《Android 系统下 Java 编程详解》

作者：华清远见

第 6 章 面向对象编程进阶

本章简介

本章详细介绍了类的继承的概念，介绍了如何控制属性和方法的访问权限。介绍了 super 和 this 关键字的使用。详细论述了重载和覆盖的概念。揭示了对象初始化的细节。介绍了简单数据类型的封装类及它们之间的关系、区别，说明了如何使用覆盖类的 toString()方法来得到表示对象的字符串。详细分析了==和 equals()两种比较操作，并介绍了通过覆盖 equals()方法来自定义对象相等的含义。

6.1 继承

- 面向对象程序设计中，可以在已有类的基础上定义新的类，而不需要把已有类的内容重新书写一遍，这就叫做继承。已有的类称为基类或父类，在此基础上建立的新类称为派生类或子类。
- 运用继承，父类的特性不必再重新定义，就可以被其他类继承。
- 继承是面向对象编程技术的一个重要机制，较好地解决了代码重用问题。
- 任何一个类都可以作为基类，从这个基类可以派生出多个子类，这些派生的类不仅具有基类的特征，而且还可以定义自己独有的特征。

6.1.1 类的继承

面向对象程序设计的一个重要特点就是类的重用。这可以通过两种方法来实现：一种是将一个类的实例当做另一个类的属性。一种是使用类的继承来实现，通过关键字 extends，可以使一个类继承另一个类，使这个类也具有被继承类的特点。实现继承的类称为子类，而被继承的类称为父类，也称为超类。类的继承是面向对象程序设计的一个重要特点。

来看一个例子，假设现在要开发手机通讯录名片，名片中规定的角色有同事，朋友，可以给同事定义一个类，如下所示：

```
public class colleaguesCard{
  String name;//姓名
  String sex;//性别
String tele;//办公电话
  String department;//属于哪个办公室
  public void setName(String theName){… }
  public String getName(){… }
  …
}
```

而朋友名片也可以定义一个类，如下：

```
public class friendCard {
  String name;//姓名
  String sex;//性别
String tele;//联系电话
  String assdress;//年级
  public void setName(String theName){… }
  public String getName(){… }
  …
}
```

仔细分析这两个类，可以发现这两个类结构上非常类似，比如，姓名、性别、电话，唯一的区别就在于“同事”类有一个属性“department”，用于说明此同事是属于哪个办公室；而“朋友”类有一个属性“address”，表示这个朋友的住址。倘若现在需要在这两个类上新增一个属性：生日，那么，必须在这两个类上都做修改。其实，在面向对象的编程方法中，完全可以将这两个类的一些共性抽象出来，当做这两个类的“父类”。例如，这两个类中的姓名、电话、性别都是作为一个“名片”所共有的特性，因此，可以将这些特性抽取出来，作为一个新的类“Card”，如下所示：

```
public class Card{
  String name;
  String sex;
String tele ;
  public void setName(String theName){… }
  public String getName(){… }
  …
}
```

然后，根据需要，使同事和朋友这两个类都继承“Card”类，再在这个基础上添加上自己特有的一些特性，例如“朋友”类如下：

```
public class friendCard extends Card{
  //不再需要定义姓名、电话、性别这些属性了，
//它们从父类“Card”中获得
  String address;//地址
  public void setAddress(String theAddress){… }
  public String getAddress(){… }
  …
}
```

而“同事”类可以定义如下：

```
public class colleaguesCard extends Card{
  //同样不需要定义姓名、电话、性别这些属性，
//而从父类“Card”中获得
  String department;//办公室
  public void setDepartment(String theDept){… }
  public String getDepartment(){… }
  …
}
```

通过这种方式，就可以共用父类“Card”的特性，这时，如果需要给“朋友”和“同事”都加上“生日”这个属性，只需要在它们共同的父类“Card”上加上这个属性就可以了，如下所示：

```
public class Card{
  String name;
  String sex;
String tele ;
  java.util.Date birthday;//新增生日属性，类型为一个 Date 引用类型
  //加上对这个属性进行操作的方法
  public void setBirthday(java.util.Date theDate){… }
  public java.util.Date getBirthday(){… }
  …
}
```

此时，如果需要在这个管理系统中新增一个对学校职工的管理功能，需要新增一个“亲属名片”类，它也可以使用“Card”类中定义的属性，然后再加上自己特有的属性，代码如下：

```
public class relativesCard extends Card{
  //在此加上“亲属”特有的属性
}
```

在这个例子中，“Card”类称为“colleaguesCard”、“friendCard”、“relativesCard”这 3 个类的“父类”或“超类”（SuperClass），而“colleaguesCard”、“friendCard”、“relativesCard”称为“Card”类的“子类”（SubClass）。

从上面的例子中，可以看到，如果一个子类要继承父类，只要使用关键字“extends”即可。在 Java 中，类继承的基本语法如下：

```
<modifier>  class  <name>  [extends <superclass> ]{
  <declaration> *
}
```

其中，用关键字“extends”来进行类的继承，后面紧跟的是父类的类名。

在 Java 中，一个类只能从一个父类继承，而不能从多个类中继承。这种继承方式称为“单继承”。写过 C++等其他面向对象语言程序的读者注意 Java 的继承方式。

在 java.lang 包中有一个 Object 类，这个类是所有类的顶级父类。所有的 Java 类，包括标准库中的类和自己定义的类，都直接或间接地继承了这个类。这个类没有任何的属性，只是定义了一些方法。因此，只要你定义了一个 Java 类，就有一些默认的方法供你调用。

在 Java 中，如果你定义了一个类，这个类没有继承任何的父类，那么，系统会自动将这个类的父类设置为 java.lang.Object，例如上面定义的“Card”类：

```
public class Card{… }
```

它实际上等价于：

```
public class Card extends java.lang.Object{… }
```

实际上，在定义一个类的时候也可以这样写，只是，这个工作可以完全由系统来代劳。

在 Java 中，虽然一个子类只能继承一个父类，但是，一个父类却可以“派生”出任意多个的子类。这种状况有点类似于生活中的“父子关系”：一个父亲可以生几个孩子，而一个孩子却只能有一个生父。所谓“派生”，只是从父类的角度来看类的继承。也就是说，“继承”是从子类的角度来看父类和子类的关系的，而“派生”却是从父类的角度来看父类和子类的关系的。

6.1.2 任务一：利用继承实现通讯录实例

1．任务描述

完善上面通讯录名片的例子。

2．技能要点

- 掌握类继承的方法与技巧。

3．任务实现过程

（1）定义一个类“Card”，用来表示“名片”。

源文件：Card.java

```
public class Card {
 String name;
 String sex;
String tele ;
public Card(String theName){
      this.name = theName;
   }
   public Card(){}

 public String getName() {
      return name;
 }
 public void setName(String name) {
      this.name = name;
 }
 public String getSex() {
      return sex;
 }
 public void setSex(String sex) {
      this.sex = sex;
 }
 public String getTele() {
      return tele;
 }
 public void setTele(String tele) {
      this.tele = tele;
 }
}
```

它有一个构造器，用名片中的姓名来作为参数。属性 name、sex 和 tele 用来表示姓名、性别、电话，类中定义了用于存取属性的两个方法：getXXX()和 setXXX()。

