```
    }
}
```

运行结果如图 6.5 所示。

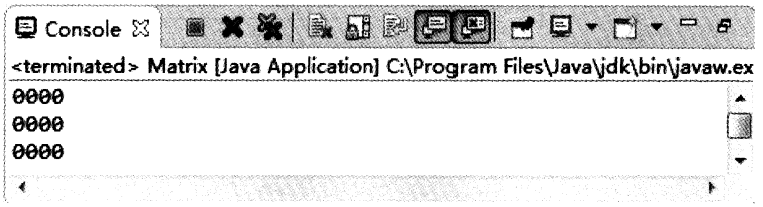

图 6.5　例 6.9 的运行结果

说明

对于整型二维数组，创建成功之后系统会赋给数组中每个元素初始值 0。

6.4　数组的基本操作

java.util 包的 Arrays 类包含了用来操作数组（如排序和搜索）的各种方法，本节就将介绍数组的基本操作。

6.4.1　遍历数组

视频讲解：光盘\TM\lx\6\遍历数组.mp4

遍历数组就是获取数组中的每个元素。通常遍历数组都是使用 for 循环来实现。遍历一维数组很简单，也很好理解，下面详细介绍遍历二维数组的方法。

遍历二维数组需使用双层 for 循环，通过数组的 length 属性可获得数组的长度。

【例 6.10】 在项目中创建类 Trap，在主方法中编写代码，定义二维数组，实现将二维数组中的元素呈梯形输出。（**实例位置：光盘\TM\sl\6.03**）

```
public class Trap {                                    //创建类
    public static void main(String[] args) {          //主方法
        int b[][] = new int[][]{{ 1 },{ 2, 3},{ 4, 5, 6 } };   //定义二维数组
        for (int k = 0; k < b.length; k++) {
            for (int c=0;c<b[k].length; c++){          //循环遍历二维数组中的每个元素
                System.out.print(b[k][c]);             //将数组中的元素输出
            }
            System.out.println();                      //输出空格
        }
    }
}
```

运行结果如图 6.6 所示。

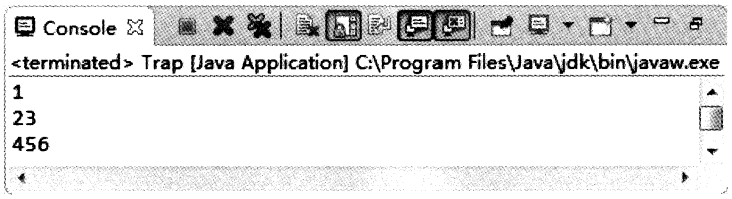

图 6.6　例 6.10 的运行结果

在遍历数组时，使用 foreach 语句可能会更简单。下面的实例就是通过 foreach 语句遍历二维数组。

【例 6.11】 在项目中创建类 Tautog，在主方法中定义二维数组，使用 foreach 语句遍历二维数组。（实例位置：光盘\TM\sl\6.04）

```
public class Tautog {                                          //创建类
    public static void main(String[] args) {                   //主方法
        int arr2[][] = { { 4, 3 }, { 1, 2 } };                 //定义二维数组
        System.out.println("数组中的元素是：");                  //提示信息
        int i = 0;                                             //外层循环计数器变量
        for (int x[] : arr2) {                                 //外层循环变量为一维数组
            i++;                                               //外层计数器递增
            int j = 0;                                         //内层循环计数器
            for (int e : x) {                                  //循环遍历每一个数组元素
                j++;                                           //内层计数器递增
                if (i == arr2.length && j == x.length) {       //判断变量是二维数组中的最后一个元素
                    System.out.print(e);                       //输出二维数组的最后一个元素
                } else
                    //如果不是二维数组中的最后一个元素
                    System.out.print(e + "、");                //输出信息
            }
        }
    }
}
```

运行结果如图 6.7 所示。

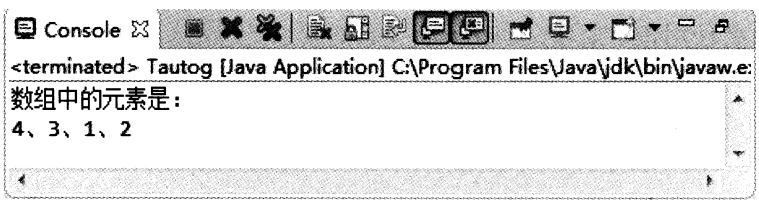

图 6.7　例 6.11 的运行结果

6.4.2　填充替换数组元素

视频讲解：光盘\TM\lx\6\填充替换数组元素.mp4

数组中的元素定义完成后，可通过 Arrays 类的静态方法 fill()来对数组中的元素进行替换。该方法

通过各种重载形式可完成对任意类型的数组元素的替换。fill()方法有两种参数类型，下面以 int 型数组为例介绍 fill()方法的使用方法。

（1）fill(int[] a,int value)

该方法可将指定的 int 值分配给 int 型数组的每个元素。

语法如下：

```
fill(int[] a,int value)
```

☑ a：要进行元素替换的数组。

☑ value：要存储数组中所有元素的值。

【例 6.12】 在项目中创建类 Swap，在主方法中创建一维数组，并实现通过 fill()方法填充数组元素，最后将数组中的各个元素输出。（实例位置：光盘\TM\sl\6.05）

```
import java.util.Arrays;                          //导入 java.util.Arrays 类
public class Swap {                               //创建类
    public static void main(String[] args) {      //主方法
        int arr[] = new int[5];                   //创建 int 型数组
        Arrays.fill(arr, 8);                      //使用同一个值对数组进行填充
        for (int i = 0; i < arr.length; i++) {    //循环遍历数组中的元素
            //将数组中的元素依次输出
            System.out.println("第" + i + "个元素是：" + arr[i]);
        }
    }
}
```

运行结果如图 6.8 所示。

（2）fill(int[] a,int fromIndex,int toIndex,int value)

该方法将指定的 int 值分配给 int 型数组指定范围中的每个元素。填充的范围从索引 fromIndex（包括）一直到索引 toIndex（不包括）。如果 fromIndex == toIndex，则填充范围为空。

语法如下：

```
fill(int[] a,int fromIndex,int toIndex,int value)
```

☑ a：要进行填充的数组。

☑ fromIndex：要使用指定值填充的第一个元素的索引（包括）。

☑ toIndex：要使用指定值填充的最后一个元素的索引（不包括）。

☑ value：要存储在数组所有元素中的值。

注意

如果指定的索引位置大于或等于要进行填充的数组的长度，则会报出 ArrayIndexOutOfBoundsException（数组越界异常，关于异常的知识将在后面的章节讲解）异常。

【例 6.13】 在项目中创建类 Displace，创建一维数组，并通过 fill()方法替换数组元素，最后将数组中的各个元素输出。（实例位置：光盘\TM\sl\6.06）

```
import java.util.Arrays;                          //导入 java.util.Arrays 类
public class Displace {                           //创建类
```

```
    public static void main(String[] args) {          //主方法
        int arr[] = new int[] { 45, 12, 2, 10 };      //定义并初始化 int 型数组 arr
        Arrays.fill(arr, 1, 2, 8);                    //使用 fill()方法对数组进行初始化
        for (int i = 0; i < arr.length; i++) {        //循环遍历数组中的元素
            //将数组中的每个元素输出
            System.out.println("第" + i + "个元素是：" + arr[i]);
        }
    }
}
```

运行结果如图 6.9 所示。

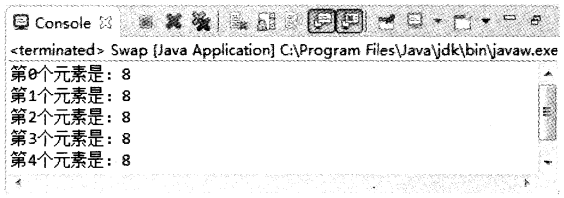

图 6.8　例 6.12 的运行结果

图 6.9　例 6.13 的运行结果

6.4.3　对数组进行排序

视频讲解：光盘\TM\lx\6\对数组进行排序.mp4

通过 Arrays 类的静态 sort()方法可以实现对数组的排序。sort()方法提供了多种重载形式，可对任意类型的数组进行升序排序。

语法如下：

```
Arrays.sort(object)
```

其中，object 是指进行排序的数组名称。

【例 6.14】 在项目中创建类 Taxis，在主方法中创建一维数组，将数组排序后输出。（**实例位置：光盘\TM\sl\6.07**）

```
import java.util.Arrays;                          //导入 java.util.Arrays 类
public class Taxis {                              //创建类
    public static void main(String[] args) {      //主方法
        int arr[] = new int[] { 23, 42, 12, 8 };  //声明数组
        Arrays.sort(arr);                         //将数组进行排序
        for (int i = 0; i < arr.length; i++) {    //循环遍历排序后的数组
            System.out.println(arr[i]);           //将排序后数组中的各个元素输出
        }
    }
}
```

运行结果如图 6.10 所示。

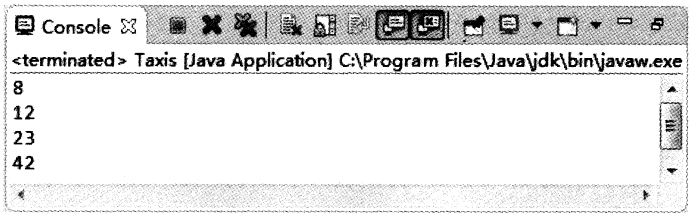

图 6.10　例 6.14 的运行结果

上述实例是对整型数组进行排序。Java 中的 String 类型数组的排序算法是根据字典编排顺序排序的，因此数字排在字母前面，大写字母排在小写字母前面。

6.4.4　复制数组

视频讲解：光盘\TM\lx\6\复制数组.mp4

Arrays 类的 copyOf()方法与 copyOfRange()方法可以实现对数组的复制。copyOf()方法是复制数组至指定长度，copyOfRange()方法则将指定数组的指定长度复制到一个新数组中。

（1）copyOf()方法

该方法提供了多种重载形式，用于满足不同类型数组的复制。

语法如下：

```
copyOf(arr,int newlength)
```

- ☑ arr：要进行复制的数组。
- ☑ newlength：int 型常量，指复制后的新数组的长度。如果新数组的长度大于数组 arr 的长度，则用 0 填充（根据复制数组的类型来决定填充的值，整型数组用 0 填充，char 型数组则使用 null 来填充）；如果复制后的数组长度小于数组 arr 的长度，则会从数组 arr 的第一个元素开始截取至满足新数组长度为止。

【例 6.15】 在项目中创建类 Cope，在主方法中创建一维数组，实现将此数组复制得到一个长度为 5 的新数组，并将新数组输出。（**实例位置：光盘\TM\sl\6.08**）

```
import java.util.Arrays;                          //导入 java.util.Arrays 类
public class Cope {                               //创建类
    public static void main(String[] args) {      //主方法
        int arr[] = new int[] { 23, 42, 12 };     //定义数组
        int newarr[] = Arrays.copyOf(arr, 5);     //复制数组 arr
        for (int i = 0; i < newarr.length; i++) { //循环变量复制后的新数组
            System.out.println(newarr[i]);        //将新数组输出
        }
    }
}
```

运行结果如图 6.11 所示。

（2）copyOfRange()方法

该方法同样提供了多种重载形式。

语法如下：

```
copyOfRange(arr,int formIndex,int toIndex)
```

- ☑ arr：要进行复制的数组对象。
- ☑ formIndex：指定开始复制数组的索引位置。formIndex 必须在 0 至整个数组的长度之间。新数组包括索引是 formIndex 的元素。
- ☑ toIndex：要复制范围的最后索引位置。可大于数组 arr 的长度。新数组不包括索引是 toIndex 的元素。

【例 6.16】 在项目中创建类 Repeat，在主方法中创建一维数组，并将数组中索引位置是 0~3 之间的元素复制到新数组中，最后将新数组输出。（**实例位置：光盘\TM\sl\6.09**）

```
import java.util.Arrays;                                    //导入 java.util.Arrays
public class Repeat {                                       //创建类
    public static void main(String[] args) {                //主方法
        int arr[] = new int[] { 23, 42, 12, 84, 10 };       //定义数组
        int newarr[] = Arrays.copyOfRange(arr, 0, 3);       //复制数组
        for (int i = 0; i < newarr.length; i++) {           //循环遍历复制后的新数组
            System.out.println(newarr[i]);                  //将新数组中的每个元素输出
        }
    }
}
```

运行结果如图 6.12 所示。

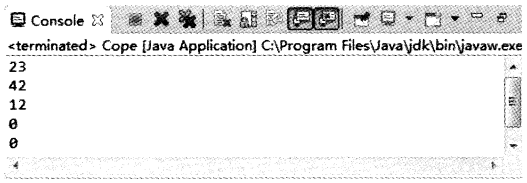

图 6.11　例 6.15 的运行结果

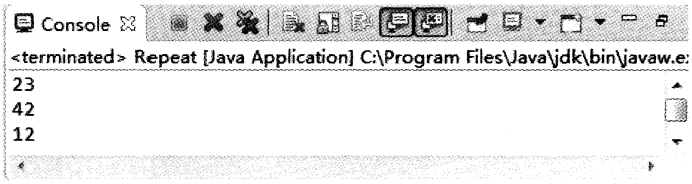

图 6.12　例 6.16 的运行结果

6.4.5　数组查询

Arrays 类的 binarySearch()方法，可使用二分搜索法来搜索指定数组，以获得指定对象。该方法返回要搜索元素的索引值。binarySearch()方法提供了多种重载形式，用于满足各种类型数组的查找需要。binarySearch()方法有两种参数类型。

（1）binarySearch(Object[],Object key)

语法如下：

```
binarySearch(Object[] a,Object key)
```

- ☑ a：要搜索的数组。
- ☑ key：要搜索的值。

如果 key 包含在数组中，则返回搜索值的索引；否则返回-1 或“-”（插入点）。插入点是搜索键将

要插入数组的那一点，即第一个大于此键的元素索引。

【例 6.17】 查询数组元素，实例代码如下：

```
int arr[] = new int[] { 4, 25, 10 };                    //创建并初始化数组
Arrays.sort(arr);                                       //将数组进行排序
int index = Arrays.binarySearch(arr, 0, 1, 8);
```

上面的代码中变量 index 的值是元素“8”在数组 arr 中索引在 0~1 内的索引位置。由于在指定的范围内并不存在元素“8”，index 的值是“-”（插入点）。如果对数组进行排序，元素“8”应该在“25”的前面，因此插入点应是元素“25”的索引值 2，所以 index 的值是-2。

如果数组中的所有元素都小于指定的键，则为 a.length（注意，这保证了当且仅当此键被找到时，返回的值将大于等于 0）。

注意

必须在进行此调用之前对数组进行排序（通过sort()方法）。如果没有对数组进行排序，则结果是不确定的。如果数组包含多个带有指定值的元素，则无法保证找到的是哪一个。

【例 6.18】 在项目中创建类 Reference，在主方法中创建一维数组 ia，实现查找元素 4 在数组 ia 中的索引位置。（**实例位置：光盘\TM\sl\6.10**）

```
import java.util.Arrays;                                //导入 java.util.Arrays 类
public class Example {                                  //创建类
    public static void main(String[] args) {            //主方法
        int ia[] = new int[] { 1, 8, 9, 4, 5 };          //定义 int 型数组 ia
        Arrays.sort(ia);                                //将数组进行排序
        int index = Arrays.binarySearch(ia, 4);          //查找数组 ia 中元素 4 的索引位置
        System.out.println("4 的索引位置是：" + index);    //将索引输出
    }
}
```

运行结果如图 6.13 所示。

说明

返回值“1”是对数组 ia 进行排序后元素 4 的索引位置。

（2）binarySearch(Object[],int fromIndex,int toIndex,Object key)

该方法在指定的范围内检索某一元素。

语法如下：

```
binarySearch(Object[] a,int fromIndex,int toIndex,Object key)
```

- ☑ a：要进行检索的数组。
- ☑ fromIndex：指定范围的开始处索引（包含）。
- ☑ toIndex：指定范围的结束处索引（不包含）。

☑ key：要搜索的元素。

在使用该方法之前同样要对数组进行排序，来获得准确的索引值。如果要搜索的元素 key 在指定的范围内，则返回搜索键的索引；否则返回-1 或“-”（插入点）。如果范围中的所有元素都小于指定的键，则为 toIndex（注意，这保证了当且仅当此键被找到时，返回的值将大于等于 0）。

注意

如果指定的范围大于或等于数组的长度，则会报出 ArrayIndexOutOfBoundsException 异常。

【例 6.19】 在项目中创建类 Rakel，在主方法中创建 String 数组，实现查找元素“cd”在指定范围的数组 str 中的索引位置。（**实例位置：光盘\TM\sl\6.11**）

```
import java.util.Arrays;                                    //导入 java.util.Arrays 类
public class Rakel {                                        //创建类
    public static void main(String[] args) {                //主方法
        //定义 String 型数组 str
        String str[] = new String[] { "ab", "cd", "ef", "yz" };
        Arrays.sort(str);                                   //将数组进行排序
        //在指定的范围内搜索元素“cd”的索引位置
        int index = Arrays.binarySearch(str, 0, 2, "cd");
        System.out.println("cd 的索引位置是：" + index);      //将索引输出
    }
}
```

运行结果如图 6.14 所示。

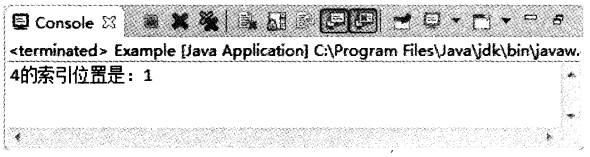

图 6.13　例 6.18 的运行结果

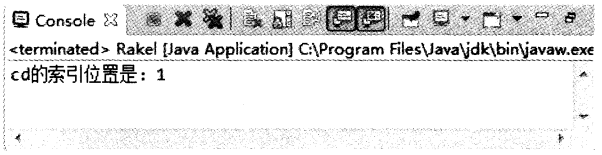

图 6.14　例 6.19 的运行结果

6.5　数组排序算法

数组有很多常用的算法，本节将介绍常用的排序算法，包括冒泡排序、直接选择排序和反转排序。

6.5.1　冒泡排序

视频讲解：光盘\TM\lx\6\冒泡排序.mp4

在程序设计中，经常需要将一组数列进行排序，这样更加方便统计与查询。程序常用的排序方法有冒泡排序、选择排序和快速排序等。本节将介绍冒泡排序方法，它以简洁的思想与实现方法而备受青睐，是广大学习者最先接触的一种排序算法。

冒泡排序是最常用的数组排序算法之一，它排序数组元素的过程总是将小数往前放、大数往后放，类似水中气泡往上升的动作，所以称做冒泡排序。

1. 基本思想

冒泡排序的基本思想是对比相邻的元素值，如果满足条件就交换元素值，把较小的元素移动到数组前面，把大的元素移动到数组后面（也就是交换两个元素的位置），这样较小的元素就像气泡一样从底部上升到顶部。

2. 算法示例

冒泡算法由双层循环实现，其中外层循环用于控制排序轮数，一般为要排序的数组长度减 1 次，因为最后一次循环只剩下一个数组元素，不需要对比，同时数组已经完成排序了。而内层循环主要用于对比数组中每个邻近元素的大小，以确定是否交换位置，对比和交换次数随排序轮数而减少。例如，一个拥有 6 个元素的数组，在排序过程中每一次循环的排序过程和结果如图 6.15 所示。

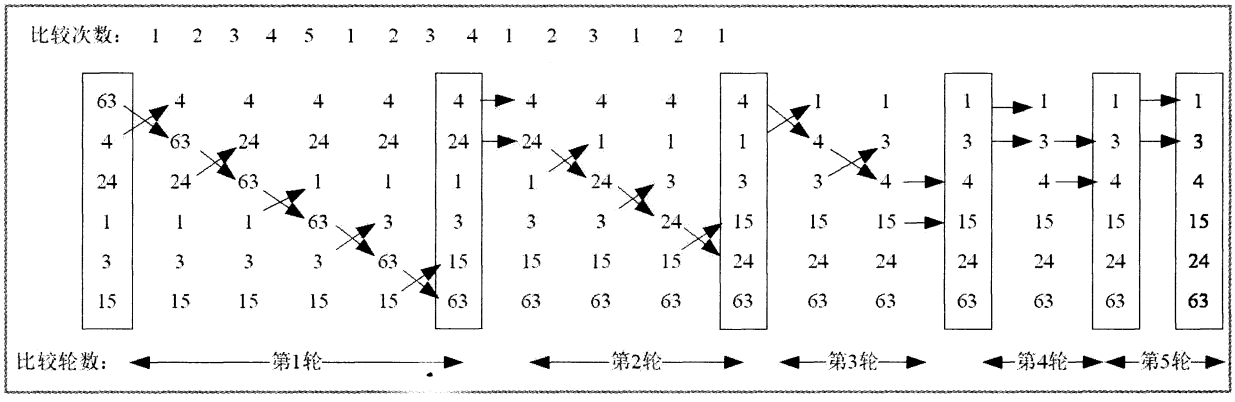

图 6.15　6 个元素数组的排序过程

第一轮外层循环时把最大的元素值 63 移动到了最后面（相应地，比 63 小的元素向前移动，类似气泡上升），第二轮外层循环不再对比最后一个元素值 63，因为它已经被确认为最大（不需要上升），应该放在最后，需要对比和移动的是其他剩余元素，这次将元素 24 移动到了 63 的前一个位置。其他循环将以此类推，继续完成排序任务。

3. 算法实现

下面来介绍一下冒泡排序的具体用法。

【例 6.20】 在项目中创建 BubbleSort 类，这个类的代码将实现冒泡排序的一个演示，其中排序使用的是正排序，读者可以根据本实例编写一个倒排序的例子。（**实例位置：光盘\TM\sl\6.12**）

```
public class BubbleSort {
    public static void main(String[] args) {
        //创建一个数组，这个数组元素是乱序的
        int[] array = { 63, 4, 24, 1, 3, 15 };
        //创建冒泡排序类的对象
        BubbleSort sorter = new BubbleSort();
```

```
        //调用排序方法将数组排序
        sorter.sort(array);
    }

    /**
     *冒泡排序
     *
     * @param array
     *              要排序的数组
     */
    public void sort(int[] array) {
        for (int i = 1; i < array.length; i++) {
            //比较相邻两个元素，较大的数往后冒泡
            for (int j = 0; j < array.length - i; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];   //把第一个元素值保存到临时变量中
                    array[j] = array[j + 1];//把第二个元素值保存到第一个元素单元中
                    array[j + 1] = temp;   //把临时变量（也就是第一个元素原值）保存到第二个元素中
                }
            }
        }
        showArray(array);                  //输出冒泡排序后的数组元素
    }

    /**
     * 显示数组中的所有元素
     *
     * @param array
     *              要显示的数组
     */
    public void showArray(int[] array) {
        for (int i : array) {              //遍历数组
            System.out.print(" >" + i);    //输出每个数组元素值
        }
        System.out.println();
    }
}
```

运行结果如图 6.16 所示。

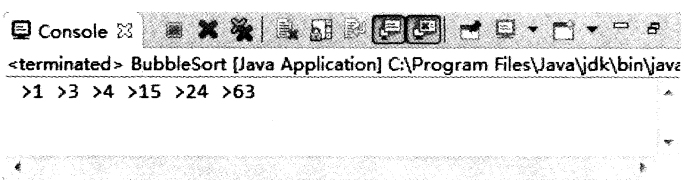

图 6.16　例 6.20 的运行结果

从实例的运行结果来看，数组中的元素已经按从小到大的顺序排列好了。冒泡排序的主要思想就是：把相邻两个元素进行比较，如满足一定条件则进行交换（如判断大小或日期前后等），每次循环都

将最大（或最小）的元素排在最后，下一次循环是对数组中其他的元素进行类似操作。

6.5.2 直接选择排序

视频讲解：光盘\TM\lx\6\直接选择排序.mp4

直接选择排序方法属于选择排序的一种，它的排序速度要比冒泡排序快一些，也是常用的排序算法，初学者应该掌握。

1. 基本思想

直接选择排序的基本思想是将指定排序位置与其他数组元素分别对比，如果满足条件就交换元素值，注意这里区别冒泡排序，不是交换相邻元素，而是把满足条件的元素与指定的排序位置交换（如从最后一个元素开始排序），这样排序好的位置逐渐扩大，最后整个数组都成为已排序好的格式。

这就好比有一个小学生，从包含数字 1~10 的乱序的数字堆中分别选择合适的数字，组成一个 1~10 的排序，而这个学生首先从数字堆中选出 1，放在第一位，然后选出 2（注意这时数字堆中已经没有 1 了），放在第二位，依此类推，直到其找到数字 9，放到 8 的后面，最后剩下 10，就不用选择了，直接放到最后就可以了。

与冒泡排序相比，直接选择排序的交换次数要少很多，所以速度会快些。

2. 算法示例

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序地放在已排好序的数列的最后，直到全部待排序的数据元素排完。

例如：

```
初始数组资源    【63   4    24   1    3    15】
第一趟排序后    【15   4    24   1    3 】63
第二趟排序后    【15   4    3    1】  24   63
第三趟排序后    【1    4    3】   15   24   63
第四趟排序后    【1    3】  4    15   24   63
第五趟排序后    【1】  3    4    15   24   63
```

3. 算法实现

下面来介绍一下直接选择排序的具体用法。

【例 6.21】 在项目中创建 SelectSort 类，这个类的代码将作为直接选择排序的一个演示，其中排序使用的是正排序，读者可以根据本实例编写一个倒排序的例子。（**实例位置：光盘\TM\sl\6.13**）

```
/**
 * 直接选择排序算法实例
 *
 * @author Li Zhong Wei
 */
public class SelectSort {
    public static void main(String[] args) {
```

```
        //创建一个数组，这个数组元素是乱序的
        int[] array = { 63, 4, 24, 1, 3, 15 };
        //创建直接排序类的对象
        SelectSort sorter = new SelectSort();
        //调用排序对象的方法将数组排序
        sorter.sort(array);
    }

    /**
     *直接选择排序
     *
     * @param array
     *              要排序的数组
     */
    public void sort(int[] array) {
        int index;
        for (int i = 1; i < array.length; i++) {
            index = 0;
            for (int j = 1; j <= array.length - i; j++) {
                if (array[j] > array[index]) {
                    index = j;
                }
            }
            //交换在位置 array.length-i 和 index(最大值)上的两个数
            int temp = array[array.length - i];       //把第一个元素值保存到临时变量中
            array[array.length - i] = array[index];   //把第二个元素值保存到第一个元素单元中
            array[index] = temp;                      //把临时变量也就是第一个元素原值保存到第二个元素中
        }
        showArray(array);                             //输出直接选择排序后的数组值
    }
    /**
     * 显示数组中的所有元素
     *
     * @param array
     *              要显示的数组
     */
    public void showArray(int[] array) {
        for (int i : array) {                         //遍历数组
            System.out.print(" >" + i);               //输出每个数组元素值
        }
        System.out.println();
    }
}
```

实例运行结果如图 6.17 所示。

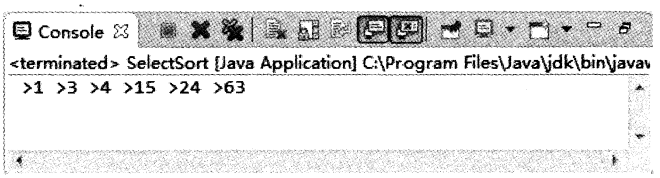

图 6.17　直接选择排序算法运行结果

6.5.3 反转排序

顾名思义，反转排序就是以相反的顺序把原有数组的内容重新排序。反转排序算法在程序开发中也经常用到。

1. 基本思想

反转排序的基本思想比较简单，也很好理解，其实现思路就是把数组最后一个元素与第一个元素替换，倒数第二个元素与第二个元素替换，依此类推，直到把所有数组元素反转替换。

2. 算法示例

反转排序是对数组两边的元素进行替换，所以只需要循环数组长度的半数次，如数组长度为 7，那么 for 循环只需要循环 3 次。

例如：

```
初始数组资源  【10   20   30   40   50   60】
第一趟排序后   60  【20   30   40   50】  10
第二趟排序后   60   50  【30   40】  20   10
第三趟排序后   60   50   40   30   20   10
```

3. 算法实现

下面来介绍一下反转排序的具体用法。

【例 6.22】 在项目中创建 ReverseSort 类，这个类的代码将作为反转排序的一个演示。（**实例位置：光盘\TM\sl\6.14**）

```
/**
 * 反转排序算法实例
 *
 * @author Li Zhong Wei
 */
public class ReverseSort {
    public static void main(String[] args) {
        //创建一个数组
        int[] array = { 10, 20, 30, 40, 50, 60 };
        //创建反转排序类的对象
        ReverseSort sorter = new ReverseSort();
        //调用排序对象的方法将数组反转
        sorter.sort(array);
    }

    /**
     *反转排序
     *
     * @param array
     *                要排序的数组
```

```
     */
    public void sort(int[] array) {
        System.out.println("数组原有内容：");
        showArray(array);            //输出排序前的数组值
        int temp;
        int len = array.length;
        for (int i = 0; i < len / 2; i++) {
            temp = array[i];
            array[i] = array[len - 1 - i];
            array[len - 1 - i] = temp;
        }
        System.out.println("数组反转后内容：");
        showArray(array);            //输出排序后的数组值
    }

    /**
     * 显示数组中的所有元素
     *
     * @param array
     *              要显示的数组
     */
    public void showArray(int[] array) {
        for (int i : array) {                    //遍历数组
            System.out.print("\t" + i);  //输出每个数组元素值
        }
        System.out.println();
    }
}
```

实例运行结果如图 6.18 所示。

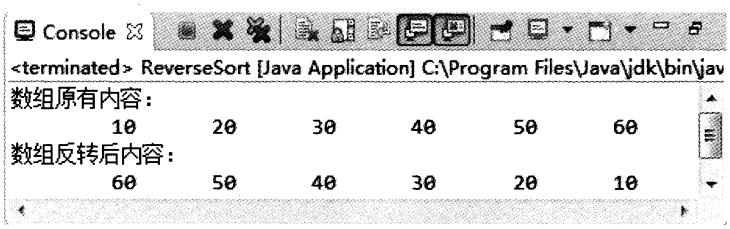

图 6.18　实例 6.22 的运行结果

6.6　小　　结

本章介绍的是数组的创建及使用方法。需要读者注意的是，数组的下标是从 0 开始的，最后一个元素的表示总是“数组名.length-1”。本章的重点是遍历数组以及使用 Arrays 类中的各种方法对数组进行操作，如填充替换数组、复制数组等。此外，Arrays 类还提供了其他操作数组的方法，有兴趣的读者可以查阅相关资料。

6.7 实践与练习

1．编写 Java 程序，创建数组 arr1 和 arr2，将数组 arr1 中索引位置是 0~3 中的元素复制到数组 arr2 中，最后将数组 arr1 和 arr2 中的元素输出。（**答案位置：光盘\TM\sl\6.15**）

2．编写 Java 程序，将数组中最小的数输出。（**答案位置：光盘\TM\sl\6.16**）

3．编写 Java 程序，实现将数组 arr 中索引位置是 2 的元素替换为“bb”，并将替换前数组中的元素和替换后数组中的元素全部输出。（**答案位置：光盘\TM\sl\6.17**）

