Chp10 Object 类与常用类，内部类

本章导读

本章首先将为读者介绍 Object 类。这个类在 Java 中具有很特殊的地位，这是所有类的父类，是 Java 继承树的根。因此，这个类中的很多方法都很值得研究。

然后为大家介绍的是 Java 中的包装类和 String 类。这些都是 Java 开发中比较常用的类，也有很多常用的方法。

另外，本章还会介绍 Java 中内部类的一些语法。

1 Object 类

Object 类是 Java 中所有类的父类。例如下面的代码：

```
class ClassA{}
class ClassB extends ClassA{}
```

在上面的代码中，ClassB 类明确的写明了，继承自 ClassA 类。那 ClassA 类呢？像这种没有明确写明 extends 的类，都继承自 java.lang.Object，也就是本章我们要介绍的 Object 类。正因为有这个特性，在 Java 中任何一个类，如果追根溯源的话，归根结底都是 Object 类的直接或者间接子类。

前面我们分析过，Java 中所有的类会组成一种树状关系，而 Object 类，就是这棵类继承关系树的树根。

既然 Object 类是所有类的父类，那我们就得好好研究一下这个类。

首先，Object 类既然是所有类型的父类，那么在 Java 中所有的对象，都能够赋值给 Object 类型的引用。这是因为子类对象可以直接赋值给父类引用，而所有 Java 中的类都是 Object 类的子类。

其次，由于子类中能够继承父类中的公开方法。因此，Object 类中所有的公开方法都能被子类继承。也就是说，Object 类中的公开方法，是 Java 中所有对象都拥有的方法。

接下来，我们就仔细来研究一下 Object 类的公开方法。

1.1 finalize

finalize 是一个 protected 的方法。虽然不是 public 的，但是这个方法也同样能够被所有子类继承（考虑一下，protected 能够被同包以及非同包的子类访问，也就是能够被所有子类访问）。

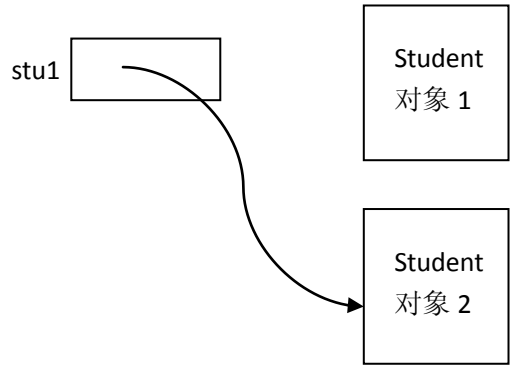
finalize 方法有什么特点呢？这个方法会在对象被垃圾回收时由垃圾回收器调用。

如何来理解垃圾回收呢？请看下面的代码：

```
class Student{
    String name;
    int age;
}
public class TestStudent{
    public static void main(String args[]){
        Student stu1 = new Student();
```

```
        stu1 = new Student();
    }
}
```

在这段代码中，创建了两个不同的 Student 对象。内存中的结构如下图：



可以看出，一开始分配的那块内存（Student 对象 1），由于之后把 stu1 指向了 Student 对象 2（创建了一个新对象并把首地址赋值给 stu1 引用），因此再也没有引用指向 Student 对象 1。因此结果就是，这个对象占据着内存空间，但是没有引用指向这个对象，因此这个对象无法被使用。于是，这种没法使用但又占据内存空间的对象，就被称为垃圾对象。这些垃圾对象占用内存空间，如果一直不处理的话，会造成内存空间的浪费，严重的话会造成程序的崩溃。

那遇到垃圾对象怎么解决呢？在传统的编程语言中，程序员既要负责分配空间，又要负责回收内存资源，这样为程序员编程增加了很大的负担。而在 Java 中，程序员只需要负责分配空间（也就是 new 对象），而不需要操心处理垃圾对象的问题。JVM 有一个自动垃圾回收的机制，这个机制能够自动回收垃圾对象所占用的内存。这样，程序员就只需要负责分配空间、创建对象而不用担心内存的回收。

垃圾回收机制对于大部分程序员来说，都是天大的好事儿：在家里举办宴会之前，都愿意把家里布置的干干净净漂漂亮亮。但是，当宴会散场之后，收拾屋子、洗碗扫地这些工作，总是让人觉得很麻烦很折腾。而 Java 垃圾回收器就担当了义务的家政服务员。这就是 Java 垃圾回收给程序员带来的福利：程序员大可以自由自在的去使用内存空间，但完全不用关系任何事后收拾的任务。

当 JVM 进行一个对象的垃圾回收工作时，会自动调用这个对象的 finalize 方法。我们应该如何来看待这个 finalize 方法呢？这要从垃圾回收的时机说起。

在 JVM 的规范中，只规定了 JVM 必须要有垃圾回收机制，但是什么时候回收却没有明确说明。也就是说，对象成为了垃圾对象之后，并不一定会马上就被垃圾回收。怎么来理解这个概念呢？

举一个生活中的例子：当顾客去餐厅吃饭，吃完饭之后，桌子上就留下了顾客吃剩的饭菜、汤水以及用过的餐具。这些餐具占用了餐厅的空间，但是却无法重复使用，因此，这些用过的餐具就可以被当做是垃圾对象。在餐厅，顾客不用管收拾桌子和餐具，而由服务员负责回收这些垃圾，并且清理空间，这就是“自动垃圾回收”。