（2）定义一个子类 FriendCard，这个子类继承自 Card，因此，此时这个类也拥有了父类 Cardl 的属性和相应的方法。在此基础上，又在子类上新增了一个用于描述地址的属性：address。

源文件：FriendCard.java

```
public class FriendCard extends Card{
 String address;//子类新增属性

 public String getAddress() {
      return address;
 }
```

```
    public void setAddress(String address) {
        this.address = address;
    }
}
```

再来定义 ColleaguesCard 类，该类有新增属性 department，并且该类定义了自己的构造器，参数为父类 Card 的属性 name，子类可以直接使用此属性。

源文件：ColleaguesCard.java

```
public class ColleaguesCard extends Card{
    String department;//子类新增属性
    public ColleaguesCard(String name){
        this.name = name;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
}
```

最后来看如何实例化 FriendCard 和 ColleaguesCard 类，以及如何体现它和父类 Card 的关系。

源文件：MyCard.java

```
public class MyCard {

    public static void main(String[] args) {
        FriendCard fc = new FriendCard();
        fc.setName("Alex");
        fc.setAddress("平安大街 23 号");
        ColleaguesCard cc = new ColleaguesCard("Mia");
        cc.setDepartment("B305");
        System.out.println("Friend:  "+fc.getName()+" Address: "+fc.getAddress());
        System.out.println("Colleagues: "+ cc.getName() + " Department: "+ cc.getDepartment());
    }
}
```

在 MyCard 类中，首先调用 FriendCard 的默认构造器来实例化了一个 FriendCard 类，此时得到一个 FriendCard 对象，然后，调用 FriendCard 从父类继承的 setName()、setAddress()方法来设置它的属性 name 和 address，以及调用从父类继承的 getName()、getAddress()方法来获取属性 name 和 address 的值。而 ColleaguesCard 则使用带参数构造器实例化 ColleaguesCard 对象，确定了属性 name 的值。使用继承自父类的 getDepartment()、setDepartment()方法确定和获取 department 属性值。运行这个程序，将在控制台上打印出如下信息：

```
Friend:  Alex Address: 平安大街 23 号
Colleagues: Mia Department: B305
```

6.1.3 访问控制

在第 2 章已经知道，通过将属性设置为 private（私有的）的，可以限制对相应属性的访问。在 Java 中，可以在类、类的属性及类的方法前面加上一个修饰符（modifier），来对类进行一些访问上的控制。比如，在前面已经讨论过的，一般情况下将类的属性定义为私有（private）的，而通过公共的（public）方法来对这些属性进行访问。在这个类程序外的其他程序只能通过公共的方法来访问这个类的属性，这样，实现了信息的隐藏和封装。但是，有时候也需要让其他的程序直接访问类的属性，或者只能让子类访问父类的属性，这时就不能用 private 来限制这些属性了。

在 Java 中，定义了 3 个修饰符用来控制类、类的属性及类的方法等的访问范围。通过这 3 个修饰符，可以定义 4 种程度的限制。下面将对这些修饰符做详细的说明。

private：这是限制最严格的一个修饰符，使用这个关键字来限制的属性或者方法，只能在同一个类中被访问。也就是说，在这个类文件之外，这些属性或方法是被隐藏的。这个修饰符最常用于修饰类中的全局变量。注意，这个修饰符不能用在类前面。

Default：Default 不是关键字，只是对类、类的属性及类的方法的访问权限的一种称呼。如果在类、类的属性、类的方法前面没有添加任何的修饰符，则说它的访问权限是 default 的。在这种情况下，只有类本身或者同一个包中的其他类可以访问这些属性或方法，而对于其他包中的类而言是不可以访问的。

protected：protected 修饰符修饰的属性或方法，可以被同一个类、同一个包中的类及子类访问。注意，这个修饰符同样不能用于类前面。

public：这个修饰符对类、类的属性及类的方法均可用。它是最宽松的一种限制，使用这个修饰符修饰的类属性、类的方法可以被任何其他的类访问，无论这个类是否在同一个包中，以及是否是子类等。

一般来说，应该将和其他类无关的属性或者方法设置为 private 的，只有需要将它给其他的类访问的属性或方法才将它设置为 public 或者 protected，或者不加任何修饰符，让其为 default。

表 6-1 列出了各种访问修饰符的限制范围。

表 6-1　修饰符的限制范围

修饰符	同一个类中	同一个包中	子类中	全局
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

访问控制修饰符的限制程度从高到低为：private、default、protected、public。

注意：
default 不是 Java 关键符，它只是表明了一种访问限制状态。

6.2　super 关键字

在从子类继承父类的过程中，可能需要在子类中调用父类中的成员，如属性、方法或者构造器，这个时候，可以使用 super 关键字来完成。super 的作用是用于引用父类的成员，如属性、方法或者是构造器。

6.2.1　调用父类构造器

用于调用父类的构造器，是 super 的用法之一，它的基本格式如下：

```
super([arg_list])
```

直接用 super()加上父类构造器所需要的参数，就可以调用父类的构造器了。如果父类中有多个构造器，系统将自动根据 super()中的参数个数和参数类型来找出父类中相匹配的构造器。

来看下面这个例子，类“FriendCard”继承了父类“Card”构造器中调用了父类的构造器：

源文件：friendCard.java

```
public class FriendCard extends Card{
  public FriendCard (String friendName) {
      super(friendName);
  }
//其他代码
      …
}
```

在 FriendCard 这个子类中，有一个构造器 FriendCard (String friendName)里有一个 super（friendName）的语句，意思是用参数 friendName 来调用父类的构造器，也就是说，如果调用 FriendCard 的构造器来构建一个对象，它将会调用父类的构造器来完成这个任务。

父类必须自己负责初始化自己的状态而不是让子类来做，因此，如果子类的构造器中没有显式地调用父类构造器，也没有在构造器中调用重载的其他构造器，则系统将会默认调用父类中无参数的构造器。所以，父类中必须定义无参的默认构造器，否则编译器将会报错。

例如，如果将 Card 类定义成如下：

```
public class Card {
 String name;
 String sex;
   String tele ;
   public Card(String theName){
      this.name = theName;
   }
 // 其他代码
}
```

则对于 Card 的子类 FriendCard 来说，如果定义为如下的方式：

```
public class FriendCard extends Card {
 private String address;

 public String getAddress() {
     return address;
 }
 public void setAddress(String address) {
     this.address = address;
 }
 // 其他代码
}
```

那么，编译这个类的时候，将会出现如下的错误：

```
FriendCard.java:1: cannot resolve symbol
symbol  : constructor Card ()
location: class Card
public class friendCard extends Card
      ^
1 error
```

这是因为在调用类 FriendCard 的默认无参数的构造器创建对象的时候，它会去调用父类 Card 的不带参数构造器，而在类 Card 中，因为没有定义不带参数的构造器，所以，编译器因为无法成功调用 Card 的不带参数的构造器而报错。

6.2.2 调用父类属性和方法

当 super 用于引用父类中的属性或方法时，使用下面的形式：

```
super.属性
super.方法()
```

例如，可以在 FriendCard 子类中通过下面的方式来调用父类中的方法：

```
 super.getName();
```

注意，这时，父类的属性或方法必须是那些 protected（受保护）或者 public（公共）等可以让子类访问的属性或者方法。

super 用于调用父类中的方法主要用于在子类中定义了和父类中同名的属性，或进行了方法的覆盖，而又要在子类中访问父类中的同名属性或覆盖前的方法的时候。

6.2.3 任务二：super 关键字的使用

1．任务描述

使用 super 调用父类中的方法和属性。

2．技能要点

- 掌握 super 关键字的使用方法。

3．任务实现过程

源文件：Card.java

```
public class Card {
 private String name;
 private String sex;
  private String tele ;
   public Card(String theName){
      this.name = theName;
   }
 public Card(){}
 public String showName() {
      return name;
 }
//其他代码

}
```

