看来，这只是一个“电话”对象，因此，这些人只会调用该对象的“打电话”，“接电话”方法。而对于另外一些人来说，可能还会调用手机的“拍照”，“玩游戏”，“上网”等方法。其实这些功能也是手机对象所具备的。

我们来看以下的代码：

```
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a = new Dog(); //将一个狗对象看作是一个动物对象
        a.eat(); // 正确
        a.sleep(); // 正确
        a.lookAfterHouse(); // 编译错误
    }
}
```

当小强把 `Dog` 对象放入 `Animal` 类型的引用时，只能对这个引用调用 `eat` 方法，`sleep` 方法，而当他试图调用 `lookAfterHouse` 方法时，将会得到一个编译错误。因为他把狗对象看作是一个动物对象，他并不了解这个对象还具有 `lookAfterHouse` 方法。

也就是说，当我们对一个引用调用方法时，只能调用这个引用的引用类型中定义的方法。例如在上述代码中，`a` 引用的引用类型为 `Animal`，而 `Animal` 类中定义了 `eat` 方法和 `sleep` 方法，因此我们可以对 `a` 引用调用这两个方法。而由于 `Animal` 类中没有定义 `lookAfterHouse` 方法，我们就无法对 `a` 引用调用这个方法。

5 运行时，根据对象类型调用子类覆盖之后的方法

通过代码，我们看到：`Animal` 类中定义了 `sleep` 方法，打印“sleep 8 hours”。而 `Dog` 类作为 `Animal` 的子类，在 `sleep` 方法的实现方式上有自己独特的实现，因此，`Dog` 类覆盖了 `Animal` 类中的 `sleep` 方法，打印“sleep 6 hours”。

我们再来思考那个场景。小强是知道眼前的对象具有“睡觉”方法的，因此，他可以调用这个对象的“睡觉”方法。下面的代码是能够编译通过的：

```
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a = new Dog(); //将一个狗对象看作是一个动物对象
        a.sleep(); // 对这个对象发出“睡觉”的指令
    }
}
```

而运行时，这段代码会打印什么呢？

我们可以这样来理解这个问题，一般的动物一天睡 8 个小时，而狗作为特殊的动物，一天睡 6 个小时。那么，当小强对眼前这个狗对象发出“睡觉”指令的时候，这个对象会睡几个小时呢？当然是 6 个小时！这与小强把这个对象看作狗还是看作动物是无关的，这个对象实际上就是一只狗，它当然会按照自己的方式去睡。

代码运行结果如下：

```
sleep 6 hours
```

从中我们可以得到结论：引用类型和对象类型之间如果存在方法的覆盖，那么在程序运行的时候，JVM 会根据引用中所存储的对象类型，去调用对象类型中覆盖之后的方法。例如：Animal 类型的 a 引用中存放的是 Dog 对象，而 Dog 类覆盖了 Animal 类中的 sleep 方法，则 JVM 会调用 Dog 类中覆盖之后的 sleep 方法，而不是 Animal 类中的 sleep 方法。

我们可以通过下面这个简单的代码来总结一下多态的基本语法特性：

```
class A{
    public void method(){
        System.out.println("method in A");
    }
}
class B extends A{
    public void method(){
        System.out.println("method in B");
    }
}
```

我们可以把子类对象放入父类引用中。如： A a = new B();

由于 A 类中定义了 method 方法，因此我们可以对 a 引用调用： a.method();

而 a 引用中实际存放的是 B 类对象，因此运行时调用 B 类覆盖之后的 method 方法，将打印 “method in B”。

3.3 强制类型转换和 instanceof

我们前面分析过，对一个引用，只能调用引用类型中定义的方法。因此，下面的代码是错误的：

```
Animal a = new Dog();
a.lookAfterHouse();
```

那么如果我们希望调用 lookAfterHouse 这个方法，应该怎么做呢？很显然，由于这个方法定义在 Dog 类中，我们必须采用 Dog 类型的引用来调用这个方法。如：

```
Animal a = new Dog();
Dog d = a;
d.lookAfterHouse();
```

在这里，a 引用和 d 引用共同指向了一个 Dog 对象。不一样的是，d 引用的引用类型为 Dog，因此，使用 d 引用可以调用 lookAfterHouse 方法。