但是服务员回收垃圾有不同的方式。在人潮拥挤的快餐店里，时时会有一个服务员到处在餐厅转悠，一旦有地方产生了垃圾，她会马上进行垃圾回收的工作。

然而，如果是八十年代的一些老饭馆，服务员的服务态度就比较差了。当顾客离开的时候，剩下的垃圾很有可能一直不去收拾，必须要等到新顾客来了以后没地方坐了，服务员才

会去收拾一下。

Sun 公司采用的垃圾回收的方式，是“最少回收”的方式：只有当内存不够的时候才会进行垃圾回收。形象的说，Sun 公司的 JVM，采用的是八十年代老饭馆的方式。这是因为回收垃圾必然需要占用 CPU，最少回收虽然可能会浪费一点空间，但是能够减少垃圾回收的次数，从而降低垃圾回收对 CPU 时间的占用，提高程序的执行效率。

但是“最少回收”的机制下，当对象成为垃圾对象之后，到 JVM 真正回收这个资源，可能之间会有很长的一段时间。

正因为如此，我们不应该在 finalize 中写上释放资源的代码。原因就像我们刚刚所说，当一个对象成为垃圾对象以后，可能并没有马上进行垃圾回收。如果把释放资源的代码写在 finalize 中，那么从对象成为垃圾对象，到对象真正被垃圾回收的这段时间，资源始终没有释放，这样就会造成资源的浪费。

例如，如果一个类要使用数据库资源，如果把数据库资源的释放写在 finalize 里，那么当这个对象成为垃圾，而没有被垃圾回收的时候，这段时间数据库资源始终被这个对象占用，可能就会造成其他对象访问不了数据库。因此，释放资源的代码不应该写在 finalize 里，而应该采用别的方式，一旦资源不使用了马上就释放，而不是依赖垃圾回收及 finalize 方法。

在 Java 中还有一个 System.gc()方法。调用这个方法，就相当于通知 JVM，程序员希望能够进行垃圾回收。但是调用 gc 也不能保证马上就能进行垃圾回收，这一切都要看当时 JVM 的运行状态。举例来说，System.gc()方法就相当于大厨在后灶吆喝：“前面的服务员，赶紧收拾一下桌子，厨房都没有干净盘子用了！”。如果当时服务员正闲着，有可能马上就帮大厨把垃圾回收了，但是如果当时服务员正忙，就有可能忽略大厨的要求，仍然按照自己的意愿，在高兴的时候再进行垃圾回收。

总结一下：finalize 方法在对象被垃圾回收的时候调用。但是，由于 Sun 公司的 JVM 采用的是“最少”回收的机制，因此不应当把释放资源的代码写在 finalize 方法中。

1.2 getClass

getClass 方法是 Object 类中的一个公开方法，这个方法的作用是：返回对象的实际类型。如何来理解对象的实际类型呢？

考虑如下继承关系：

```
class Animal{}
class Dog extends Animal{}
class Courser extends Dog{}
```

在这个继承关系中，Dog 类表示狗，继承自 Animal 类；Courser 类继承自 Dog 类，表示猎犬。

考虑下面的逻辑：写一个函数，接受一个 Animal 类型的参数，当这个 Animal 类型的引用指向一个 Dog 类型的对象时，返回 true，否则返回 false。

这个函数应该如何写呢？根据我们之前学习的多态的内容，可以利用 instanceof 操作符来进行判断。示例代码如下：

```
public static boolean isDog(Animal ani){
    if (ani instanceof Dog){
        return true;
    }else return false;
```

```
}
```

这样的代码却有一个问题，当传入的 ani 参数所指向的对象是一个 Courser 对象时，这个函数同样返回 true。因为根据多态，Courser 是 Dog 类的子类，因此当 ani 指向一个 Courser 对象时，返回值也是 true。这与我们的要求并不一致，我们希望的是，仅仅当 ani 参数指向一个实际类型为 Dog 类的对象时才返回 true。

应该怎么解决这个问题呢？针对这个问题，我们可以使用 getClass 方法来解决。getClass 方法能够返回一个对象的实际类型，通过比较两个对象 getClass 的返回值，就能够判断着两个对象是否是同一个类型。例如有如下代码：

```
Dog d1 = new Dog();
Dog d2 = new Dog();
Dog d3 = new Courser();
//d1 和 d2 实际类型相同，因此 getClass 方法返回值相同，输出 true
System.out.println(d1.getClass() == d2.getClass());
//d1 和 d3 实际类型不同，因此 getClass 方法返回值不同，输出 false
System.out.println(d1.getClass() == d3.getClass());
```

因此，利用 getClass 方法，就能够避免 instanceof 操作符的麻烦。改写 isDog 方法如下：

```
public static boolean isDog(Animal ani){
    Dog d = new Dog();
    if (ani.getClass() == d.getClass() ){
        return true;
    }else return false;
}
```

关于 getClass 方法的更多内容，我们将在反射章节中做更加详细的介绍。

1.3 equals

equals 方法是 Object 类中定义的方法，其方法签名为：

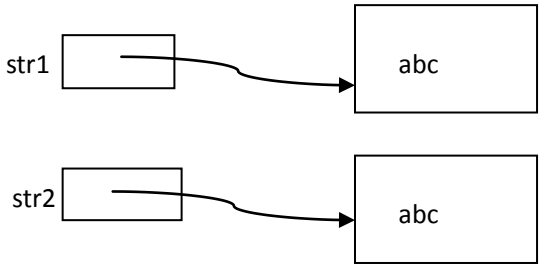
```
public boolean equals(Object obj)
```

由于 equals 是 Object 类中的公开方法，这意味着在 java 中所有对象都包含了 equals 方法。这个方法用来判断两个对象内容是否相等。要注意的是 equals 方法和双等号“==”之间的区别。例如，考虑下面的代码：

```
String str1 = new String("abc");
String str2 = new String("abc");
System.out.println(str1 == str2);
```