在这个例子中，将 Card 类中的属性都是 private 的，通过各自的 public 的方法来存取。

源文件：friendCard.java

```
public class friendCard extends Card{
 String address;
 public friendCard(String friendName){
      super(friendName);
 }

 public String getAddress() {
      return address;
 }
 public void setAddress(String address) {
      this.address = address;
   }
 public String showFriendName(){
      return "My Friend "+ super.showName();
 }
}
```

在子类 friendCard 中，如果使用下面的代码段所示直接返回父类中的属性 name：

```
return "My Friend "+ super.name;
```

在编译的时候，将会因为访问权限问题而出错，将会出现下面的错误：

```
friendCard.java:17: name has private access in Card
            return "My Friend "+ super.name;
1 error
```

这个时候，可以在 showFriendName()方法中调用被覆盖方法，得到需要的“name”属性的值：

```
 public String showFriendName(){
      return "My Friend "+ super.showName();
 }
```

这样就解决了访问权限的问题，也清晰地指明了在这个覆盖方法中调用的 showName()方法是父类中的方法。

注意：

如果不使用 super 指明此处调用的 showName()方法的出处，系统依然会自动调用父类中的 showName()方法。但是当子类中有同名方法覆盖父类方法时，系统将会当做是子类自身的方法，当运行此程序时，将会递归地调用方法本身，引起程序错误。为避免混淆，super 关键字不要默认。

6.3 this 关键字

6.3.1 知识准备：使用 this 获得当前对象的引用

编写类的方法时，会希望获得当前对象的引用，Java 引入关键字 this。this 代表其所在方法的当前对象，包括：

- 构造器中指该构造器所创建的新对象。
- 方法中调用该方法的对象。
- 在类本身的方法或构造器中引用该类的实例变量和方法。

this 只能用在构造器或者方法中，表示对“调用此方法的那个对象”的引用。可以和任何的对象引用一样来处理 this 对象。如果在方法内部调同一个类的另一个方法，可以不必显示地使用 this，直接调用即可，效果是一样的。

（1）调用当前对象的属性。

源文件：Person.java

```
public class Person {
 private String name;

 private int age;

 private String sex;

 public String showName() {
      return this.name;
 }

 public void setName(String theName) {
      this.name = theName;
 }
 // …
}
```

在类“Person”中，定义了两个方法用于存取 name 属性。来看一下 showName()这个方法，它将返回当前对象的 name 属性的值，在这里使用了 this 表示当前对象的属性，在方法 setName()中也有类似的用法。其实，如果只是在类的某个方法或构造器中调用另一个方法，可以不用显式使用 this。这样的写法虽然并非必要，但可以使你的程序清晰易读，特别是在你的方法中的参数名称和属性名称一样的时候，例如，如果方法 setName()中的参数名称也为 name，那么，如果没有用 this 来标示对象的属性，方法中的代码将如下：

```
…
public void setName(String name){
 name = name;
}
…
```

（2）引用对象本身。

还有一种情况是必须使用 this 关键字的，就是当需要在对象中明确地指明当前的对象引用是本对象的时候。比如，当你需要返回当前的对象的时候，就需要用到 this 关键字了。假设需要设置手机用户账户、行号

源文件：MobileAccount.java

```
public class MobileAccount {

 private int accountId = 0;
 public MobileAccount createAccount() {
      accountId++;
      return this;
 }
 public int getAccountId() {
      return accountId;
 }
 public void setAccountId(int accountId) {
      this.accountId = accountId;
 }
 public static void main(String[] args) {
      MobileAccount account = new MobileAccount();
      System.out.println("账号是："
               + account.createAccount().createAccount().getAccountId());
 }
}
```

由于 createAccount()方法返回了同一个对象，所以可以在这个对象上多次调用方法 createAccount()。

编译并运行上面的程序，将得到如下的输出：

```
手机用户账号是：2
```

6.3.2 知识准备：在构造器中调用构造器

在一个类中，由于初始化条件不同，可能定义了多个构造器，这称为构造器的重载。在这些构造器中，可能一个构造器中的一段代码和另一个构造器完全一样，那么，就可以在这个构造器中直接调用另一个构造器，这样可以避免编写相同的代码，this 关键字可以做到这一点。this 不再表示对象本身对当前对象的引用。在构造器中，为 this 添加了参数列表，可以调用类本身其他的构造器。语法如下：

```
this([args_list]);
```

如果该类中有多个其他构造器定义，系统将自动根据 this()中的参数个数和参数类型来找出类中相匹配的构造器。

我们来看一个在构造器中使用 this()的例子：

源文件：Card.java

```
public class Card {

 String name;
 String sex;
   String tele ;
   public Card(){
      System.out.println("Card()被调用");
   }
   public Card(String theName){
      this();
      this.name = theName;
      System.out.println("Card(String theName)被调用");
   }
   public Card(String theName,String theTele,String theSex ){
      this("theName");
      this.sex = theSex;
      this.tele = theTele;
      System.out.println("Card(String theName,String theTele,String theSex )被调用");
   }
```

```
 /*
getter,setter
*/
 public static void main(String[] args) {
      Card c  = new Card("Alex","186****","male");
 }
}
```

这个示例中定义了一个类“Person”，这个类中定义了 3 个构造器：没有参数的构造器、有一个参数的构造器，以及有两个参数的构造器。没有参数的构造器将以“Male”值来初始化新建对象的 sex 属性。下面重点来看后面两个构造器，在带一个参数的构造器中，接收一个 String 类型的参数 theName（姓名）来创建对象，这个构造器代码块里有一条语句：

```
this();
```

这条语句的作用是用于调用对象的没有参数的构造器，也就是 Person()；而在带两个参数的构造器中，接收 theName（姓名）和 theAge（年龄）来创建对象，它也通过 this()来调用了该对象中的另外一个构造器：

```
this(theName);
```

此时，这条语句调用的是带一个 String 类型参数的构造器，在这里是 Person(String theName)构造器。

编译并运行上面的程序，将得到如下的输出：

```
Card()被调用
Card(String theName)被调用
Card(String theName,String theTele,String theSex )被调用
```

注意：

在构造器中可以通过 this()方式来调用其他的构造器，但在一个构造器中最多只能调用一次其他的构造器。并且，对其他构造器的调用动作必须在构造器的起始处，否则编译的时候将会出现错误。

另外，不能在构造器以外的地方以这种方式调用构造器。

6.3.3 知识准备：static 的含义

了解 this 关键字后，我们可以更加全面地了解 static 方法的含义。static 方法没有 this 的静态方法，不可以通过 this 来调用静态方法。在 static 方法内部不能调用非静态方法，可以在没有创建任何对象的时候就通过类本身来调用 static 方法，这也是 static 方法的主要用途。