但是很遗憾，代码 “Dog d = a” 是错误的。

d 引用的类型为 Dog，因此 d 引用中必须存放一个 Dog 类的对象（或者是 Dog 类的子类的对象），但是 a 引用的类型为 Animal，a 引用中可能存放 Dog 类的对象，也可能存放 Animal 类其他子类的对象（比如 Cat）。如果 a 引用中存放了 Cat 类的对象，我们将 a 引用赋值给 d 引用，就错误的把一个 Cat 对象装在了 Dog 类型的引用中，这显然违反了 Java 语言强类型的约束。因此，代码 “Dog d = a” 在编译时，编译器会报错。

为了解决这个问题，我们必须把这句代码改写为：

```
Dog d = (Dog) a;
```

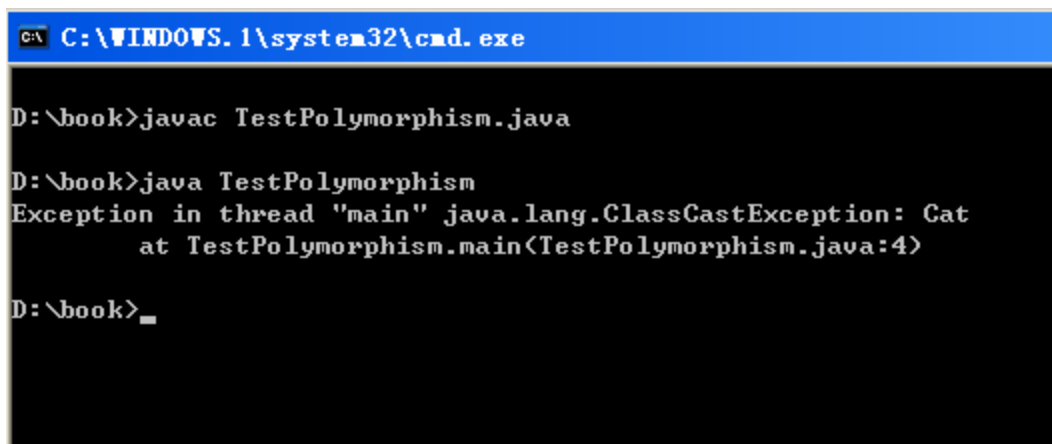
这个语法我们似曾相识，在介绍基本类型的时候，我们用过类似的语法：强制类型转换。

在这里，强制类型转换的含义是：我们认为，a 引用中存放的就是一个 Dog 类的对象，我们“强制性的要求”系统把 a 引用中的对象，作为 Dog 对象存放在 d 引用中。

由于我们表明了这样一种“强硬”的态度，编译器在编译的时候，就不会对这句话报出任何错误了。但是请注意，这并不意味着这句话一定是对的。因为 a 引用中依然可能存放 Cat 对象，一旦这种情况发生，虽然编译通过，但是运行时，JVM 会抛出一个 ClassCastException，类型转换异常。代码如下：

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal a = new Cat();  
        Dog d = (Dog) a;  
        d.lookAfterHouse();  
    }  
}
```

该代码运行时，结果如下：



```
C:\WINDOWS.1\system32\cmd.exe  
  
D:\book>javac TestPolymorphism.java  
  
D:\book>java TestPolymorphism  
Exception in thread "main" java.lang.ClassCastException: Cat  
    at TestPolymorphism.main(TestPolymorphism.java:4)  
  
D:\book>_
```

我们可以看到，尽管编译正确，但是运行时系统抛出了 ClassCastException。

因此我们可以得出结论：

1. 子类的引用可以直接赋值给父类引用。（这是多态的基本用法）
2. 父类的引用赋值给子类引用，必须强制类型转换，并有可能在运行时得到一个类型转换异常。

那么，如何避免类型转换异常的发生呢？下面我们介绍 Java 中的一个关键字：instanceof。

instanceof 的基本语法如下：

引用 instanceof 类名

instanceof 是一个二元运算符，用来组成一个布尔表达式。用来判断某个引用所指向的对象是否和某个类型兼容。例如，我们假定：a 引用中存放的是 Cat 类型的对象。即：