上面的代码，输出结果为 false。原因在于，使用双等号比较两个对象类型，比较的是两个引用中保存的地址，而不是对象的值。

在上面的三行代码执行之后，在内存中的示意图如下：



也就是说，str1 和 str2 这两个引用分别指向两个不同的对象。由于 str1 和 str2 指向不同

的对象，也就意味着 str1 中保存的内存地址和 str2 中保存的内存地址一定不相同，因此，那双等号比较这两个引用，返回值为 false。

但是，从另一个意义上说，有没有办法能够比较两个对象的内容是否相等呢？因为 str1 和 str2 这两个引用所指向的对象都包含字符串“abc”，因此，从对象的内容上来看，str1 和 str2 应该是相等的。

那如何比较两个对象的内容呢？我们可以调用 equals 方法来比较两个对象的内容是否相等。例如下面的代码：

```
String str1 = new String(“abc”);
String str2 = new String(“abc”);
//比较两个引用是否指向同一个对象，输出 false
System.out.println(str1 == str2);
//比较两个对象内容是否相等，输出为 true
System.out.println(str1.equals(str2));
```

从上面的例子可知，equals 方法是用来判断对象的内容是否相等。

注意 equals 方法的签名：这个方法接受一个 Object 类型的对象 obj 做为参数。equals 方法比较的就是当前对象（this）和 obj 这两个对象的内容。例如，对于 str1.equals(str2)这个比较而言，当前对象就是 str1 对象，而 obj 对象就是 str2 对象。

要注意的是，在某些情况下需要程序员覆盖 equals 方法。例如下面的代码：

```
class Student {
    String name;
    int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}
public class TestStudent{
    public static void main(String args[]){
        Student stu1 = new Student(“Tom”, 18);
        Student stu2 = new Student(“Tom”, 18);
        System.out.println(stu1 == stu2);
        System.out.println(stu1.equals(stu2));
    }
}
```

上面这个程序，输出结果为两个 false。第一个 false 好理解，stu1 和 stu2 两个引用分别指向两个不同的对象，两个引用中存放的地址不同，因此返回值为 false。

但是第二个输出语句中，我们调用了 equals 方法来比较两个对象。在这两个学生对象中，两个学生对象的姓名相同，两个学生对象的年龄也相同，但是比较的结果依然是 false。这是怎么回事呢？

在我们的 Student 对象中并没有定义 equals 方法，因此在调用 stu1.equals 方法时，调用的实际上是 Student 类从 Object 类中继承的 equals 方法。那 Object 类中的 equals 方法进行判断时，判断的就是引用是否相等。以下代码来源于 Sun 公司 JDK6 的源码（扩展知识：在你的 JDK 安装目录下，有一个 src.zip 文件，这个压缩包中包含了 JDK 的源码。下面的代码，

就来源于这个文件中的 java/lang/Object.java 文件）

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

可以看到，在 Object 类中的 equals 方法，使用的是双等号进行的引用的比较，因此，Object 类中的 equals 方法不能够帮助我们进行两个学生对象是否相等的比较。

为了替换掉 Object 类中对 equals 的实现，我们应当覆盖 equals 方法。也就是说，程序员应当自己来指定两个对象的比较准则。

要注意的是，实现的 equals 方法，应当满足以下几个条件：

- 自反性。自反性指的是，如果一个引用 x!=null，则 x.equals(x)应当为 true。也就是说，无论什么情况，一个对象自己跟自己比较，必须为 true。
- 对称性。对称性指的是，如果有两个不为 null 的引用 x 和 y，如果 x.equals(y)为 true，则 y.equals(x)也为 true；而如果 x.equals(y)为 false，则 y.equals(x)也为 false。
  这一条表明，在使用 equals 比较的时候，x.equals(y)和 y.equals(x)的结果是一样的。从理解上来说，可以把这一条当做交换律来理解。
- 传递性。这条准则指的是，如果有三个不为 null 的引用 a、b、c，如果 a.equals(b)为 true，b.equals(c)为 true，则 a.equals(c)也必然为 true。
  从逻辑上说，如果 a 和 b 相等，b 和 c 相等，则 a 和 c 必然相等。
  举一个反例：假设，判断两个学生是否相等的时候，我们制定这样的比较规则：看这两个学生的成绩相差是否在 5 分以内。
  这样判断的话，如果学生 A 考了 85 分，学生 B 考了 89 分，则 A 和 B 两个学生相等。如果学生 C 考虑 93 分，则 B 和 C 学生相等。但是，93-85>5，因此 A 和 C 不相等。这就违反了传递性原则，因此我们制定的比较规则是错误的。
- 一致性。这条准则比较简单，指的是如果有两个不为 null 的引用 x 和 y，在不改变 x 和 y 的属性的前提下，每次调用 x.equals(y)返回的值都相同。也就是说，如果你不改变 x 和 y 的属性，则每次比较这两个对象，结果应该一致，不能因为时间的变化等因素而影响比较的结果。
- 如果 x 不为 null，则 x.equals(null)应当返回 false。

根据这些条件，在开发 equals 方法时，有一套固定的模式。只要按照这个模式进行开发，就一定能写出满足 equals 方法的这几条准则，并且能符合程序员要求的代码。以上文的 Student 类为例，equals 方法的写法如下：

```
public boolean equals(Object obj){
    //判断 obj 是否和 this 相等，保证自反性
    if (obj == this) return true;
    //判断 obj 是否为 null，保证最后一条准则
    if (obj == null) return false;
    //判断两个对象的实际类型是否相等，
    //如果不相等，则说明比较的是两个不同种类的对象，应当返回 false
    if (obj.getClass() != this.getClass()) return false;
    //强制类型转换
    //由于之前已经使用 getClass 判断过实际类型，因此这里强转是安全的
    Student stu = (Student) obj;
    //判断每个属性是否相等
    // 对于基本类型的属性用“==”比较，对象类型的属性用 equals 比较
```

```
        if (this.age == stu.age && this.name.equals(stu.name) )
            return true;
        else return false;
    }
```