这可能会引起一些质疑，static 方法是否违背了 Java“面向对象”的概念呢？因为 static 方法具有全局函数的特点；由于使用 static 方法时不能通过 this，所以不是通过“向对象发送消息”的方式来完成调用的。所以，如果你的代码里大量出现了 static 方法，应该重新考虑自己的程序设计问题了。但是不可否认 static 的实用性，很多地方需要用到它，至于是否有违“面向对象”的概念，我们大可不必深究，在实际中灵活运用才是好的。

6.4 方法的覆盖与重载

6.4.1 知识准备：方法覆盖

当一个子类继承了一个父类时，它也同时继承了父类的属性和方法。可以直接使用父类的属性和方法，或者，如果父类的方法不能满足子类的需求，则可以在子类中对父类的方法进行“改造”，“改造”的过程在 Java 中称为“覆盖（override）”。

比如，在任务二的 Card 类中定义了一个方法用于显示名片的姓名，代码如下：

```
public String showName() {
      return name;
```

```
}
```

那么，子类 FriendCard 就会继承这个父类的 showName()方法，此时，即使 FriendCard 类中没有定义 showName()方法，也可以在它的实例（instance）中使用这个方法。比如，实例化一个 Teacher 的实例，这时就可以直接调用从 Teacher 的父类 Person 中继承的 showName()方法。代码如下：

```
FriendCard friend = new FriendCard ();
friend.showName();
```

但是，在 FriendCard 类中，希望输出姓名前面加上“friend:”，在任务二中，给 FriendCard 定义了方法 showFriendName，其实可以直接对从父类中继承的 showName()方法进行“改造”，也就是在子类 FriendCard 中覆盖父类 Card 的方法 showName()：

```
public class friendCard extends Card{
 …
 public String showName(){
      return "My Friend "+ this.name;
 }
}
```

在子类 FriendCard 中，覆盖了父类的 showName 方法，在覆盖的过程中，需要提供和父类中的被覆盖方法相同的方法名称、输入参数（此处为空）及返回类型。

另外，在子类对父类的方法进行覆盖的过程中，不能使用比父类中的被覆盖方法更严格的访问权限。比如，父类 Person 中的方法 showName()的修饰符是 public，那么，在子类 FriendCard 中的覆盖方法 showName()就不能用 protected、默认（Default）或者 private 等来限制。

从前面的讨论可以看出，类的继承主要可以从两个方面来看：

（1）对父类的扩充。如在子类中加入新的属性、新的方法。

（2）对父类的改造。比如，对方法的覆盖。

首先定义一个父类：Person，它有 3 个属性，分别由各自的存取方法来存取。

源文件：Person.java

```
public class Card {
 String name;
 String sex;
   String tele ;
   public Card(String theName){
      this.name = theName;
   }
 public Card(){}
 public String getName() {
      return name;
 }
 public void setName(String name) {
      this.name = name;
 }
 public String getSex() {
      return sex;
 }
 public void setSex(String sex) {
      this.sex = sex;
 }
 public String getTele() {
      return tele;
 }
 public void setTele(String tele) {
      this.tele = tele;
 }
 public String showName() {
      return name;
 }
```

```
}
```

在这里，将 Person 的属性的访问控制定义为 Default，是因为在子类中需要访问这些属性。

接着，定义一个类“FriendCard”，它继承“Card”类。

源文件：FriendCard.java

```
public class friendCard extends Card{
 String address;
 public friendCard(String friendName){
      super(friendName);
 }

 public String getAddress() {
      return address;
 }
 public void setAddress(String address) {
      this.address = address;
   }
 public String showName(){
      return "My Friend "+ this.name;
 }
}
```

在这个子类中，继承了父类，新增了一个“地址”的属性：address，并新增了相应的存取方法。这是对父类属性和方法的扩充。改写（覆盖）了父类中的 showName()方法，在 name 前加上“My Friend”后返回。因为在同一个包中的子类中用到了父类的属性 name，所以，父类 Card 中的 name 属性不能定义为 private 的。这是对父类方法的改造，即方法覆盖。

6.4.2 知识准备：方法重载

在 Java 程序中，如果同一个类中有两个相同的方法（方法名相同、返回值相同、参数列表相同）是不行的，因为这样编译器无法将方法的调用和特定的方法联系起来。但是，在一个类中，如果有多个方法具有相同的名称，而有不同的参数，这种情况是允许的，称这种行为为方法的重载（overload）。经常使用 println 来向控制台输出各种类型的数据，这些 println 方法就是实现了方法的重载。

在进行方法的重载时，方法的参数列表必须不同（参数个数或者参数数据类型，或者两者皆不同）。而方法的返回值可以相同，也可以不同。

来看一个例子。还是以上面的“Card”类的 showName 方法来显示对象姓名。需要取得“Card”对象的“name”属性，这里假设会有两种情况：一种是返回值直接为“name”属性；还有一种情况是在获取的对象属性“name”的方法上输入一个参数，将这个参数和“name”结合起来，比如，输入的参数是“先生”，则通过方法返回的是：“XXX 先生”，输入的参数是“女士”，则方法返回的是“XXX 女士”。可以为这两个需求定义两个方法，但是，为了显示这两个方法的相似点，更倾向于使用方法重载来完成：

```
public class Card{
 …
 public String showName() {
      return name;
 }
 public String showName(String personCall){
      return name+personCall;
 }
 …
}
```

这样，如果只需要输出“Card”对象的“name”属性，调用不带参数的 showName()方法就可以了；如果需要得到“Card”对象的“name”属性并且需要指明此人的称谓，则可以调用带一个参数的 showName()方法。

在进行方法的重载时，有 4 条基本原则需要遵守：

（1）方法名相同。

（2）参数列表必须不同。
（3）返回值可以不同。
（4）可以相互调用。

注意：
方法的返回值不是方法签名（Signature）的一部分，所以，进行方法重载的时候，不能将返回值类型的不同当成两个方法的区别。也就是说，在同一个类中，不能有这样的两个方法，它们的方法名相同、参数相同，只是方法的返回值类型不同。

6.4.3 知识准备：方法重载构造器重载

在前面说过，构造器在某种程度上可以看成是一个特殊的方法：它没有返回值，它的方法名称必须和类的名称一致。因此，构造器也常常被称为“构造方法”。作为 Java 类的组成成分之一，构造器也可以进行重载。例如下面的代码中，类 Card 就定义了 3 个重载的构造器以满足不同的需要。

源文件：Card.java

```
public class Card {

 String name;
 String sex;
   String tele ;
   public Card(){
      System.out.println("Card()被调用");
   }
   public Card(String theName){
      this.name = theName;
      System.out.println("Card(String theName)被调用");
   }
   public Card(String theTele,String theSex ){
      this.sex = theSex;
      this.tele = theTele;
      System.out.println("Card(String theTele,String theSex )被调用");
   }
 /*
getter,setter
*/
 public static void main(String[] args) {
      Card c  = new Card("186****","male");
 }
}
```

在这个类中，定义了 3 个构造器，这 3 个构造器中，各自的参数个数都不一样。在创建对象的时候，编译器会根据参数类型和参数个数来确定到底调用哪一个构造器。

编译并运行上面这个程序，它将会向控制台打印出如下信息：

```
Card(String theTele,String theSex )被调用
```

这说明，它的带两个参数的构造器被调用来创建 Card 对象了。

6.5 通常需要覆盖的几种方法

6.5.1 知识准备：对象的 toString 方法

在 Object 类中，定义了一个 toString()方法，用来返回一个表示这个对象的字符串：

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

在这个方法中，它将返回一个由类名、紧随其后的“@”符号和 hash 码的无符号的十六进制字符串，用来表示这个对象。我们知道所有类都是继承 Object，所以“所有对象都有这个方法”，其作用就是为了方便所有类的字符串操作。