4．编写 Java 程序，将二维数组中的行列互调显示出来。（**答案位置：光盘\TM\sl\6.18**）

例如：

1 2 3
4 5 6
7 8 9

显示出的结果为：

1 4 7
2 5 8
3 6 9

第 7 章

类和对象

（视频讲解：1 小时 27 分钟）

在 Java 语言中经常被提到的两个词是类与对象，实质上可以将类看作是对象的载体，它定义了对象所具有的功能。学习 Java 语言必须要掌握类与对象，这样可以从深层次去理解 Java 这种面向对象语言的开发理念，从而更好、更快地掌握 Java 编程思想与编程方式，因此，掌握类与对象是学习 Java 语言的基础。本章将详细介绍类的各种方法以及对象，为了使初学者更容易入门，在讲解过程中列举了大量实例。

通过阅读本章，您可以：

- 了解面向对象编程思想
- 掌握如何定义类
- 掌握类的成员变量、成员方法
- 掌握修饰权限
- 掌握局部变量以及作用范围
- 掌握 this、static 关键字
- 掌握构造方法以及通过构造方法创建对象
- 掌握类中的主方法以及如何运行带参数的 Java 程序
- 掌握使用对象获取对象的属性和行为
- 掌握对象的创建、比较和销毁

7.1 面向对象概述

视频讲解：光盘\TM\lx\7\面向对象概述.mp4

在程序开发初期人们使用结构化开发语言，但随着软件的规模越来越庞大，结构化语言的弊端也逐渐暴露出来，开发周期被延长，产品的质量也不尽如人意，结构化语言已经不再适合当前的软件开发。这时人们开始将另一种开发思想引入程序中，即面向对象的开发思想。面向对象思想是人类最自然的一种思考方式，它将所有预处理的问题抽象为对象，同时了解这些对象具有哪些相应的属性以及展示这些对象的行为，以解决这些对象面临的一些实际问题，这样就在程序开发中引入了面向对象设计的概念，面向对象设计实质上就是对现实世界的对象进行建模操作。

7.1.1 对象

视频讲解：光盘\TM\lx\7\对象.mp4

现实世界中，随处可见的一种事物就是对象。对象是事物存在的实体，如人类、书桌、计算机、高楼大厦等。人类解决问题的方式总是将复杂的事物简单化，于是就会思考这些对象都是由哪些部分组成的。通常都会将对象划分为两个部分，即静态部分与动态部分。静态部分，顾名思义，就是不能动的部分，这个部分被称为“属性”，任何对象都会具备其自身属性，如一个人，其属性包括高矮、胖瘦、性别、年龄等。然而具有这些属性的人会执行哪些动作也是一个值得探讨的部分，这个人可以哭泣、微笑、说话、行走，这些是这个人具备的行为（动态部分），人类通过探讨对象的属性和观察对象的行为了解对象。

在计算机的世界中，面向对象程序设计的思想要以对象来思考问题，首先要将现实世界的实体抽象为对象，然后考虑这个对象具备的属性和行为。例如，现在面临一只大雁要从北方飞往南方这样一个实际问题，试着以面向对象的思想来解决这一实际问题。步骤如下：

（1）首先可以从这一问题中抽象出对象，这里抽象出的对象为大雁。

（2）然后识别这个对象的属性。对象具备的属性都是静态属性，如大雁有一对翅膀、黑色的羽毛等。这些属性如图 7.1 所示。

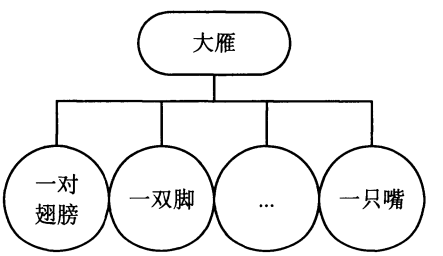

图 7.1 识别对象的属性

（3）接着识别这个对象的动态行为，即这只大雁可以进行的动作，如飞行、觅食等，这些行为都是这个对象基于其属性而具有的动作。这些行为如图 7.2 所示。

（4）识别出这个对象的属性和行为后，这个对象就被定义完成了，然后可以根据这只大雁具有的特性制定这只大雁要从北方飞向南方的具体方案以解决问题。

究其本质，所有的大雁都具有以上的属性和行为，可以将这些属性和行为封装起来以描述大雁这类动物。由此可见，类实质上就是封装对象属性和行为的载体，而对象则是类抽象出来的一个实例，

两者之间的关系如图 7.3 所示。

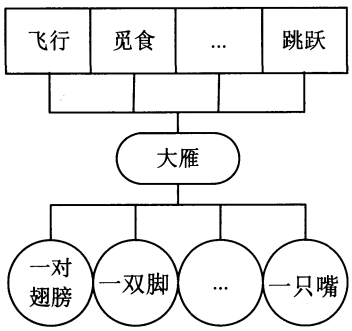

图 7.2　识别对象具有的行为

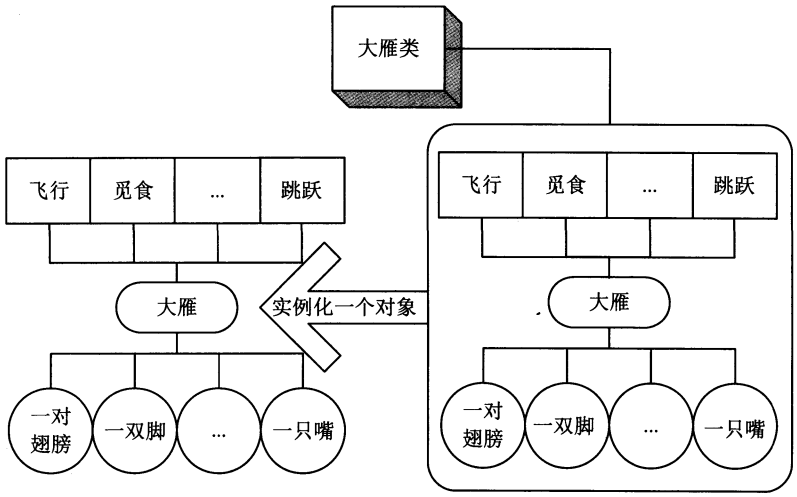

图 7.3　描述对象与类之间的关系

7.1.2　类

视频讲解：光盘\TM\lx\7\类.mp4

不能将所谓的一个事物描述成一类事物，如一只鸟不能称为鸟类。如果需要对同一类事物统称，就不得不说明类这个概念。

类就是同一类事物的统称，如果将现实世界中的一个事物抽象成对象，类就是这类对象的统称，如鸟类、家禽类、人类等。类是构造对象时所依赖的规范，如一只鸟有一对翅膀，它可以用这对翅膀飞行，而基本上所有的鸟都具有有翅膀这个特性和飞行的技能，这样具有相同特性和行为的一类事物就称为类，类的思想就是这样产生的。在图 7.3 中已经描述过类与对象之间的关系，对象就是符合某个类的定义所产生出来的实例。更为恰当的描述是，类是世间事物的抽象称呼，而对象则是这个事物相对应的实体。如果面临实际问题，通常需要实例化类对象来解决。例如，解决大雁南飞的问题，这里只能拿这只大雁来处理这个问题，而不能拿大雁类或鸟类来解决。

类是封装对象的属性和行为的载体，反过来说具有相同属性和行为的一类实体被称为类。例如，鸟类封装了所有鸟的共同属性和应具有的行为，其结构如图 7.4 所示。

定义完鸟类之后，可以根据这个类抽象出一个实体对象，最后通过实体对象来解决相关的实际问题。

在 Java 语言中，类中对象的行为是以方法的形式定义的，对象的属性是以成员变量的形式定义的，而类包括对象的属性和方法，有关类的具体实现会在后续章节中进行介绍。

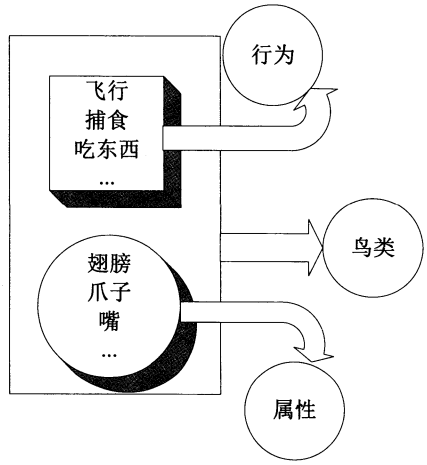

图 7.4　鸟类结构

7.1.3 封装

面向对象程序设计具有以下特点：

☑ 封装性。

☑ 继承性。

☑ 多态性。

封装是面向对象编程的核心思想。将对象的属性和行为封装起来，其载体就是类，类通常对客户隐藏其实现细节，这就是封装的思想。例如，用户使用计算机时，只需要使用手指敲击键盘就可以实现一些功能，无须知道计算机内部是如何工作的，即使可能知道计算机的工作原理，但在使用计算机时也并不完全依赖于计算机工作原理这些细节。

采用封装的思想保证了类内部数据结构的完整性，应用该类的用户不能轻易地直接操作此数据结构，只能执行类允许公开的数据。这样就避免了外部操作对内部数据的影响，提高了程序的可维护性。

使用类实现封装特性如图 7.5 所示。

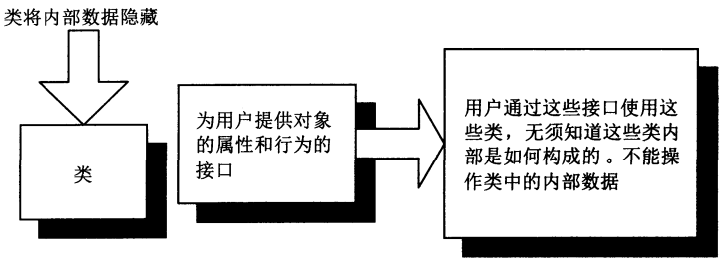

图 7.5 封装特性示意图

7.1.4 继承

类与类之间同样具有关系，如一个百货公司类与销售员类相联系，类之间的这种关系被称为关联。关联主要描述两个类之间的一般二元关系，例如，一个百货公司类与销售员类就是一个关联，学生类与教师类也是一个关联。两个类之间的关系有很多种，继承是关联中的一种。

当处理一个问题时，可以将一些有用的类保留下来，在遇到同样问题时拿来复用。假如这时需要解决信鸽送信的问题，我们很自然就会想到图 7.4 所示的鸟类。由于鸽子属于鸟类，具有与鸟类相同的属性和行为，便可以在创建信鸽类时将鸟类拿来复用，并且保留鸟类具有的属性和行为。不过，并不是所有的鸟都有送信的习惯，因此还需要再添加一些信鸽具有的独特属性及行为。鸽子类保留了鸟类的属性和行为，这样就节省了定义鸟和鸽子共同具有的属性和行为的时间，这就是继承的基本思想。可见设计软件的代码时使用继承思想可以缩短软件开发的时间，复用那些已经定义好的类可以提高系统性能，减少系统在使用过程中出现错误的几率。

继承性主要利用特定对象之间的共有属性。例如，平行四边形是四边形，正方形、矩形也都是四边形，平行四边形与四边形具有共同特性，就是拥有 4 个边，可以将平行四边形类看作四边形的延伸，

平行四边形复用了四边形的属性和行为，同时添加了平行四边形独有的属性和行为，如平行四边形的对边平行且相等。这里可以将平行四边形类看作是从四边形类中继承的。在 Java 语言中将类似于平行四边形的类称为子类，将类似于四边形的类称为父类或超类。值得注意的是，可以说平行四边形是特殊的四边形，但不能说四边形是平行四边形，也就是说子类的实例都是父类的实例，但不能说父类的实例是子类的实例。图 7.6 阐明了图形类之间的继承关系。

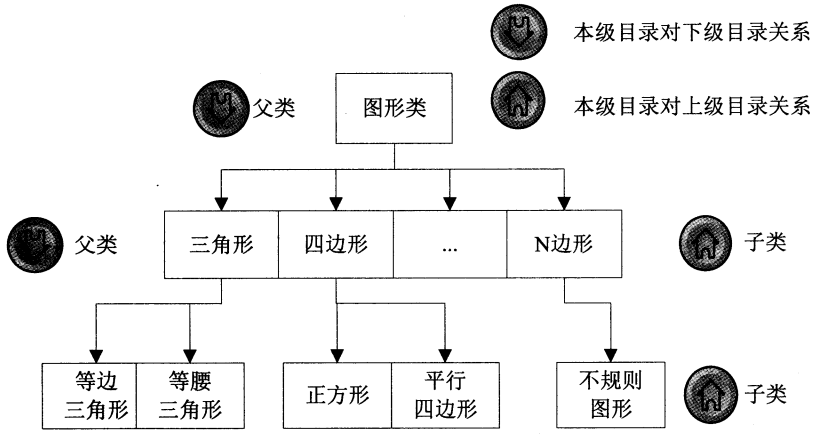

图 7.6　图形类层次结构示意图

从图 7.6 中可以看出，继承关系可以使用树形关系来表示，父类与子类存在一种层次关系。一个类处于继承体系中，它既可以是其他类的父类，为其他类提供属性和行为，也可以是其他类的子类，继承父类的属性和方法，如三角形既是图形类的子类也是等边三角形的父类。

7.1.5　多态

在 7.1.4 节中介绍了继承，了解了父类和子类，其实将父类对象应用于子类的特征就是多态。依然以图形类来说明多态，每个图形都拥有绘制自己的能力，这个能力可以看作是该类具有的行为，如果将子类的对象统一看作是父类的实例对象，这样当绘制图形时，简单地调用父类也就是图形类绘制图形的方法即可绘制任何图形，这就是多态最基本的思想。

多态性允许以统一的风格编写程序，以处理种类繁多的已存在的类及相关类。该统一风格可以由父类来实现，根据父类统一风格的处理，可以实例化子类的对象。由于整个事件的处理都只依赖于父类的方法，所以日后只要维护和调整父类的方法即可，这样就降低了维护的难度，节省了时间。

提到多态，就不得不提抽象类和接口，因为多态的实现并不依赖具体类，而是依赖于抽象类和接口。

再回到绘制图形的实例上来。图形类作为所有图形的父类，具有绘制图形的能力，这个方法可以称为“绘制图形”，但如果要执行这个“绘制图形”的命令，没有人知道应该画什么样的图形，并且如果要在图形类中抽象出一个图形对象，没有人能说清这个图形究竟是什么图形，所以使用“抽象”这个词来描述图形类比较恰当。在 Java 语言中称这样的类为抽象类，抽象类不能实例化对象。在多态的机制中，父类通常会被定义为抽象类，在抽象类中给出一个方法的标准，而不给出实现的具体流程。实质上这个方法也是抽象的，如图形类中的“绘制图形”方法只提供一个可以绘制图形的标准，并没

有提供具体绘制图形的流程，因为没有人知道究竟需要绘制什么形状的图形。

在多态的机制中，比抽象类更方便的方式是将抽象类定义为接口。由抽象方法组成的集合就是接口。接口的概念在现实中也极为常见，如从不同的五金商店买来螺丝帽和螺丝钉，螺丝帽很轻松地就可以拧在螺丝钉上，可能螺丝帽和螺丝钉的厂家不同，但这两个物品可以轻易地组合在一起，这是因为生产螺丝帽和螺丝钉的厂家都遵循着一个标准，这个标准在 Java 中就是接口。依然拿“绘制图形”来说明，可以将“绘制图形”作为一个接口的抽象方法，然后使图形类实现这个接口，同时实现“绘制图形”这个抽象方法，当三角形类需要绘制时，就可以继承图形类，重写其中的“绘制图形”方法，并改写这个方法为“绘制三角形”，这样就可以通过这个标准绘制不同的图形。

7.2 类

在 7.1.2 节中已经讲解过类是封装对象的属性和行为的载体，而在 Java 语言中对象的属性以成员变量的形式存在，对象的方法以成员方法的形式存在。本节将介绍在 Java 语言中类是如何定义的。

7.2.1 成员变量

在 Java 中对象的属性也称为成员变量。为了了解成员变量，首先定义一个图书类，成员变量对应于类对象的属性，在 Book 类中设置 3 个成员变量，分别为 id、name 和 category，分别对应于图书编号、图书名称和图书类别 3 个图书属性。

【例 7.1】 在项目中创建 Book 类，在该类中定义并使用成员变量。

```
public class Book {
    private String name;                        //定义一个 String 型的成员变量

    public String getName() {                   //定义一个 getName()方法
        int id = 0;                             //局部变量
        setName("Java");                        //调用类中其他方法
        return id + this.name;                  //设置方法返回值
    }

    private void setName(String name) {         //定义一个 setName()方法
        this.name = name;                       //将参数值赋予类中的成员变量
    }

    public Book getBook() {
        return this;                            //返回 Book 类引用
    }
}
```

根据以上代码，读者可以看到在 Java 中使用 class 关键字来定义类，Book 是类的名称。同时在 Book 类中定义了 3 个成员变量，成员变量的类型可以设置为 Java 中合法的数据类型，其实成员变量就是普

通的变量，可以为它设置初始值，也可以不设置初始值。如果不设置初始值，则会有默认值。读者应该注意到在 3 个成员变量前面的 private 关键字，它用来定义一个私有成员（关于权限修饰符的说明将在 7.2.3 节中进行介绍）。

7.2.2　成员方法

在 Java 语言中使用成员方法对应于类对象的行为。以 Book 类为例，它包含 getName()和 setName()两个方法，这两个成员方法分别为获取图书名称和设置图书名称的方法。

定义成员方法的语法格式如下：

```
权限修饰符 返回值类型 方法名(参数类型 参数名){
    …//方法体
    return 返回值;
}
```

一个成员方法可以有参数，这个参数可以是对象，也可以是基本数据类型的变量，同时成员方法有返回值和不返回任何值的选择，如果方法需要返回值，可以在方法体中使用 return 关键字，使用这个关键字后，方法的执行将被终止。

注意

Java 中的成员方法无返回值，可以使用 void 关键字表示。

成员方法的返回值可以是计算结果，也可以是其他想要的数值和对象，返回值类型要与方法返回的值类型一致。

在成员方法中可以调用其他成员方法和类成员变量，如在例7.1中的getName()方法中就调用了setName()方法将图书名称赋予一个值。同时在成员方法中可以定义一个变量，这个变量为局部变量（局部变量的内容将在7.2.4节中进行介绍）。

说明

如果一个方法中含有与成员变量同名的局部变量，则方法中对这个变量的访问以局部变量进行。例如，变量 id 在 getName()方法中值为 0，而不是成员变量中 id 的值。

类成员变量和成员方法也可以统称为类成员。

7.2.3　权限修饰符

Java 中的权限修饰符主要包括 private、public 和 protected，这些修饰符控制着对类和类的成员变量以及成员方法的访问。如果一个类的成员变量或成员方法被修饰为 private，则该成员变量只能在本类中被使用，在子类中是不可见的，并且对其他包的类也是不可见的。如果将类的成员变量和成员方法

的访问权限设置为 public，那么除了可以在本类使用这些数据之外，还可以在子类和其他包的类中使用。如果一个类的访问权限被设置为 private，这个类将隐藏其内的所有数据，以免用户直接访问它。如果需要使类中的数据被子类或其他包中的类使用，可以将这个类设置为 public 访问权限。如果一个类使用 protected 修饰符，那么只有本包内的该类的子类或其他类可以访问此类中的成员变量和成员方法。

这么看来，public 和 protected 修饰的类可以由子类访问，如果子类和父类不在同一包中，那么只有修饰符为 public 的类可以被子类进行访问。如果父类不允许通过继承产生的子类访问它的成员变量，那么必须使用 private 声明父类的这个成员变量。表 7.1 中描述了 private、protected 和 public 修饰符的修饰权限。

表 7.1　Java 语言中的修饰符权限

访问包位置	类 修 饰 符		
	private	protected	public
本类	可见	可见	可见
同包其他类或子类	不可见	可见	可见
其他包的类或子类	不可见	不可见	可见

注意

当声明类时不使用 public、protected 和 private 修饰符设置类的权限，则这个类预设为包存取范围，即只有一个包中的类可以调用这个类的成员变量或成员方法。

【例 7.2】 在项目中的 com.lzw 包下创建 AnyClass 类，该类使用默认的访问权限。

```
package com.lzw;
class AnyClass {
    public void doString(){
        …//方法体
    }
}
```

在上述代码中，由于类的修饰符为默认修饰符，即只有一个包内的其他类和子类可以对该类进行访问，而 AnyClass 类中的 doString()方法却又被设置为 public 访问权限，即使这样，doString()方法的访问权限依然与 AnyClass 类的访问权限相同，因为 Java 语言规定，类的权限设定会约束类成员的权限设定，所以上述代码等同于例 7.3 的代码。

【例 7.3】 本实例等同于例 7.2 的代码。

```
package com.lzw;
class AnyClass {
    void doString(){
        …//方法体
    }
}
```

7.2.4　局部变量

在 7.2.2 节中已经讲述过成员方法，如果在成员方法内定义一个变量，那么这个变量被称为局部变量。

例如，在例 7.1 定义的 Book 类中，getName()方法的 id 变量即为局部变量。实际上方法中的形参也可作为一个局部变量，如在定义 setName(String name)方法时，String name 这个形参就被看作是局部变量。

局部变量是在方法被执行时创建，在方法执行结束时被销毁。局部变量在使用时必须进行赋值操作或被初始化，否则会出现编译错误。

【例 7.4】 在项目中创建一个类文件，在该类中定义 getName()方法并进行调用。

```
public String getName(){          //定义一个 getName()方法
    int id=0;                     //局部变量
    setName("Java");              //调用类中其他方法
    return id+this.name;          //设置方法返回值
}
```

如果将 id 这个局部变量的初始值去掉，编译器将出现错误。

7.2.5　局部变量的有效范围

可以将局部变量的有效范围称为变量的作用域，局部变量的有效范围从该变量的声明开始到该变量的结束为止。图 7.7 描述了局部变量的作用范围。

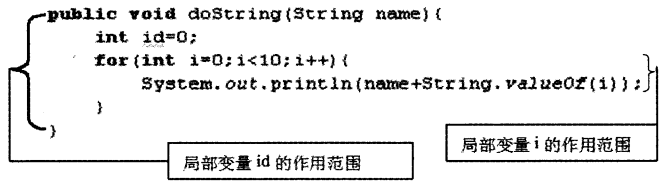

图 7.7　局部变量的作用范围

在相互不嵌套的作用域中可以同时声明两个名称和类型相同的局部变量，如图 7.8 所示。

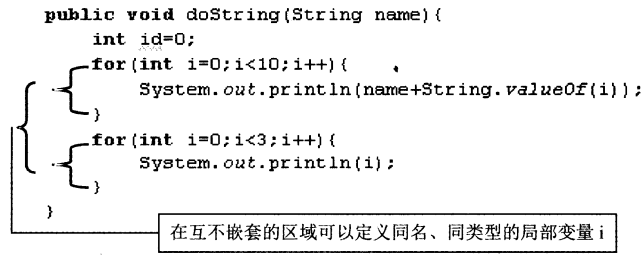

图 7.8　在不同嵌套区域可以定义相同名称和类型的局部变量

但是在相互嵌套的区域中不可以这样声明，如果将局部变量 id 在方法体的 for 循环中再次定义，编译器将会报错，如图 7.9 所示。

```
public void doString(String name){
    int id=0;
    for(int i=0;i<10;i++){
        System.out.println(name+String.valueOf(i));
    }
    for(int i=0;i<3;i++){
        System.out.println(i);
        int id=7;
    }
}
```

在嵌套区域中重复定义局部变量 id

图 7.9　在嵌套区域中不可以定义相同名称和类型的局部变量

注意

在作用范围外使用局部变量是一个常见的错误，因为在作用范围外没有声明局部变量的代码。

7.2.6　this 关键字

【例 7.5】 在项目中创建一个类文件，该类中定义了 setName()，并将方法的参数值赋予类中的成员变量。

```
private void setName(String name){      //定义一个 setName()方法
    this.name=name;                     //将参数值赋予类中的成员变量
}
```

在上述代码中可以看到，成员变量与 setName()方法中的形式参数的名称相同，都为 name，那么该如何在类中区分使用的是哪一个变量呢？在 Java 语言中规定使用 this 关键字来代表本类对象的引用，this 关键字被隐式地用于引用对象的成员变量和方法，如在上述代码中，this.name 指的就是 Book 类中的 name 成员变量，而 this.name=name 语句中的第二个 name 则指的是形参 name。实质上 setName()方法实现的功能就是将形参 name 的值赋予成员变量 name。

在这里读者明白了 this 可以调用成员变量和成员方法，但 Java 语言中最常规的调用方式是使用“对象.成员变量”或“对象.成员方法”进行调用（关于使用对象调用成员变量和方法的问题，将在后续章节中进行讲述）。

既然 this 关键字和对象都可以调用成员变量和成员方法，那么 this 关键字与对象之间具有怎样的关系呢？

事实上，this 引用的就是本类的一个对象。在局部变量或方法参数覆盖了成员变量时，如上面代码的情况，就要添加 this 关键字明确引用的是类成员还是局部变量或方法参数。

如果省略 this 关键字直接写成 name = name，那只是把参数 name 赋值给参数变量本身而已，成员变量 name 的值没有改变，因为参数 name 在方法的作用域中覆盖了成员变量 name。

其实，this 除了可以调用成员变量或成员方法之外，还可以作为方法的返回值。

【例 7.6】 在项目中创建一个类文件，在该类中定义 Book 类型的方法，并通过 this 关键字进行返回。

```
public Book getBook(){
    return this;          //返回 Book 类引用
}
```

在 getBook()方法中，方法的返回值为 Book 类，所以方法体中使用 return this 这种形式将 Book 类的对象进行返回。

7.3　类的构造方法

视频讲解：光盘\TM\lx\7\类的构造方法.mp4

在类中除了成员方法之外，还存在一种特殊类型的方法，那就是构造方法。构造方法是一个与类同名的方法，对象的创建就是通过构造方法完成的。每当类实例化一个对象时，类都会自动调用构造方法。

构造方法的特点如下：

☑ 构造方法没有返回值。

☑ 构造方法的名称要与本类的名称相同。

注意

在定义构造方法时，构造方法没有返回值，但这与普通没有返回值的方法不同，普通没有返回值的方法使用 public void methodEx()这种形式进行定义，但构造方法并不需要使用 void 关键字进行修饰。

构造方法的定义语法格式如下：

```
public book(){
    …//构造方法体
}
```

☑ public：构造方法修饰符。

☑ book：构造方法的名称。

在构造方法中可以为成员变量赋值，这样当实例化一个本类的对象时，相应的成员变量也将被初始化。

如果类中没有明确定义构造方法，编译器会自动创建一个不带参数的默认构造方法。

注意

如果在类中定义的构造方法都不是无参的构造方法，那么编译器也不会为类设置一个默认的无参构造方法，当试图调用无参构造方法实例化一个对象时，编译器会报错。所以只有在类中没有定义任何构造方法时，编译器才会在该类中自动创建一个不带参数的构造方法。

在 7.2.6 节中介绍过 this 关键字，了解了 this 可以调用类的成员变量和成员方法，事实上 this 还可以调用类中的构造方法。看下面的实例。

【例 7.7】 在项目中创建 AnyThting 类，在该类中使用 this 调用构造方法。

```
public class AnyThting {
    public AnyThting() {                          //定义无参构造方法
        this("this 调用有参构造方法");              //使用 this 调用有参构造方法
        System.out.println("无参构造方法");
    }

    public AnyThting(String name) {               //定义有参构造方法
        System.out.println("有参构造方法");
    }
}
```

在例 7.7 中可以看到定义了两个构造方法，在无参构造方法中可以使用 this 关键字调用有参的构造方法。但使用这种方式需要注意的是只可以在无参构造方法中的第一句使用 this 调用有参构造方法。

7.4 静态变量、常量和方法

视频讲解：光盘\TM\lx\7\静态变量、常量和方法（static 关键字）.mp4

在介绍静态变量、常量和方法之前首先需要介绍 static 关键字，因为由 static 修饰的变量、常量和方法被称做静态变量、常量和方法。

有时，在处理问题时会需要两个类在同一个内存区域共享一个数据。例如，在球类中使用 PI 这个常量，可能除了本类需要这个常量之外，在另外一个圆类中也需要使用这个常量。这时没有必要在两个类中同时创建 PI 常量，因为这样系统会将这两个不在同一个类中定义的常量分配到不同的内存空间中。为了解决这个问题，可以将这个常量设置为静态的。PI 常量在内存中被共享的布局如图 7.10 所示。

被声明为 static 的变量、常量和方法被称为静态成员。静态成员属于类所有，区别于个别对象，可以在本类或其他类使用类名和“.”运算符调用静态成员。

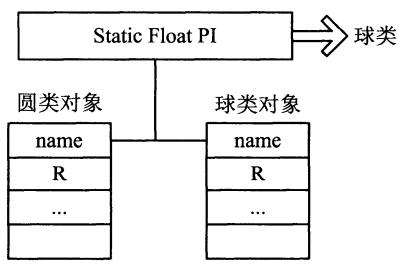

图 7.10 PI 常量在内存中被共享情况

语法如下：

```
类名.静态类成员
```

【例 7.8】 在项目中创建 StaticTest 类，该类中的主方法调用静态成员并在控制台中输出。

```
public class StaticTest {
    final static double PI = 3.1415;          //在类中定义静态常量
    static int id;                            //在类中定义静态变量

    public static void method1() {            //在类中定义静态方法
        //do Something
    }

    public void method2() {
        System.out.println(StaticTest.PI);    //调用静态常量
        System.out.println(StaticTest.id);    //调用静态变量
        StaticTest.method1();                 //调用静态方法
    }
}
```

在例 7.8 中设置了 3 个静态成员，分别为常量、变量和方法，然后在 method2()方法中分别调用这 3 个静态成员，直接使用“类名.静态成员”形式进行调用即可。

注意

虽然静态成员也可以使用“对象.静态成员”的形式进行调用，但通常不建议用这样的形式，因为这样容易混淆静态成员和非静态成员。

静态数据与静态方法的作用通常是为了提供共享数据或方法，如数学计算公式等，以 static 声明并实现，这样当需要使用时，直接使用类名调用这些静态成员即可。尽管使用这种方式调用静态成员比较方便，但静态成员同样遵循着 public、private 和 protected 修饰符的约束。

【例 7.9】 在项目中创建 StaticTest 类，该类中的主方法调用静态成员并在控制台中输出。

```
public class StaticTest {
    static double PI = 3.1415;                //在类中定义静态常量
    static int id;                            //在类中定义静态变量
    public static void method1() {            //在类中定义静态方法
        //do Something
    }
    public void method2() {                   //在类中定义一个非静态方法
        System.out.println(StaticTest.PI);    //调用静态常量
        System.out.println(StaticTest.id);    //调用静态变量
        StaticTest.method1();                 //调用静态方法
    }
    public static StaticTest method3() {      //在类中定义一个静态方法
        method2();                            //调用非静态方法
        return this;                          //在 return 语句中使用 this 关键字
```

```
    }
}
```

读者也许会发现在 Eclipse 中输入上述代码后，编译器会发生错误，这是因为 method3()方法为一个静态方法，而在其方法体中调用了非静态方法和 this 关键字。在 Java 语言中对静态方法有两点规定：