总结一下，equals 方法的五个步骤：

1、 判断 this == obj

2、 判断 obj == null

3、 判断两个对象的实际类型（使用 getClass()方法）

4、 强制类型转换

5、 依次判断两个对象的属性是否相等

1.4 toString

toString 方法也是 Object 类中定义的方法，这意味着这个方法是 Java 中所有对象都有的方法。这个方法的签名如下：

```
public String toString()
```

这个方法没有参数，返回值类型是一个 String 类型。这个方法的返回值是某个对象的字符串表现形式。如何来理解“字符串表现形式”呢？请看这个例子：

```
class Student{
    String name;
    int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}
public class TestToString{
    public static void main(String args[]){
        Student stu = new Student();
        System.out.println(stu.toString());
    }
}
```

在这个例子中，定义了 Student 类型，每一个 Student 对象都代表了一个学生。为了打印学生的信息，在打印语句中我们可以调用 toString 方法，这个方法返回了学生对象的字符串表现形式。也就是说，这个方法返回了学生对象的信息，这些信息组成了表示学生对象的字符串，也就是学生信息的字符串表现形式。

我们也可以这样使用打印语句：

```
System.out.println(stu.toString()); //手动调用 toString 方法
System.out.println(stu); //直接打印 stu 对象
```

上面的两行代码运行结果如下：

```
Student@c17164
Student@c17164
```

可以看出，无论是手动调用 toString 方法，还是直接打印 stu 对象，所得的结果都是一样的。这意味着，如果打印 stu 对象的话，就相当于打印 stu 的 toString 方法返回值。

另外一方面，打印出的信息是什么意思呢？其中，Student 表示的是对象的类名，而后面的@XXXXX 表示的是对象相应的内存地址。事实上，这种信息往往不是我们需要的信息，因为我们打印学生对象的信息时，不需要知道其地址，事实上也无法直接操作其地址。

我们调用学生对象的 toString 方法，目的是获知学生对象的相关信息，例如学生的姓名和年龄等。而由于在上面的代码中，我们没有为 Student 类写 toString 方法，这意味着学生对象中的 toString 方法是从 Object 类中继承来的，因此打印出的是类名以及相关地址。为了让 toString 方法能打印出我们想要的内容，我们可以覆盖这个方法如下：

```
public String toString(){
    return name + " " + age;
}
```

这样，打印学生对象时就会显示相应的学生信息。

完整代码如下：

```
class Student{
    String name;
    int age;
    public Student(){}
    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return name + " " + age;
    }

}
public class TestToString{
    public static void main(String args[]){
        Student stu = new Student("Tom", 18);
        System.out.println(stu.toString());
        System.out.println(stu);
    }
}
```

运行结果如下：

```
Tom 18
Tom 18
```

可以看出，无论用哪种方式，都打印的是 Student 类中 toString()方法的返回值。

此外，我们在介绍 String 类型时曾经提过，String 类型与任何其他类型相加，结果都是 String 类型。这其中，“其他类型”既包括简单类型，又包括对象类型。例如下面的代码：

```
Student stu = new Student("Tom", 18);
String str = "my string " + stu;
System.out.println(str);
```

上面的例子中，我们使用一个“my string”字符串与一个学生对象相加，结果还是一个字符串。在生成这个字符串的时候，会把一个对象转换成一个字符串，转换的方式，就是调

用这个对象的 toString 方法并获取其返回值。上面的字符串加法的代码：

```
String str = "my string " + stu;
```

这句代码表示把“my string”字符串和 stu 的 toString 方法返回值相加，并把相加的结果赋值给 str 变量。

上述代码的运行结果如下：

```
my string Tom 18
```

2 包装类

由于 Object 类是所有对象的父类，因此 Object 类型的引用能够接受 Java 中所有类型的对象。但是，对于基本类型，Object 类就无能为力了，毕竟 Object 只能处理对象类型。

有没有什么办法，能够让 Object 类处理 Java 中所有的数据类型呢？在 Java 中解决这个问题的方法就是使用包装类。

2.1 包装类简介

包装类是为了把基本类型包装成对象类型，从而让其也成为对象类型，从而能被 Object 类型统一管理。那如何包装呢？下面我们以 int 类型的包装类为例，创建一个我们自己的 int 类型的包装类。

```
class MyInteger{
    private int value;
    public MyInteger(int value){
        this.value = value;
    }
    public int intValue(){
        return value;
    }
}
```

这样，我们定义了一个对象类型 MyInteger，这个类型封装了一个 int 类型的 value，从而把一个简单类型 int 类型封装成了一个对象类型。用这个 MyInteger 类的对象，同样可以表示一个整数。

事实上，在 Sun 公司的 JDK 中，已经为我们封装好了类似的对象，我们可以直接使用这些对象。简单类型和其对应的包装类类型如下表：

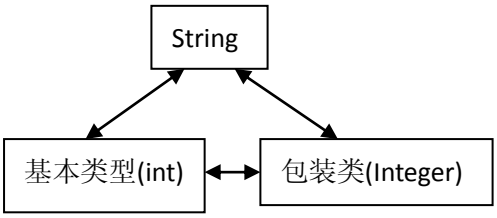
基本类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

这就是 Sun 公司为我们提供的八种包装类。

2.2 六种转换

有了包装类之后，就要研究一下包装类和基本类型之间应该如何转换。除此之外，还有其他类型的转换：字符串和基本类型以及包装类之间的转换。由于在互联网上传输时，大部分情况下不会进行数值的传递，很多情况下传递的是字符串。例如，在一些购物网站上下订单进行消费时，会输入产品数量等信息。这些信息从浏览器传输到服务器时，并不是按照数值的方式进行传递，而是把数值转化成字符串进行网络传输。这样，就要求我们在服务器端完成字符串类型到数值类型的转换。