来看一下调用“Person”对象的 toString()方法返回的值。

```
…
Person person = new Person();
System.out.println(person);
…
```

上述代码将打印出表示“Person”对象的字符串。它将打印出类似如下的信息：

```
Person@15ff48b
```

显然，这个信息对于我们来说没有什么用。因此，通常情况下，需要覆盖父类中的方法 toString()，用来提供某对象的自定义信息。在 Java 的 API 文档中也指出“建议所有子类都重写此方法”。一般来说，大多数类的 toString()方法覆盖后返回的用于表示对象的字符串都遵循如下的格式：

```
类名[属性 1=值 1,属性 2=值 2,…]
```

覆盖 toString()方法的一个基本原则是，它应该返回包含在对象中的所有令人感兴趣的信息，比如对象的属性的值。

6.5.2 任务三：覆盖 toString 方法

1．任务描述

在任务一的 Card 类中添加自定义 toString 方法以覆盖 Object 中的 toString()方法。

2．技能要点

- 掌握自定义 toString()方法。

3．任务实现过程

源文件：Card.java

```
public class Card {
 String name;
 String sex;
   String tele ;
   public Card(String theName){
      this.name = theName;
   }
 public Card(){}
 public String getName() {
      return name;
 }
 public void setName(String name) {
      this.name = name;
 }
 public String getSex() {
      return sex;
 }
 public void setSex(String sex) {
      this.sex = sex;
 }
 public String getTele() {
      return tele;
 }
 public void setTele(String tele) {
```

```
        this.tele = tele;
    }

    // 覆盖 toString()方法
    public String toString() {
        return getClass() + "[" + "name = " + name + ",sex = " + sex+ ",tele = " + tele + "]";
    }
}
```

在这个类中，覆盖了父类（在这里是 Object 类）的 toString()方法，让它按照上面所说的惯例来返回一个用以表示对象的字符串。

除显式调用对象的 toString()方法外，在进行 String 与其他类型数据的连接操作时，会自动调用 toString()方法，它可以分为两种情况：

（1）如果 String 类型数据和引用数据类型连接，则引用类型数据直接调用其 toString()方法返回表示该对象的字符串.

（2）如果 String 类型数据和简单类型数据连接，则简单类型数据先转换为对应的封装类型，再调用该封装类对象的 toString()方法转换为 String 类型。

6.5.3 知识准备：==和 equals()

在 Java 程序设计中，经常需要比较两个变量值是否相等，例如：

```
a = 10;
b = 12;
if(a == b)    {
    //statements
}
再如：
ClassA a = new ClassA("abc");
ClassA b = new ClassA("abc");
if (a == b){
    //statements
}
```

对于第一个例子，比较的是简单类型数据，它们是明显相同的，所以 a==b 值为 true。但是，对于第二个例子中的比较表达式，它们比较的是对象的引用，即判断这两个变量 a 和 b 是否指向同一个对象引用。在这里，因为 a 指向一个对象，而 b 指向另一个对象，所以，它们并不指向同一个对象，因此，a==b 返回的值是 false。

为了方便说明简单类型和引用类型比较，此处用 string 类型和它的封装类 String 来说明简单类型和封装类型进行比较时的区别。请看下面的例子：

源文件：TestEqual.java

```
public class TestEqual {
 public static void main(String[] args) {
      // 简单类型比较
      string1="aaa";
string2="aaa";
      System.out.println("string1== string2? " + (string1 == string2));

      // 引用类型比较
String string3=new String("aaa");
String string4=new String("aaa");


      System.out.println("string3== string4? " + (string3 == string4));
 }
}
```

运行这个程序，在控制台上打印出如下信息：

```
string1== string2? true
```

```
string3== string4? false
```

可以看出，比较两个引用类型的时候，虽然用了同一个参数构造两个变量，但它们并不相同。我们知道，对于引用类型，它们指向的是两个不同的对象：这两个对象的值都为 100。因为它们指向的对象是两个对象，因此，比较这两个变量会得到 false 的值。也就是说，对于引用类型变量，运算符“==”比较的是两个对象是否引用同一个对象。那么，如何比较对象的值是否相等？

Java 中提供了一个 equals()方法，用于比较对象的值。下面将上面的程序稍做修改：

```
String string3=new String("aaa");
String string4=new String("aaa");
     System.out.println("string3 equals string4? " + (string3 == string4));
```

这时表达式 c.equals(d)会得到一个 true，这是因为，方法 equals()进行的是“深层比较”，它会去比较两个对象的值是否相等。

equals()方法是由谁来实现的呢？在所有类的父类 Object 中，已经定义了一个 equals()方法，但是这个方法实际上也只是测试两个对象引用是否指向同一个对象。所以，可以使用这个方法来进行比较操作，但是，它并不一定能得到所期望的效果。所以，经常的，还需要自己将定义的类中的 equals()进行覆盖。像 Integer 封装类，就覆盖了 Object 中的 equals()方法。

关于==和 equals()两种比较方式，在使用的时候要小心选择。如果测试两个简单类型的数值是否相等，则一定要使用“==”来进行比较；如果要比较两个引用变量对象的值是否相等，则用对象的 equals()方法来进行比较；如果需要比较两个引用变量是否指向同一个对象，则使用“==”来进行比较。对于自定义的类，应该视情况覆盖 Object 或其父类中的 equals()方法。equals()方法只有在比较的两者是同一个对象的时候，才返回 true。

下面，来考虑如下的这种场景：假设在一个 Java 应用程序中，创建了两个“公民”对象。现在，需要比较两个“公民”对象是否相等，此时，关心的是这两个“公民”对象代表的是否为同一个人，而并不关心它们是否是同一个内存区域里的对象。

假设有一个类“Citizen”用于表示公民，它有一个属性 id 用来表示这个公民的身份证号，假设身份证号不会重复，也就是说，一个身份证号对应一个公民。它的类定义如下（为简单起见，省略了其他的属性而只留下用于表示身份证号的 id 属性）：

```
public class Citizen {
  // 身份证号
  String id;

  // 其他属性略
  public Citizen(String theId) {
       id = theId;
  }
}
```

假设在一个 Java 应用程序中，建立了两个“Citizen”对象，然后，在某个点上需要判断这两个对象是否代表了同一个公民：

```
Person p1 = new Person("id00001");
Person p2 = new Person("id00001");
…
if (p1.equals(p2)){
  …
}
```

在这个程序中，因为这两个“Citizen”对象的身份证号一样，所以，它们代表的应该是同一个人，但如果用 Object 的默认方式，它只会去比较两个对象引用变量是否指向同一个对象，因此，它返回 false。显然，这个结果不是我们所期待的。此时，就需要在“Citizen”中覆盖 Object 类中的 equals()方法，以满足我们的需求。