```
Animal a = new Cat();
```

那么表达式“a instanceof Cat”的值为“true”，因为 a 引用所指向的对象确实为 Cat 类型的对象。显然，表达式“a instanceof Dog”的值为“false”。而表示式“a instanceof Animal”也会为“true”。因为 a 引用所指向的对象也可以认为是一个 Animal 类的对象。

完整的代码如下：

```
public class TestInstanceof {
```



```

        public static void main(String[] args) {
            Animal a =new Cat();
            System.out.println(a instanceof Cat);
            System.out.println(a instanceof Dog);
            System.out.println(a instanceof Animal);
        }
    }
}

```

运行结果为:

```

true
false
true

```

我们可以把“instanceof”理解为“是不是”三个字，这样可以帮助我们方便的记忆这个关键字的用法。例如：“a instanceof Dog”就可以理解为“a 引用所指向的对象是不是狗类的对象”。

由于 instanceof 关键字可以用来判断一个引用中的对象类型，因此，我们在对一个引用进行强制类型转换之前，就可以用这个关键字先进行判断，从而避免类型转换异常的发生。例如以下代码：

```

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a = new Cat();
        if (a instanceof Dog){
            Dog d = (Dog) a;
            d.lookAfterHouse();
        }
    }
}

```

由于我们在对 a 引用进行强制类型转换之前，用 instanceof 进行了判断，如果 a 引用中存放的不是 Dog 类型的对象，则不会运行强制类型转换的代码，因此，就避免了类型转换异常的发生。

3.4 多态的作用

多态最主要的作用在于：我们可以将若干不同子类的对象都当做统一的父类对象来使用，这样就会提高程序的通用性，屏蔽不同子类之间的差异。

我们先来看第一种情形：我们希望开发一个 Feeder 类，表示一个饲养员，饲养员应该具有一个“喂养动物”的方法。动物园中的所有动物都由这个饲养员来喂养。代码如下：

```

class TestPolymorphism{
    public static void main(String[] args){
        Feeder f = new Feeder();

        Dog d = new Dog();
        f.feed(d);

        Cat c = new Cat();
        f.feed(c);
    }
}

```

```

    }
}
class Feeder{
    public void feed(Dog d){
        d.eat();
    }
    public void feed(Cat c){
        c.eat();
    }
}

```

我们看到，在 `Feeder` 类中有两个重载的 `feed` 方法，分别以 `Dog` 和 `Cat` 作为参数类型，表示喂养狗和喂养猫。那么如果 `Animal` 类不止 2 个子类呢？假设动物园中有 1000 种动物，`Animal` 类有 1000 个子类，是不是意味着，我们要为 `Feeder` 类重载 1000 个 `feed` 方法呢？

其实没有必要这样，因为无论是狗还是猫，对于饲养员来说都是一样的，都是“动物”，因此，我们在设计 `feed` 方法参数类型的时候，完全可以采用父类 `Animal` 类作为形参类型。代码如下：

```

class TestPolymorphism{
    public static void main(String[] args){
        Feeder f = new Feeder();

        Dog d = new Dog();
        f.feed(d);

        Cat c = new Cat();
        f.feed(c);
    }
}
class Feeder{
    public void feed(Animal a ){
        a.eat();
    }
}

```

我们将 `feed` 方法的参数类型设计为“`Animal`”，这就意味着，该方法接受一个 `Animal` 对象作为实参。而 `Animal` 的任何一个子类对象都可以看作是一个 `Animal` 对象。因此，无论以 `Dog` 对象作为参数，还是以 `Cat` 对象作为参数，都可以调用同一个 `feed` 方法。这样的一个 `feed` 方法，就可以为成千上万种动物“服务”了。

我们把这种用法称为“把多态用在方法的参数类型上”。我们在定义方法时，可以把形参定义为父类类型的引用，而调用方法时，完全可以把子类类型的对象作为实参。很显然，形参为父类类型的方法更加通用，能够接受更多种不同子类类型的实参对象。

作为初学者，我们应该牢牢记住，形如：

```
public void m (A a )
```

这样的方法，完全可以用 `A` 类对象或是 `A` 类的某个子类对象作为参数来调用。

我们再来看第二种情形：我们希望写一个方法，这个方法能够返回一个 `Dog` 对象。代

码如下：