接下来，我们就为大家介绍三种类型之间的六种转换。所谓的三种类型，指的是字符串类型、基本类型、包装类类型。如下图示：



首先介绍一下包装类和基本类型之间的转换。以 Integer 和 int 为例，由 int 转换为 Integer 的方式，是利用 Integer 的构造方法，以 int 作为参数创建相应的 Integer 对象。如下面代码所示：

```
int i1 = 10;
//int 类型转化为 Integer 类型
Integer ii1 = new Integer(i);
```

那如何由 Integer 类型转换为 int 类型呢？我们可以直接调用 Integer 对象的 intValue 方法，用这个方法就可以把 Integer 类型转换为 int 类型。如下面代码所示：

```
int i2 = ii1.intValue();
```

下面是 String 类型与包装类之间的转换。字符串转为包装类类型非常简单，同样是利用包装类的构造方法，例如：

```
String str = "123";
Integer ii2 = new Integer(str);
```

这样就可以把字符串类型转为相应的包装类类型。而包装类类型转为字符串类型也非常简单，直接调用包装类对象的 toString 方法，就能返回该对象的字符串表现形式，也就是把对象转化为字符串。如下面代码所示：

```
String str2 = ii2.toString();
```

那字符串类型和基本类型之间应该如何转化呢？首先是基本类型转换成字符串。这种转化有很多种不同的转换方式，下面为大家介绍两种不同的方式。

一种方式是使用 String 类中定义的一个静态方法：valueOf。因为这个方法是静态方法，因此可以用 String 类名来直接调用。在 String 类中定义了一系列重载的 valueOf 方法，分别表示把不同的基本类型转换为字符串类型。示例代码如下：

```
int i = 10;
String str3 = String.valueOf(i);
```

第二种方式，我们可以利用字符串的特性，使用字符串加法来完成基本类型和字符串类型之间的转换。例如：

```
String str4 = "" + i;
```

由于字符串类型加上任何其他类型，结果都是字符串，因此我们可以用一个空字符串加上一个整数，从而把这个整数转化为一个字符串。

那字符串类型如何转化为基本类型呢？还是以 int 为例，如果要转换的话，可以调用 Integer 类的 parseInt 方法，把字符串转化为 int。示例代码如下：

```
int i = Integer.parseInt("123");
```

由此，我们就给大家介绍完了三种类型（字符串、基本类型、包装类）之间的相互转化。建议各位读者到这里先暂停一下，练习一下这些转化，试着把字符串、Double 和 double 类型之间进行六种转换。

2.3 JDK5.0 新特性：自动封箱拆箱

在掌握了三种类型、六种转换之后，我们介绍一个 JDK5.0 的新特性：自动封箱和拆箱。这个特性的核心就是一句话：在 JDK5.0 中，由系统自动完成基本类型和包装类之间的转换。例如，对于 JDK5.0 以前的代码，要进行 int 和 Integer 类型之间的转换，必须要把代码写成下面的形式：

```
//把 int 类型的 10 转换为 Integer 类型
Integer myInteger = new Integer(10);
//把 Integer 类型的 myInteger 转换为 int 类型
int i = myInteger.intValue();
```

而对于 JDK5.0 以后的版本来说，下面的代码也能编译通过：

```
//把 int 类型直接赋值给 Integer 类型
Integer myInteger = 10;
//把 Integer 类型直接赋值给 int 类型
int i = myInteger;
```

上面的这两行代码，可以把 int 类型的变量直接赋值给 Integer 类型，也可以把 Integer 类型的变量直接赋值给 int 类型。也就是说，可以对 Integer 类型和 int 类型自动进行转换。

需要注意的是，这个新特性只是语法方面的改变，而底层实现的原理，并没有发生改变。也就是说，从语法上说，

```
Integer myInteger = 10;
```

是把一个 int 类型的值直接赋值给一个 Integer 类型的值。但是，在 JDK5.0 的编译器编译这段代码时，会自动把上面的代码翻译成下面的形式：

```
Integer myInteger = new Integer(10);
```

也就是说，从底层的工作原理上说，与 JDK5.0 以前的版本是一样的。只不过 JDK5.0 的编译器提供了一种新的语法，能够帮助我们让代码变得更清晰。

同样的，从语法上说，

```
int i = myInteger;
```

是把一个 Integer 类型的变量直接赋值给 int 类型的变量。但是在底层实现中，JDK5.0 的编译器会自动把上面的代码翻译成如下形式：

```
int i = myInteger.intValue();
```

例如，下面的例子：

```
public class TestAutoBoxing {
    public static void main(String[] args) {
        Integer myInteger = null;
        int i = myInteger;
    }
```

```
}
```

上面的代码在运行时，会产生一个 NullPointerException 的异常。为什么会产生这个异常呢？因为

```
int i = myInteger;
```

这行代码，编译器会自动翻译成：

```
int i = myInteger.intValue();
```

在这个过程中，由于调用 intValue 方法时，myInteger 的值为 null，相当于对一个 null 值调用了 intValue()方法。因此，这行代码会抛出 NullPointerException 这个异常。

自动封箱和拆箱的语法和功能并不复杂，但是有时候，这个特性能够帮我们解决很大的问题。例如，如果我们有一个 Integer 类型的值：

```
Integet myInt = new Integer(10);
```

现在的需求是：让 myInt 的值加 1。如果不使用自动封箱和拆箱，则代码应当这么写：

```
Integer myInt = new Integer(10);
int t = myInt.intValue();
t++;
myInt = new Integer(t);
```