源文件：Citizen.java

```
public class Account {
  String accountName;
```

```
    public Account(String theName){
        this.accountName = theName;
    }
    public String getAccountName() {
        return accountName;
    }

    public void setAccountName(String accountName) {
        this.accountName = accountName;
    }

    public boolean equals(Account a) {
        // 首先判断需要比较的 Object 是否为 null，
        // 如果为 null，返回 false
        if (a == null) {
            return false;
        }
        // 判断测试的是否为同一个对象，
        // 如果是同一个对象，毋庸置疑，它应该返回 true
        if (this == a) {
            return true;
        }
        // 判断它们的类型是否相等，
        // 如果不相等，则肯定返回 false
        if (this.getClass() != a.getClass()) {
            return false;
        }

        // 只需比较两个对象的 id 属性是否一样，
        // 就可以得出这两个对象是否相等
        return accountName.equals(a.accountName);
    }

    public void print(Account ac){
        if (this.equals(ac)){
            System.out.println("Welcome "+this.accountName);
        }
        else
        {
            System.out.println("用户名错误");
        }
    }
    public static void main(String[] args) {
        Account a1 = new Account("Alex");
        Account a2 = new Account("Mary");
        a1.print(new Account("Alex"));
        a2.print(new Account("Alex"));
    }
}
```

在类“Citizen”中，覆盖了父类（Object）中的 equals()，使它能够根据对象的 id 属性是否相等来判断这两个对象是否相等。这个覆盖方法中，已经加入了足够的注释，请读者参考程序中的注释来理解这个方法的定义。

源文件：TestCitizen.java

```
public class TestCitizen {
    public static void main(String[] args) {
        Citizen p1 = new Citizen("id00001");
        Citizen p2 = new Citizen("id00001");
        System.out.println(p1.equals(p2));
    }
}
```

这个类中，定义了一个 main()方法，它是一个应用程序。在 main()方法中，使用身份证号“id00001”创建了两个 Citizen 对象，然后，用覆盖的 equals()方法比较这两个对象的相等性，此时，因为它们的身份证号相同，equals()方法因此将返回 true。编译并运行这个程序，将向控制台输出如下信息：

```
true
```

读者可以试着将覆盖的 equals()方法删除或注释掉，然后重新编译运行 TestCitizen.java，看看此时的结果，然后想想为什么。

在这个覆盖的 equals()方法中，还使用到了另一个方法 getClass()，它将返回对应的对象的运行期类（runtime class）。关于 getClass()的更多信息，请读者查看 API 文档，在此不再赘述。

另外，如果一个类的父类不是 Object，那么，你首先需要检查它的父类是否定义了 equals()方法，如果是的话，在覆盖父类的 equals()方法的时候，需要在子类的 equals()方法中使用下面的方法来调用父类的 equals()方法，以确保父类中的相关比较能够执行：

```
super.equals(obj)
```

例如，假设有一个类“Armyman”用来表示军人，它是一个“Citizen”的子类，此时，如果在 Armyman 中覆盖 Citizen 的 equals()方法，则需要在覆盖的 equals()方法中调用被覆盖的 Citizen 类中的 equals()方法：

```
public boolean equals(Object obj){
return super.equals(obj)&&(其他比较语句);
}
```

6.6 对象的初始化

当调用类的构造器来创建对象时，它将给新建的对象分配内存，并对对象进行初始化操作。现在我们来探讨对对象进行初始化操作时的细节。

对象的初始化操作将递归如下的步骤来进行：

（1）设置实例变量的值为默认的初始值（0, false, null），不同的数据类型有不同的初始值。

（2）调用类的构造器（但是还没有执行构造方法体），绑定构造器参数。

（3）如果构造器中有 this()调用，则根据 this()调用的参数调用相应的重载构造器，然后转到步骤（5）；否则转到步骤（4）。

（4）除 java.lang.Object 类外，调用父类中的初始化块初始化父类的属性，然后调用父类构造器，如果在构造器中有 super()调用，则根据 super()中的参数调用父类中相应的构造器。

（5）使用初始化程序和初始化块初始化成员。

（6）执行构造器方法体中的其他语句。

所谓的初始化块，就是我们在前面章节提到的所谓“游离块”。不管使用哪个构造器创建对象，它都会被首先运行，然后才是构造器的主体部分被执行。

我们来看一个例子。

源文件：TestPerson.java

```
class Person {
  private String name;

  private int age;

  private String sex;

  public Person() {
       System.out.println("构造器 Person()被调用");
       sex = "Male";
       System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
  }
```

```
  public Person(String theName) {
       // 调用构造器 Person()
       this();
       System.out.println("构造器 Person(String theName)被调用");
       name = theName;
       System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
  }

  public Person(String theName, int theAge) {
       // 调用构造器 Person(String theName)
       this(theName);
       System.out.println("构造器 Person(String theName,int theAge)被调用");
       age = theAge;
       System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
  }

  // 初始化块
  {
       name = "Tony Blair";
       age = 50;
       sex = "Female";
       System.out.println("初始化块执行后：name=" + name
+ " , age=" + age + " ,sex=" + sex);
  }
}

public class TestPerson {
  public static void main(String[] args) {
       Person person = new Person();
  }
}
```

编译执行上面的程序，将会得到如下的输出：

```
初始化块执行后：name=Tony Blair ,age=50 ,sex=Female
构造器 Person()被调用
name=Tony Blair ,age=50 ,sex=Male
```

可以看到，初始化块会先于构造器调用执行。读者可以将 main()方法中调用的创建 Person 对象的构造器换成其他两个，再观察它的结果，同样可以得出上面的结论。

提示：

初始化块的机制并不是必须的，完全可以将属性的初始化和属性的声明结合在一起，例如：

```
    String name =  "Tony Blair";
```

下面看一个对象初始化的例子，以加深对对象初始化的理解：

源文件：Person.java

```
class Person {
  private String name;

  private int age;

  private String sex;

  public Person() {
       System.out.println("构造器 Person()被调用");
       sex = "Male";
       System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
  }

  public Person(String theName) {
       System.out.println("构造器 Person(String theName)被调用");
       name = theName;
```

```
        System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
    }

    public Person(String theName, int theAge) {
        System.out.println("构造器 Person(String theName,int theAge)被调用");
        name = theName;
        age = theAge;
        System.out.println("name=" + name + " ,age=" + age + " ,sex=" + sex);
    }

    // 初始化块
    {
        name = "Tony Blair";
        age = 50;
        sex = "Female";
        System.out.println("Person 初始化块执行后：name=" + name
  + " ,age=" + age    + " ,sex=" + sex);
    }
}
```

这里定义了一个父类 Person，它里面定义了 3 个构造器以及 1 个初始化块。

我们再来定义一个 Person 类的子类 Teacher，代码如下。

源文件：Teacher.java

```
//Person 子类
class Teacher extends Person {
    // 部门
    String department;

    // 教龄
    int schoolAge;

    public Teacher() {
        System.out.println("构造器 Teacher()被调用");
    }

    public Teacher(String name) {
        // 调用父类中的构造器 Person(String theName)
        super(name);
        System.out.println("构造器 Teacher(String name)被调用");
    }

    public Teacher(int theSchoolAge) {
        schoolAge = theSchoolAge;
    }

    public Teacher(String dept, int theSchoolAge) {
        // 调用本类中重载的构造器 Teacher(int theSchoolAge)
        this(theSchoolAge);
        department = dept;
    }

    // 初始化块
    {
        department = "教务部";
        System.out.println("Teacher 初始化块执行后：name=" + name + " ,age=" + age+ " ,sex=" + sex);
    }
}
```