☑ 在静态方法中不可以使用 this 关键字。

☑ 在静态方法中不可以直接调用非静态方法。

注意

在 Java 中规定不能将方法体内的局部变量声明为 static 的。例如下述代码就是错误的：

```
public class example {
    public void method() {
        static int i = 0;
    }
}
```

技巧

如果在执行类时，希望先执行类的初始化动作，可以使用 static 定义一个静态区域。例如：

```
public class example {
    static {
        // some
    }
}
```

当这段代码被执行时，首先执行 static 块中的程序，并且只会执行一次。

7.5 类的主方法

视频讲解：光盘\TM\lx\7\类的主方法.mp4

主方法是类的入口点，它定义了程序从何处开始；主方法提供对程序流向的控制，Java 编译器通过主方法来执行程序。主方法的语法如下：

```
public static void main(String[] args){
    //方法体
}
```

在主方法的定义中可以看到其具有以下特性：

☑ 主方法是静态的，所以如要直接在主方法中调用其他方法，则该方法必须也是静态的。

☑ 主方法没有返回值。

☑ 主方法的形参为数组。其中 args[0]~args[n]分别代表程序的第一个参数到第 n 个参数，可以使用 args.length 获取参数的个数。

【例 7.10】 在项目中创建 TestMain 类，在主方法中编写以下代码，并在 Eclipse 中设置程序参数。（实例位置：光盘\TM\sl\7.01）

```
public class TestMain {
    public static void main(String[] args) {          //定义主方法
        for (int i = 0; i < args.length; i++) {       //根据参数个数做循环操作
            System.out.println(args[i]);              //循环打印参数内容
        }
    }
}
```

在 Eclipse 中运行本例，结果如图 7.11 所示。

在 Eclipse 中设置程序参数的步骤如下：

（1）在 Eclipse 中，在包资源管理器的项目名称节点上右击，在弹出的快捷菜单中选择 Run As / Run Configrations 命令，弹出 Run Configrations 对话框。

（2）在 Run Configurations 对话框中选择 Arguments 选项卡，在 Program arguments 文本框中输入相应的参数，每个参数间按 Enter 键隔开。具体设置如图 7.12 所示。

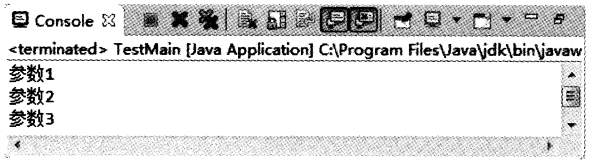

图 7.11　带参数程序的运行结果

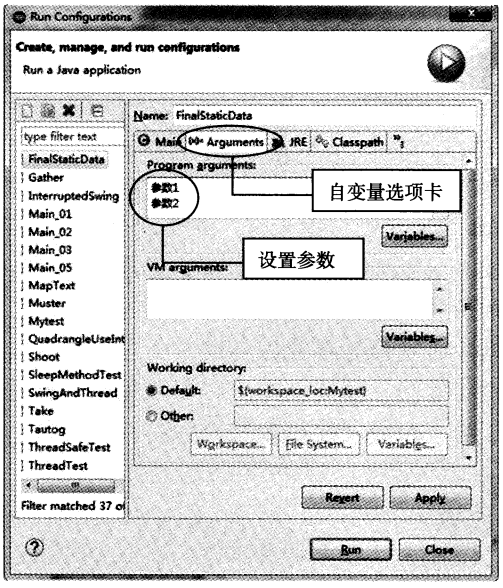

图 7.12　Eclipse 中的 Run Configurations 对话框

7.6　对　　象

Java 是一门面向对象的程序设计语言，对象是由类抽象出来的，所有的问题都通过对象来处理，对象可以操作类的属性和方法解决相应的问题，所以了解对象的产生、操作和消亡是十分必要的。本

节就来讲解对象在 Java 语言中的应用。

7.6.1 对象的创建

在 7.1 节中曾经介绍过对象，对象可以认为是在一类事物中抽象出某一个特例，可以通过这个特例来处理这类事物出现的问题。在 Java 语言中通过 new 操作符来创建对象。前文在讲解构造方法时介绍过每实例化一个对象就会自动调用一次构造方法，实质上这个过程就是创建对象的过程。准确地说，可以在 Java 语言中使用 new 操作符调用构造方法创建对象。

语法如下：

```
Test test=new Test();
Test test=new Test("a");
```

其参数说明如表 7.2 所示。

表 7.2 创建对象语法中的参数说明

设 置 值	描 述
Test	类名
test	创建 Test 类对象
new	创建对象操作符
"a"	构造方法的参数

test 对象被创建出来时，就是一个对象的引用，这个引用在内存中为对象分配了存储空间，7.3 节中介绍过，可以在构造方法中初始化成员变量，当创建对象时，自动调用构造方法。也就是说，在 Java 语言中初始化与创建是被捆绑在一起的。

每个对象都是相互独立的，在内存中占据独立的内存地址，并且每个对象都具有自己的生命周期，当一个对象的生命周期结束时，对象就变成垃圾，由 Java 虚拟机自带的垃圾回收机制处理，不能再被使用（对于垃圾回收机制的知识将在 7.6.5 小节中进行介绍）。

注意

在 Java 语言中对象和实例事实上可以通用。

下面来看一个创建对象的实例。

【例 7.11】 在项目中创建 CreateObject 类，在该类中创建对象并在主方法中创建对象。（**实例位置：光盘\TM\sl\7.02**）

```
public class CreateObject {
    public CreateObject() {                    //构造方法
        System.out.println("创建对象");
    }
    public static void main(String args[]) {   //主方法
        new CreateObject();                    //创建对象
```

```
    }
}
```

在 Eclipse 中运行上述代码，结果如图 7.13 所示。

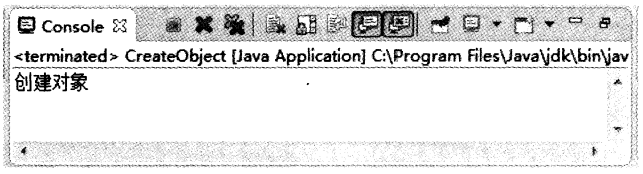

图 7.13　创建对象运行结果

在上述实例的主方法中使用 new 操作符创建对象，创建对象的同时，将自动调用构造方法中的代码。

7.6.2　访问对象的属性和行为

用户使用 new 操作符创建一个对象后，可以使用“对象.类成员”来获取对象的属性和行为。前文已经提到过，对象的属性和行为在类中是通过类成员变量和成员方法的形式来表示的，所以当对象获取类成员时，也相应地获取了对象的属性和行为。

【例 7.12】 在项目中创建 TransferProperty 类，在该类中说明对象是如何调用类成员的。（**实例位置：光盘\TM\sl\7.03**）

```
public class TransferProperty {
    int i = 47;                                          //定义成员变量
    public void call() {                                 //定义成员方法
        System.out.println("调用 call()方法");
        for (i = 0; i < 3; i++) {
            System.out.print(i + " ");
            if (i == 2) {
                System.out.println("\n");
            }
        }
    }
    public TransferProperty() {                          //定义构造方法
    }
    public static void main(String[] args) {
        TransferProperty t1 = new TransferProperty();    //创建一个对象
        TransferProperty t2 = new TransferProperty();    //创建另一个对象
        t2.i = 60;                                       //将类成员变量赋值为 60
        //使用第一个对象调用类成员变量
        System.out.println("第一个实例对象调用变量 i 的结果：" + t1.i++);
        t1.call();                                       //使用第一个对象调用类成员方法
        //使用第二个对象调用类成员变量
        System.out.println("第二个实例对象调用变量 i 的结果：" + t2.i);
        t2.call();                                       //使用第二个对象调用类成员方法
    }
}
```

在 Eclipse 中运行上述代码，结果如图 7.14 所示。

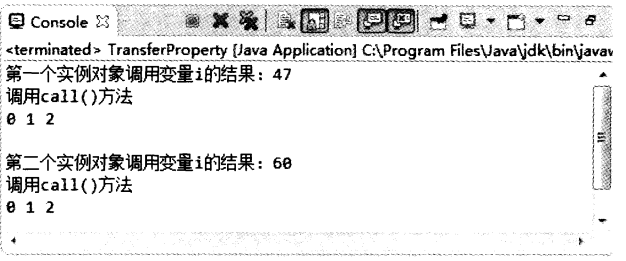

图 7.14　使用对象调用类成员运行结果

在上述代码的主方法中首先实例化一个对象，然后使用“.”操作符调用类的成员变量和成员方法。但是在运行结果中可以看到，虽然使用两个对象调用同一个成员变量，结果却不相同，因为在打印这个成员变量的值之前将该值重新赋值为 60，但在赋值时使用的是第二个对象 t2 调用成员变量，所以在第一个对象 t1 调用成员变量打印该值时仍然是成员变量的初始值。由此可见，两个对象的产生是相互独立的，改变了 t2 的 i 值，不会影响到 t1 的 i 值。在内存中这两个对象的布局如图 7.15 所示。

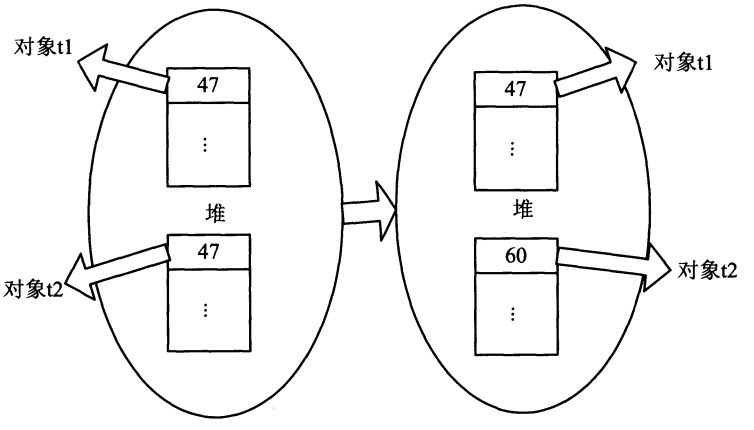

图 7.15　内存中 t1、t2 两个对象的布局

如果希望成员变量不被其中任何一个对象改变，可以使用 static 关键字（前文曾经介绍过一个被声明为 static 的成员变量的值可以被本类或其他类的对象共享）将上述代码改写为例 7.13 的形式。

【例 7.13】 在项目中创建 AccessProperty 类，该类举例说明对象调用静态成员变量。（**实例位置：光盘\TM\sl\7.04**）

```
public class AccessProperty {
    static int i = 47;                          //定义静态成员变量
    public void call() {                        //定义成员方法
        System.out.println("调用 call()方法");
        for (i = 0; i < 3; i++) {
            System.out.print(i + " ");
            if (i == 2) {
                System.out.println("\n");
            }
        }
```

```
    }
    public AccessProperty() {                          //定义构造方法
    }
    public static void main(String[] args) {           //定义主方法
        AccessProperty t1 = new AccessProperty();      //创建一个对象
        AccessProperty t2 = new AccessProperty();      //创建另一个对象
        t2.i = 60;                                     //将类成员变量赋值为 60
        //使用第一个对象调用类成员变量
        System.out.println("第一个实例对象调用变量 i 的结果：" + t1.i++);
        t1.call();                                     //使用第一个对象调用类成员方法
        //使用第二个对象调用类成员变量
        System.out.println("第二个实例对象调用变量 i 的结果：" + t2.i);
        t2.call();                                     //使用第二个对象调用类成员方法
    }
}
```

在 Eclipse 中运行上述代码，结果如图 7.16 所示。

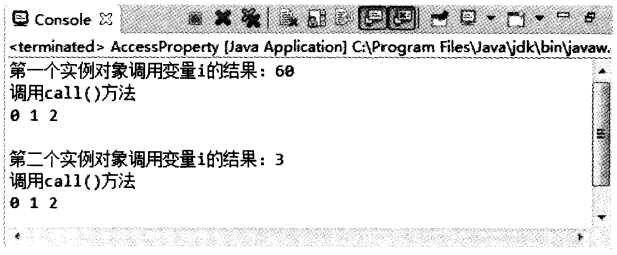

图 7.16　对象调用静态成员变量运行结果

从上述运行结果中可以看到，由于使用 t2.i=60 语句改变了静态成员变量的值，使用对象 t1 调用成员变量的值也为 60，这正是 i 值被定义为静态成员变量的效果，即使使用两个对象对同一个静态成员变量进行操作，依然可以改变静态成员变量的值，因为在内存中两个对象同时指向同一块内存区域。t1.i++语句执行完毕后，i 值变为 3。当再次调用 call()方法时又被重新赋值为 0，做循环打印操作。

7.6.3　对象的引用

在 Java 语言中尽管一切都可以看作对象，但真正的操作标识符实质上是一个引用，那么引用在 Java 中是如何体现的呢？来看下面的语法。

语法如下：

```
类名 对象引用名称
```

如一个 Book 类的引用可以使用以下代码：

```
Book book;
```

通常一个引用不一定需要有一个对象相关联。引用与对象相关联的语法如下：

```
Book book=new Book();
```

- ☑ Book：类名。
- ☑ book：对象。
- ☑ new：创建对象操作符。

注意

引用只是存放一个对象的内存地址，并非存放一个对象。严格地说，引用和对象是不同的，但是可以将这种区别忽略，如可以简单地说 book 是 Book 类的一个对象，而事实上应该是 book 包含 Book 对象的一个引用。

7.6.4 对象的比较

在 Java 语言中有两种对象的比较方式，分别为“==”运算符与 equals()方法。实质上这两种方式有着本质区别，下面举例说明。

【例 7.14】 在项目中创建 Compare 类，该类说明了“==”运算符与 equals()方法的区别。（**实例位置：光盘\TM\sl\7.05**）

```
public class Compare {
    public static void main(String[] args) {
        String c1 = new String("abc");              //创建两个 String 型对象引用
        String c2 = new String("abc");
        String c3 = c1;                             //将 c1 对象引用赋予 c3
        //使用“==”运算符比较 c2 与 c3
        System.out.println("c2==c3 的运算结果为：" + (c2 == c3));
        //使用 equals()方法比较 c2 与 c3
        System.out.println("c2.equals(c3)的运算结果为：" + (c2.equals(c3)));
    }
}
```

在 Eclipse 中运行本例，结果如图 7.17 所示。

从上述运行结果中可以看出，“==”运算符和 equals()方法比较的内容是不相同的，equals()方法是 String 类中的方法，它用于比较两个对象引用所指的内容是否相等；而“==”运算符比较的是两个对象引用的地址是否相等。由于 c1 与 c2 是两个不同的对象引用，两者在内存中的位置不同，而“String c3=c1;”语句将 c1 的引用赋给 c3，所以 c1 与 c3 这两个对象引用是相等的，也就是打印 c1==c3 这样的语句将返回 true 值。对象 c1、c2 和 c3 在内存中的布局如图 7.18 所示。

图 7.17　比较两个对象引用的运行结果

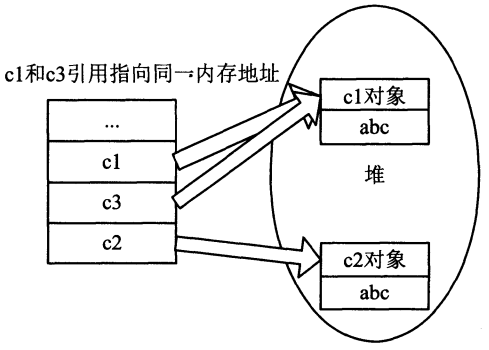

图 7.18　对象 c1、c2 和 c3 在内存中的布局

7.6.5　对象的销毁

每个对象都有生命周期，当对象的生命周期结束时，分配给该对象的内存地址将会被回收。在其他语言中需要手动回收废弃的对象，但是 Java 拥有一套完整的垃圾回收机制，用户不必担心废弃的对象占用内存，垃圾回收器将回收无用的但占用内存的资源。

在谈到垃圾回收机制之前，首先需要了解何种对象会被 Java 虚拟机视为垃圾。主要包括以下两种情况：

☑ 对象引用超过其作用范围，这个对象将被视为垃圾，如图 7.19 所示。

☑ 将对象赋值为 null，如图 7.20 所示。

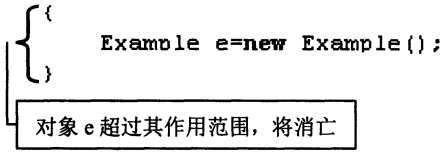

图 7.19　对象超过作用范围将消亡

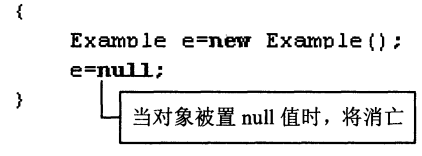

图 7.20　对象被置为 null 值时将消亡

虽然垃圾回收机制已经很完善，但垃圾回收器只能回收那些由 new 操作符创建的对象。如果某些对象不是通过 new 操作符在内存中获取一块内存区域，这种对象可能不能被垃圾回收机制所识别，所以在 Java 中提供了一个 finalize()方法。这个方法是 Object 类的方法，它被声明为 protected，用户可以在自己的类中定义这个方法。如果用户在类中定义了 finalize()方法，在垃圾回收时会首先调用该方法，在下一次垃圾回收动作发生时，才能真正回收被对象占用的内存。

说明

有一点需要明确的是，垃圾回收或 finalize()方法不保证一定会发生，如 Java 虚拟机内存损耗待尽时，它是不会执行垃圾回收的。

由于垃圾回收不受人为控制，具体执行时间也不确定，所以 finalize()方法也就无法执行，为此，Java 提供了 System.gc()方法强制启动垃圾回收器，这与给 120 打电话通知医院来救护病人的道理一样，

来告知垃圾回收器进行清理。

7.7 小　　结

本章学习了面向对象的概念、类的定义、成员方法、类的构造方法、主方法以及对象的应用等。

通过对本章的学习，读者应该掌握面向对象的编程思想，这对 Java 的学习十分有帮助，同时在此基础上读者可以编写类，定义类成员、构造方法、主方法以解决一些实际问题，类以及类成员可以使用权限修饰符进行修饰，读者应该了解这些修饰符的具体范围。由于在 Java 中通过对象来处理问题，所以对象的创建、比较、销毁的应用就显得非常重要。初学者应该反复揣摩这些基本概念和面向对象的编程思想，为 Java 语言的学习打下坚实的基础。

7.8 实践与练习

1．尝试编写一个类，定义一个修饰权限为 private 的成员变量，并定义两个成员方法，一个成员方法实现为此成员变量赋值，另一个成员方法获取这个成员变量的值，保证其他类继承该类时能获取该类的成员变量的值。（**答案位置：光盘\TM\sl\7.06**）

2．尝试编写一个矩形类，将长与宽作为矩形类的属性，在构造方法中将长、宽初始化，定义一个成员方法求此矩形的面积。（**答案位置：光盘\TM\sl\7.07**）

3．根据运行参数的个数决定循环打印变量 i 值的次数。（**答案位置：光盘\TM\sl\7.08**）

第 8 章

包装类

（视频讲解：11 分钟）

Java 是一种面向对象语言，Java 中的类把方法与数据连接在一起，构成了自包含式的处理单元。但在 Java 中不能定义基本类型（Primitive Type）对象，为了能将基本类型视为对象进行处理，并能连接相关的方法，Java 为每个基本类型都提供了包装类，如 int 型数值的包装类 Integer 和 boolean 型数值的包装类 Boolean 等，这样便可以把这些基本类型转换为对象来处理了。需要说明的是，Java 是可以直接处理基本类型的，但在有些情况下需要将其作为对象来处理，这时就需要将其转换为包装类了。本章将介绍 Java 中提供的各种包装类。

通过阅读本章，您可以：

- 掌握 Integer 对象的创建以及 Integer 类提供的各种方法
- 掌握 Long 对象的创建以及 Long 类提供的各种方法
- 掌握 Short 对象的创建以及 Short 类提供的各种方法
- 掌握 Boolean 对象的创建以及 Boolean 类提供的各种方法
- 掌握 Byte 对象的创建以及 Byte 类提供的各种方法
- 掌握 Character 对象的创建以及 Character 类提供的各种方法
- 掌握 Double 对象的创建以及 Double 类提供的各种方法
- 掌握 Float 对象的创建以及 Float 类提供的各种方法
- 了解所有数字类的父类 Number

8.1 Integer

视频讲解：光盘\TM\lx\8\Integer.exe

java.lang 包中的 Integer 类、Long 类和 Short 类，分别将基本类型 int、long 和 short 封装成一个类。由于这些类都是 Number 的子类，区别就是封装不同的数据类型，其包含的方法基本相同，所以本节以 Integer 类为例介绍整数包装类。

Integer 类在对象中包装了一个基本类型 int 的值。该类的对象包含一个 int 类型的字段。此外，该类提供了多个方法，能在 int 类型和 String 类型之间互相转换，同时还提供了其他一些处理 int 类型时非常有用的常量和方法。

1. 构造方法

Integer 类有以下两种构造方法。

☑ Integer (int number)

该方法以一个 int 型变量作为参数来获取 Integer 对象。

【例 8.1】 以 int 型变量作为参数创建 Integer 对象，实例代码如下：

```
Integer number = new Integer(7);
```

☑ Integer (String str)

该方法以一个 String 型变量作为参数来获取 Integer 对象。

【例 8.2】 以 String 型变量作为参数创建 Integer 对象，实例代码如下：

```
Integer number = new Integer("45");
```

注意

要用数值型 String 变量作为参数，如 123，否则将会抛出 NumberFormatException 异常。

2. 常用方法

Integer 类的常用方法如表 8.1 所示。

表 8.1 Integer 类的常用方法

方 法	返 回 值	功 能 描 述
byteValue()	byte	以 byte 类型返回该 Integer 的值
compareTo(Integer anotherInteger)	int	在数字上比较两个 Integer 对象。如果这两个值相等，则返回 0；如果调用对象的数值小于 anotherInteger 的数值，则返回负值；如果调用对象的数值大于 anotherInteger 的数值，则返回正值
equals(Object IntegerObj)	boolean	比较此对象与指定的对象是否相等
intValue()	int	以 int 型返回此 Integer 对象
shortValue()	short	以 short 型返回此 Integer 对象

续表

方　法	返 回 值	功 能 描 述
toString()	String	返回一个表示该 Integer 值的 String 对象
valueOf(String str)	Integer	返回保存指定的 String 值的 Integer 对象
parseInt(String str)	int	返回包含在由 str 指定的字符串中的数字的等价整数值

Integer 类中的 parseInt()方法返回与调用该方法的数值字符串相应的整型（int）值。下面通过一个实例来说明 parseInt()方法的应用。

【例 8.3】 在项目中创建类 Summation，在主方法中定义 String 数组，实现将 String 类型数组中的元素转换成 int 型，并将各元素相加。（**实例位置：光盘\TM\sl\8.01**）

```
public class Summation {                                    //创建类 Summation
    public static void main(String args[]) {                //主方法
        String str[] = { "89", "12", "10", "18", "35" };    //定义 String 数组
        int sum = 0;                                        //定义 int 型变量 sum
        for (int i = 0; i < str.length; i++) {              //循环遍历数组
            int myint=Integer.parseInt(str[i]);             //将数组中的每个元素都转换为 int 型
            sum = sum + myint;                              //将数组中的各元素相加
        }
        System.out.println("数组中的各元素之和是：" + sum);    //将计算后结果输出
    }
}
```

运行结果如图 8.1 所示。

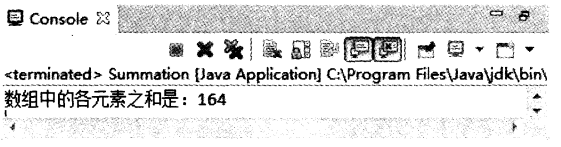

图 8.1　例 8.3 的运行结果

Integer 类的 toString()方法，可将 Integer 对象转换为十进制字符串表示。toBinaryString()、toHexString()和 toOctalString()方法分别将值转换成二进制、十六进制和八进制字符串。实例 8.4 介绍了这 3 种方法的用法。

【例 8.4】 在项目中创建类 Charac，在主方法中创建 String 变量，实现将字符变量以二进制、十六进制和八进制形式输出。（**实例位置：光盘\TM\sl\8.02**）

```
public class Charac {                                       //创建类 Charac
    public static void main(String args[]) {                //主方法
        String str = Integer.toString(456);                 //获取数字的十进制表示
        String str2 = Integer.toBinaryString(456);          //获取数字的二进制表示
        String str3 = Integer.toHexString(456);             //获取数字的十六进制表示
        String str4 = Integer.toOctalString(456);           //获取数字的八进制表示
        System.out.println("'456'的十进制表示为：" + str);
        System.out.println("'456'的二进制表示为：" + str2);
        System.out.println("'456'的十六进制表示为：" + str3);
        System.out.println("'456'的八进制表示为：" + str4);
```

```
    }
}
```

运行结果如图 8.2 所示。

3. 常量

Integer 类提供了以下 4 个常量。

☑ MAX_VALUE：表示 int 类型可取的最大值，即 $2^{31}-1$。

☑ MIN_VALUE：表示 int 类型可取的最小值，即 -2^{31}。

☑ SIZE：用来以二进制补码形式表示 int 值的位数。

☑ TYPE：表示基本类型 int 的 Class 实例。

可以通过程序来验证 Integer 类的常量。

【例 8.5】 在项目中创建类 GetCon，在主方法中实现将 Integer 类的常量值输出。（**实例位置：光盘\TM\sl\8.03**）

```
public class GetCon {                                          //创建类 GetCon
    public static void main(String args[]) {                  //主方法
        int maxint = Integer.MAX_VALUE;                        //获取 Integer 类的常量值
        int minint = Integer.MIN_VALUE;
        int intsize = Integer.SIZE;
        System.out.println("int 类型可取的最大值是：" + maxint); //将常量值输出
        System.out.println("int 类型可取的最小值是：" + minint);
        System.out.println("int 类型的二进制位数是：" + intsize);
    }
}
```

运行结果如图 8.3 所示。

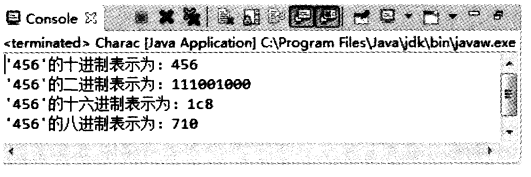

图 8.2 例 8.4 的运行结果

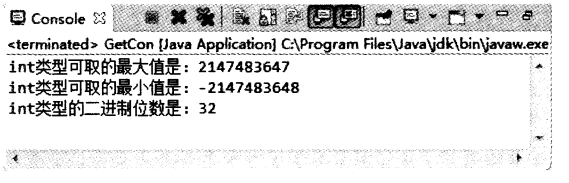

图 8.3 例 8.5 的运行结果

8.2 Boolean

视频讲解：光盘\TM\lx\8\Boolean.exe

Boolean 类将基本类型为 boolean 的值包装在一个对象中。一个 Boolean 类型的对象只包含一个类型为 boolean 的字段。此外，此类还为 boolean 和 String 的相互转换提供了许多方法，并提供了处理 boolean 时非常有用的其他一些常量和方法。

1．构造方法

☑　Boolean(boolean value)

该方法创建一个表示 value 参数的 Boolean 对象。

【例 8.6】 创建一个表示 value 参数的 Boolean 对象，实例代码如下：

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

☑　Boolean(String str)

该方法以 String 变量作为参数创建 Boolean 对象。如果 String 参数不为 null 且在忽略大小写时等于 true，则分配一个表示 true 值的 Boolean 对象，否则获得一个 false 值的 Boolean 对象。

【例 8.7】 以 String 变量作为参数，创建 Boolean 对象。实例代码如下：

```
Boolean bool = new Boolean("ok");
```

2．常用方法

Boolean 类的常用方法如表 8.2 所示。

表 8.2　Boolean 类的常用方法

方　法	返 回 值	功 能 描 述
booleanValue()	boolean	将 Boolean 对象的值以对应的 boolean 值返回
equals(Object obj)	boolean	判断调用该方法的对象与 obj 是否相等。当且仅当参数不是 null，而且与调用该方法的对象一样都表示同一个 boolean 值的 Boolean 对象时，才返回 true
parseBoolean(String s)	boolean	将字符串参数解析为 boolean 值
toString()	String	返回表示该 boolean 值的 String 对象
valueOf(String s)	boolean	返回一个用指定的字符串表示值的 boolean 值

【例 8.8】 在项目中创建类 GetBoolean，在主方法中以不同的构造方法创建 Boolean 对象，并调用 booleanValue()方法将创建的对象重新转换为 boolean 数据输出。（**实例位置：光盘\TM\sl\8.04**）

```
public class GetBoolean {                          //创建类 GetBoolean
    public static void main(String args[]) {       //主方法
        Boolean b1 = new Boolean(true);             //创建 Boolean 对象
        Boolean b2 = new Boolean("ok");             //创建 Boolean 对象
        System.out.println("b1：" + b1.booleanValue());
        System.out.println("b2：" + b2.booleanValue());
    }
}
```

运行结果如图 8.4 所示。

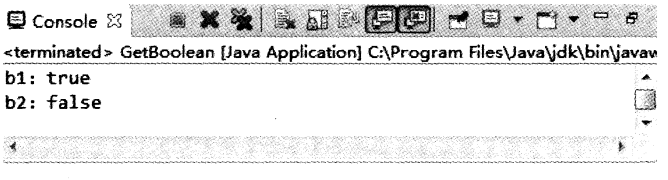

图 8.4　例 8.8 的运行结果

3. 常量

Boolean 提供了以下 3 个常量。

☑ TRUE：对应基值 true 的 Boolean 对象。

☑ FALSE：对应基值 false 的 Boolean 对象。

☑ TYPE：基本类型 boolean 的 Class 对象。

8.3 Byte

视频讲解：光盘\TM\lx\8\Byte.exe

Byte 类将基本类型为 byte 的值包装在一个对象中。一个 Byte 类型的对象只包含一个类型为 byte 的字段。此外，该类还为 byte 和 String 的相互转换提供了方法，并提供了其他一些处理 byte 时非常有用的常量和方法。

1. 构造方法

Byte 类提供了以下两种构造方法的重载形式来创建 Byte 类对象。

☑ Byte(byte value)

通过这种方法创建的 Byte 对象，可表示指定的 byte 值。

【例 8.9】 以 byte 型变量作为参数，创建 Byte 对象。实例代码如下：

```
byte mybyte = 45;
Byte b = new Byte(mybyte);
```

☑ Byte(String str)

通过这种方法创建的 Byte 对象，可表示 String 参数所指示的 byte 值。

【例 8.10】 以 String 型变量作为参数，创建 Byte 对象。实例代码如下：

```
Byte mybyte = new Byte("12");
```

注意

要用数值型 String 变量作为参数，如 123，否则将会抛出 NumberFormatException 异常。

2. 常用方法

Byte 类的常用方法如表 8.3 所示。

表 8.3 Byte 类的常用方法

方　法	返 回 值	功 能 描 述
byteValue()	byte	以一个 byte 值返回 Byte 对象
compareTo(Byte anotherByte)	int	在数字上比较两个 Byte 对象
doubleValue()	double	以一个 double 值返回此 Byte 的值