而如果使用自动封箱的话，只用一句：

```
myInt++;
```

就可以完成上述的功能。这个例子说明，自动封箱和拆箱能够很大程度上，帮助程序员简化代码，让程序的结构更清晰，可读性更强。

3 内部类

内部类是 Java 中很特殊的一个语法。一方面，内部类能够一定程度上的减少代码量，并且能够为程序员提供一些语法方面的比较方便的功能。另一方面来说，内部类有可能带来非常古怪的语法，这些语法会造成代码的可读性下降。因此，内部类是一把双刃剑。从规范上说，我们并不提倡程序员过多的使用内部类，不过依然应该了解一些内部类的基本语法和部分内部类的使用方法。

Java 中的内部类分为四种：成员内部类、静态内部类、局部内部类、匿名内部类，我们下面对这四种内部类分别进行介绍。

3.1 成员内部类

我们在定义类的时候，会定义成员变量。例如下面的程序：

```
class MyOuterClass{
    private int value;
}
```

其中，value 被成为 MyOuterClass 的属性，也被称为成员变量。在 value 的位置定义一个类，则这个类就被称为成员内部类。如下面代码所示：

```
class MyOuterClass{
    private int value;
    private class InnerClass{
```

```
        public void m(){
            System.out.println(value);
        }
    }
}
```

上面这段代码演示了如何定义一个成员内部类。首先，定义成员内部类的位置，在某一个类的里面，在任何一个函数的外面（也就是定义成员变量的位置）。

其次，由于成员内部类是属于外部类的内部，因此在成员内部类中可以访问外部类的私有成员。例如，上面的程序中，在 InnerClass 的 m 方法中，就访问了外部类 MyOuterClass 的私有属性 value。

第三，成员内部类可以作为 private 的。之前我们学习修饰符的时候曾经提过，private 不能修饰类。但是 private 可以修饰成员内部类，这表示只能在 MyOuterClass 的内部是用 InnerClass 类。

编译上面的程序，会在相应的目录中生成两个.class 文件：一个 MyOuterClass.class 文件和一个 MyOuterClass$InnerClass.class 文件。这说明，在编译时，内部类也会生成独立的.class 文件，换句话说，内部类也是独立的类。

下面我们来看一下，如何来使用成员内部类创建对象。第一种情况，在外部类中，可以直接创建内部类的对象。（在上面的例子中，指的是在 MyOuterClass 类中，可以直接创建内部类的对象）。例如，我们可以为 MyOuterClass 添加一个 f 方法，在 f 方法中创建 InnerClass 对象。示例代码如下：

```
class MyOuterClass{
    private int value = 100;
    private class InnerClass{
        public void m(){
            System.out.println(value);
        }
    }
    public void f(){
        InnerClass ic = new InnerClass();
        ic.m();
    }
}

public class TestInnerClass {
    public static void main(String[] args) {
        MyOuterClass moc = new MyOuterClass();
        moc.f();
    }
}
```

在上面的这段程序中，运行结果输出 100，也就是 value 的值。

第二种情况，在外部类的外部创建对象。以上面的例子来说，也就是在 MyOuterClass 类的外部如何创建内部类对象。我们来修改上面的例子，在 TestInnerClass 类中创建 InnerClass 类型的对象。

首先，为了能够让 InnerClass 在 MyOuterClass 类的外部创建，因此必须要把 InnerClass

的访问权限修改一下，去掉 private 修饰符。修改后的 MyOuterClass 以及 InnerClass 的代码如下：

```
class MyOuterClass{
    private int value = 100;
    //增加一个 value 属性的 get/set 方法
    public void setValue(int value){
        this.value = value;
    }
    public int getValue(){
        return value;
    }
    //去掉 private 修饰符
    class InnerClass{
        public void m(){
            System.out.println(value);
        }
    }
    public void f(){
        InnerClass ic = new InnerClass();
        ic.m();
    }
}
```

修改之后，接下来就要在 TestInnerClass 类中创建 InnerClass 类型的对象。

首先，内部类对象的类型，需要写成“外部类.内部类”。也就是说，在主方法中首先定义一个变量，代码如下：

```
MyOuterClass.Inner inner;
```

其次，创建一个内部类对象的时候，必须首先创建一个外部类的对象。代码如下：

```
MyOuterClass moc = new MyOuterClass();
```

下面到了最激动人心的时刻：我们要正式创建成员内部类对象了！创建的语法：

```
inner = moc.new InnerClass();
```

你没有看错，在 new 关键字前面，是一个 MyOuterClass 类型的引用名！这就是最最奇怪的地方。这说明，每一个成员内部类对象都要跟一个外部类对象相关联。

为什么成员内部类对象要跟外部类对象关联呢？考虑这样的问题：在 InnerClass 的 m 方法中，打印了外部类的 value 属性。这个属性是外部类的实例变量，也就是说，必定是某一个外部类对象的属性。假设创建了两个外部类对象：

```
MyOuterClass moc1 = new MyOuterClass();
moc1.setValue(10);
MyOuterClass moc2 = new MyOuterClass();
moc2.setValue(20);
```

这样，就创建了两个对象，这两个对象分别有自己的 value 属性。接下来需要创建成员内部类对象。由于内部类对象能够访问 value 属性，因此在创建成员内部类对象的时候，必须要说清楚，这个成员内部类对象究竟以那个对象为外部对象，究竟打印的是哪个 value 值。

例如，如果有下面的代码：

```
MyOuterClass.InnerClass in1 = moc1.new InnerClass();
```

上面的代码说明了 in1 这个对象的外部对象为 moc1，因此调用 in1 的 m 方法时，打印的是 moc1 对象的 value 值，因此打印的是 10。

完整的 TestInnerClass 对象的代码如下：

```
public class TestInnerClass {
    public static void main(String[] args) {
        MyOuterClass moc = new MyOuterClass();
        MyOuterClass.InnerClass ic = moc.new InnerClass();
        ic.m();
    }
}
```