这个类中定义了 4 个构造器：一个不带参数的构造器；一个带一个 String 数据类型参数的构造器，它通过 super()显式调用父类的构造器；一个带一个 int 数据类型参数的构造器；一个带两个参数的构造器，通过 this()来调用类中带 int 类型参数的构造器。

最后我们来看测试程序。

源文件：TestInit.java

```
public class TestInit {
 public static void main(String[] args) {
      System.out.println("-----------------------------------");
      Teacher t1 = new Teacher();
      System.out.println("");

      System.out.println("-----------------------------------");
      Teacher t2 = new Teacher("Tom");
      System.out.println("");

      System.out.println("-----------------------------------");
      Teacher t3 = new Teacher("财务部", 20);
 }
}
```

这个程序通过 3 种构造器来创建 3 个 Teacher 对象，因为调用的构造器不同，所以对象初始化的步骤也有所不同。因为在这几个程序中，几个关键部分都已经有信息打印到控制台，所以，只要执行这个程序，就可以看出各个调用构造器创建对象的运行细节。

编译并运行 TestInit，可以在控制台上得到如下的信息：

```
-----------------------------------
Person 初始化块执行后：name=Tony Blair ,age=50 ,sex=Female
构造器 Person()被调用
name=Tony Blair ,age=50 ,sex=Male
Teacher 初始化块执行后：name=Tony Blair ,age=50 ,sex=Male
构造器 Teacher()被调用

-----------------------------------
Person 初始化块执行后：name=Tony Blair ,age=50 ,sex=Female
构造器 Person(String theName)被调用
name=Tom ,age=50 ,sex=Female
Teacher 初始化块执行后：name=Tom ,age=50 ,sex=Female
构造器 Teacher(String name)被调用

-----------------------------------
Person 初始化块执行后：name=Tony Blair ,age=50 ,sex=Female
构造器 Person()被调用
name=Tony Blair ,age=50 ,sex=Male
Teacher 初始化块执行后：name=Tony Blair ,age=50 ,sex=Male
```

请读者参考上面列出的对象初始化的 6 个步骤和本程序的执行结果，充分理解和掌握对象初始化的过程。

6.7 封装类

6.7.1 知识准备：Java 中的封装类

虽然 Java 语言是典型的面向对象编程语言，但其中的 8 种基本数据类型并不支持面向对象的编程机制，基本类型的数据不具备“对象”的特性——没有属性、没有方法可调用。沿用它们只是为了迎合程序员根深蒂固的习惯，并能简单、有效地进行常规数据处理。

这种借助于非面向对象技术的做法有时也会带来不便，比如引用类型数据均继承了 Object 类的特性，要转换为 String 类型（经常有这种需要）时只要简单调用 Object 类中定义的 toString()即可（关于 toString()方法，参见 6.7.2 节内容），而基本数据类型转换为 String 类型则要麻烦得多。为解决此类问题，Java 语言引入了封装类的概念，在 JDK 中针对各种基本数据类型分别定义相应的引用类型，并称之为封装类（Wrapper Classes）。

所有的封装类对象都可以向各自的构造器传入一个简单类型数据来构造：

```
boolean b = true;
Boolean B = new Boolean(b);

byte by = '42';
Byte By = new Byte(by);

int i = 123;
Integer I = new Integer(i);

…
```

除了 Character 外，还可以通过向构造器传入一个字符串数据来构造，如果传入的字符串不能用于表示对应的值，除了 Boolean 类型外，将会抛出一个 NumberFormatException 异常：

```
Boolean B = new Boolean("true");
Boolean B1 = new Boolean("a");// 对，不抛出异常

try {
 Byte By = new Byte("42");
 Short S = new Short("121212");
 Integer I = new Integer("123456789");
 // …
} catch (NumberFormatException e) {
 e.printStackTrace();
}
```

封装在封装类中的值，可以通过各自的 xxxValue()方法来转换成简单类型。

- Boolean：public boolean booleanValue()。
- Byte：public byte byteValue()。
- Character：public char charValue()。
- Double：public double doubleValue()。
- Float：public float floatValue()。
- Integer：public int intValue()。
- Long：public long longValue()。
- Short：public short shortValue()。

下面我们来看一个封装类的例子。

源文件：WrapperClass.java

```
public class WrapperClass {
 public static void main(String[] args) {
      Integer i = new Integer(10);
      Integer j = new Integer(10);
      System.out.println(i == j);
 }
}
```

在这个类中，创建了两个 int 的封装类 Integer 对象，并且比较它们是否相等。运行这个程序，将在控制台上输出：

```
false
```

可以从结果看出，它们并不相等。这是因为，i 和 j 各自指向的对象是不一样的。

6.7.2 知识准备：自动拆箱和装箱

在 JDK 5.0 中，引入了自动装箱/拆箱（Autoboxing/Unboxing）功能，可以让我们方便地在简单类型和对应的封装类型数据之间转换，例如，我们来看下面这个例子：

```
Integer iObject = 100;
```

这行代码在 JDK 5.0 之前是非法的：不能将一个简单类型的数据赋值给引用类型变量，而在 JDK 5.0 中，通过自动装箱功能，可以自动进行“装箱”——将简单类型数据“装”到对应的封装类型中。相反的，通过自动拆箱功能，可以将封装类型的数据赋值给对应简单类型变量，例如：

```
int i = new Integer(100);
```

这个功能带来的一个直接好处就是，以后在那些本来只接收引用类型数据的方法中，可以直接使用简单类型数据，而不再需要先在程序中将它转换成引用类型数据。例如，下面的方法在 JDK 5.0 之前是非法的，而在 JDK 5.0 中，它是合法的：

```
public class TestBoxing{
 public void test(Object o) {
      System.out.println(o);
 }
 public static void main(String[] args){
      TestBoxing tb = new TestBoxing();
      tb.test(100);   //自动装箱
 }
}
```

在这个例子中，我们定义了一个 test()方法，它有一个参数 Object，如果在 JDK 5.0 之前，在调用这个方法的时候，必须传递一个引用类型的数据给它，而在 JDK 5.0 中，可以直接给它传递一个简单类型数据。

另外，在 JDK 对 5.0 中，使用自动装箱的时候还有一个问题需要特别注意，下面我们来讨论这个问题。

在 Java 中，为了节省创建对象的时间和空间，对于一些常用的对象，会将它在内存中缓存起来。在前面，我们已经学到了 String 对象就是这样的，当直接使用“String s = "str"”这种形式来产生 String 对象时，如果在内存中已经有一个使用这种方式产生的字符串对象，那么就不会再新建对象，而是直接使用已经存在的那个 String 对象。而实际上，对于如下范围内的简单数据类型：

- boolean 类型的值。
- 所有 byte 类型的值。
- 在-128~127 之间的 short 类型的值。
- 在-128~127 之间的 int 类型的值。
- 在\u0000~\u007F 之间的 char 类型的值。

它们在使用自动装箱转换成相关封装类型对象的时候，其行为也和 String 类似。上面列表范围中的数据在进行自动装箱的时候，将首先检查内存中是否已经有使用自动装箱产生的具有相同值的对象。如果已经有一个“值”相同的对象存在，那么，并不会产生新的对象。这个机制和使用 String s = "test"这种方式产生一个字符串对象类似。也就是说，当简单类型的数据是上面列表中的数据类型和对应范围内的值的时候，使用自动装箱得到的对象在内存中可能已经存在，而不是新产生的，就跟 String 类型数据一样。