续表

方　法	返 回 值	功 能 描 述
intValue()	int	以一个 int 值返回此 Byte 的值
parseByte(String s)	byte	将 String 型参数解析成等价的字节（byte）形式
toString()	String	返回表示此 Byte 的值的 String 对象
valueOf(String str)	byte	返回一个保持指定 String 所给出的值的 Byte 对象
equals(Object obj)	boolean	将此对象与指定对象比较，如果调用该方法的对象与 obj 相等，则返回 true，否则返回 false

3. 常量

Byte 类中提供了如下 4 个常量。

☑ MIN_VALUE：byte 类型可取的最小值。
☑ MAX_VALUE：byte 类型可取的最大值。
☑ SIZE：用于以二进制补码形式表示 byte 值的位数。
☑ TYPE：表示基本类型 byte 的 Class 实例。

8.4　Character

视频讲解：光盘\TM\lx\8\Character.exe

Character 类在对象中包装一个基本类型为 char 的值。一个 Character 类型的对象包含类型为 char 的单个字段。该类提供了几种方法，以确定字符的类别（小写字母、数字等），并将字符从大写转换成小写，反之亦然。

1. 构造方法

Character 类的构造方法的语法如下：

```
Character(char value)
```

该类的构造函数必须是一个 char 类型的数据。通过该构造函数创建的 Character 类对象包含由 char 类型参数提供的值。一旦 Character 类被创建，它包含的数值就不能改变了。

【例 8.11】 以 char 型变量作为参数，创建 Character 对象。实例代码如下：

```
Character mychar = new Character('s');
```

2. 常用方法

Character 类提供了很多方法来完成对字符的操作，常用的方法如表 8.4 所示。

表 8.4　Character 类的常用方法

方　法	返 回 值	功 能 描 述
charvalue()	char	返回此 Character 对象的值
compareTo(Character anotherCharacter)	int	根据数字比较两个 Character 对象，若这两个对象相等则返回 0

续表

方　法	返 回 值	功 能 描 述
equals(Object obj)	Boolean	将调用该方法的对象与指定的对象相比较
toUpperCase(char ch)	char	将字符参数转换为大写
toLowerCase(char ch)	char	将字符参数转换为小写
toString()	String	返回一个表示指定 char 值的 String 对象
charValue()	char	返回此 Character 对象的值
isUpperCase(char ch)	boolean	判断指定字符是否为大写字符
isLowerCase(char ch)	boolean	判断指定字符是否为小写字符

下面通过实例来介绍 Character 对象的某些方法的使用。

【例 8.12】 在项目中创建类 UpperOrLower，在主方法中创建 Character 类的对象，并判断字符的大小写状态。（实例位置：光盘\TM\sl\8.05）

```
public class UpperOrLower {                          //创建类 UpperOrLower
    public static void main(String args[]) {         //主方法
        Character mychar1 = new Character('A');       //声明 Character 对象
        Character mychar2 = new Character('a');       //声明 Character 对象
        System.out.println(mychar1 + "是大写字母吗？"
                + Character.isUpperCase(mychar1));
        System.out.println(mychar2 + "是小写字母吗？"
                + Character.isLowerCase(mychar2));
    }
}
```

运行结果如图 8.5 所示。

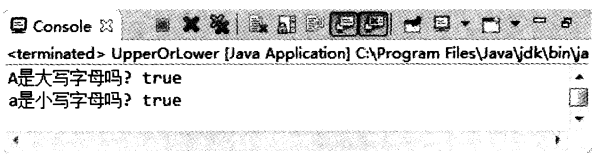

图 8.5　实例 8.12 的运行结果

3．常量

Character 类提供了大量表示特定字符的常量。例如：

☑ CONNECTOR_PUNCTUATION：返回 byte 型值，表示 Unicode 规范中的常规类别“Pc”。

☑ UNASSIGNED：返回 byte 型值，表示 Unicode 规范中的常规类别“Cn”。

☑ TITLECASE_LETTER：返回 byte 型值，表示 Unicode 规范中的常规类别“Lt”。

8.5　Double

视频讲解：光盘\TM\lx\8\Double.exe

Double 和 Float 包装类是对 double、float 基本类型的封装，它们都是 Number 类的子类，又都是对

小数进行操作，所以常用方法基本相同，本节将对 Double 类进行介绍。对于 Float 类可以参考 Double 类的相关介绍。

Double 类在对象中包装一个基本类型为 double 的值。每个 Double 类的对象都包含一个 double 类型的字段。此外，该类还提供多个方法，可以将 double 转换为 String，也可以将 String 转换为 double，也提供了其他一些处理 double 时有用的常量和方法。

1. 构造方法

Double 类提供了以下两种构造方法来获得 Double 类对象。

☑ Double(double value)：基于 double 参数创建 Double 类对象。

☑ Double(String str)：构造一个新分配的 Double 对象，表示用字符串表示的 double 类型的浮点值。

注意

如果不是以数值类型的字符串作为参数，则抛出 NumberFormatException 异常。

2. 常用方法

Double 类的常用方法如表 8.5 所示。

表 8.5　Double 类的常用方法

方　法	返 回 值	功 能 描 述
byteValue()	byte	以 byte 形式返回 Double 对象值（通过强制转换）
compareTo(Double d)	int	对两个 Double 对象进行数值比较。如果两个值相等，则返回 0；如果调用对象的数值小于 d 的数值，则返回负值；如果调用对象的数值大于 d 的值，则返回正值
equals(Object obj)	boolean	将此对象与指定的对象相比较
intValue()	int	以 int 形式返回 double 值
isNaN()	boolean	如果此 double 值是非数字（NaN）值，则返回 true；否则返回 false
toString()	String	返回此 Double 对象的字符串表示形式
valueOf(String str)	Double	返回保存用参数字符串 str 表示的 double 值的 Double 对象
doubleValue()	double	以 double 形式返回此 Double 对象
longValue()	long	以 long 形式返回此 double 的值（通过强制转换为 long 类型）

3. 常量

Double 类提供了以下常量。

☑ MAX_EXPONENT：返回 int 值，表示有限 double 变量可能具有的最大指数。

☑ MIN_EXPONENT：返回 int 值，表示标准化 double 变量可能具有的最小指数。

☑ NEGATIVE_INFINITY：返回 double 值，表示保存 double 类型的负无穷大值的常量。

☑ POSITIVE_INFINITY：返回 double 值，表示保存 double 类型的正无穷大值的常量。

8.6 Number

视频讲解：光盘\TM\lx\8\Number.exe

抽象类 Number 是 BigDecimal、BigInteger、Byte、Double、Float、Integer、Long 和 Short 类的父类，Number 的子类必须提供将表示的数值转换为 byte、double、float、int、long 和 short 的方法。例如，doubleValue()方法返回双精度值，floatValue()方法返回浮点值。这些方法如表 8.6 所示。

表 8.6 Number 类的方法

方 法	返 回 值	功 能 描 述
byteValue()	byte	以 byte 形式返回指定的数值
intValue()	int	以 int 形式返回指定的数值
floatValue()	float	以 float 形式返回指定的数值
shortValue()	short	以 short 形式返回指定的数值
longValue()	long	以 long 形式返回指定的数值
doubleValue()	double	以 double 形式返回指定的数值

Number 类的方法分别被 Number 的各子类所实现，也就是说，在 Number 类的所有子类中都包含以上这几种方法。

8.7 小 结

本章介绍了 Java 中表示数字、字符、布尔值的包装类，其中 Number 是所有数字类的父类，其子类包括 Integer、Float 等；Character 类是字符的包装类，该类提供了对字符的各种处理方法；Boolean 类是布尔类型值的包装类。通过学习本章，读者应该熟练掌握各种包装类所提供的方法，在实际开发中要灵活运用。

8.8 实践与练习

1．创建 Integer 类对象，并以 int 类型将 Integer 的值返回。（**答案位置：光盘\TM\sl\8.06**）

2．创建两个 Character 对象，通过 equals()方法比较它们是否相等；之后将这两个对象分别转换成小写形式，再通过 equals()方法比较这两个 Character 对象是否相等。（**答案位置：光盘\TM\sl\8.07**）

3．编写程序，实现通过字符型变量创建 boolean 值，再将其转换成字符串输出，观察输出后的字符串与创建 Boolean 对象时给定的参数是否相同。（**答案位置：光盘\TM\sl\8.08**）

第 9 章

数字处理类

（视频讲解：16 分钟）

在解决实际问题时，对数字的处理是非常普遍的，如数学问题、随机问题、商业货币问题、科学计数问题等。为了应对以上问题，Java 提供了处理相关问题的类，包括 DecimalFormat 类（用于格式化数字）、Math 类（为各种数学计算提供了工具方法）、Random 类（为 Java 处理随机数问题提供了各种方法）、BigInteger 类与 BigDecimal 类（为所有大数字的处理提供了相应的数学运算操作方法）。本章将学习这些数字处理类。

通过阅读本章，您可以：

- 掌握对数字进行格式化
- 掌握 Math 类中的各种数学运算方法
- 掌握生成任意范围内的随机数
- 掌握大整数与大小数的数学运算方式

9.1 数字格式化

视频讲解：光盘\TM\lx\9\数字格式化.exe

数字的格式化在解决实际问题时使用非常普遍，如表示某超市的商品价格，需要保留两位有效数字。Java 主要对浮点型数据进行数字格式化操作，其中浮点型数据包括 double 型和 float 型数据，在 Java 中使用 java.text.DecimalFormat 格式化数字，本节将着重讲解 DecimalFormat 类。

在 Java 中没有格式化的数据遵循以下原则：

☑ 如果数据绝对值大于 0.001 并且小于 10000000，Java 将以常规小数形式表示。

☑ 如果数据绝对值小于 0.001 或者大于 10000000，使用科学记数法表示。

由于上述输出格式不能满足解决实际问题的要求，通常将结果格式化为指定形式后输出。在 Java 中可以使用 DecimalFormat 类进行格式化操作。

DecimalFormat 是 NumberFormat 的一个子类，用于格式化十进制数字。它可以将一些数字格式化为整数、浮点数、百分数等。通过使用该类可以为要输出的数字加上单位或控制数字的精度。一般情况下可以在实例化 DecimalFormat 对象时传递数字格式，也可以通过 DecimalFormat 类中的 applyPattern() 方法来实现数字格式化。

当格式化数字时，在 DecimalFormat 类中使用一些特殊字符构成一个格式化模板，使数字按照一定的特殊字符规则进行匹配。在表 9.1 中列举了格式化模板中的特殊字符及其所代表的含义。

表 9.1 DecimalFormat 类中特殊字符说明

字　符	说　明
0	代表阿拉伯数字，使用特殊字符“0”表示数字的一位阿拉伯数字，如果该位不存在数字，则显示 0
#	代表阿拉伯数字，使用特殊字符“#”表示数字的一位阿拉伯数字，如果该位存在数字，则显示字符；如果该位不存在数字，则不显示
.	小数分隔符或货币小数分隔符
-	负号
,	分组分隔符
E	分隔科学记数法中的尾数和指数
%	本符号放置在数字的前缀或后缀，将数字乘以 100 显示为百分数
\u2030	本符号放置在数字的前缀或后缀，将数字乘以 1000 显示为千分数
\u00A4	本符号放置在数字的前缀或后缀，作为货币记号
'	本符号为单引号，当上述特殊字符出现在数字中时，应为特殊符号添加单引号，系统会将此符号视为普通符号处理

下面以实例说明数字格式化的使用。

【例 9.1】 在项目中创建 DecimalFormatSimpleDemo 类，在类中分别定义 SimgleFormat()方法和 UseApplyPatternMethodFormat()方法实现两种格式化数字的方式。（**实例位置：光盘\TM\sl\9.01**）

```
import java.text.DecimalFormat;

public class DecimalFormatSimpleDemo {
```

```
    //使用实例化对象时设置格式化模式
    static public void SimgleFormat(String pattern, double value) {
        //实例化 DecimalFormat 对象
        DecimalFormat myFormat = new DecimalFormat(pattern);
        String output = myFormat.format(value);                    //将数字进行格式化
        System.out.println(value + " " + pattern + " " + output);
    }

    //使用 applyPattern()方法对数字进行格式化
    static public void UseApplyPatternMethodFormat(String pattern, double value) {
        DecimalFormat myFormat=new DecimalFormat();                //实例化 DecimalFormat 对象
        myFormat.applyPattern(pattern);                            //调用 applyPattern()方法设置格式化模板
        System.out
                .println(value + " " + pattern + " " + myFormat.format(value));
    }

    public static void main(String[] args) {
        SimgleFormat("###,###.###", 123456.789);                   //调用静态 SimgleFormat()方法
        SimgleFormat("00000000.###kg", 123456.789);                //在数字后加上单位
        //按照格式模板格式化数字，不存在的位以 0 显示
        SimgleFormat("000000.000", 123.78);
        //调用静态 UseApplyPatternMethodFormat()方法
        UseApplyPatternMethodFormat("#.###%", 0.789);              //将数字转换为百分数形式
        //将小数点后格式化为两位
        UseApplyPatternMethodFormat("###.##", 123456.789);
        //将数字转化为千分数形式
        UseApplyPatternMethodFormat("0.00\u2030", 0.789);
    }
}
```

最后在 Eclipse 中运行上述代码，结果如图 9.1 所示。

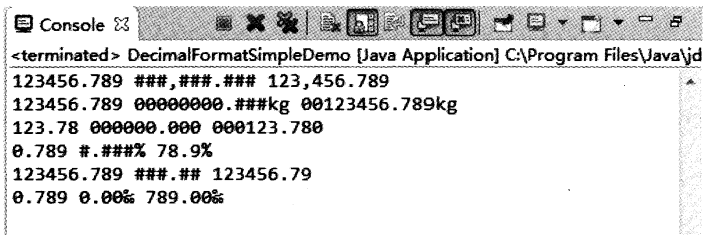

图 9.1　数字格式化

在本实例中可以看到，代码的第一行使用 import 关键字将 java.text.DecimalFormat 这个类包含进来，这是首先告知系统下面的代码将使用到 DecimalFormat 类；然后定义两个格式化数字的方法，这两个方法的参数个数都为两个，分别代表数字格式化模板和具体需要格式化的数字，虽然这两个方法都可以实现格式化数字的操作，但使用的方式有所不同，SimgleFormat()方法是在实例化 DecimalFormat 对象时设置数字格式化模板，而 UseApplyPatternMethodFormat()方法是在实例化 DecimalFormat 对象后调用 applyPattern()方法设置数字格式化模板；最后在主方法中根据不同形式模板格式化数字。在结果中可以

看到以“0”特殊字符构成的模板进行格式化时，当数字某位不存在时，将显示 0；而以“#”特殊字符构成的模板进行格式化操作时，格式化后的数字位数与数字本身的位数一致。

在 DecimalFormat 类中除了可以设置格式化模式来格式化数字之外，还可以使用一些特殊方法对数字进行格式化设置。例如：

```
DecimalFormat myFormat=new DecimalFormat();                    //实例化 DecimalFormat 类对象
myFormat.setGroupingSize(2);                                   //设置将数字分组的大小
myFormat.setGroupingUsed(false);                               //设置是否支持分组
```

在上述代码中，setGroupingSize()方法设置格式化数字的分组大小，setGroupingUsed()方法设置是否可以对数字进行分组操作。为了使读者更好地理解这两个方法的使用，来看下面的实例。

【例 9.2】在项目中创建 DecimalMethod 类，在类的主方法中调用 setGroupingSize()与 setGroupingUsed()方法实现数字的分组。（**实例位置：光盘\TM\sl\9.02**）

```
import java.text.DecimalFormat;

public class DecimalMethod {
    public static void main(String[] args) {
        DecimalFormat myFormat = new DecimalFormat();
        myFormat.setGroupingSize(2);              //设置将数字分组为 2
        String output = myFormat.format(123456.789);
        System.out.println("将数字以每两个数字分组 " + output);
        myFormat.setGroupingUsed(false);          //设置不允许数字进行分组
        String output2 = myFormat.format(123456.789);
        System.out.println("不允许数字分组 " + output2);
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.2 所示。

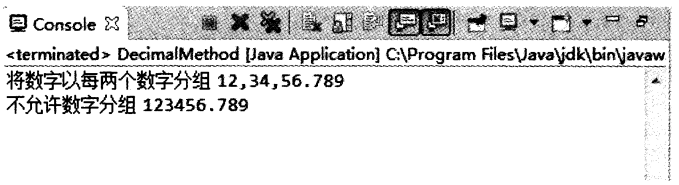

图 9.2　使用 setGroupingSize()与 setGroupingUsed()方法设置数字格式

9.2　数 学 运 算

在 Java 语言中提供了一个执行数学基本运算的 Math 类，该类包括常用的数学运算方法，如三角函数方法、指数函数方法、对数函数方法、平方根函数方法等一些常用数学函数，除此之外还提供了一些常用的数学常量，如 PI、E 等。本节将介绍 Math 类以及其中的一些常用函数方法。

9.2.1　Math 类

视频讲解：光盘\TM\lx\9\Math 类.exe

在 Math 类中提供了众多数学函数方法，主要包括三角函数方法、指数函数方法、取整函数方法、取最大值、最小值以及平均值函数方法，这些方法都被定义为 static 形式，所以在程序中应用比较简便。可以使用如下形式调用：

```
Math.数学方法
```

在 Math 类中除了函数方法之外还存在一些常用数学常量，如 PI、E 等。这些数学常量作为 Math 类的成员变量出现，调用起来也很简单。可以使用如下形式调用：

```
Math.PI
Math.E
```

9.2.2　常用数学运算方法

视频讲解：光盘\TM\lx\9\常用数学运算方法.exe

在 Math 类中的常用数学运算方法较多，大致可以将其分为 4 大类别，分别为三角函数方法、指数函数方法、取整函数方法以及取最大值、最小值和绝对值函数方法。

1．三角函数方法

在 Math 类中包含的三角函数方法如下。

- ☑ public static double sin(double a)：返回角的三角正弦。
- ☑ public static double cos(double a)：返回角的三角余弦。
- ☑ public static double tan(double a)：返回角的三角正切。
- ☑ public static double asin(double a)：返回一个值的反正弦。
- ☑ public static double acos(double a)：返回一个值的反余弦。
- ☑ public static double atan(double a)：返回一个值的反正切。
- ☑ public static double toRadians(double angdeg)：将角度转换为弧度。
- ☑ public static double toDegrees(double angrad)：将弧度转换为角度。

以上每个方法的参数和返回值都是 double 型的。将这些方法的参数的值设置为 double 型是有一定道理的，参数以弧度代替角度来实现，其中 1° 等于 π/180 弧度，所以 180° 可以使用 π 弧度来表示。除了可以获取角的正弦、余弦、正切、反正弦、反余弦、反正切之外，Math 类还提供了角度和弧度相互转换的方法 toRadians()和 toDegrees()。但需要注意的是，角度与弧度的转换通常是不精确的。

【例 9.3】 在项目中创建 TrigonometricFunction 类，在类的主方法中调用 Math 类提供的各种三角函数运算方法，并输出运算结果。（**实例位置：光盘\TM\sl\9.03**）

```
public class TrigonometricFunction {
    public static void main(String[] args) {
        //取 90° 的正弦
        System.out.println("90 度的正弦值：" + Math.sin(Math.PI / 2));
        System.out.println("0 度的余弦值：" + Math.cos(0));        //取 0° 的余弦
        //取 60° 的正切
        System.out.println("60 度的正切值：" + Math.tan(Math.PI / 3));
        //取 2 的平方根与 2 商的反正弦
        System.out.println("2 的平方根与 2 商的反弦值："
                + Math.asin(Math.sqrt(2) / 2));
        //取 2 的平方根与 2 商的反余弦
        System.out.println("2 的平方根与 2 商的反余弦值："
                + Math.acos(Math.sqrt(2) / 2));
        System.out.println("1 的反正切值：" + Math.atan(1));       //取 1 的反正切
        //取 120° 的弧度值
        System.out.println("120 度的弧度值：" + Math.toRadians(120.0));
        //取 π/2 的角度
        System.out.println("π/2 的角度值：" + Math.toDegrees(Math.PI / 2));
    }
}
```

在 Eclipse 中运行上述代码，运行结果如图 9.3 所示。

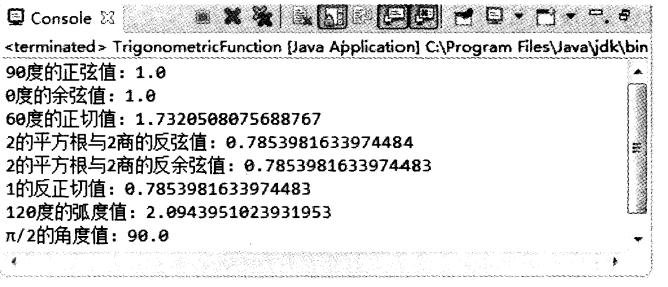

图 9.3　在程序中使用三角函数方法

通过运行结果可以看出，90° 的正弦值为 1，0° 的余弦值为 1，60° 的正切与 Math.sqrt(3)的值应该是一致的，也就是取 3 的平方根。在结果中可以看到第 4~6 行的值是基本相同的，这个值换算后正是 45°，也就是获取的 Math.sqrt(2)/2 反正弦、反余弦值与 1 的反正切值都是 45°。最后两行打印语句实现的是角度和弧度的转换，其中 Math.toRadians(120.0)语句是获取 120° 的弧度值，而 Math.toDegrees(Math.PI/2)语句是获取 π/2 的角度。读者可以将这些具体的值使用 π 的形式表示出来，与上述结果应该是基本一致的，这些结果不能做到十分精确，因为 π 本身也是一个近似值。

2．指数函数方法

Math 类中与指数相关的函数方法如下。

- ☑ public static double exp(double a)：用于获取 e 的 a 次方，即取 e^a。
- ☑ public static double log(double a)：用于取自然对数，即取 lna 的值。
- ☑ public static double log10(double a)：用于取底数为 10 的对数。
- ☑ public static double sqrt(double a)：用于取 a 的平方根，其中 a 的值不能为负值。

☑ public static double cbrt(double a)：用于取 a 的立方根。

☑ public static double pow(double a,double b)：用于取 a 的 b 次方。

指数运算包括求方根、取对数以及求 n 次方的运算。为了使读者更好地理解这些运算函数方法的用法，下面举例说明。

【例 9.4】 在项目中创建 ExponentFunction 类，在类的主方法中调用 Math 类中的方法实现指数函数的运算，并输出运算结果。（**实例位置：光盘\TM\sl\9.04**）

```
public class ExponentFunction {
    public static void main(String[] args) {
        System.out.println("e 的平方值：" + Math.exp(2));          //取 e 的 2 次方
        //取以 e 为底 2 的对数
        System.out.println("以 e 为底 2 的对数值：" + Math.log(2));
        //取以 10 为底 2 的对数
        System.out.println("以 10 为底 2 的对数值：" + Math.log10(2));
        System.out.println("4 的平方根值：" + Math.sqrt(4));       //取 4 的平方根
        System.out.println("8 的立方根值：" + Math.cbrt(8));       //取 8 的立方根
        System.out.println("2 的 2 次方值：" + Math.pow(2, 2));    //取 2 的 2 次方
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.4 所示。

在本实例中可以看到，使用 Math 类中的方法比较简单，直接使用 Math 类名调用相应的方法即可。

3．取整函数方法

在具体的问题中，取整操作使用也很普遍，所以 Java 在 Math 类中添加了数字取整方法。在 Math 类中主要包括以下几种取整方法。

☑ public static double ceil(double a)：返回大于等于参数的最小整数。

☑ public static double floor(double a)：返回小于等于参数的最大整数。

☑ public static double rint(double a)：返回与参数最接近的整数，如果两个同为整数且同样接近，则结果取偶数。

☑ public static int round(float a)：将参数加上 0.5 后返回与参数最近的整数。

☑ public static long round(double a)：将参数加上 0.5 后返回与参数最近的整数，然后强制转换为长整型。

下面以 1.5 作为参数，获取取整函数的返回值。在坐标轴上表示如图 9.5 所示。

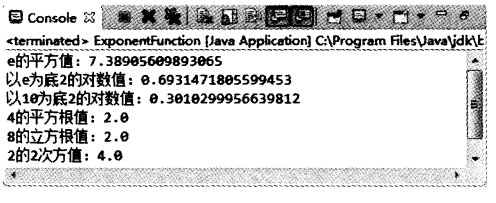

图 9.4　在程序中使用指数函数方法

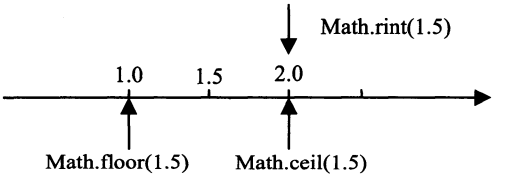

图 9.5　取整函数的返回值

注意

由于数 1.0 和数 2.0 距离数 1.5 都是 0.5 个单位长度，因此返回偶数 2.0。

下面举例说明 Math 类中取整方法的使用。

【例 9.5】 在项目中创建 IntFunction 类，在类的主方法中调用 Math 类中的方法实现取整函数的运算，并输出运算结果。（**实例位置：光盘\TM\sl\9.05**）

```
public class IntFunction {
    public static void main(String[] args) {
        //返回第一个大于等于参数的整数
        System.out.println("使用 ceil()方法取整：" + Math.ceil(5.2));
        //返回第一个小于等于参数的整数
        System.out.println("使用 floor()方法取整：" + Math.floor(2.5));
        //返回与参数最接近的整数
        System.out.println("使用 rint()方法取整：" + Math.rint(2.7));
        //返回与参数最接近的整数
        System.out.println("使用 rint()方法取整：" + Math.rint(2.5));
        //将参数加上 0.5 后返回最接近的整数
        System.out.println("使用 round()方法取整：" + Math.round(3.4f));
        //将参数加上 0.5 后返回最接近的整数，并将结果强制转换为长整型
        System.out.println("使用 round()方法取整：" + Math.round(2.5));
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.6 所示。

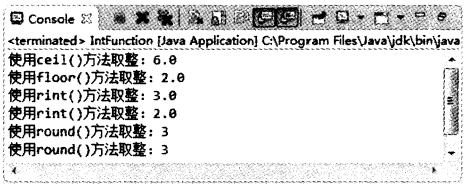

图 9.6 在程序中使用取整函数方法

4．取最大值、最小值、绝对值函数方法

在程序中最常用的方法就是取最大值、最小值、绝对值等，在 Math 类中包括的这些操作方法如下。

- ☑ public static double max(double a,double b)：取 a 与 b 之间的最大值。
- ☑ public static int min(int a,int b)：取 a 与 b 之间的最小值，参数为整型。
- ☑ public static long min(long a,long b)：取 a 与 b 之间的最小值，参数为长整型。
- ☑ public static float min(float a,float b)：取 a 与 b 之间的最小值，参数为浮点型。
- ☑ public static double min(double a,double b)：取 a 与 b 之间的最小值，参数为双精度型。
- ☑ public static int abs(int a)：返回整型参数的绝对值。
- ☑ public static long abs(long a)：返回长整型参数的绝对值。
- ☑ public static float abs(float a)：返回浮点型参数的绝对值。
- ☑ public static double abs(double a)：返回双精度型参数的绝对值。

下面举例说明上述方法的使用。

【例 9.6】 在项目中创建 AnyFunction 类，在类的主方法中调用 Math 类中的方法实现求两数的最大值、最小值和取绝对值运算，并输出运算结果。（**实例位置：光盘\TM\sl\9.06**）

```
public class AnyFunction {
    public static void main(String[] args) {
        System.out.println("4 和 8 较大者:" + Math.max(4, 8));
         //取两个参数的最小值
        System.out.println("4.4 和 4 较小者：" + Math.min(4.4, 4));
        System.out.println("-7 的绝对值：" + Math.abs(-7)); //取参数的绝对值
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.7 所示。

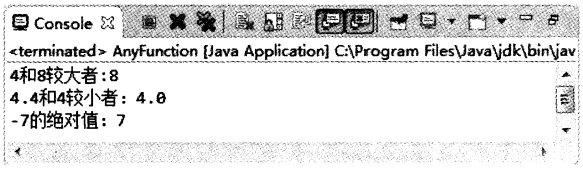

图 9.7　在程序中使用 Math 类取最大值、最小值、绝对值的方法

9.3　随　机　数

在实际开发中产生随机数的使用是很普遍的，所以在程序中进行产生随机数操作很重要。在 Java 中主要提供了两种方式产生随机数，分别为调用 Math 类的 random()方法和 Random 类提供的产生各种数据类型随机数的方法。

9.3.1　Math.random()方法

视频讲解：光盘\TM\lx\9\Math.random()方法.exe

在 Math 类中存在一个 random()方法，用于产生随机数字，这个方法默认生成大于等于 0.0 且小于 1.0 的 double 型随机数，即 0<=Math.random()<1.0，虽然 Math.random()方法只可以产生 0~1 之间的 double 型数字，其实只要在 Math.random()语句上稍加处理，就可以使用这个方法产生任意范围的随机数，如图 9.8 所示。

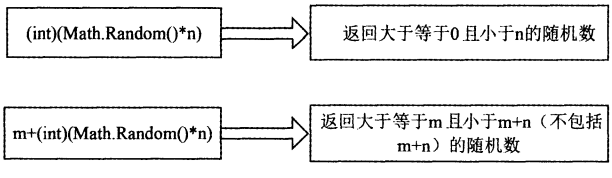

图 9.8　使用 random()方法示意图

为了更好地解释这种产生随机数的方式，下面举例说明。

【例 9.7】在项目中创建 MathRondom 类，在类中编写 GetEvenNum()方法产生两数之间的随机数，并在主方法中输出这个随机数。（**实例位置：光盘\TM\sl\9.07**）

```
public class MathRondom {
    /**
     * 定义产生偶数的方法
     * @param num1 起始范围参数
     * @param num2 终止范围参数
     * @return 随机的范围内偶数
     */
    public static int GetEvenNum(double num1, double num2) {
        //产生 num1~num2 之间的随机数
        int s = (int) num1 + (int) (Math.random() * (num2 - num1));
        if (s % 2 == 0) {              //判断随机数是否为偶数
            return s;                  //返回
        } else
            //如果是奇数
            return s + 1;              //将结果加 1 后返回
    }
    public static void main(String[] args) {
        //调用产生随机数方法
        System.out.println("任意一个 2~32 之间的偶数：" + GetEvenNum(2, 32));
    }
}
```

在 Eclipse 中运行本实例，结果如图 9.9 所示。

本实例每次运行时结果都不相同，这就实现了随机产生数据的功能，并且每次产生的值都是偶数。为了实现这个功能，这里定义了一个方法 GetEvenNum()，该方法的参数分别为产生随机数字的上限与下限。因为 m+(int)(Math.random()*n)语句可以获取 m~m+n 之间的随机数，所以“2+(int)(Math.random()*(32-2));”这个表达式就可以求出 2~32 之间的随机数。当获取到这个区间的随机数以后需要判断这个数字是否为偶数时，对该数字做对 2 取余操作即可。如果该数字为奇数，将该奇数加 1 也可以返回偶数。

使用 Math 类的 random()方法也可以随机生成字符，可以使用如下代码生成 a~z 之间的字符。

```
(char)('a'+Math.random()*('z'-'a'+1));
```