```
public Dog getAnimal() {  
    return new Dog();  
}
```

这个方法非常简单。但是，如果我们的需求发生了改变，希望这个方法接受一个 `int` 类型的参数，当这个参数为偶数时返回 `Dog` 对象，为奇数时返回 `Cat` 对象。我们会遇到一个困难：

```
public ??? getAnimal(int i) {  
    if (i % 2 == 0) {  
        return new Dog();  
    }  
    else {  
        return new Cat();  
    }  
}
```

看到了？我们无法定义这个方法的返回值类型。因为这个方法可能返回 `Dog` 对象，也可能返回 `Cat` 对象。

聪明的读者一定想到了，代码完全可以写成这个样子：

```
public Animal getAnimal(int i) {  
    if (i % 2 == 0) {  
        return new Dog();  
    }  
    else {  
        return new Cat();  
    }  
}
```

这个方法可能返回 `Dog` 对象，也可能返回 `Cat` 对象，但返回的一定是个 `Animal` 对象。因此，我们可以用 `Animal` 类型作为这个方法的返回值类型。也就是说，我们在方法声明中“承诺”方法会返回一个 `Animal` 对象，在方法的实现中，完全可以将 `Animal` 的某个子类对象作为返回值返回。这并不违反我们在声明中的“承诺”。

我们把这种用法称为“把多态用在方法的返回值类型上”，很显然，我们把返回值类型定义为父类，会使得方法更加的通用。在方法的实现中，我们可以返回任何一个子类的对象。

我们应该牢牢记住，形如：

```
public A m ()
```

这样的方法，可能返回 `A` 类的对象，也完全可能返回 `A` 类的某个子类的对象。

我们来看一个完整的代码：

```
class TestPolymorphism{  
    public static void main(String[] args) {  
        B b = new B();  
        m1(b);  
  
        A a = m2(5);  
        a.method();  
    }  
}
```

```

        static void m1(A a){
            a.method();
        }
        static A m2(int i){
            if (i % 2 == 0){
                return new B();
            }
            else {
                return new C();
            }
        }
    }
}
class A{
    public void method(){
        System.out.println("method in A");
    }
}
class B extends A{
    public void method(){
        System.out.println("method in B");
    }
}
class C extends A{
    public void method(){
        System.out.println("method in C");
    }
}

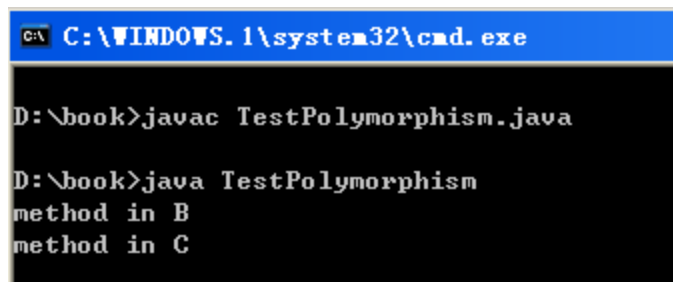
```

这段代码运行结果是什么？

对于 “m1(b);”，我们以一个 B 对象作为参数调用 m1 方法，当实参传给形参时，B 对象就会被赋值给 m1 方法的形参 a。也就是说，a 引用中存放的是 B 类的对象。那么对 a 引用调用 method 方法，自然会调用 B 类覆盖之后的 method 方法，因此打印 “method in B”。

对于 “Aa = m2(5);”，我们用 5 作为参数调用 m2 方法，根据代码，m2 方法会返回一个 C 对象。也就是说，a 引用中存放的是 C 类的对象。此时对 a 调用 method 方法，会调用 C 类覆盖之后的 method 方法，因此打印出 “method in C”。

代码运行结果如下：



```

C:\WINDOWS.1\system32\cmd.exe

D:\book>javac TestPolymorphism.java

D:\book>java TestPolymorphism
method in B
method in C

```

在面向对象的三大特性中，多态最为灵活，不易理解和掌握。请读者勤加练习，多加揣摩。在编程实践中逐渐领悟多态的语法和作用。