最后要说明的一点，由于成员内部类必须与外部类某一个对象相关联，因此成员内部类中不能定义静态方法。

3.2 静态内部类

静态内部类与成员内部类相似，只有一点不同：在定义成员内部类之前，增加一个 static 关键字，则定义的内部类就成为一个静态内部类。示例代码如下：

```
class MyOuterClass{
    //成员内部类
    class MemberInner{}
    //静态内部类
    static class StaticInner{}
}
```

需要说明的是，static 关键字不能够用来修饰类，但是能够用来修饰内部类。

接下来我们看一下静态内部类以及成员内部类对外部类成员的访问。

请看如下代码：

```
class MyOuterClass{
    int value1 = 100;
    public void m1(){}

    static int value2 = 200;
    public static void m2(){}

    class InnerClass1{
        public void f1(){
            //成员内部类中能够访问外部类的静态成员以及非静态成员
            System.out.println(value1);
            System.out.println(value2);
            m1();
            m2();
        }
        //!public static void f2(){}
        //编译出错，成员内部类不能定义静态方法
    }
```

```
    static class InnerClass2{
        public void f1(){
            //静态内部类中只能访问外部类的静态成员
            //System.out.println(value1);  出错!
            System.out.println(value2);
            //! m1();
            m2();
        }
        //静态内部类中可以定义静态方法
        public static void f2(){}
    }
}
```

由上面的代码，我们可以得出以下结论：

成员内部类中不能定义静态方法;成员内部类中能够访问外部类的所有静态以及非静态的成员；

静态内部类中可以定义静态方法，静态内部类中只能访问外部类的静态成员（即访问外部类的静态属性或者调用外部类的静态方法。）

至于如何创建静态内部类的对象，也分两种情况

第一种情况，在外部类中，可以直接创建内部类的对象。（在上面的例子中，指的是在 MyOuterClass 类中，可以直接创建内部类的对象）。这种情况与成员内部类的情况相同，在此不再赘述。

第二种情况，在外部类的外部创建对象。以上面的例子来说，也就是在 MyOuterClass 类的外部如何创建内部类对象。这种情况下，静态内部类比成员内部类要简单的多。直接使用“外部类.内部类”的方式就能够创建。示例代码如下：

```
MyOuterClass.InnerClass2 in2 = new MyOuterClass.InnerClass2();
```

完整的代码如下：

```
class MyOuterClass{
    private int value1 = 100;
    private static int value2 = 200;

    static class InnerClass2{
        public void m(){
            System.out.println(value2);
            m2();
        }

    }

    class InnerClass1{
        public void m(){
            System.out.println(value1);
            System.out.println(value2);
            m1();
```

```
            m2();

        }
    }

    public void m1(){}
    public static void m2(){}

    public void f1(){
        InnerClass1 in1 = new InnerClass1();
        InnerClass2 in2 = new InnerClass2();
    }

}


public class TestInnerClass {

    public static void main(String[] args) {
        //创建成员内部类对象
        MyOuterClass moc = new MyOuterClass();
        MyOuterClass.InnerClass1 inc1 = moc.new InnerClass1();
        //创建静态内部类对象
        MyOuterClass.InnerClass2 inc2 =
                new MyOuterClass.InnerClass2();
    }

}
```

3.3 局部内部类

首先，我们回顾一下局部变量的概念：所谓局部变量，指的是定义在方法内部的变量。局部内部类与这个概念类似：所谓局部内部类，指的是定义在方法内部的类。示例代码如下：

```
class Outer{
    private int value = 100;
    private static int value2 = 200;
    public void m(){
        int localValue = 300;
        class Inner{
            public void f(){
                System.out.println(value);
                System.out.println(value2);
            }
        }
        Inner in = new Inner();
```

```
            in.f();
        }
    }
```

在上面的代码中，在外部类 Outer 的方法 m 内部，我们定义了一个 Inner 类，这个类就是一个局部内部类。

与局部变量一样，局部内部类访问范围就是在方法内部。也就是说，定义的这个 Inner 类只有在 m 方法内部是用，而无法在 m 方法外部使用。

从上面的代码我们还可以看出，局部内部类能够访问外部类的私有属性、静态属性。

除此之外，在 m 方法内部，定义了一个 localValue 变量。这个变量是一个局部变量，其作用范围是在 m 方法内部。然而，Inner 类也在 m 方法内部，那在 Inner 类中能否访问外部类的局部变量呢？

试着修改一下 Inner 类的 f 方法如下：

```
public void f(){
    System.out.println(value);
    System.out.println(value2);
    System.out.println(localValue);
}
```

加上一句输出 localValue 的语句，结果会产生一个编译时错误！

为什么呢？请牢牢记住下面的规则：在局部内部类中能够访问外部类的局部变量，但是要求该变量必须是 final 的！

修改之后的代码如下：

```
class Outer{
    private int value = 100;
    private static int value2 = 200;
    public void m(){
        final int localValue = 300;
        class Inner{
            public void f(){
                System.out.println(value);
                System.out.println(value2);
                System.out.println(localValue);
            }
        }
        Inner in = new Inner();
        in.f();
    }
}
```

这就是局部内部类的语法。

那么局部内部类有什么作用呢？考虑下面的代码：

```
interface Teacher{
    void teach();
}

class Tom implements Teacher{
```

```
    public void teach(){
        System.out.println("Tom teach");
    }
}
class Jim implements Teacher{
    public void teach(){
        System.out.println("Jim teach");
    }
}
public class TestTeacher{
    public static void main(String args[]){
        Teacher t = getTeacher(10);
        t.teach();
    }
    public static Teacher getTeacher(int n){
        if (n == 10) return new Tom();
        else return new Jim();
    }
}
```