通过上述表达式可以求出更多的随机字符，如 A~Z 之间的随机字符，进而推理出求任意两个字符之间的随机字符，可以使用以下语句表示：

```
(char)(cha1+Math.random()*(cha2-cha1+1));
```

在这里可以将这个表达式设计为一个方法，参数设置为随机产生字符的上限与下限。下面举例说明。

【例 9.8】 在项目中创建 MathRandomChar 类，在类中编写 GetRandomChar()方法产生随机字符，并在主方法中输出该字符。（**实例位置：光盘\TM\sl\9.08**）

```
public class MathRandomChar {
    //定义获取任意字符之间的随机字符
    public static char GetRandomChar(char cha1, char cha2) {
        return (char) (cha1 + Math.random() * (cha2 - cha1 + 1));
```

```
    }
    public static void main(String[] args) {
        //获取 a~z 之间的随机字符
        System.out.println("任意小写字符" + GetRandomChar('a', 'z'));
        //获取 A~Z 之间的随机字符
        System.out.println("任意大写字符" + GetRandomChar('A', 'Z'));
        //获取 0~9 之间的随机字符
        System.out.println("0 到 9 任意数字字符" + GetRandomChar('0', '9'));
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.10 所示。

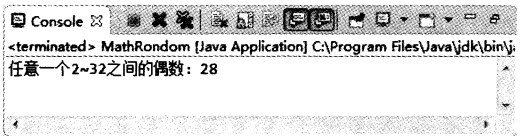

图 9.9　随机产生 2~32 之间的偶数

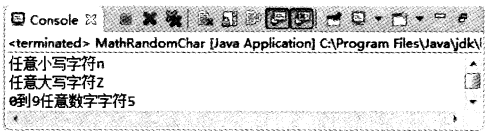

图 9.10　获取任意区间的随机字符

注意

random()方法返回的值实际上是伪随机数，它通过复杂的运算而得到一系列的数。该方法是通过当前时间作为随机数生成器的参数，所以每次执行程序都会产生不同的随机数。

9.3.2　Random 类

视频讲解：光盘\TM\lx\9\Random 类.exe

除了 Math 类中的 random()方法可以获取随机数之外，Java 中还提供了一种可以获取随机数的方式，那就是 java.util.Random 类。可以通过实例化一个 Random 对象创建一个随机数生成器。

语法如下：

```
Random r=new Random();
```

其中，r 是指 Random 对象。

以这种方式实例化对象时，Java 编译器以系统当前时间作为随机数生成器的种子，因为每时每刻的时间不可能相同，所以产生的随机数将不同，但是如果运行速度太快，也会产生两次运行结果相同的随机数。

同时也可以在实例化 Random 类对象时，设置随机数生成器的种子。

语法如下：

```
Random r=new Random(seedValue);
```

☑　r：Random 类对象。

☑　seedValue：随机数生成器的种子。

在 Random 类中提供了获取各种数据类型随机数的方法，下面列举几个常用的方法。

☑　public int nextInt()：返回一个随机整数。

- ☑ public int nextInt(int n)：返回大于等于 0 且小于 n 的随机整数。
- ☑ public long nextLong()：返回一个随机长整型值。
- ☑ public boolean nextBoolean()：返回一个随机布尔型值。
- ☑ public float nextFloat()：返回一个随机浮点型值。
- ☑ public double nextDouble()：返回一个随机双精度型值。
- ☑ public double nextGaussian()：返回一个概率密度为高斯分布的双精度值。

【例 9.9】 在项目中创建 RandomDemo 类，在类的主方法中创建 Random 类的对象，使用该对象生成各种类型的随机数，并输出结果。（**实例位置：光盘\TM\sl\9.09**）

```
import java.util.Random;

public class RandomDemo {
    public static void main(String[] args) {
        Random r = new Random(); //实例化一个 Random 类
        //随机产生一个整数
        System.out.println("随机产生一个整数:" + r.nextInt());
        //随机产生一个大于等于 0 且小于 10 的整数
        System.out.println("随机产生一个大于等于 0 小于 10 的整数：" + r.nextInt(10));
        //随机产生一个布尔型的值
        System.out.println("随机产生一个布尔型的值：" + r.nextBoolean());
        //随机产生一个双精度型的值
        System.out.println("随机产生一个双精度型的值：" + r.nextDouble());
        //随机产生一个浮点型的值
        System.out.println("随机产生一个浮点型的值：" + r.nextFloat());
        //随机产生一个概率密度为高斯分布的双精度值
        System.out.println("随机产生一个概率密度为高斯分布的双精度值："
                + r.nextGaussian());
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.11 所示。

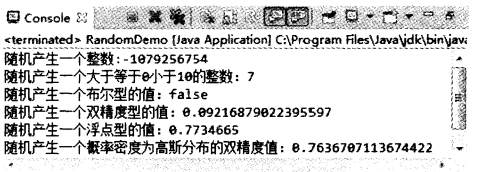

图 9.11 使用 Random 类中的方法产生随机数

9.4 大数字运算

在 Java 中提供了大数字的操作类，即 java.math.BigInteger 类与 java.math.BigDecimal 类。这两个类用于高精度计算，其中 BigInteger 类是针对大整数的处理类，而 BigDecimal 类则是针对大小数的处理类。

9.4.1　BigInteger

视频讲解：光盘\TM\lx\9\BigInteger.exe

BigInteger类型的数字范围较Integer类型的数字范围要大得多。前文介绍过Integer是int的包装类，int的最大值为$2^{31}-1$，如果要计算更大的数字，使用Integer数据类型就无法实现了，所以Java中提供了BigInteger类来处理更大的数字。BigInteger支持任意精度的整数，也就是说在运算中BigInteger类型可以准确地表示任何大小的整数值而不会丢失任何信息。

在BigInteger类中封装了多种操作，除了基本的加、减、乘、除操作之外，还提供了绝对值、相反数、最大公约数以及判断是否为质数等操作。

使用BigInteger类，可以实例化一个BigInteger对象，并自动调用相应的构造函数。BigInteger类具有很多构造函数，但最直接的一种方式是参数以字符串形式代表要处理的数字。

语法如下：

```
public BigInteger(String val)
```

其中，val是十进制字符串。

如果将2转换为BigInteger类型，可以使用以下语句进行初始化操作：

```
BigInteger twoInstance=new BigInteger("2");                    //将十进制 2 转换为 BigInteger 形式
```

注意

参数2的双引号不能省略，因为参数是以字符串的形式存在的。

一旦创建了对象实例，就可以调用BigInteger类中的一些方法进行运算操作，包括基本的数学运算和位运算以及一些取相反数、取绝对值等操作。下面列举了BigInteger类中常用的几种运算方法。

- ☑ public BigInteger add(BigInteger val)：做加法运算。
- ☑ public BigInteger subtract(BigInteger val)：做减法运算。
- ☑ public BigInteger multiply(BigInteger val)：做乘法运算。
- ☑ public BigInteger divide(BigInteger val)：做除法运算。
- ☑ public BigInteger remainder(BigInteger val)：做取余操作。
- ☑ public BigInteger[] divideAndRemainder(BigInteger val)：用数组返回余数和商，结果数组中第一个值为商，第二个值为余数。
- ☑ public BigInteger pow(int exponent)：进行取参数的exponent次方操作。
- ☑ public BigInteger negate()：取相反数。
- ☑ public BigInteger shiftLeft(int n)：将数字左移n位，如果n为负数，做右移操作。
- ☑ public BigInteger shiftRight(int n)：将数字右移n位，如果n为负数，做左移操作。
- ☑ public BigInteger and(BigInteger val)：做与操作。
- ☑ public BigInteger or(BigInteger val)：做或操作。
- ☑ public int compareTo(BigInteger val)：做数字比较操作。

☑ public boolean equals(Object x)：当参数 x 是 BigInteger 类型的数字并且数值相等时，返回 true。

☑ public BigInteger min(BigInteger val)：返回较小的数值。

☑ public BigInteger max(BigInteger val)：返回较大的数值。

【例 9.10】 在项目中创建 BigIntegerDemo 类，在类的主方法中创建 BigInteger 类的实例对象，调用该对象的各种方法实现大整数的加、减、乘、除和其他运算，并输出运算结果。（**实例位置：光盘\TM\sl\9.10**）

```
import java.math.BigInteger;

public class BigIntegerDemo {
    public static void main(String[] args) {
        BigInteger bigInstance = new BigInteger("4"); //实例化一个大数字
        //取该大数字加 2 的操作
        System.out.println("加法操作：" + bigInstance.add(new BigInteger("2")));
        //取该大数字减 2 的操作
        System.out.println("减法操作："
                + bigInstance.subtract(new BigInteger("2")));
        //取该大数字乘以 2 的操作
        System.out.println("乘法操作："
                + bigInstance.multiply(new BigInteger("2")));
        //取该大数字除以 2 的操作
        System.out.println("除法操作："
                + bigInstance.divide(new BigInteger("2")));
        //取该大数字除以 3 的商
        System.out.println("取商："
                + bigInstance.divideAndRemainder(new BigInteger("3"))[0]);
        //取该大数字除以 3 的余数
        System.out.println("取余数："
                + bigInstance.divideAndRemainder(new BigInteger("3"))[1]);
        //取该大数字的 2 次方
        System.out.println("做 2 次方操作：" + bigInstance.pow(2));
        //取该大数字的相反数
        System.out.println("取相反数操作：" + bigInstance.negate());
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.12 所示。

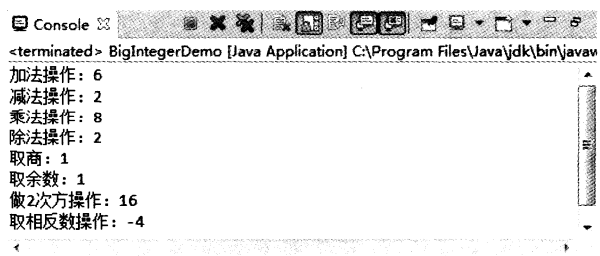

图 9.12　操作大数字

在本实例中需要注意的是 divideAndRemainder()方法，这个方法做除法操作，以数组的形式返回，数组中第一个值为做除法的商，第二个值为做除法的余数。

9.4.2 BigDecimal

视频讲解：光盘\TM\lx\9\BigDecimal.exe

BigDecimal 和 BigInteger 都能实现大数字的运算，不同的是 BigDecimal 加入了小数的概念。一般的 float 型和 double 型数据只可以用来做科学计算或工程计算，但由于在商业计算中要求数字精度比较高，所以要用到 java.math.BigDecimal 类。BigDecimal 类支持任何精度的定点数，可以用它来精确计算货币值。

在 BigDecimal 类中常用的两个构造方法如下。

☑ public BigDecimal(double val)：实例化时将双精度型转换为 BigDecimal 类型。

☑ public BigDecimal(String val)：实例化时将字符串形式转换为 BigDecimal 类型。

BigDecimal 类型的数字可以用来做超大的浮点数的运算，如加、减、乘、除等，但是在所有的运算中除法是最复杂的，因为在除不尽的情况下末位小数点的处理是需要考虑的。

下面列举了 BigDecimal 类中实现加、减、乘、除的方法。

☑ public BigDecimal add(BigDecimal augend)：做加法操作。

☑ public BigDecimal subtract(BigDecimal subtrahend)：做减法操作。

☑ public BigDecimal multiply(BigDecimal multiplicand)：做乘法操作。

☑ public BigDecimal divide(BigDecimal divisor,int scale,int roundingMode)：做除法操作，方法中 3 个参数分别代表除数、商的小数点后的位数、近似处理模式。

在上述方法中，BigDecimal类中divide()方法有多种设置，用于返回商末位小数点的处理，这些模式的名称与含义如表9.2所示。

表 9.2　BigDecimal 类中 divide()方法的多种处理模式

模　式	含　义
BigDecimal.ROUND_UP	商的最后一位如果大于 0，则向前进位，正负数都如此
BigDecimal.ROUND_DOWN	商的最后一位无论是什么数字都省略
BigDecimal.ROUND_CEILING	商如果是正数，按照 ROUND_UP 模式处理；如果是负数，按照 ROUND_DOWN 模式处理。这种模式的处理都会使近似值大于等于实际值
BigDecimal.ROUND_FLOOR	与 ROUND_CEILING 模式相反，商如果是正数，按照 ROUND_DOWN 模式处理；商如果是负数，则按照 ROUND_UP 模式处理。这种模式的处理都会使近似值小于等于实际值
BigDecimal.ROUND_HALF_DOWN	对商进行四舍五入操作，如果商最后一位小于等于 5，则做舍弃操作；如果最后一位大于 5，则做进位操作，如 7.5≈7
BigDecimal.ROUND_HALF_UP	对商进行四舍五入操作，如果商的最后一位小于 5 则舍弃；如果大于等于 5，进行进位操作，如 7.5≈8
BigDecimal.ROUND_HALF_EVEN	如果商的倒数第二位为奇数，则按照 ROUND_HALF_UP 处理；如果为偶数，则按照 ROUND_HALF_DOWN 处理，如 7.5≈8，8.5≈8

下面设计一个类，这个类包括任意两个 Decimal 类型数字的加、减、乘、除运算方法。

【例 9.11】 在项目中创建 BigDecimalDemo 类，在类中分别定义 add()、sub()、mul()和 div()方法实现加、减、乘、除运算，并输出运算结果。（**实例位置：光盘\TM\sl\9.11**）

```
import java.math.BigDecimal;

public class BigDecimalDemo {
    static final int location = 10;

    /**
     * 定义加法方法，参数为加数与被加数
     *
     * @param value1
     *              相加的第一个数
     * @param value2
     *              相加的第二个数
     * @return 两数之和
     */
    public BigDecimal add(double value1, double value2) {
        //实例化 Decimal 对象
        BigDecimal b1 = new BigDecimal(Double.toString(value1));
        BigDecimal b2 = new BigDecimal(Double.toString(value2));
        return b1.add(b2); //调用加法方法
    }

    /**
     * 定义减法方法，参数为减数与被减数
     *
     * @param value1
     *              被减数
     * @param value2
     *              减数
     * @return 运算结果
     */
    public BigDecimal sub(double value1, double value2) {
        BigDecimal b1 = new BigDecimal(Double.toString(value1));
        BigDecimal b2 = new BigDecimal(Double.toString(value2));
        return b1.subtract(b2); //调用减法方法
    }

    /**
     * 定义乘法方法，参数为乘数与被乘数
     *
     * @param value1
     *              第一个乘数
     * @param value2
     *              第二个乘数
     * @return
     */
```

```
    public BigDecimal mul(double value1, double value2) {
        BigDecimal b1 = new BigDecimal(Double.toString(value1));
        BigDecimal b2 = new BigDecimal(Double.toString(value2));
        return b1.multiply(b2); //调用乘法方法
    }

    /**
     * 定义除法方法，参数为除数与被除数
     *
     * @param value1 被除数
     * @param value2 除数
     * @return
     */
    public BigDecimal div(double value1, double value2) {
        return div(value1, value2, location); //调用自定义除法方法
    }

    //定义除法方法，参数分别为除数与被除数以及商小数点后的位数
    public BigDecimal div(double value1, double value2, int b) {
        if (b < 0) {
            System.out.println("b 值必须大于等于 0");
        }
        BigDecimal b1 = new BigDecimal(Double.toString(value1));
        BigDecimal b2 = new BigDecimal(Double.toString(value2));
        //调用除法方法，商小数点后保留 b 位，并将结果进行四舍五入操作
        return b1.divide(b2, b, BigDecimal.ROUND_HALF_UP);
    }

    public static void main(String[] args) {
        BigDecimalDemo b = new BigDecimalDemo();
        System.out.println("两个数字相加结果：" + b.add(-7.5, 8.9));
        System.out.println("两个数字相减结果：" + b.sub(-7.5, 8.9));
        System.out.println("两个数字相乘结果：" + b.mul(-7.5, 8.9));
        System.out.println("两个数字相除结果，结果小数后保留 10 位："+b.div(10, 2));
        System.out.println("两个数字相除，保留小数后 5 位："+b.div(-7.5,8.9, 5));
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 9.13 所示。

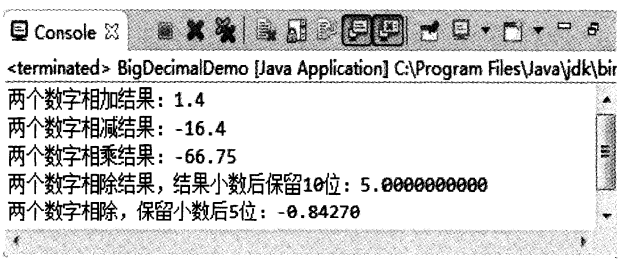

图 9.13　Decimal 类型数字的运算操作

9.5 小　　结

本章学习了 Java 数字格式的处理，以及数学运算、随机数、大数字处理等。其中，数学运算和随机数的产生是本章的重点，在解决实际问题中这两种技巧经常被用到，初学者应该熟练掌握。数字格式化操作在程序中使用也比较广泛，而大数字处理是针对商业货币或科学计算领域提出的解决方案，读者只要对其进行简单了解即可。

9.6 实践与练习

1．尝试开发一个程序，获取 2~32 之间（不包括 32）的 6 个偶数，并取得这 6 个偶数的和。（**答案位置：光盘\TM\sl\9.12**）

2．尝试开发一个程序，定义一个求圆面积的方法，其中以圆半径作为参数，并将计算结果保留 5 位小数。（**答案位置：光盘\TM\sl\9.13**）

3．尝试改写 BigDecimalDemo 类中的 div(double value1,double value2,int b)方法，以不同近似处理模式处理商的精度。（**答案位置：光盘\TM\sl\9.14**）

核心技术

本篇将介绍接口、继承与多态，类的高级特性，异常处理，Swing 程序设计，集合类，I/O（输入/输出），反射，枚举类型与泛型，多线程，网络通信和数据库操作等内容。学习完本篇，应能够开发一些小型应用程序。

第10章

接口、继承与多态

（视频讲解：23 分钟）

学习好继承和多态是面向对象开发语言中非常重要的一个环节，如果在程序中使用继承和多态得当，整个程序的架构将变得非常有弹性，同时可以减少代码的冗余性。继承机制的使用可以复用一些定义好的类，减少重复代码的编写。多态机制的使用可以动态调整对象的调用，降低对象之间的依存关系。为了优化继承与多态，一些类除了继承父类外还使用接口的形式。Java 中的类可以同时实现多个接口，接口被用来建立类与类之间关联的标准。正因为使用了这些机制，才使 Java 语言更具有生命力。

通过阅读本章，您可以：

- 掌握类的继承
- 掌握 Object 类中的几个重要方法
- 掌握对象类型的转换
- 掌握使用 instanceof 操作符判断对象类型
- 掌握方法的重载
- 掌握多态技术
- 掌握使用抽象类与接口

10.1　类的继承

视频讲解：光盘\TM\lx\10\类的继承.exe

继承在面向对象开发思想中是一个非常重要的概念，它使整个程序架构具有一定的弹性，在程序中复用一些已经定义完善的类不仅可以减少软件开发周期，也可以提高软件的可维护性和可扩展性。本节将详细讲解类的继承。

在第 7 章中曾简要介绍过继承，其基本思想是基于某个父类的扩展，制定出一个新的子类，子类可以继承父类原有的属性和方法，也可以增加原来父类所不具备的属性和方法，或者直接重写父类中的某些方法。例如，平行四边形是特殊的四边形，可以说平行四边形类继承了四边形类，这时平行四边形类将所有四边形具有的属性和方法都保留下来，并基于四边形类扩展了一些新的平行四边形类特有的属性和方法。

下面演示一下继承性。创建一个新类 Test，同时创建另一个新类 Test2 继承 Test 类，其中包括重写的父类成员方法（重写的概念将在下文中进行详细介绍）以及新增成员方法等。在图 10.1 中描述了类 Test 与 Test2 的结构以及两者之间的关系。

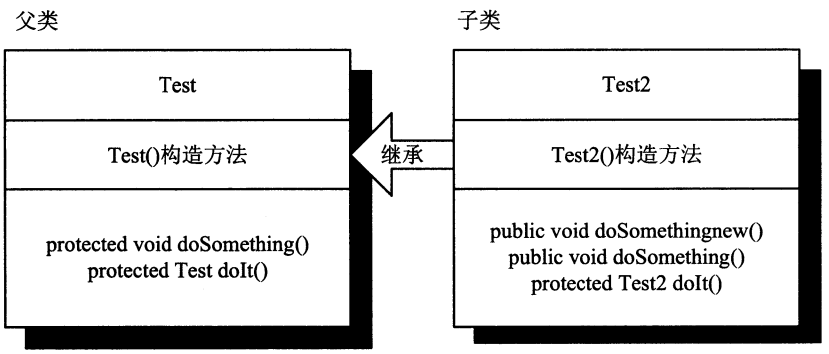

图 10.1　Test 与 Test2 类之间的继承关系

在 Java 中使用 extends 关键字来标识两个类的继承关系，下面将图 10.1 中的继承关系以代码的形式给出，如实例 10.1 所示。

【例 10.1】 在项目中分别创建 Test 类和 Test2 类，在 Test 类中编写成员方法 doSomething()和 doIt()，使 Test2 类继承 Test 类，重写父类的这两个方法和构造方法，并新增 doSomethingnew()方法。其中 Test2 类的构造方法中使用 super 关键字调用父类的构造方法和成员方法等。（**实例位置：光盘\TM\sl\10.01**）

```
class Test {
    public Test() {                          //构造方法
        //SomeSentence
    }
    protected void doSomething() {           //成员方法
        //SomeSentence
    }
    protected Test doIt() {                  //方法返回值类型为 Test 类型
```

```
        return new Test();
    }
}
class Test2 extends Test {                    //继承父类
    public Test2() {                          //构造方法
        super();                              //调用父类构造方法
        super.doSomething();                  //调用父类成员方法
    }
    public void doSomethingnew() {            //新增方法
        //SomeSentence
    }
    public void doSomething() {               //重写父类方法
        //SomeNewSentence
    }
    protected Test2 doIt() {                  //重写父类方法，方法返回值类型为 Test2 类型
        return new Test2();
    }
}
```

例 10.1 中定义了两个类，其中 Test2 类继承 Test 类，可以说 Test 类为 Test2 的父类，Test2 类为 Test 类的子类。在子类中可以连同初始化父类构造方法来完成子类初始化操作，既可以在子类的构造方法中使用 super()语句调用父类的构造方法，也可以在子类中使用 super 关键字调用父类的成员方法等。但是子类没有权限调用父类中被修饰为 private 的方法，只可以调用父类中修饰为 public 或 protected 的成员方法。例如，子类构造方法中可以使用 super 关键字调用父类的 doSomething()方法，因为 doSomething()方法的权限修饰符为 protected。同时在子类中也可以定义一些新方法，如子类中的 doSomethingnew()方法。

继承并不只是扩展父类的功能，还可以重写父类的成员方法。重写（还可以称为覆盖）就是在子类中将父类的成员方法的名称保留，重写成员方法的实现内容，更改成员方法的存储权限，或是修改成员方法的返回值类型（重写父类成员方法的返回值类型是基于 J2SE 5.0 版本以上编译器提供的新功能）。例如，子类中的 doSomething()方法，除了重写方法的实现内容之外，还将方法的修饰权限修改为 public。

在继承中还有一种特殊的重写方式，子类与父类的成员方法返回值、方法名称、参数类型及个数完全相同，唯一不同的是方法实现内容，这种特殊重写方式被称为重构。

注意

当重写父类方法时，修改方法的修饰权限只能从小的范围到大的范围改变，例如，父类中的 doSomething()方法的修饰权限为 protected，继承后子类中的方法 doSomething()的修饰权限只能修改为 public，不能修改为 private。如图 10.2 所示的重写关系就是错误的。

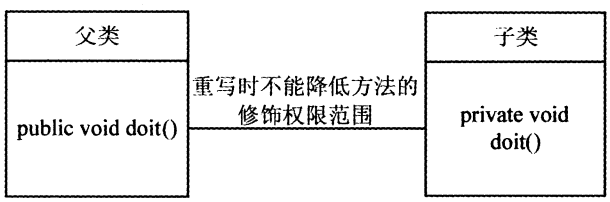

图 10.2　重写时不能降低方法的修饰权限范围

子类重写父类的方法还可以修改方法的返回值类型，但这只是在 J2SE 5.0 以上的版本中支持的新功能。例如，例 10.1 子类中的 doIt()方法就使用了这个新功能，父类中的 doIt()方法的返回值类型为 Test，而子类中的 doIt()方法的返回值类型为 Test2，子类中重写了父类的 doIt()方法。这种重写方式需要遵循一个原则，即重写的返回值类型必须是父类中同一方法返回值类型的子类，而 Test2 类正是 Test 类的子类。

在 Java 中一切都以对象的形式进行处理，在继承的机制中，创建一个子类对象，将包含一个父类子对象，这个对象与父类创建的对象是一样的。两者的区别在于后者来自外部，而前者来自子类对象的内部。当实例化子类对象时，父类对象也相应被实例化，换句话说，在实例化子类对象时，Java 编译器会在子类的构造方法中自动调用父类的无参构造方法。为了验证这个理论，来看下面的实例。

【例 10.2】 在项目中创建 Subroutine 类和两个父类，分别为 Parent 和 SubParent。这 3 个类的继承关系是 Subroutine 类继承 SubParent 类，而 SubParent 类继承 Parent 类。分别在这 3 个类的构造方法中输出构造方法名称，然后创建 Subroutine 类的实例对象，继承机制将使该类的父类对象自动初始化。**（实例位置：光盘\TM\sl\10.02）**

```
class Parent { //父类
    Parent() {
        System.out.println("调用父类的 Parent()构造方法");
    }
}
class SubParent extends Parent {                    //继承 Parent 类
    SubParent() {
        System.out.println("调用子类的 SubParent()构造方法");
    }
}
public class Subroutine extends SubParent {         //继承 SubParent 类
    Subroutine() {
        System.out.println("调用子类的 Subroutine()构造方法");
    }
    public static void main(String[] args) {
        Subroutine s = new Subroutine();             //实例化子类对象
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 10.3 所示。

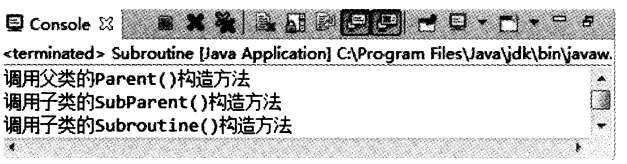

图 10.3　实例化子类对象自动调用父类构造方法

从本实例的运行结果可以看出，在子类 Subroutine 的主方法中只调用子类的构造方法实例化子类对象，并且在子类构造方法中没有调用父类构造方法的任何语句，但是在实例化子类对象时它相应调用了父类的构造方法。在结果中可以看到调用构造方法的顺序，首先是顶级父类，然后是上一级父类，最后是子类。也就是说，实例化子类对象时首先要实例化父类对象，然后再实例化子类对象，所以在子类构造方法访问父类的构造方法之前，父类已经完成实例化操作。

说明

在实例化子类对象时，父类无参构造方法将被自动调用，但有参构造方法并不能被自动调用，只能依赖于 super 关键字显式地调用父类的构造方法。

技巧

如果使用 finalize()方法对对象进行清理，需要确保子类的 finalize()方法的最后一个动作是调用父类的 finalize()方法，以保证当垃圾回收对象占用内存时，对象的所有部分都能被正常终止。

10.2 Object 类

视频讲解：光盘\TM\lx\10\Object 类.exe

在开始学习使用 class 关键字定义类时，就应用到了继承原理，因为在 Java 中，所有的类都直接或间接继承了 java.lang.Object 类。Object 类是比较特殊的类，它是所有类的父类，是 Java 类层中的最高层类。当创建一个类时，总是在继承，除非某个类已经指定要从其他类继承，否则它就是从 java.lang.Object 类继承而来的，可见 Java 中的每个类都源于 java.lang.Object 类，如 String、Integer 等类都是继承于 Object 类；除此之外，自定义的类也都继承于 Object 类。由于所有类都是 Object 子类，所以在定义类时，省略了 extends Object 关键字，如图 10.4 所示便描述了这一原则。

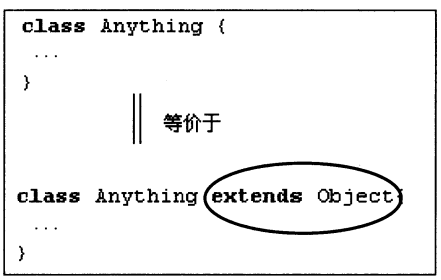

图 10.4 定义类时可以省略 extends Object 关键字

在 Object 类中主要包括 clone()、finalize()、equals()、toString()等方法，其中常用的两个方法为 equals()和 toString()方法。由于所有的类都是 Object 类的子类，所以任何类都可以重写 Object 类中的方法。

注意

Object 类中的 getClass()、notify()、notifyAll()、wait()等方法不能被重写，因为这些方法被定义为 final 类型。

下面详细讲述 Object 类中的几个重要方法。

1．getClass()方法

getClass()方法是 Object 类定义的方法，它会返回对象执行时的 Class 实例，然后使用此实例调用 getName()方法可以取得类的名称。

语法如下：

```
getClass().getname();
```

可以将 getClass()方法与 toString()方法联合使用。

2．toString()方法

toString()方法的功能是将一个对象返回为字符串形式，它会返回一个 String 实例。在实际的应用中通常重写 toString()方法，为对象提供一个特定的输出模式。当这个类转换为字符串或与字符串连接时，将自动调用重写的 toString()方法。

【例 10.3】 在项目中创建 ObjectInstance 类，在类中重写 Object 类的 toString()方法，并在主方法中输出该类的实例对象。（**实例位置：光盘\TM\sl\10.03**）

```
public class ObjectInstance {
    public String toString() {                          //重写 toString()方法
        return "在" + getClass().getName() + "类中重写 toString()方法";
    }
    public static void main(String[] args) {
        System.out.println(new ObjectInstance());       //打印本类对象
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 10.5 所示。

图 10.5　在 ObjectInstance 类中重写 toString()方法

在本实例中重写父类 Object 的 toString()方法，在子类的 toString()方法中使用 Object 类中的 getClass()方法获取当前运行的类名，定义一段输出字符串，当用户打印 ObjectInstance 类对象时，将自动调用 toString()方法。

3．equals()方法

第 7 章中曾讲解过 equals()方法，当时是比较“==”运算符与 equals()方法，说明“==”比较的是两个对象的引用是否相等，而 equals()方法比较的是两个对象的实际内容。带着这样一个理论来看下面的实例。