例如，我们来看下面的例子。

源文件：TestAutoBoxing.java

```
public class TestAutoBoxing{
 public static void main(String[] args){
      Integer t1 = new Integer(127);
      Integer t2 = new Integer(127);
      System.out.println("t1 == t2 ? "+(t1 == t2));
      Integer t3 = 127;
      Integer t4 = 127;
      System.out.println("t3 == t4 ? "+(t3 == t4));
      System.out.println("t1 == t4 ? "+(t1 == t4));
      Integer t5 = 128;
      Integer t6 = 128;
      System.out.println("t5 == t6 ? "+(t5 == t6));
 }
}
```

编译并运行这个程序，将得到如下的输出：

```
t1 == t2 ? false
```

```
t3 == t4 ? true
t1 == t4 ? false
t5 == t6 ? false
```

对于 t1 和 t2 的关系，相信读者很容易就可以得出正确的判断。而对于 t3/t4 及 t5/t6 的关系，可能不是那么容易就能得到正确答案了。根据上面所述的规则，当 int 类型数据在-128~127 之间的时候，它通过自动装箱所产生的 Integer 对象会缓存在内存中，而当试图通过自动装箱方式产生另一个相等值的 Integer 对象的时候，系统将不会重新生成新的对象，而是直接使用内存中已经存在的 Integer 对象。当数值范围不在该数据类型所对应的范围内的时候，自动装箱产生的数据等同于用 new 方式产生的数据。

提示：

在 JDK 5.0 以后，原来很多只接收对象参数的类、方法，现在可以接收简单类型数据了，这并不意味着这些类、方法改变了它们的行为方式，而是自动装箱功能帮我们自动完成了将简单类型数据转换成对应引用类型数据的动作。

6.7.3 知识拓展：在 Java 中实现小数的精确计算

我们在编写 Java 程序的时候，可能经常需要用到小数，例如，给某个厂商编写一个采购平台，需要计算货物的价格等。

现在假设有两个商品，价格分别为 0.05 元和 0.01 元，需要计算这两个商品的价格的和，并且将它打印出来。我们可以编写如下的代码：

```
System.out.println(0.05+0.01);
```

但是，当你执行上面的代码时，将会得到下面的输出：

```
0.060000000000000005
```

显然，这个输出并不是我们所期望的结果。为什么会出现这样的情况呢？这是因为计算机中的数字表示的方式原因（所有的数值都需要转换成二进制数），0.05 不能被精确表示为一个 double（默认情况下，小数类型数据为 double 类型）类型数据，而是被表示为最接近它的 double 值。因此，对于需要精确运算结果的地方，请勿使用 double 或者 float 类型的数据来表示。要实现数值的精确运算，有两种选择：使用 int 或者 long 类型数据，得到最后结果后，再除以适当的 10 的倍数，来获得精确的小数。或者使用 BigDecimal 来进行精确的小数运算。对于 int 或者 long 类型数据的运算，在此不再赘述，本节主要讨论如何使用 BigDecimal 进行精确的小数运算。

BigDecimal 位于 java.math 包中，它有 4 个构造器，其中，有两个接收 BigInteger 参数，而我们要用的是使用 String 或者 double 类型参数的构造器，但是，基于上面的理由，不要使用 double 类型参数的构造器来构造 BigDecimal 对象，而要使用 String 类型参数的构造器。在这个类上定义了很多进行加减乘除运算的方法：add()、subtract()、multiply()、divide()，另外还有进行小数点移位运算的 movePointLeft()和 movePointRight()等。

我们来看一个使用 BigDecimal 来实现精确运算小数的例子。

源文件：TestFloat.java

```
import java.math.BigDecimal;

public class TestFloat{
public static void main(String args[]){
       //直接使用 double 类型数据进行运算
       System.out.println(0.05+0.01);
       //使用 BigDecimal 的 double 参数的构造器
       BigDecimal bd1 = new BigDecimal(0.05);
       BigDecimal bd2 = new BigDecimal(0.01);
       System.out.println(bd1.add(bd2));
       //使用 BigDecimal 的 String 参数的构造器
       BigDecimal bd3 = new BigDecimal("0.05");
       BigDecimal bd4 = new BigDecimal("0.01");
```

```
        System.out.println(bd3.add(bd4));
    }
}
```

编译运行这个程序，可以得到如下输出：

```
0.060000000000000005
0.060000000000000002983724378680108202388510107994079589843 75
0.06
```

从这个例子中可以看出，如果使用 double 类型参数的构造器来获得 BigDecimal 对象进行运算，也可能得不到想要的结果，而如果使用 String 参数的构造器来获得 BigDecimal，将可以得到正确的运算结果。

6.8 本章小结

本章详细介绍了类的继承的概念，着重介绍了 super 和 this 关键字的使用，并介绍了重载和覆盖的概念，揭示了对象初始化的细节。通过本章的学习，读者对类的继承应该有了一定的了解和掌握，并掌握了对 toString()方法、equals()方法的使用和覆盖，这些在今后的编程中是常常会用到的。继承是 Java 面向对象的一大特点，在以后的学习中读者们会有深入的体会。

课后练习题

一、选择题

1．关于继承，下列说法正确的是（　　）。
A．Java 中的类是单继承的，但接口可以继承多个接口
B．一个类只能有一个子类
C．子类引用可以指向一个父类对象
D．子类可以继承父类所有的属性和方法
2．关于方法重写（Override），下列说法正确的是（　　）。
A．方法重写是指一个类里面有多个同名的方法
B．方法重写是指子类和父类有同名的方法，并且参数列表和返回类型都完全一样
C．方法重写是指子类和父类有同名的方法，但要有不同的参数列表
D．重写的方法可以比原方法有更严格的访问权限
3．关于方法重载（Overload），下列说法正确的是（　　）。
A．方法重载是子类和父类有同名的方法。
B．实现方法重载必须是方法名相同，参数列表不同
C．可以用返回值不同来区分两个重载的方法
D．重载的方法可以相互调用
4．下列说法正确的是（）。
A．用 protected 修饰的属性只能被同一个包中的类访问
B．不加任何访问修饰符的属性只能被同一个类及子类访问
C．protected 和 private 不可以用来修饰类
D．protected 和 private 和 public 既可以修饰类，又可修饰属性
5．下列说法正确的是（　　）。
A．每个子类对象创建时都会默认调用父类的所有构造器
B．每个子类对象创建时都会默认调用父类的无参构造器
C．子类中不能显式调用父类的构造器
D．子类中显式调用了父类的某个构造器后，就不会再调用父类的无参构造器了

二、简答题

1．简述 Java 中的继承。

2．简述 Java 中有哪些访问控制修饰符及各自的作用。

3．简述方法覆盖和方法重载的区别。

三、编程题

1. 创建一个 Animal 类，它有一个默认构造器，在默认构造器中输出“I am an animal”。创建一个 Animal 类的对象。

2．创建一个 Dog 类，它继承 Animal 类，也有一个默认构造器，在默认构造器中输出“I am a dog”，创建一个 Dog 类的对象。

3．在 Dog 类中添加一个重载的构造器，接收一个字符串 dogname，在构造器中调用默认构造器，并输出“myname is”加上你接收到的参数。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218