上面的代码中，我们创建了一个 Teacher 接口，这个类型用来表示一个老师。然后，Tom 和 Jim 可以认为是两个不同的老师，他们都实现了 Teacher 接口。在 getTeacher 方法内部，根据参数的不同，可以创建 Tom 和 Jim 对象并作为返回值返回。在主方法中，可以调用 getTeacher 方法，从而获得 Teacher 类型实现类的对象，之后就可以调用实现类的 teach 方法。

上面的代码，由于在主方法中操作的都是接口 Teacher 类型，而没有与实现类接触。因此，我们可以认为主方法与 Teacher 接口的实现类（也就是 Tom 和 Jim 这两个类）是弱耦合的关系。

换成生活中的例子，我们可以把主方法当做是求学的学生：学生调用学校的 getTeacher 方法，由学校选派一位老师，然后学生调用老师的 teach 方法。在这个过程中，学校应该尽量避免让学生直接指定具体的老师，而是通过 Teacher 接口，让学生和老师之间形成一个弱耦合的关系。

但是，上面的代码却依然存在着强耦合的可能性。考虑下面这种主方法的写法：

```
public static void main(String args[]){
    Teacher t = new Tom();
    t.teach();
}
```

在这段代码中，创建了一个 Tom 对象。明明有 getTeacher 方法可以实现弱耦合，但是如果程序员不使用这个方法来获得 Teacher 对象，而是自己直接创建一个 Tom 对象，代码依然是强耦合的。

现在我们把代码修改一下：我们把 Tom 类和 Jim 类都放入到 getTeacher 方法的内部，这样，就把这两个类变成了局部内部类。修改后的代码如下：

```
interface Teacher{
    void teach();
}
```

```
public class TestTeacher{
    public static void main(String args[]){
        Teacher t = getTeacher(10);
        t.teach();
    }
    public static Teacher getTeacher(int n){
        class Tom implements Teacher{
            public void teach(){
                System.out.println("Tom teach");
            }
        }
        class Jim implements Teacher{
            public void teach(){
                System.out.println("Jim teach");
            }
        }
        if (n == 10) return new Tom();
        else return new Jim();
    }
}
```

这样，在 getTeacher 方法的外部，无法访问到 Tom 类和 Jim 类，也就无法直接创建出 Teacher 接口的实现类。也就是说，程序员只能通过调用 getTeacher 方法来获得 Teacher 对象，而无法直接创建某个实现类的对象。这样，就能够实现“强制弱耦合”，即强制程序员必须要利用 Teacher 接口来写程序，从而实现弱耦合。

3.4 匿名内部类

匿名内部类是一种特殊的局部内部类。如果一个局部内部类满足这样两个特点：1、该内部类继承自某个类或者实现某个接口；2、该内部类在整个方法中只创建了一个对象。满足这两个特点的局部内部类就能够改写成匿名内部类。

考虑上面 TestTeacher 程序中的两个局部内部类 Jim 和 Tom。这两个类都实现了 Teacher 接口，并且在 getTeacher 方法中各自只创建了一个对象，因此这两个类能够改写为匿名内部类的形式。下面我们演示一下如何把 Tom 类改写成匿名内部类：

```
    public static Teacher getTeacher(int n){
        //Jim 依然采用局部内部类的写法
        class Jim implements Teacher{
            public void teach(){
                System.out.println("Jim teach");
            }
        }

        //匿名内部类
        if (n == 10) return new Teacher(){
```

```
            public void teach(){
                System.out.println("Tom teach");
            }
        };
        else return new Jim();
    }
```

上面这段代码，把 Tom 类修改成了一个匿名内部类。我们分析一下这段代码。首先，return 语句返回了一个 Teacher 类型的对象。由于 Teacher 是一个接口，不能创建对象，因此所谓的 new Teacher()，实际上是创建了一个实现 Teacher 接口的类的对象。

那创建的对象是什么类型的呢？这个实现类没有名字，这也就是为什么这种语法要叫做“匿名”内部类的原因。而且正因为这个类没有名字，因此没有办法通过 new 类名()的方式创建对象。也就是说，匿名内部类没有名字，并且只能在一次方法调用中创建一个匿名内部类的对象。

那这个匿名内部类是如何实现 Teacher 接口的呢？为了实现 Teacher 接口，要在一对圆括号()后增加一个代码块，在这个代码块中写上对接口的实现。在上面的这个例子中，就是实现了 Teacher 接口中的 teach 方法。

最后，在这一对花括号{}之后，还有一个分号。要注意的是，这个分号并不是匿名内部类的语法特点，这个分号是 return 语句的结尾。对比 else 后面的那个 return 语句：这个语句创建一个 Jim 对象，并且以分号结尾；而 if 的那个分支中创建了一个匿名内部类对象，最后那个分号也是 return 语句的结尾。

终上所述，使用匿名内部类，是一种简便的写法。它将实现接口（或者是创建类）的代码和创建类的对象的代码结合在了一起。形成了 Java 中比较有特色的语法：

new 接口名 (){ 实现接口的代码 };

以上的代码首先用一对{}来实现了接口，然后用 new 关键字创建出了一个对象。请读者千万注意，这句话创建的绝不是接口的对象（接口是特殊的抽象类，无法创建对象），而创建的是一个实现了接口的，没有名字的内部类的对象！

既然匿名内部类是特殊的局部内部类，因此也具有局部内部类的特点。例如：匿名内部类不仅可以访问外部类的私有成员，还可以访问外部类的局部变量，但是要求这个局部变量被声明为“final”。

另外，由于匿名内部类没有明确的给出类的名字，因此，无法在匿名内部类中定义任何构造方法。因为我们知道，构造方法的名字必须和类名相同！