【例 10.4】 在项目中创建 OverWriteEquals 类，在类的主方法中定义两个字符串对象，调用 equals()方法判断两个字符串对象是否相等。（**实例位置：光盘\TM\sl\10.04**）

```
class V {                                          //自定义类 V
}
public class OverWriteEquals {
    public static void main(String[] args) {
        String s1 = "123";                         //实例化两个对象，内容相同
        String s2 = "123";
        System.out.println(s1.equals(s2));         //使用 equals()方法调用
        V v1 = new V();                            //实例化两个 V 类对象
        V v2 = new V();
        System.out.println(v1.equals(v2));         //使用 equals()方法比较 v1 与 v2 对象
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 10.6 所示。

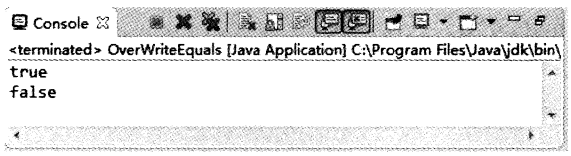

图 10.6　使用 equals()方法比较两个对象

从本实例的结果中可以看出，在自定义的类中使用 equals()方法进行比较时，将返回 false，因为 equals()方法的默认实现是使用“==”运算符比较两个对象的引用地址，而不是比较对象的内容，所以要想真正做到比较两个对象的内容，需要在自定义类中重写 equals()方法。

10.3　对象类型的转换

对象类型的转换在 Java 编程中经常遇到，主要包括向上转型与向下转型操作。本节将详细讲解对象类型转换的内容。

10.3.1　向上转型

视频讲解：光盘\TM\lx\10\向上转型.exe

因为平行四边形是特殊的四边形，也就是说平行四边形是四边形的一种，那么就可以将平行四边形对象看作是一个四边形对象。例如，鸡是家禽的一种，而家禽是动物中的一类，那么也可以将鸡对象看作是一个动物对象。可以使用例 10.5 所示的代码表示平行四边形与四边形的关系。

【例 10.5】 在项目中创建 Parallelogram 类，再创建 Quadrangle 类，并使 Parallelogram 类继承 Quadrangle 类，然后在主方法中调用父类的 draw()方法。（**实例位置：光盘\TM\sl\10.05**）

```
class Quadrangle {                                 //四边形类
    public static void draw(Quadrangle q) {        //四边形类中的方法
        //SomeSentence
```

```
    }
}
public class Parallelogram extends Quadrangle {          //平行四边形类，继承了四边形类
    public static void main(String args[]) {
        Parallelogram p = new Parallelogram();            //实例化平行四边形类对象引用
        draw(p);                                          //调用父类方法
    }
}
```

在例 10.5 中，平行四边形类继承了四边形类，四边形类存在一个 draw()方法，它的参数是 Quadrangle（四边形类）类型，而在平行四边形类的主方法中调用 draw()时给予的参数类型却是 Parallelogram（平行四边形类）类型的。这里一直在强调一个问题，就是平行四边形也是一种类型的四边形，所以可以将平行四边形类的对象看作是一个四边形类的对象，这就相当于“Quadrangle obj = new Parallelogram();”，就是把子类对象赋值给父类类型的变量，这种技术被称为“向上转型”。试想一下正方形类对象可以作为 draw()方法的参数，梯形类对象同样也可以作为 draw()方法的参数，如果在四边形类的 draw()方法中根据不同的图形对象设置不同的处理，就可以做到在父类中定义一个方法完成各个子类的功能，这样可以使同一份代码毫无差别地运用到不同类型之上，这就是多态机制的基本思想（在 10.6 节中将对多态进行详细介绍）。

图 10.7 中演示了平行四边形类继承四边形类的关系。

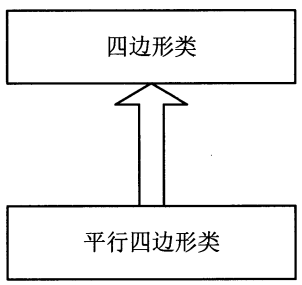

图 10.7　平行四边形类与四边形类的关系

从图 10.7 中可以看出，平行四边形类继承了四边形类，常规的继承图都是将顶级类设置在页面的顶部，然后逐渐向下，所以将子类对象看作是父类对象被称为“向上转型”。由于向上转型是从一个较具体的类到较抽象的类的转换，所以它总是安全的，如可以说平行四边形是特殊的四边形，但不能说四边形是平行四边形。

10.3.2　向下转型

视频讲解：光盘\TM\lx\10\向下转型.exe

通过向上转型可以推理出向下转型是将较抽象类转换为较具体的类。这样的转型通常会出现问题，例如，不能说四边形是平行四边形的一种、所有的鸟都是鸽子，因为这非常不合乎逻辑。可以说子类对象总是父类的一个实例，但父类对象不一定是子类的实例。如果修改例 10.5，将四边形类对象赋予平行四边形类对象，来看一下在程序中如何处理这种情况。

【例 10.6】 修改例 10.5，在 Parallelogram 类的主方法中将父类 Quadrangle 的对象赋值给子类

Parallelogram 的对象的引用变量将使程序产生错误。

```
class Quadrangle {
    public static void draw(Quadrangle q) {
        //SomeSentence
    }
}
public class Parallelogram extends Quadrangle {
    public static void main(String args[]) {
        draw(new Parallelogram());
        //将平行四边形类对象看作是四边形对象，称为向上转型操作
        Quadrangle q = new Parallelogram();
        //Parallelogram p=q;
        //将父类对象赋予子类对象，这种写法是错误的
        //将父类对象赋予子类对象，并强制转换为子类型，这种写法是正确的
        Parallelogram p = (Parallelogram) q;
    }
}
```

从例 10.6 中可以看到，如果将父类对象直接赋予子类，会发生编译器错误，因为父类对象不一定是子类的实例。例如，一个四边形不一定就是指平行四边形，它也许是梯形，也许是正方形，也许是其他带有四条边的不规则图形，图 10.8 表明了这些图形的关系。

从图 10.8 中可以看出，越是具体的对象具有的特性越多，越抽象的对象具有的特性越少。在做向下转型操作时，将特性范围小的对象转换为特性范围大的对象肯定会出现问题，所以这时需要告知编译器这个四边形就是平行四边形。将父类对象强制转换为某个子类对象，这种方式称为显式类型转换。

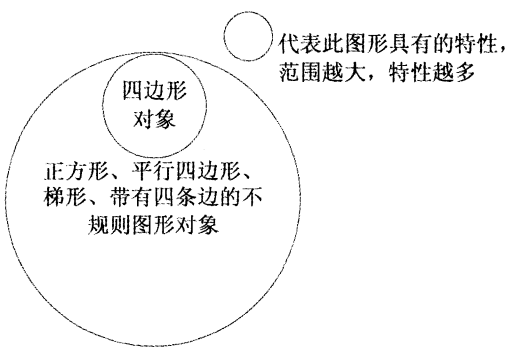

图 10.8　四边形与具体的四边形的关系

当在程序中使用向下转型技术时，必须使用显式类型转换，向编译器指明将父类对象转换为哪一种类型的子类对象。

10.4　使用 instanceof 操作符判断对象类型

视频讲解：光盘\TM\lx\10\使用 instanceof 操作符判断对象类型.exe

当在程序中执行向下转型操作时，如果父类对象不是子类对象的实例，就会发生 ClassCastException

异常，所以在执行向下转型之前需要养成一个良好的习惯，就是判断父类对象是否为子类对象的实例。这个判断通常使用 instanceof 操作符来完成。可以使用 instanceof 操作符判断是否一个类实现了某个接口（接口会在 10.6 节中进行介绍），也可以用它来判断一个实例对象是否属于一个类。

instanceof 的语法格式如下：

```
myobject instanceof ExampleClass
```

☑ myobject：某类的对象引用。

☑ ExampleClass：某个类。

使用 instanceof 操作符的表达式返回值为布尔值。如果返回值为 true，说明 myobject 对象为 ExampleClass 的实例对象；如果返回值为 false，说明 myobject 对象不是 ExampleClass 的实例对象。

注意

instanceof 是 Java 语言的关键字，在 Java 语言中的关键字都为小写。

下面来看一个向下转型与 instanceof 操作符结合的例子。

【例 10.7】 在项目中创建 Parallelogram 类和 3 个内部类 Quadrangle、Square、Anything。其中 Parallelogram 类和 Square 类继承 Quadrangle 类，在 Parallelogram 类的主方法中分别创建这些类的对象，然后使用 instanceof 操作符判断它们的类型并输出结果。（**实例位置：光盘\TM\sl\10.06**）

```
class Quadrangle {
    public static void draw(Quadrangle q) {
        //SomeSentence
    }
}
class Square extends Quadrangle {
    //SomeSentence
}
class Anything {
    //SomeSentence
}
public class Parallelogram extends Quadrangle {
    public static void main(String args[]) {
        Quadrangle q = new Quadrangle();                    //实例化父类对象
        //判断父类对象是否为 Parallelogram 子类的一个实例
        if (q instanceof Parallelogram) {
            Parallelogram p = (Parallelogram) q;            //向下转型操作
        }
        //判断父类对象是否为 Square 子类的一个实例
        if (q instanceof Square) {
            Square s = (Square) q;                          //进行向下转型操作
        }
        //由于 q 对象不为 Anything 类的对象，所以这条语句是错误的
        //System.out.println(q instanceof Anything);
    }
}
```

在本实例中将 instanceof 操作符与向下转型操作结合使用。在程序中定义了两个子类，即平行四边形类和正方形类，这两个类分别继承四边形类。在主方法中首先创建四边形类对象，然后使用 instanceof 操作符判断四边形类对象是否为平行四边形类的一个实例，是否为正方形类的一个实例，如果判断结果为 true，将进行向下转型操作。

10.5 方法的重载

视频讲解：光盘\TM\lx\10\方法的重载.exe

在第 7 章中曾学习过构造方法，知道构造方法的名称已经由类名决定，所以构造方法只有一个名称，但如果希望以不同的方式来实例化对象，就需要使用多个构造方法来完成。由于这些构造方法都需要根据类名进行命名，为了让方法名相同而形参不同的构造方法同时存在，必须用到“方法重载”。虽然方法重载起源于构造方法，但是它也可以应用到其他方法中。本节将讲述方法的重载。

方法的重载就是在同一个类中允许同时存在一个以上的同名方法，只要这些方法的参数个数或类型不同即可。为了更好地解释重载，来看下面的实例。

【例 10.8】 在项目中创建 OverLoadTest 类，在类中编写 add()方法的多个重载形式，然后在主方法中分别输出这些方法的返回值。（**实例位置：光盘\TM\sl\10.07**）

```
public class OverLoadTest {
    public static int add(int a, int b) {          //定义一个方法
        return a + b;
    }
    //定义与第一个方法相同名称、参数类型不同的方法
    public static double add(double a, double b) {
        return a + b;
    }
    public static int add(int a) {                  //定义与第一个方法参数个数不同的方法
        return a;
    }
    public static int add(int a, double b) {    //定义一个成员方法
        return 1;
    }
    //这个方法与前一个方法参数次序不同
    public static int add(double a, int b) {
        return 1;
    }
    public static void main(String args[]) {
        System.out.println("调用 add(int,int)方法：" + add(1, 2));
        System.out.println("调用 add(double,double)方法：" + add(2.1, 3.3));
        System.out.println("调用 add(int)方法：" + add(1));
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 10.9 所示。

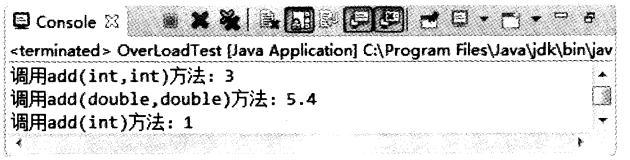

图 10.9　方法的重载

在本实例中分别定义了 5 个方法，在这 5 个方法中，前两个方法的参数类型不同，并且方法的返回值类型也不同，所以这两个方法构成重载关系；前两个方法与第 3 个方法相比，第 3 个方法的参数个数少于前两个方法，所以这 3 个方法也构成了重载关系；最后两个方法相比，发现除了参数的出现顺序不同之外，其他都相同，这样同样可以根据这个区别将两个方法构成重载关系。图 10.10 表明了所有可以构成重载的条件。

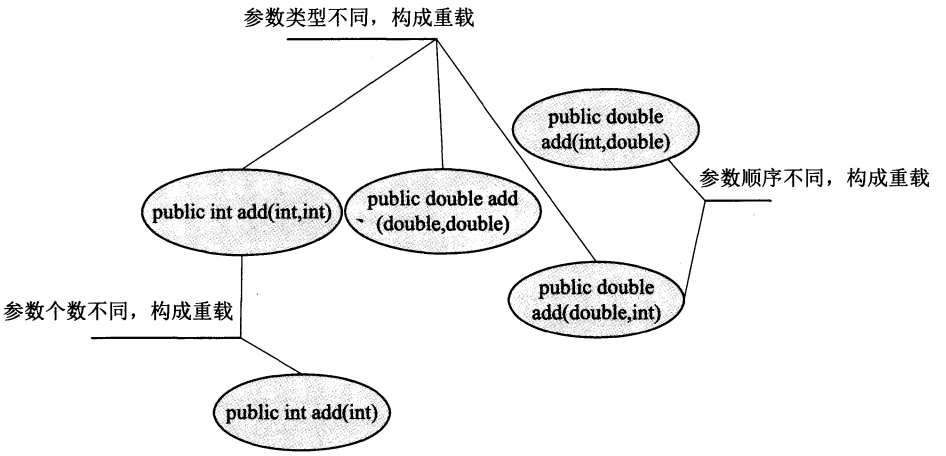

图 10.10　构成方法重载的条件

注意

虽然在方法重载中可以使两个方法的返回类型不同，但只有返回类型不同并不足以区分两个方法的重载，还需要通过参数的个数以及参数的类型来设置。

根据图 10.10 所示的构成方法重载的条件，可以总结出编译器是利用方法名、方法各参数类型和参数的个数以及参数的顺序来确定类中的方法是否唯一。方法的重载使得方法以统一的名称被管理，使程序代码有条理。

在谈到参数个数可以确定两个方法是否具有重载关系时，会想到定义不定长参数方法。

【例 10.9】 修改例 10.8，在例 10.8 中添加如下方法。

```
public static int add(int...a){                    //定义不定长参数方法
    int s=0;
    for(int i=0;i<a.length;i++)
    s+=a[i];                                       //做参数累加操作
    return s;                                      //将结果返回
}
```

上述方法又是一个 add()重载方法，它与例 10.8 中方法的不同之处在于该方法为不定长参数方法。不定长方法的语法如下：

```
返回值 方法名(参数数据类型…参数名称)
```

在参数列表中使用“…”形式定义不定长参数，其实这个不定长参数 a 就是一个数组，编译器会将(int…a)这种形式看作是(int[]a)，所以在 add()方法体做累加操作时使用到了 for 循环语句，在循环中是根据数组 a 的长度作为循环条件的，最后将累加结果返回。

如果将上述代码放在例 10.8 中，关键代码如例 10.10。

【例 10.10】 在项目中创建 OverLoadTest2 类，在类中编写 add()方法的多种重载形式，并编写该方法的不定长参数形式。然后在主方法中调用这些重载方法，并输出返回值。（**实例位置：光盘\TM\sl\10.08**）

```
public class OverLoadTest2 {
    public static int add(int a, int b) {
        return a + b;
    }
    public static double add(double a, double b) {
        return a + b;
    }
    public static int add(int a) {
        return 1;
    }
    public static int add(int a, double b) {
        return 1;
    }
    public static int add(double a, int b) {
        return 1;
    }
    public static int add(int... a) {        //定义不定长参数方法
        int s = 0;
        for (int i = 0; i < a.length; i++)
            //根据参数个数做循环操作
            s += a[i];                       //将每个参数累加
        return s;                            //将计算结果返回
    }
    public static void main(String args[]) {
        System.out.println("调用 add(int,int)方法：" + add(1, 2));
        System.out.println("调用 add(double,double)方法：" + add(2.1, 3.3));
        System.out.println("调用 add(int)方法：" + add(1));
        //调用不定长参数方法
        System.out.println("调用不定长参数方法：" + add(1,2, 3,4, 5,6, 7, 8, 9));
        System.out.println("调用不定长参数方法：" + add(1));
    }
}
```

在 Eclipse 中运行例 10.10，运行结果如图 10.11 所示。

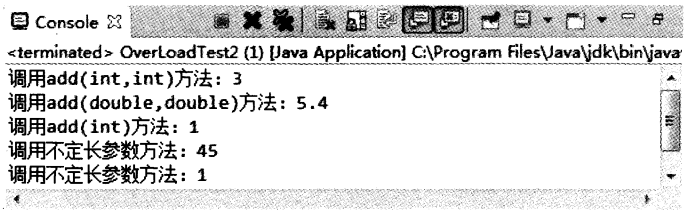

图 10.11　调用不定长参数方法

从例 10.10 中可以看出，定义不定长参数依然可以作为 add()方法的重载方法，由于它的参数是不定长的，所以满足根据参数个数区分重载的条件。

10.6　多　　态

视频讲解：光盘\TM\lx\10\多态.exe

利用多态可以使程序具有良好的扩展性，并可以对所有类对象进行通用的处理。在 10.3 节中已经学习过对象可以作为父类的对象实例使用，这种将子类对象视为父类对象的做法称为“向上转型”。假如现在需要绘制一个平行四边形，这时可以在平行四边形类中定义一个 draw()方法，具体实现代码如例 10.11 所示。

【例 10.11】 定义一个平行四边形的类 Parallelogram，在类中定义一个 draw()方法。

```
public class Parallelogram {
    //实例化保存平行四边形对象的数组对象
    public void draw(Parallelogram p){//定义 draw()方法，参数为本类对象
        ...//绘图语句
    }
}
```

如果需要定义一个绘制正方形的方法，通过定义一个正方形类来处理正方形对象，会出现代码冗余的缺点；通过定义一个正方形和平行四边形的综合类，分别处理正方形和平行四边形对象，也没有太大意义。

如果定义一个四边形类，让它处理所有继承该类的对象，根据“向上转型”原则可以使每个继承四边形类的对象作为 draw()方法的参数，然后在 draw()方法中作一些限定就可以根据不同图形类对象绘制相应的图形，从而以更为通用的四边形类来取代具体的正方形类和平行四边形类。这样处理能够很好地解决代码冗余问题，同时也易于维护，因为可以加入任何继承父类的子类对象，而父类方法也无须修改。

创建四边形类的具体实现代码如例 10.12 所示。

【例 10.12】 创建 Quadrangle 类，再分别创建两个内部类 Square 和 Parallelogramgle，它们都继承了 Quadrangle 类。编写 draw()方法，该方法接收 Quadrangle 类的对象作为参数，即使用这两个内部类的父类作为方法参数。在主方法中分别以两个内部类的实例对象作为参数执行 draw()方法。（**实例位置：光盘\TM\sl\10.09**）

```
public class Quadrangle {
    //实例化保存四边形对象的数组对象
    private Quadrangle[] qtest = new Quadrangle[6];
    private int nextIndex = 0;
    public void draw(Quadrangle q) {                    //定义 draw()方法，参数为四边形对象
        if (nextIndex < qtest.length) {
            qtest[nextIndex] = q;
            System.out.println(nextIndex);
            nextIndex++;
        }
    }
    public static void main(String[] args) {
        //实例化两个四边形对象，用于调用 draw()方法
        Quadrangle q = new Quadrangle();
        q.draw(new Square());                           //以正方形对象为参数调用 draw()方法
        q.draw(new Parallelogramgle());                 //以平行四边形对象为参数调用 draw()方法
    }
}
//定义一个正方形类，继承四边形类
class Square extends Quadrangle {
    public Square() {
        System.out.println("正方形");
    }
}
//定义一个平行四边形类，继承四边形类
class Parallelogramgle extends Quadrangle {
    public Parallelogramgle() {
        System.out.println("平行四边形");
    }
}
```

运行 Quadrangle 类，结果如图 10.12 所示。

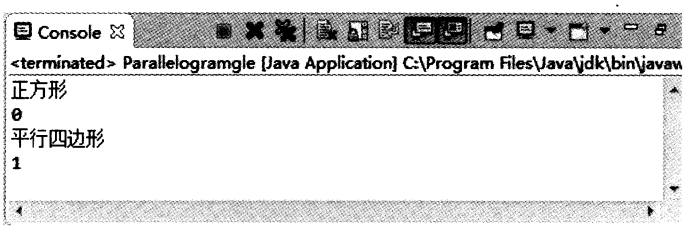

图 10.12　多态的实现

从本实例的运行结果中可以看出，以不同类对象为参数调用 draw()方法可以处理不同的图形问题。使用多态节省了开发和维护时间，因为程序员无须在所有的子类中定义执行相同功能的方法，避免了大量重复代码的开发，同时只要实例化一个继承父类的子类对象即可调用相应的方法，这里只要维护父类中的这个方法即可。

10.7　抽象类与接口

通常可以说四边形具有 4 条边，或者更具体一点，平行四边形是具有对边平行且相等特性的特殊四边形，等腰三角形是其中两条边相等的三角形，这些描述都是合乎情理的，但对于图形对象却不能使用具体的语言进行描述，它有几条边，究竟是什么图形，没有人能说清楚，这种类在 Java 中被定义为抽象类。

10.7.1　抽象类

视频讲解：光盘\TM\lx\10\抽象类.exe

在解决实际问题时，一般将父类定义为抽象类，需要使用这个父类进行继承与多态处理。回想继承和多态原理，继承树中越是在上方的类越抽象，如鸽子类继承鸟类、鸟类继承动物类等。在多态机制中，并不需要将父类初始化对象，我们需要的只是子类对象，所以在 Java 语言中设置抽象类不可以实例化对象，因为图形类不能抽象出任何一种具体图形，但它的子类却可以。

抽象类的语法如下：

```
public abstract class Test {
    abstract void testAbstract(); //定义抽象方法
}
```

其中，abstract 是定义抽象类的关键字。

使用 abstract 关键字定义的类称为抽象类，而使用这个关键字定义的方法称为抽象方法。抽象方法没有方法体，这个方法本身没有任何意义，除非它被重写，而承载这个抽象方法的抽象类必须被继承，实际上抽象类除了被继承之外没有任何意义。

反过来讲，如果声明一个抽象的方法，就必须将承载这个抽象方法的类定义为抽象类，不可能在非抽象类中获取抽象方法。换句话说，只要类中有一个抽象方法，此类就被标记为抽象类。

抽象类被继承后需要实现其中所有的抽象方法，也就是保证相同的方法名称、参数列表和相同返回值类型创建出非抽象方法，当然也可以是抽象方法。图 10.13 说明了抽象类的继承关系。

从图 10.13 中可以看出，继承抽象类的所有子类需要将抽象类中的抽象方法进行覆盖。这样在多态机制中，就可以将父类修改为抽象类，将 draw()方法设置为抽象方法，然后每个子类都重写这个方法来处理。但这又会出现我们刚探讨多态时讨论的问题，程序中会有太多冗余的代码，同时这样的父类局限性很大，也许某个不需要 draw()方法的子类也不得不重写 draw()方法。如果将 draw()方法放置在另外一个类中，这样让那些需要 draw()方法的类继承该类，而不需要 draw()方法的类继承图形类，但所有的子类都需要图形类，因为这些类是从图形类中被导出的，同时某些类还需要 draw()方法，但是在 Java 中规定，类不能同时继承多个父类，面临这种问题，接口的概念便出现了。

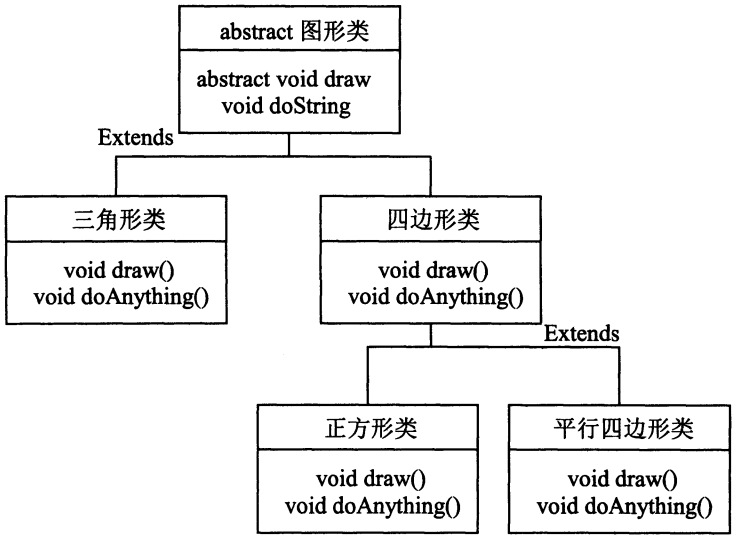

图 10.13　抽象类继承关系

10.7.2　接口

视频讲解：光盘\TM\lx\10\接口.exe

1. 接口简介

接口是抽象类的延伸，可以将它看作是纯粹的抽象类，接口中的所有方法都没有方法体。对于 10.7.1 小节中遗留的问题，可以将 draw()方法封装到一个接口中，使需要 draw()方法的类实现这个接口，同时也继承图形类，这就是接口存在的必要性。在图 10.14 中描述了各个子类继承图形类后使用接口的关系。

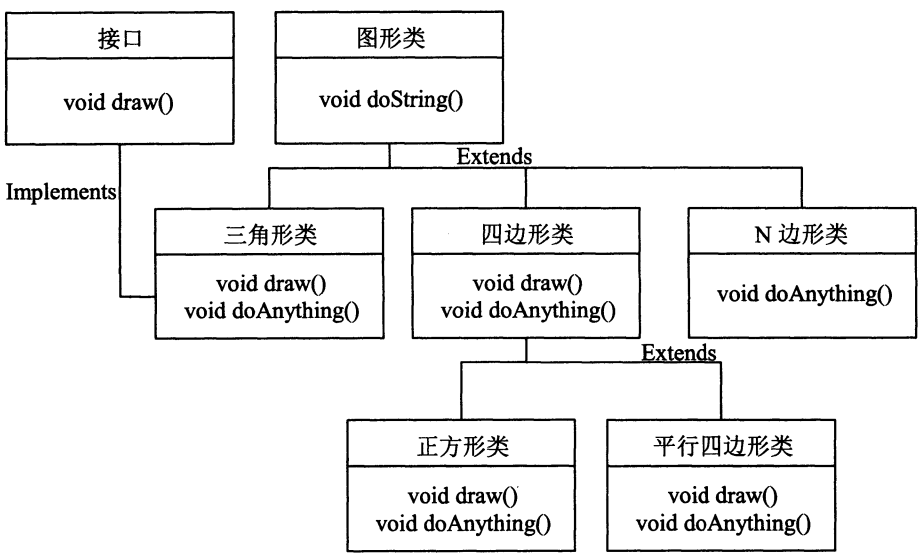

图 10.14　使用接口继承关系

接口使用 interface 关键字进行定义，其语法如下：

```
public interface drawTest {
    void draw(); //接口内的方法，省略 abstract 关键字
}
```

☑ public：接口可以像类一样被权限修饰符修饰，但 public 关键字仅限用于接口在与其同名的文件中被定义。

☑ interface：定义接口关键字。

☑ drawTest：接口名称。

一个类实现一个接口可以使用 implements 关键字，代码如下：

```
public class Parallelogram extends Quadrangle implements drawTest{
    ···//
}
```

注意

在接口中定义的方法必须被定义为 public 或 abstract 形式，其他修饰权限不被 Java 编译器认可，即使不将该方法声明为 public 形式，它也是 public。

说明

在接口中定义的任何字段都自动是 static 和 final 的。

下面将修改例 10.12，将多态技术与接口相结合，如例 10.13 所示。

【例 10.13】在项目中创建 QuadrangleUseInterface 类，在类中创建两个继承该类的内部类 ParallelogramgleUseInterface 和 SquareUseInterface；再创建 drawTest 接口，并使前两个内部类实现该接口；然后在主方法中分别调用这两个内部类的 draw()方法。（**实例位置：光盘\TM\sl\10.10**）

```
interface drawTest {                    //定义接口
    public void draw();                 //定义方法
}
//定义平行四边形类，该类继承了四边形类，并实现了 drawTest 接口
class ParallelogramgleUseInterface extends QuadrangleUseInterface
        implements drawTest {
    public void draw() {                //由于该类实现了接口，所以需要覆盖 draw()方法
        System.out.println("平行四边形.draw()");
    }

    void doAnyThing() {                 //覆盖父类方法
        //SomeSentence
    }
}
class SquareUseInterface extends QuadrangleUseInterface implements
        drawTest {
```

```
    public void draw() {
        System.out.println("正方形.draw()");
    }
    void doAnyThing() {
        //SomeSentence
    }
}
class AnyThingUseInterface extends QuadrangleUseInterface {
    void doAnyThing() {
    }
}
public class QuadrangleUseInterface {   //定义四边形类
    public void doAnyTthing() {
        //SomeSentence
    }

    public static void main(String[] args) {
        drawTest[] d = {                    //接口也可以进行向上转型操作
        new SquareUseInterface(), new ParallelogramgleUseInterface() };
        for (int i = 0; i < d.length; i++) {
            d[i].draw();                    //调用 draw()方法
        }
    }
}
```

在 Eclipse 中运行 QuadrangleUseInterface 类，运行结果如图 10.15 所示。

图 10.15　多态与接口结合

在本实例中，正方形类与平行四边形类分别实现了 drawTest 接口并继承了四边形类，所以需要覆盖接口中的方法。在调用 draw()方法时，首先将平行四边形类对象与正方形类对象向上转型为 drawTest 接口形式。这里也许很多读者会有疑问，接口是否可以向上转型？其实在 Java 中无论是将一个类向上转型为父类对象，还是向上转型为抽象父类对象，或者向上转型为该类实现接口，都是没有问题的。然后使用 d[i]数组中的每一个对象调用 draw()，由于向上转型，所以 d[i]数组中的每一个对象分别代表正方形类对象与平行四边形类对象，最后结果分别调用正方形类与平行四边形类中覆盖的 draw()方法。

2．接口与继承

我们知道在 Java 中不允许多重继承，但使用接口就可以实现多重继承，因为一个类可以同时实现多个接口，这样可以将所有需要继承的接口放置在 implements 关键字后并使用逗号隔开，但这可能会在一个类中产生庞大的代码量，因为继承一个接口时需要实现接口中所有的方法。

多重继承的语法如下：

```
class 类名 implements 接口 1,接口 2,…,接口 n
```

【例 10.14】 在定义一个接口时使该接口继承另外一个接口。

```
interface intf1 {
}
interface intf2 extends intf1 {
}
```

10.8 小　　结

通过对本章的学习，读者可以了解继承与多态的机制，掌握重载、类型转换等技术，学会使用接口与抽象类，从而对继承和多态有一个比较深入的了解。尽管读者已经学习过本章，但还是建议初学者仔细揣摩继承与多态机制，因为继承和多态本身是比较抽象的概念，深入理解需要一段时间，使用多态机制必须扩展自己的编程视野，将编程的着眼点放在类与类之间的共同特性以及关系上，使软件开发具有更快的速度、更完善的代码组织架构，以及更好的扩展性和维护性。

10.9 实践与练习

1．创建一个抽象类，验证它是否可以实例化对象。（**答案位置：光盘\TM\sl\10.11**）

2．尝试创建一个父类，在父类中创建两个方法，在子类中覆盖第二个方法，为子类创建一个对象，将它向上转型到基类并调用这个方法。（**答案位置：光盘\TM\sl\10.12**）

3．尝试创建一个父类和子类，分别创建构造方法，然后向父类和子类添加成员变量和方法，并总结构建子类对象时的顺序。（**答案位置：光盘\TM\sl\10.13**）

第11章

类的高级特性

（视频讲解：23 分钟）

类除了具有普通的特性之外，还具有一些高级特性，如包、内部类等。包在整个管理过程中起到了非常重要的作用，使用包可以有效地管理繁杂的类文件，解决类重名的问题，当在类中配合包与权限修饰符使用时，可以控制其他人对类成员的访问。同时在 Java 中一个更为有效的隐藏实现细节的技巧是使用内部类，通过使用内部类机制可以向上转型为被内部类实现的公共接口。由于在类中可以定义多个内部类，实现接口的方式也不止一个，只要将内部类中的方法设置为类最小范围的修饰权限即可将内部类的实现细节有效地隐藏。

通过阅读本章，您可以：

- 掌握包的创建规则
- 掌握在程序中导入其他类包
- 掌握 final 变量、方法、类
- 掌握内部类

11.1　Java 类包

在 Java 中每定义好一个类，通过 Java 编译器进行编译之后，都会生成一个扩展名为.class 的文件，当这个程序的规模逐渐庞大时，就很容易发生类名称冲突的现象。那么 JDK API 中提供了成千上万具有各种功能的类，又是如何管理的呢？Java 中提供了一种管理类文件的机制，就是类包。

11.1.1　类名冲突

视频讲解：光盘\TM\lx\11\类名冲突.exe

Java 中每个接口或类都来自不同的类包，无论是 Java API 中的类与接口还是自定义的类与接口都需要隶属于某一个类包，这个类包包含了一些类和接口。如果没有包的存在，管理程序中的类名称将是一件非常麻烦的事情。如果程序只由一个类定义组成，并不会给程序带来什么影响，但是随着程序代码的增多，难免会出现类同名的问题。例如，在程序中定义一个 Login 类，因业务需要，还要定义一个名称为 Login 的类，但是这两个类所实现的功能完全不同，于是问题就产生了，编译器不会允许存在同名的类文件。解决这类问题的办法是将这两个类放置在不同的类包中。

11.1.2　完整的类路径

视频讲解：光盘\TM\lx\11\完整的类路径.exe

在第 9 章中曾经讲述过 Math 类，其实 Math 类并不是它的完整名称，就如同一个人的名称需要有名有姓一样，Math 类的完整名称如图 11.1 所示。

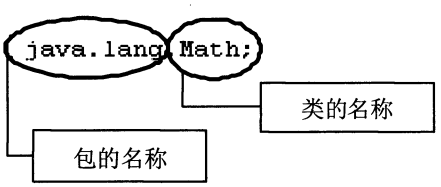

图 11.1　定义完整的类名

从图 11.1 中可以看出，一个完整的类名需要包名与类名的组合，每个类都隶属于一个类包，只要保证同一类包中的类不同名，就可以有效地避免同名类冲突的情况。例如，一个程序中同时使用到 java.util.Date 类与 java.sql.Date 类，如果在程序中不指定完整类路径，编译器不会知道这段代码使用的是 java.util 类包中的 Date 类还是 java.sql 类包中的 Date 类，所以需要在指定代码中给出完整的类路径。

【例 11.1】 在程序中使用两个不同 Date 类的完整类路径，可以使用如下代码：

```
java.util.Date date=new java.util.Date();
java.sql.Date date2=new java.sql.Date(233);
```

在 Java 中采用类包机制非常重要，类包不仅可以解决类名冲突问题，还可以在开发庞大的应用程序时，帮助开发人员管理庞大的应用程序组件，方便软件复用。下面来看一下在 Java 中如何创建类包（以下简称包）。

说明

同一个包中的类相互访问时，可以不指定包名。

注意

同一个包中的类不必存放在同一个位置，如 com.lzw.class1 和 com.lzw.class2 可以一个放置在 C 盘，一个放置在 D 盘，只要将 CLASSPATH 分别指向这两个位置即可。

11.1.3 创建包

视频讲解：光盘\TM\lx\11\创建包.exe

在 Eclipse 中创建包的步骤如下：

（1）在项目的 src 节点上右击，选择 New / Package 命令。

（2）弹出 New Java Package 对话框，在 Name 文本框中输入新建的包名，如 com.lzw，然后单击 Finish 按钮，如图 11.2 所示。

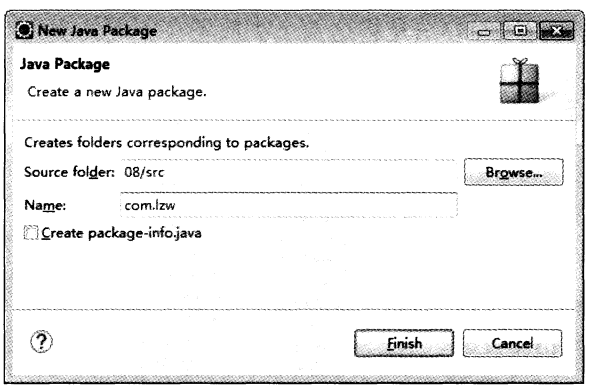

图 11.2 New Java Package 对话框

（3）在 Eclipse 中创建类时，可以在新建立的包上右击，选择 New 命令，这样新建的类会默认保存在该包中。另外也可以在 New Java Class 对话框中指定新建类所在的包。有关新建类的步骤，可参见 2.2.2 小节。

在 Java 中包名设计应与文件系统结构相对应，如一个包名为 com.lzw，那么该包中的类位于 com 文件夹下的 lzw 子文件夹下。没有定义包的类会被归纳在预设包（默认包）中。在实际开发中，应该为所有类设置包名，这是良好的编程习惯。

在类中定义包名的语法如下：

```
package 包名
```

在类中指定包名时需要将 package 表达式放置在程序的第一行，它必须是文件中的第一行非注释代码，当使用 package 关键字为类指定包名之后，包名将会成为类名中的一部分，预示着这个类必须指定

全名。例如，在使用位于 com.lzw 包下的 Dog.java 类时，需要使用形如 com.lzw.Dog 这样的表达式。

注意

Java 包的命名规则是全部使用小写字母。

在 11.1.1 小节中已经谈到类名的冲突问题，也许有的读者会产生疑问，如此之多的包不会产生包名冲突现象吗？这是有可能的。为了避免这样的问题，在 Java 中定义包名时通常使用创建者的 Internet 域名的反序，由于 Internet 域名是独一无二的，包名自然不会发生冲突。下面来看一个实例。

【例 11.2】 在项目中创建 Math 类，在创建类的对话框中指定包名为 com.lzw，并在主方法中输出说明该类并非 java.lang 包的 Math 类。（**实例位置：光盘\TM\sl\11.01**）

```
package com.lzw; //指定包名

public class Math {
    public static void main(String[] args) {
        System.out.println("不是 java.lang.Math 类，而是 com.lzw.Math 类");
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 11.3 所示。

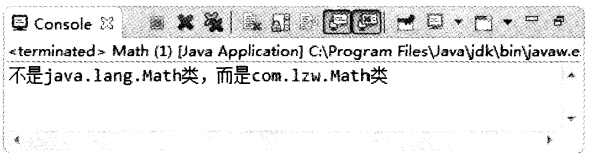

图 11.3　在程序中指定包名

在本实例中，在程序的第一行指定包名，同时在 com.lzw 包中定义 Math 类，读者不禁会想到 java.lang.Math 类，而本实例定义的为 com.lzw.Math 类，可以看出在不同包中定义相同类名也是没有问题的，所以在 Java 中使用包可以有效管理各种功能的类。

11.1.4　导入包

视频讲解：光盘\TM\lx\11\导入包.exe

1．使用 import 关键字导入包

如果某个类中需要使用 Math 类，那么如何告知编译器当前应该使用哪一个包中的 Math 类，是 java.lang.Math 类还是 com.lzw.Math 类？这是一个令人困扰的问题。此时，可以使用 Java 中的 import 关键字指定。例如，如果在程序中使用 import com.lzw 表达式，在程序中使用 Math 类时就会选择 com.lzw.Math 类来使用，当然也可以直接在程序中使用 Math 类时指定 com.lzw.Math 类。

import 关键字的语法如下：

```
import com.lzw.*;              //指定 com.lzw 包中的所有类在程序中都可以使用
```

```
import com.lzw.Math          //指定 com.lzw 包中的 Math 类在程序中可以使用
```

在使用 import 关键字时，可以指定类的完整描述，如果为了使用包中更多的类，可以在使用 import 关键字指定时在包指定后加上*，这表示可以在程序中使用包中的所有类。

注意

如果类定义中已经导入 com.lzw.Math 类，在类体中再使用其他包中的 Math 类时则必须指定完整的带有包格式的类名，如这种情况再使用 java.lang 包的 Math 类时就要使用全名格式 java.lang.Math。

在程序中添加 import 关键字时，就开始在 CLASSPATH 指定的目录中进行查找，查找子目录 com.lzw，然后从这个目录下编译完成的文件中查找是否有名称符合者，最后寻找到 Math.class 文件。另外，当使用 import 指定了一个包中的所有类时，并不会指定这个包的子包中的类，如果用到这个包中的子类，需要再次对子包作单独引用。

在 Java 中将 Java 源文件与类文件放在一起管理是极为不好的管理方式。可以在编译时使用-d 参数设置编译后类文件产生的位置。使用 DOS 进入程序所在的根目录下，执行如下命令：

```
javac -d ./bin/ ./com/lzw/*.java
```

这样编译成功后将在当前运行路径下的 bin 目录中产生 com/lzw 路径，并在该路径下出现相应源文件的类文件。如果使用 Eclipse 编译器，并在创建项目时设置了源文件与输出文件的路径，编译后的类文件会自动保存在输出文件的路径中。

说明

如果不能在程序所在的根目录下使用 javac.exe 命令，注意在 path 环境变量中设置 Java 编译器所在的位置，假如是 C:\Java\jdk1.6.0_03\bin，可以使用 set path=C:\Java\jdk1.6.0_03\bin;%path%命令在 DOS 中设置 path 环境变量。

2. 使用 import 导入静态成员

import 关键字除了导入包之外，还可以导入静态成员，这是 JDK 5.0 以上版本提供的新功能。导入静态成员可以使程序员编程更为方便。

使用 import 导入静态成员的语法如下：

```
import static 静态成员
```

为了使读者了解如何使用 import 关键字导入静态成员，来看下面的实例。

【例 11.3】 在项目中创建 ImportTest 类，在该类中使用 import 关键字导入静态成员。（**实例位置：光盘\TM\sl\11.02**）

```
package com.lzw;
```

```
import static java.lang.Math.max;                    //导入静态成员方法
import static java.lang.System.out;                  //导入静态成员变量

public class ImportTest {
    public static void main(String[] args) {
        //在主方法中可以直接使用这些静态成员
        out.println("1 和 4 的较大值为：" + max(1, 4));
    }
}
```

本实例在 Eclipse 中的运行结果如图 11.4 所示。

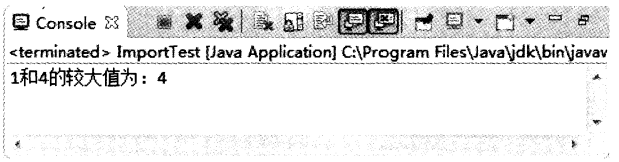

图 11.4　使用 import 关键字导入静态成员

从本实例中可以看出，分别使用 import static 导入了 java.lang.Math 类中的静态成员方法 max()和 java.lang.System 类中的 out 成员变量，这时就可以在程序中直接引用这些静态成员，如在主方法中的 out.println()表达式以及直接使用 max()方法。

11.2　final 变量

视频讲解：光盘\TM\lx\11\final 变量.exe

final 关键字可用于变量声明，一旦该变量被设定，就不可以再改变该变量的值。通常，由 final 定义的变量为常量。例如，在类中定义 PI 值，可以使用如下语句：

```
final double PI=3.14;
```

当在程序中使用到 PI 这个常量时，它的值就是 3.14，如果在程序中再次对定义为 final 的常量赋值，编译器将不会接受。

final 关键字定义的变量必须在声明时对其进行赋值操作。final 除了可以修饰基本数据类型的常量，还可以修饰对象引用。由于数组也可以被看作一个对象来引用，所以 final 可以修饰数组。一旦一个对象引用被修饰为 final 后，它只能恒定指向一个对象，无法将其改变以指向另一个对象。一个既是 static 又是 final 的字段只占据一段不能改变的存储空间。为了深入了解 final 关键字，来看下面的实例。

【例 11.4】 在项目的 com.lzw 包中创建 FinalData 类，在该类中创建 Test 内部类，并定义各种类型的 final 变量。（**实例位置：光盘\TM\sl\11.03**）

```
package com.lzw;
import static java.lang.System.out;
import java.util.Random;
class Test {
```

```
    int i = 0;
}
public class FinalData {
    static Random rand = new Random();
    private final int VALUE_1 = 9;                  //声明一个 final 常量
    private static final int VALUE_2 = 10;          //声明一个 final、static 常量
    private final Test test = new Test();           //声明一个 final 引用
    private Test test2 = new Test();                //声明一个不是 final 的引用
    private final int[] a = {1,2,3,4,5,6 };         //声明一个定义为 final 的数组
    private final int i4 = rand.nextInt(20);
    private static final int i5 = rand.nextInt(20);
    public String toString() {
        return i4 + " " + i5 + " ";
    }
    public static void main(String[] args) {
        FinalData data = new FinalData();
        data.test=new Test();
        //可以对指定为 final 的引用中的成员变量赋值
        //但不能将定义为 final 的引用指向其他引用
        data.value2++;
        //不能改变定义为 final 的常量值
        data.test2 = new Test();                    //可以将没有定义为 final 的引用指向其他引用
        for (int i = 0; i < data.a.length; i++) {
            a[i]=9;
            //不能对定义为 final 的数组赋值
        }
        out.println(data);
        out.println("data2");
        out.println(new FinalData());
        out.println(data);
    }
}
```

在本实例中，被定义为 final 的常量定义时需要使用大写字母命名，并且中间使用下划线进行连接，这是 Java 中的编码规则。同时，定义为 final 的数据无论是常量、对象引用还是数组，在主函数中都不可以被改变。

我们知道一个被定义为 final 的对象引用只能指向唯一一个对象，不可以将它再指向其他对象，但是一个对象本身的值却是可以改变的，那么为了使一个常量真正做到不可更改，可以将常量声明为 static final。为了验证这个理论，来看下面的实例。

【例 11.5】 在项目的 com.lzw 包中创建 FinalStaticData 类，在该类中创建 Random 类的对象，在主方法中分别输出类中定义的 final 变量 a1 与 a2。（**实例位置：光盘\TM\sl\11.04**）

```
package com.lzw;
import java.util.Random;
import static java.lang.System.out;
public class FinalStaticData {
    private static Random rand = new Random(); //实例化一个 Random 类对象
    //随机产生 0~10 之间的随机数赋予定义为 final 的 a1
```

```
    private final int a1 = rand.nextInt(10);
    //随机产生 0~10 之间的随机数赋予定义为 static final 的 a2
    private static final int a2 = rand.nextInt(10);
    public static void main(String[] args) {
        FinalStaticData fdata = new FinalStaticData(); //实例化一个对象
        //调用定义为 final 的 a1
        out.println("重新实例化对象调用 a1 的值：" + fdata.a1);
        //调用定义为 static final 的 a2
        out.println("重新实例化对象调用 a1 的值：" + fdata.a2);
        //实例化另外一个对象
        FinalStaticData fdata2 = new FinalStaticData();
        out.println("重新实例化对象调用 a1 的值：" + fdata2.a1);
        out.println("重新实例化对象调用 a2 的值：" + fdata2.a2);
    }
}
```

在 Eclipse 中运行上述实例，运行结果如图 11.5 所示。

从本实例的运行结果中可以看出，定义为 final 的常量不是恒定不变的，将随机数赋予定义为 final 的常量，可以做到每次运行程序时改变 a1 的值。但是 a2 与 a1 不同，由于它被声明为 static final 形式，所以在内存中为 a2 开辟了一个恒定不变的区域，当再次实例化一个 FinalStaticData 对象时，仍然指向 a2 这块内存区域，所以 a2 的值保持不变。a2 是在装载时被初始化，而不是每次创建新对象时都被初始化；而 a1 会在重新实例化对象时被更改。

技巧

在 Java 中定义全局常量，通常使用 public static final 修饰，这样的常量只能在定义时被赋值。

可以将方法的参数定义为 final 类型，这预示着无法在方法中更改参数引用所指向的对象。

最后总结一下在程序中 final 数据可以出现的位置。图 11.6 清晰地表明了在程序中哪些位置可以定义 final 数据。

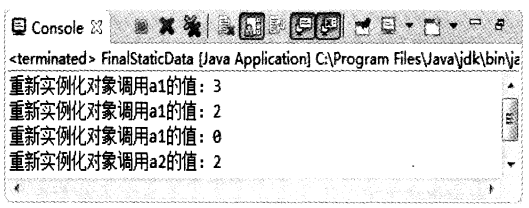

图 11.5　比较 static final 与 final 定义数据的区别

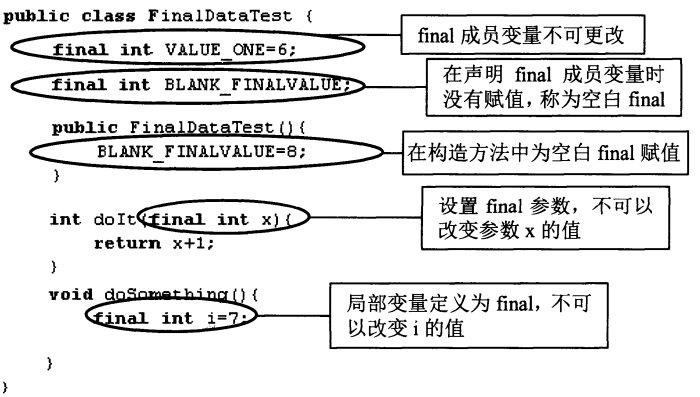

图 11.6　程序中可以定义为 final 的数据

11.3 final 方法

视频讲解：光盘\TM\lx\11\final 方法.exe

首先，读者应该了解定义为 final 的方法不能被重写。

将方法定义为 final 类型可以防止子类修改该类的定义与实现方式，同时定义为 final 的方法的执行效率要高于非 final 方法。在修饰权限中曾经提到过 private 修饰符，如果一个父类的某个方法被设置为 private 修饰符，子类将无法访问该方法，自然无法覆盖该方法，所以一个定义为 private 的方法隐式被指定为 final 类型，这样无须将一个定义为 private 的方法再定义为 final 类型。例如下面的语句：

```
private final void test(){
    …//省略一些程序代码
}
```

但是在父类中被定义为 private final 的方法似乎可以被子类覆盖，来看下面的实例。

【例 11.6】 在项目中创建 FinalMethod 类，在该类中创建 Parents 类和继承该类的 Sub 类，在主方法中分别调用这两个类中的方法，并查看 final 类型方法能否被覆盖。（**实例位置：光盘\TM\sl\11.05**）

```
class Parents {
    private final void doit() {
        System.out.println("父类.doit()");
    }
    final void doit2() {
        System.out.println("父类.doit2()");
    }
    public void doit3() {
        System.out.println("父类.doit3()");
    }
}
class Sub extends Parents {
    public final void doit() {          //在子类中定义一个 doit()方法
        System.out.print("子类.doit()");
    }
//  final void doit2(){                 //final 方法不能覆盖
//      System.out.println("子类.doit2()");
//  }
    public void doit3() {
        System.out.println("子类.doit3()");
    }
}
public class FinalMethod {
    public static void main(String[] args) {
        Sub s = new Sub();              //实例化
        s.doit();                       //调用 doit()方法
        Parents p = s;                  //执行向上转型操作
        //p.doit();                     //不能调用 private 方法
```

```
        p.doit2();
        p.doit3();
    }
}
```

在 Eclipse 中运行本实例，结果如图 11.7 所示。

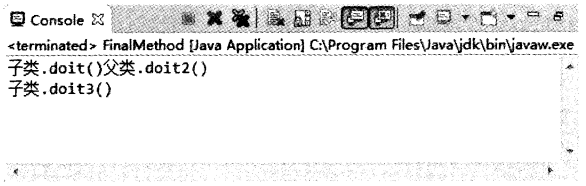

图 11.7　验证是否可以覆盖 private final 方法

从本实例中可以看出，final 方法不能被覆盖，例如，doit2()方法不能在子类中被重写，但是在父类中定义了一个 private final 的 doit()方法，同时在子类中也定义了一个 doit()方法，从表面上来看，子类中的 doit()方法覆盖了父类的 doit()方法，但是覆盖必须满足一个对象向上转型为它的基本类型并调用相同方法这样一个条件。例如，在主方法中使用“Parents p=s;”语句执行向上转型操作，对象 p 只能调用正常覆盖的 doit3()方法，却不能调用 doit()方法，可见子类中的 doit()方法并不是正常覆盖，而是生成一个新的方法。

11.4　final 类

视频讲解：光盘\TM\lx\11\final 类.exe

定义为 final 的类不能被继承。

如果希望一个类不允许任何类继承，并且不允许其他人对这个类进行任何改动，可以将这个类设置为 final 形式。

final 类的语法如下：

```
final 类名{}
```

如果将某个类设置为 final 形式，则类中的所有方法都被隐式设置为 final 形式，但是 final 类中的成员变量可以被定义为 final 或非 final 形式。

【例 11.7】 在项目中创建 FinalClass 类，在类中定义 doit()方法和变量 a，实现在主方法中操作变量 a 自增。（**实例位置：光盘\TM\sl\11.06**）

```
final class FinalClass {
    int a = 3;
    void doit() {
    }
    public static void main(String args[]) {
        FinalClass f = new FinalClass();
        f.a++;
        System.out.println(f.a);
```

```
    }
}
```

11.5 内 部 类

前面曾经学习过在一个文件中定义两个类，但其中任何一个类都不在另一个类的内部，而如果在类中再定义一个类，则将在类中再定义的那个类称为内部类。内部类可分为成员内部类、局部内部类以及匿名类。本节将具体进行介绍。

11.5.1 成员内部类

 **视频讲解：光盘\TM\lx\11\成员内部类.exe**

1. 成员内部类简介

在一个类中使用内部类，可以在内部类中直接存取其所在类的私有成员变量。本节首先介绍成员内部类。

成员内部类的语法如下：

```
public class OuterClass {                    //外部类
    private class InnerClass {               //内部类
        //…
    }
}
```

在内部类中可以随意使用外部类的成员方法以及成员变量，尽管这些类成员被修饰为 private。图 11.8 充分说明了内部类的使用，尽管成员变量 i 以及成员方法 f()都在外部类中被修饰为 private，但在内部类中可以直接使用外部类中的类成员。

内部类的实例一定要绑定在外部类的实例上，如果从外部类中初始化一个内部类对象，那么内部类对象就会绑定在外部类对象上。内部类初始化方式与其他类初始化方式相同，都是使用 new 关键字。下面来看一个实例。

【例 11.8】 在项目中创建 OuterClass 类，在类中定义 innerClass 内部类和 doit()方法，在主方法中创建 OuterClass 类的实例对象和 doit()方法。（**实例位置：光盘\TM\sl\11.07**）

```
public class OuterClass {
    innerClass in = new innerClass();    //在外部类实例化内部类对象引用
    public void ouf() {
        in.inf();                        //在外部类方法中调用内部类方法
    }
    class innerClass {
        innerClass() {                   //内部类构造方法
        }
```

```
        public void inf() {                    //内部类成员方法
        }
        int y = 0;                             //定义内部类成员变量
    }
    public innerClass doit() {                 //外部类方法，返回值为内部类引用
        //y=4;                                 //外部类不可以直接访问内部类成员变量
        in.y = 4;
        return new innerClass();               //返回内部类引用
    }
    public static void main(String args[]) {
        OuterClass out = new OuterClass();
        //内部类的对象实例化操作必须在外部类或外部类的非静态方法中实现
        OuterClass.innerClass in = out.doit();
        OuterClass.innerClass in2 = out.new innerClass();
    }
}
```

例 11.8 中的外部类创建内部类实例与其他类创建对象引用时相同。内部类可以访问它的外部类成员，但内部类的成员只有在内部类的范围之内是可知的，不能被外部类使用。例如，在例 11.8 中对内部类的成员变量 y 再次赋值时将会出错，但是可以使用内部类对象引用调用成员变量 y。图 11.9 说明了内部类 InnerClass 对象与外部类 OuterClass 对象的关系。

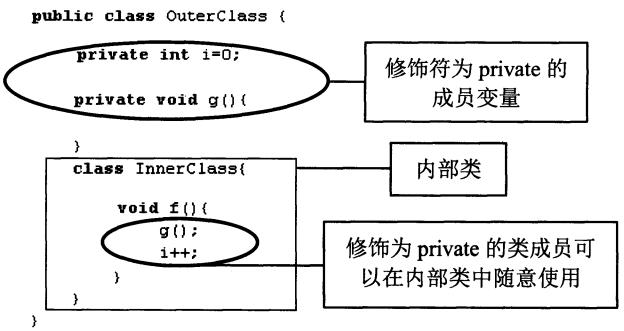

图 11.8　内部类可以使用外部类的成员

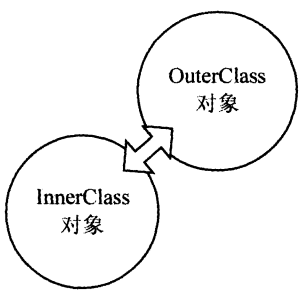

图 11.9　内部类对象与外部类对象的关系

从图 11.9 中可以看出，内部类对象与外部类对象关系非常紧密，内外可以交互使用彼此类中定义的变量。

注意

如果在外部类和非静态方法之外实例化内部类对象，需要使用外部类。内部类的形式指定该对象的类型。

在例 11.8 的主方法中如果不使用 doit()方法返回内部类对象引用，可以直接使用内部类实例化内部类对象，但由于是在主方法中实例化内部类对象，必须在 new 操作符之前提供一个外部类的引用。

【例 11.9】 在主方法中实例化一个内部类对象。

```
public static void main(String args[]){
    OuterClass out=new OuterClass();
    OuterClass.innerClass in=out.doit();
```

```
    OuterClass.innerClass in2=out.new innerClass();    //实例化内部类对象
}
```

从例 11.9 中可以看出，在实例化内部类对象时，不能在 new 操作符之前使用外部类名称实例化内部类对象，而是应该使用外部类的对象来创建其内部类的对象。

注意

内部类对象会依赖于外部类对象，除非已经存在一个外部类对象，否则类中不会出现内部类对象。

2．内部类向上转型为接口

如果将一个权限修饰符为 private 的内部类向上转型为其父类对象，或者直接向上转型为一个接口，在程序中就可以完全隐藏内部类的具体实现过程。可以在外部提供一个接口，在接口中声明一个方法。如果在实现该接口的内部类中实现该接口的方法，就可以定义多个内部类以不同的方式实现接口中的同一个方法，而在一般的类中是不能多次实现接口中同一个方法的，这种技巧经常被应用在 Swing 编程中，可以在一个类中做出多个不同的响应事件（Swing 编程技术会在后文中详细介绍）。

【例 11.10】 下面修改例 11.8，在项目中创建 InterfaceInner 类，并定义接口 OutInterface，使内部类 InnerClass 实现这个接口，最后使 doit()方法返回值类型为该接口。代码如下：（**实例位置：光盘\TM\sl\11.08**）

```
package com.lzw;
interface OutInterface {                              //定义一个接口
    public void f();
}
public class InterfaceInner {
    public static void main(String args[]) {
        OuterClass2 out = new OuterClass2();    //实例化一个 OuterClass2 对象
        //调用 doit()方法，返回一个 OutInterface 接口
        OutInterface outinter = out.doit();
        outinter.f();                                 //调用 f()方法
    }
}
class OuterClass2 {
    //定义一个内部类实现 OutInterface 接口
    private class InnerClass implements OutInterface {
        InnerClass(String s) {                        //内部类构造方法
            System.out.println(s);
        }
        public void f() {                             //实现接口中的 f()方法
            System.out.println("访问内部类中的 f()方法");
        }
    }
    public OutInterface doit() {                      //定义一个方法，返回值类型为 OutInterface 接口
        return new InnerClass("访问内部类构造方法");
```

```
    }
}
```

在 Eclipse 中运行上述实例，运行结果如图 11.10 所示。

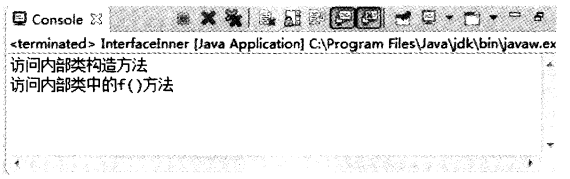

图 11.10　内部类向上转型为接口

从上述实例中可以看出，OuterClass2 类中定义了一个修饰权限为 private 的内部类，这个内部类实现了 OutInterface 接口，然后修改 doit()方法，使该方法返回一个 OutInterface 接口。由于内部类 InnerClass 修饰权限为 private，所以除了 OuterClass2 类可以访问该内部类之外，其他类都不能访问，而可以访问 doit()方法。由于该方法返回一个外部接口类型，这个接口可以作为外部使用的接口。它包含一个 f()方法，在继承此接口的内部类中实现了该方法，如果某个类继承了外部类，由于内部的权限不可以向下转型为内部类 InnerClass，同时也不能访问 f()方法，但是却可以访问接口中的 f()方法。例如，InterfaceInner 类中最后一条语句，接口引用调用 f()方法，从执行结果可以看出，这条语句执行的是内部类中的 f()方法，很好地对继承该类的子类隐藏了实现细节，仅为编写子类的人留下一个接口和一个外部类，同时也可以调用 f()方法，但是 f()方法的具体实现过程却被很好地隐藏了，这就是内部类最基本的用途。

注意

非内部类不能被声明为 private 或 protected 访问类型。

3．使用 this 关键字获取内部类与外部类的引用

如果在外部类中定义的成员变量与内部类的成员变量名称相同，可以使用 this 关键字。

【例 11.11】在项目中创建 TheSameName 类，在类中定义成员变量 x，再定义一个内部类 Inner，在内部类中也创建 x 变量，并在内部类的 doit()方法中分别操作两个 x 变量。关键代码如下：（**实例位置：光盘\TM\sl\11.09**）

```
public class TheSameName {
    private int x;
    private class Inner {
        private int x = 9;
        public void doit(int x) {
            x++;                          //调用的是形参 x
            this.x++;                     //调用内部类的变量 x
            TheSameName.this.x++;         //调用外部类的变量 x
        }
    }
}
```

在类中，如果遇到内部类与外部类的成员变量重名的情况，可以使用 this 关键字进行处理。例如，

在内部类中使用 this.x 语句可以调用内部类的成员变量 x，而使用 TheSameName.this.x 语句可以调用外部类的成员变量 x，即使用外部类名称后跟一个点操作符和 this 关键字便可获取外部类的一个引用。

图 11.11 给出了例 11.11 在内存中变量的布局情况。

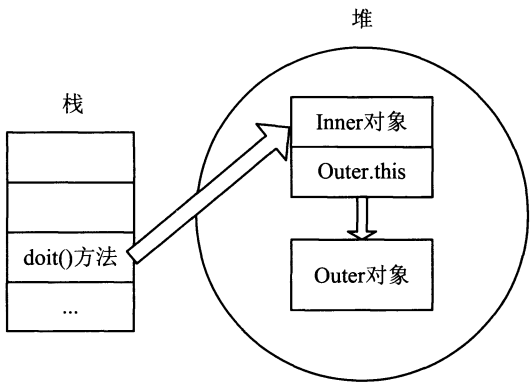

图 11.11　内部类对象与外部类对象在内存中的分布情况

读者应该明确一点，在内存中所有对象均被放置在堆中，方法以及方法中的形参或局部变量放置在栈中。在图 11.11 中，栈中的 doit()方法指向内部类的对象，而内部类的对象与外部类的对象是相互依赖的，Outer.this 对象指向外部类对象。

11.5.2　局部内部类

视频讲解：光盘\TM\lx\11\局部内部类.exe

内部类不仅可以在类中进行定义，也可以在类的局部位置定义，如在类的方法或任意的作用域中均可以定义内部类。

【例 11.12】 修改例 11.10，将 InnerClass 类放在 doit()方法的内部。关键代码如下：（**实例位置：光盘\TM\sl\11.10**）

```
interface OutInterface2 {                          //定义一个接口
}
class OuterClass3 {
    public OutInterface2 doit(final String x) { //doit()方法参数为 final 类型
        //在 doit()方法中定义一个内部类
        class InnerClass2 implements OutInterface2 {
            InnerClass2(String s) {
                s = x;
                System.out.println(s);
            }
        }
        return new InnerClass2("doit");
    }
}
```

从上述代码中可以看出，内部类被定义在了doit()方法内部。但是有一点值得注意，内部类InnerClass2是doit()方法的一部分，并非OuterClass3类中的一部分，所以在doit()方法的外部不能访问该内部类，但是该内部类可以访问当前代码块的常量以及此外部类的所有成员。

有的读者会注意到例 11.12 中的一个修改细节，就是将 doit()方法的参数设置为 final 类型。如果需要在方法体中使用局部变量，该局部变量需要被设置为 final 类型，换句话说，在方法中定义的内部类只能访问方法中 final 类型的局部变量，这是因为在方法中定义的局部变量相当于一个常量，它的生命周期超出方法运行的生命周期，由于该局部变量被设置为 final，所以不能在内部类中改变该局部变量的值。

11.5.3　匿名内部类

视频讲解：光盘\TM\lx\11\匿名内部类.exe

下面将例 11.12 中定义的内部类再次进行修改，在 doit()方法中将 return 语句和内部类定义语句合并在一起，下面通过一个实例说明。

【例 11.13】 在 return 语句中编写返回值为一个匿名内部类。

```
class OuterClass4 {
    public OutInterface2 doit() {          //定义 doit()方法
        return new OutInterface2() {       //声明匿名内部类
            private int i = 0;
            public int getValue() {
                return i;
            }
        };
    }
}
```

从例 11.13 中可以看出，笔者将 doit()方法修改得有一些莫名其妙，但这种写法确实被 Java 编译器认可，在 doit()方法内部首先返回一个 OutInterface2 的引用，然后在 return 语句中插入一个定义内部类的代码，由于这个类没有名称，所以这里将该内部类称为匿名内部类。实质上这种内部类的作用就是创建一个实现于 OutInterface2 接口的匿名类的对象。

匿名类的所有实现代码都需要在大括号之间进行编写。语法如下：

```
return new A(){
    …//内部类体
};
```

其中，A 指类名。

由于匿名内部类没有名称，所以匿名内部类使用默认构造方法来生成 OutInterface2 对象。在匿名内部类定义结束后，需要加分号标识，这个分号并不是代表定义内部类结束的标识，而是代表创建 OutInterface2 引用表达式的标识。

说明

匿名内部类编译以后，会产生以“外部类名$序号”为名称的.class 文件，序号以 1~n 排列，分别代表 1~n 个匿名内部类。

11.5.4 静态内部类

视频讲解：光盘\TM\lx\11\静态内部类.exe

在内部类前添加修饰符static，这个内部类就变为静态内部类了。一个静态内部类中可以声明static成员，但是在非静态内部类中不可以声明静态成员。静态内部类有一个最大的特点，就是不可以使用外部类的非静态成员，所以静态内部类在程序开发中比较少见。

可以这样认为，普通的内部类对象隐式地在外部保存了一个引用，指向创建它的外部类对象，但如果内部类被定义为 static，就会有更多的限制。静态内部类具有以下两个特点：

☑ 如果创建静态内部类的对象，不需要其外部类的对象。

☑ 不能从静态内部类的对象中访问非静态外部类的对象。

【例 11.14】 定义一个静态内部类 StaticInnerClass，可以使用如下代码：

```
public class StaticInnerClass {
    int x = 100;
    static class Inner {
        void doitInner() {
            // System.out.println("外部类"+x); //调用外部类的成员变量 x
        }
    }
}
```

例 11.14 中，在内部类的 doitInner()方法中调用成员变量 x，由于 Inner 被修饰为 static 形式，而成员变量 x 却是非 static 类型的，所以在 doitInner()方法中不能调用 x 变量。

进行程序测试时，如果在每一个 Java 文件中都设置一个主方法，将出现很多额外代码，而程序本身并不需要这些主方法，为了解决这个问题，可以将主方法写入静态内部类中。

【例 11.15】 在静态内部类中定义主方法。

```
public class StaticInnerClass {
    int x = 100;
    static class Inner {
        void doitInner() {
            // System.out.println("外部类"+x);
        }
        public static void main(String args[]) {
            System.out.println("a");
        }
    }
}
```

如果编译例 11.15 中的类，将生成一个名称为 StaticInnerClass$Inner 的独立类和一个 StaticInnerClass 类，只要使用 java StaticInnerClass$Inner，就可以运行主方法中的内容，这样当完成测试，需要将所有.class 文件打包时，只要删除 StaticInnerClass$Inner 独立类即可。

11.5.5　内部类的继承

视频讲解：光盘\TM\lx\11\内部类的继承.exe

内部类和其他普通类一样可以被继承，但是继承内部类比继承普通类复杂，需要设置专门的语法来完成。

【例 11.16】 在项目中创建 OutputInnerClass 类，使 OutputInnerClass 类继承 ClassA 类中的内部类 ClassB。

```
public class OutputInnerClass extends ClassA.ClassB {        //继承内部类 ClassB
    public OutputInnerClass(ClassA a) {
        a.super();
    }
}
class ClassA {
    class ClassB {
    }
}
```

在某个类继承内部类时，必须硬性给予这个类一个带参数的构造方法，并且该构造方法的参数为需要继承内部类的外部类的引用，同时在构造方法体中使用 a.super()语句，这样才为继承提供了必要的对象引用。

11.6　小　　结

在本章中读者学习了 Java 语言中的包、final 关键字的用法以及内部类。通过本章的学习，读者应该掌握在程序中如何导入包，如何定义 final 变量、final 方法以及 final 类，同时还应掌握内部类的用法，内部类是 Java 中类的高级用法，而匿名类在 Swing 编程中的使用尤为频繁，所以初学者应该多加练习，为今后的 Swing 学习打下良好的基础。

11.7　实践与练习

1．尝试在方法中编写一个匿名内部类。（**答案位置：光盘\TM\sl\11.11**）

2．尝试将主方法编写到静态内部类中，然后在 DOS 中编译运行，注意编译后出现的.class 文件。（**答案位置：光盘\TM\sl\11.12**）

3．尝试编写一个静态内部类，在主方法中创建其内部类的实例。（**答案位置：光盘\TM\sl\11.13**）

第12章

异常处理

（视频讲解：17分钟）

在程序设计和运行的过程中，发生错误是不可避免的。尽管 Java 语言的设计从根本上提供了便于写出整洁、安全的代码的方法，并且程序员也尽量地减少错误的产生，但使程序被迫停止的错误的存在仍然不可避免。为此，Java 提供了异常处理机制来帮助程序员检查可能出现的错误，保证了程序的可读性和可维护性。Java 中将异常封装到一个类中，出现错误时，就会抛出异常。本章将介绍异常处理的概念以及如何创建、激活自定义异常等知识。

通过阅读本章，您可以：

- 了解异常的概念
- 掌握捕捉异常
- 了解 Java 中常见的异常
- 掌握自定义异常
- 了解如何在方法中抛出异常
- 了解运行时异常的种类
- 了解异常处理的使用原则

12.1　异 常 概 述

视频讲解：光盘\TM\lx\12\异常概述.exe

在程序中，错误可能产生于程序员没有预料到的各种情况，或者是超出了程序员可控范围的环境因素，如用户的坏数据、试图打开一个根本不存在的文件等。在 Java 中这种在程序运行时可能出现的一些错误称为异常。异常是一个在程序执行期间发生的事件，它中断了正在执行的程序的正常指令流。

【例 12.1】 在项目中创建类 Baulk，在主方法中定义 int 型变量，将 0 作为除数赋值给该变量。**（实例位置：光盘\TM\sl\12.01）**

```
public class Baulk {                          //创建类 Baulk
    public static void main(String[] args) {  //主方法
        int result = 3 / 0;                   //定义 int 型变量并赋值
        System.out.println(result);           //将变量输出
    }
}
```

运行结果如图 12.1 所示。

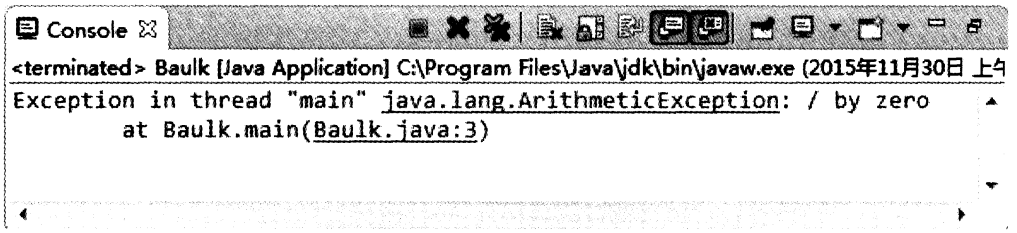

图 12.1　例 12.1 的运行结果

程序运行的结果报告发生了算术异常 ArithmeticException（根据给出的错误提示可知发生错误是因为在算术表达式“3/0”中，0 作为除数出现），系统不再执行下去，提前结束。这种情况就是所说的异常。

有许多异常的例子，如空指针、数组溢出等。Java 语言是一门面向对象的编程语言，因此，异常在 Java 语言中也是作为类的实例的形式出现的。当某一方法中发生错误时，这个方法会创建一个对象，并且把它传递给正在运行的系统。这个对象就是异常对象。通过异常处理机制，可以将非正常情况下的处理代码与程序的主逻辑分离，即在编写代码主流程的同时在其他地方处理异常。

12.2　处理程序异常错误

为了保证程序有效地执行，需要对发生的异常进行相应的处理。在 Java 中，如果某个方法抛出异常，既可以在当前方法中进行捕捉，然后处理该异常，也可以将异常向上抛出，由方法调用者来处理。本节将介绍 Java 中捕获异常的方法。

12.2.1 错误

视频讲解：光盘\TM\lx\12\错误.exe

异常产生后，如果不做任何处理，程序就会被终止。例如，将一个字符串转换为整型，可以通过 Integer 类的 parseInt()方法来实现。但如果该字符串不是数字形式，parseInt()方法就会抛出异常，程序将在出现异常的位置终止，不再执行下面的语句。

【例 12.2】 在项目中创建类 Thundering，在主方法中实现将非字符型数值转换为 int 型。运行程序，系统会报出异常提示。（**实例位置：光盘\TM\sl\12.02**）

```
public class Thundering {                          //创建类
    public static void main(String[] args) {       //主方法
        String str = "lili";                       //定义字符串
        System.out.println(str + "年龄是：");       //输出的提示信息
        int age = Integer.parseInt("20L");         //数据类型的转换
        System.out.println(age);                   //输出信息
    }
}
```

运行结果如图 12.2 所示。

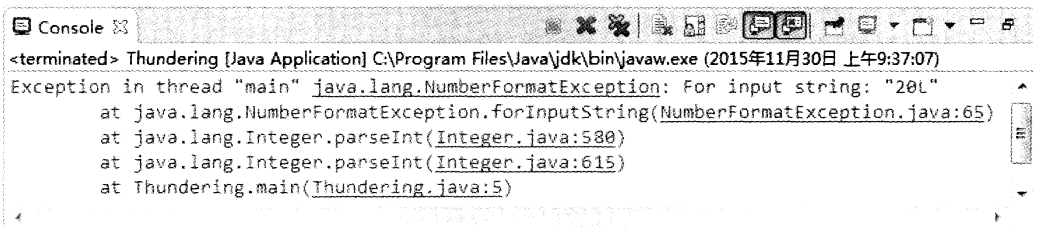

图 12.2 例 12.2 的运行结果

从图 12.2 中可以看出，本实例报出的是 NumberFormatException（字符串转换为数字）异常。提示信息“lili 年龄是”已经输出，可知该句代码之前并没有异常，而变量 age 没有输出，可知程序在执行类型转换代码时已经终止。

12.2.2 捕捉异常

视频讲解：光盘\TM\lx\12\捕捉异常.exe

Java 语言的异常捕获结构由 try、catch 和 finally 3 部分组成。其中，try 语句块存放的是可能发生异常的 Java 语句；catch 程序块在 try 语句块之后，用来激发被捕获的异常；finally 语句块是异常处理结构的最后执行部分，无论 try 语句块中的代码如何退出，都将执行 finally 语句块。

语法如下：

```
try{
    //程序代码块
```

```
}
catch(Exceptiontype1 e){
    //对 Exceptiontype1 的处理
}
catch(Exceptiontype2 e){
    //对 Exceptiontype2 的处理
}
...
finally{
    //程序块
}
```

通过异常处理器的语法可知，异常处理器大致分为 try-catch 语句块和 finally 语句块。

1．try-catch 语句块

下面将例 12.2 中的代码进行修改。

【例 12.3】 在项目中创建类 Take，在主方法中使用 try-catch 语句块将可能出现的异常语句进行异常处理。（实例位置：光盘\TM\sl\12.03）

```
public class Take {                                    //创建类
    public static void main(String[] args) {
        try {                                          //try 语句中包含可能出现异常的程序代码
            String str = "lili";                       //定义字符串变量
            System.out.println(str + "年龄是：");      //输出的信息
            int age = Integer.parseInt("20L");         //数据类型转换
            System.out.println(age);
        } catch (Exception e) {                        //catch 语句块用来获取异常信息
            e.printStackTrace();                       //输出异常性质
        }
        System.out.println("program over");            //输出信息
    }
}
```

运行结果如图 12.3 所示。

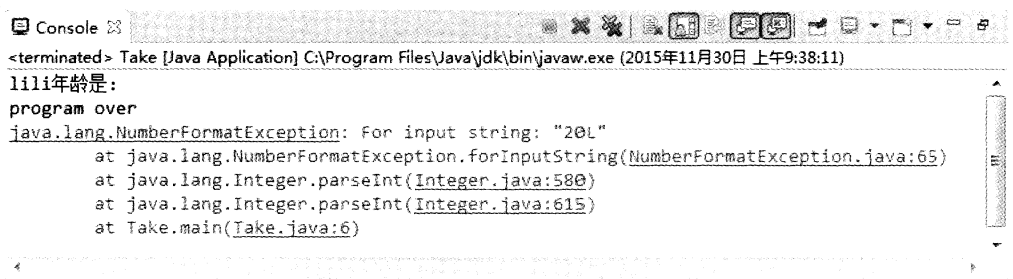

图 12.3　应用 try-catch 语句块对可能出现的异常语句进行异常处理

从图 12.3 中可以看出，程序仍然输出最后的提示信息，没有因为异常而终止。在例 12.3 中将可能出现异常的代码用 try-catch 语句块进行处理，当 try 代码块中的语句发生异常时，程序就会调转到 catch 代码块中执行，执行完 catch 代码块中的程序代码后，将继续执行 catch 代码块后的其他代码，而不会

执行 try 代码块中发生异常语句后面的代码。由此可知，Java 的异常处理是结构化的，不会因为一个异常影响整个程序的执行。

注意

Exception是try代码块传递给catch代码块的变量类型，e是变量名。catch代码块中语句“e.getMessage();”用于输出错误性质。通常，异常处理常用以下3个函数来获取异常的有关信息。

- ☑ getMessage()函数：输出错误性质。
- ☑ toString()函数：给出异常的类型与性质。
- ☑ printStackTrace()函数：指出异常的类型、性质、栈层次及出现在程序中的位置。

注意

有时为了编程简单会忽略 catch 语句后的代码，这样 try-catch 语句就成了一种摆设，一旦程序在运行过程中出现了异常，就会导致最终运行结果与期望的不一致，而错误发生的原因很难查找。因此要养成良好的编程习惯，最好在 catch 代码块中写入处理异常的代码。

2. finally 语句块

完整的异常处理语句一定要包含 finally 语句，无论程序中有无异常发生，并且无论之间的 try-catch 是否顺利执行完毕，都会执行 finally 语句。

在以下 4 种特殊情况下，finally 块不会被执行：

- ☑ 在 finally 语句块中发生了异常。
- ☑ 在前面的代码中使用了 System.exit()退出程序。
- ☑ 程序所在的线程死亡。
- ☑ 关闭 CPU。

12.3 Java 常见异常

视频讲解：光盘\TM\lx\12\Java 常见异常.exe

在 Java 中提供了一些异常用来描述经常发生的错误，其中，有的需要程序员进行捕获处理或声明抛出，有的是由 Java 虚拟机自动进行捕获处理的。Java 中常见的异常类如表 12.1 所示。

表 12.1 常见的异常类

异 常 类	说 明
ClassCastException	类型转换异常
ClassNotFoundException	未找到相应类异常
ArithmeticException	算术异常
ArrayIndexOutOfBoundsException	数组下标越界异常

续表

异　常　类	说　　明
ArrayStoreException	数组中包含不兼容的值抛出的异常
SQLException	操作数据库异常类
NullPointerException	空指针异常
NoSuchFieldException	字段未找到异常
NoSuchMethodException	方法未找到抛出的异常
NumberFormatException	字符串转换为数字抛出的异常
NegativeArraySizeException	数组元素个数为负数抛出的异常
StringIndexOutOfBoundsException	字符串索引超出范围抛出的异常
IOException	输入输出异常
IllegalAccessException	不允许访问某类异常
InstantiationException	当应用程序试图使用 Class 类中的 newInstance()方法创建一个类的实例，而指定的类对象无法被实例化时，抛出该异常
EOFException	文件已结束异常
FileNotFoundException	文件未找到异常

12.4　自定义异常

视频讲解：光盘\TM\lx\12\自定义异常.exe

使用 Java 内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户只需继承 Exception 类即可自定义异常类。

在程序中使用自定义异常类，大体可分为以下几个步骤：

（1）创建自定义异常类。

（2）在方法中通过 throw 关键字抛出异常对象。

（3）如果在当前抛出异常的方法中处理异常，可以使用 try-catch 语句块捕获并处理，否则在方法的声明处通过 throws 关键字指明要抛出给方法调用者的异常，继续进行下一步操作。

（4）在出现异常方法的调用者中捕获并处理异常。

【例 12.4】 创建自定义异常。在项目中创建类 MyException，该类继承 Exception。（**实例位置：光盘\TM\sl\12.04**）

```
public class MyException extends Exception {      //创建自定义异常，继承 Exception 类
    public MyException(String ErrorMessage) {     //构造方法
        super(ErrorMessage);                      //父类构造方法
    }
}
```

字符串 ErrorMessage 是要输出的错误信息。若想抛出用户自定义的异常对象，要使用 throw 关键字（throw 关键字的介绍可参考 12.5 节）。

【例 12.5】 在项目中创建类 Tran，该类中创建一个带有 int 型参数的方法 avg()，该方法用来检查

参数是否小于 0 或大于 100。如果参数小于 0 或大于 100，则通过 throw 关键字抛出一个 MyException 异常对象，并在 main()方法中捕捉该异常。（**实例位置：光盘\TM\sl\12.05**）

```
public class Tran {                                          //创建类
    //定义方法，抛出异常
    static int avg(int number1, int number2) throws MyException {
        if (number1 < 0 || number2 < 0) {                    //判断方法中参数是否满足指定条件
            throw new MyException("不可以使用负数");          //错误信息
        }
        if (number1 > 100 || number2 > 100) {                //判断方法中参数是否满足指定条件
            throw new MyException("数值太大了");              //错误信息
        }
        return (number1 + number2) / 2;                      //将参数的平均值返回
    }
    public static void main(String[] args) {                 //主方法
        try {                                                //try 代码块处理可能出现异常的代码
            int result = avg(102, 150);                      //调用 avg()方法
            System.out.println(result);                      //将 avg()方法的返回值输出
        } catch (MyException e) {
            System.out.println(e);                           //输出异常信息
        }
    }
}
```

运行结果如图 12.4 所示。

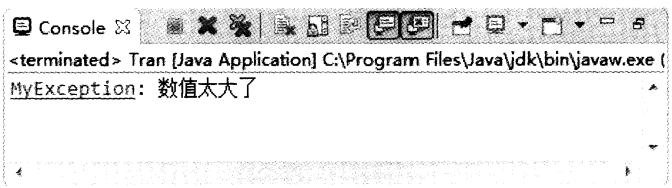

图 12.4　例 12.5 的运行结果

12.5　在方法中抛出异常

若某个方法可能会发生异常，但不想在当前方法中处理这个异常，则可以使用 throws、throw 关键字在方法中抛出异常。

12.5.1　使用 throws 关键字抛出异常

视频讲解：光盘\TM\lx\12\使用 throws 关键字抛出异常.exe

throws 关键字通常被应用在声明方法时，用来指定方法可能抛出的异常。多个异常可使用逗号分隔。

【例 12.6】 在项目中创建类 Shoot，在该类中创建方法 pop()，在该方法中抛出 NegativeArraySizeException 异常，在主方法中调用该方法，并实现异常处理。（**实例位置：光盘\TM\sl\12.06**）

```
public class Shoot {                                          //创建类
    static void pop() throws NegativeArraySizeException {
        //定义方法并抛出 NegativeArraySizeException 异常
        int[] arr = new int[-3];                              //创建数组
    }
    public static void main(String[] args) {                  //主方法
        try {                                                 //try 语句处理异常信息
            pop();                                            //调用 pop()方法
        } catch (NegativeArraySizeException e) {
            System.out.println("pop()方法抛出的异常");          //输出异常信息
        }
    }
}
```

运行结果如图 12.5 所示。

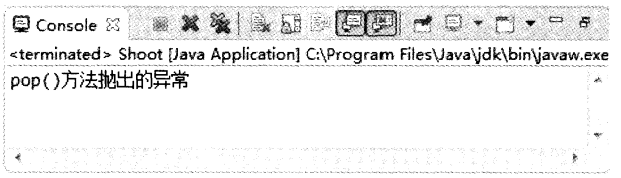

图 12.5　例 12.6 的运行结果

使用 throws 关键字将异常抛给上一级后，如果不想处理该异常，可以继续向上抛出，但最终要有能够处理该异常的代码。

说明

如果是 Error、RuntimeException 或它们的子类，可以不使用 throws 关键字来声明要抛出的异常，编译仍能顺利通过，但在运行时会被系统抛出。

12.5.2　使用 throw 关键字抛出异常

视频讲解：光盘\TM\lx\12\使用 throw 关键字抛出异常.exe

throw 关键字通常用于方法体中，并且抛出一个异常对象。程序在执行到 throw 语句时立即终止，它后面的语句都不执行。通过 throw 抛出异常后，如果想在上一级代码中来捕获并处理异常，则需要在抛出异常的方法中使用 throws 关键字在方法的声明中指明要抛出的异常；如果要捕捉 throw 抛出的异常，则必须使用 try-catch 语句块。

throw 通常用来抛出用户自定义异常。下面通过实例介绍 throw 的用法。

【例 12.7】 在项目中创建自定义异常类 MyException，继承类 Exception。（**实例位置：光盘\TM\sl\12.07**）

```
public class MyException extends Exception {      //创建自定义异常类
    String message;                               //定义 String 类型变量
    public MyException(String ErrorMessagr) {     //父类方法
        message = ErrorMessagr;
    }
    public String getMessage() {                  //覆盖 getMessage()方法
        return message;
    }
}
```

【例 12.8】 使用 throw 关键字捕捉异常。在项目中创建 Captor 类，该类中的 quotient()方法传递两个 int 型参数，如果其中的一个参数为负数，则会抛出 MyException 异常，最后在 main()方法中捕捉异常。（**实例位置：光盘\TM\sl\12.08**）

```
public class Captor {                                          //创建类
    static int quotient(int x, int y) throws MyException {     //定义方法抛出异常
        if (y < 0) {                                           //判断参数是否小于 0
            throw new MyException("除数不能是负数");            //异常信息
        }
        return x / y;                                          //返回值
    }
    public static void main(String args[]) {                   //主方法
        try {                                                  //try 语句包含可能发生异常的语句
            int result = quotient(3, -1);                      //调用方法 quotient()
        } catch (MyException e) {                              //处理自定义异常
            System.out.println(e.getMessage());                //输出异常信息
        } catch (ArithmeticException e) {                      //处理 ArithmeticException 异常
            System.out.println("除数不能为 0");                 //输出提示信息
        } catch (Exception e) {                                //处理其他异常
            System.out.println("程序发生了其他的异常");         //输出提示信息
        }
    }
}
```

运行结果如图 12.6 所示。

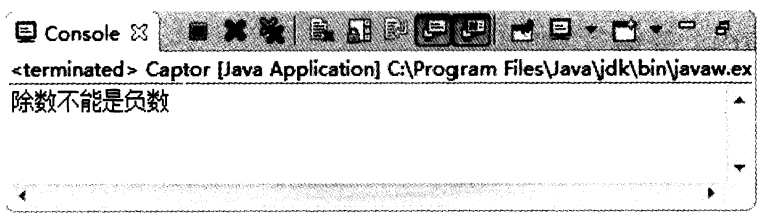

图 12.6 例 12.8 的运行结果

上面的实例使用了多个 catch 语句来捕捉异常。如果调用 quotient(3,-1)方法，将发生 MyException 异常，程序调转到 catch (MyException e)代码块中执行；如果调用 quotient(5,0)方法，会发生 ArithmeticException 异常，程序调转到 catch(ArithmeticException e)代码块中执行；如还有其他异常发生，将使用 catch(Exception e)捕捉异常。由于 Exception 是所有异常类的父类，如果将 catch(Exception e)代码块

放在其他两个代码块的前面，后面的代码块将永远得不到执行，也就没有什么意义了，所以 catch 语句的顺序不可调换。

12.6　运行时异常

视频讲解：光盘\TM\lx\12\运行时异常.exe

RuntimeException 异常是程序运行过程中产生的异常。Java 类库的每个包中都定义了异常类，所有这些类都是 Throwable 类的子类。Throwable 类派生了两个子类，分别是 Exception 和 Error 类。Error 类及其子类用来描述 Java 运行系统中的内部错误以及资源耗尽的错误，这类错误比较严重。Exception 类称为非致命性类，可以通过捕捉处理使程序继续执行。Exception 类又根据错误发生的原因分为 RuntimeException 异常和除 RuntimeException 之外的异常，如图 12.7 所示。

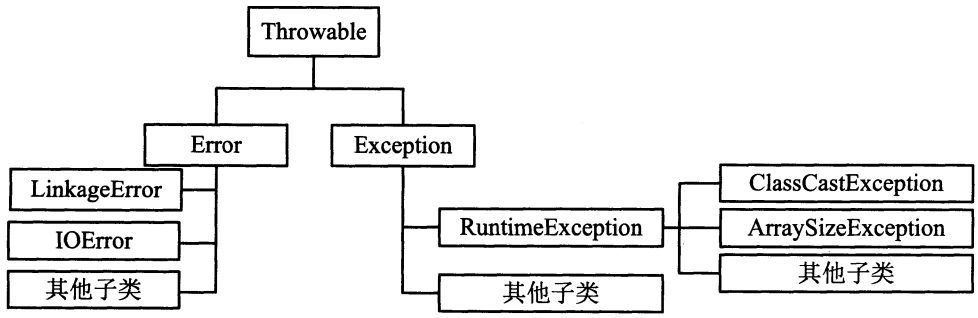

图 12.7　Java 异常类结构

Java 中提供了常见的 RuntimeException 异常，这些异常可通过 try-catch 语句捕获，如表 12.2 所示。

表 12.2　RuntimeException 异常的种类

种　类	说　明
NullPointerException	空指针异常
ArrayIndexOutOfBoundsException	数组下标越界异常
ArithmeticException	算术异常
ArrayStoreException	数组中包含不兼容的值抛出的异常
IllegalArgumentException	非法参数异常
SecurityException	安全性异常
NegativeArraySizeException	数组长度为负异常

12.7 异常的使用原则

视频讲解：光盘\TM\lx\12\异常的使用原则.exe

Java 异常强制用户去考虑程序的强健性和安全性。异常处理不应用来控制程序的正常流程，其主要作用是捕获程序在运行时发生的异常并进行相应的处理。编写代码处理某个方法可能出现的异常时，可遵循以下几条原则：

- ☑ 在当前方法声明中使用 try-catch 语句捕获异常。
- ☑ 一个方法被覆盖时，覆盖它的方法必须抛出相同的异常或异常的子类。
- ☑ 如果父类抛出多个异常，则覆盖方法必须抛出那些异常的一个子集，不能抛出新异常。

12.8 小　　结

本章向读者介绍的是 Java 中的异常处理机制。通过本章的学习读者应了解异常的概念、几种常见的异常类，掌握异常处理技术，以及如何创建、激活用户自定义的异常处理器。Java 中的异常处理是通过 try-catch 语句来实现的，也可以使用 throws 语句向上抛出。建议读者不要将异常抛出，应该编写异常处理语句。对于异常处理的使用原则，读者也应该理解。

12.9 实践与练习

1．编写一个异常类 MyException，再编写一个类 Student，该类有一个产生异常的方法 speak(int m)。要求参数 m 的值大于 1000 时，方法抛出一个 MyException 对象。最后编写主类，在主方法中创建 Student 对象，让该对象调用 speak()方法。（**答案位置：光盘\TM\sl\12.09**）

2．创建类 Number，通过类中的方法 count 可得到任意两个数相乘的结果，并在调用该方法的主方法中使用 try-catch 语句捕捉可能发生的异常。（**答案位置：光盘\TM\sl\12.10**）

3．创建类 Computer，该类中有一个计算两个数的最大公约数的方法，如果向该方法传递负整数，该方法就会抛出自定义异常。（**答案位置：光盘\TM\sl\12.11**）

第13章

Swing 程序设计

（视频讲解：1 小时 3 分钟）

Swing 较早期版本中的 AWT 更为强大、性能更加优良。Swing 中除了保留 AWT 中几个重要的重量级组件之外，其他组件都为轻量级，这样使用 Swing 开发出的窗体风格会与当前运行平台上的窗体风格一致，程序员也可以在跨平台时指定窗体统一的风格与外观。Swing 的使用很复杂，本章主要讲解 Swing 中的基本要素，包括容器、组件、窗体布局、事件和监听器。

通过阅读本章，您可以：

- 了解 Swing 组件
- 掌握常用窗体的使用方法
- 掌握在标签上设置图标的方法
- 掌握应用程序中的布局管理器的方法
- 掌握常用面板
- 掌握按钮组件
- 掌握列表组件
- 掌握文本组件
- 学会常用事件监听器的使用方法

13.1 Swing 概述

GUI（图形用户界面）为程序提供图形界面，它最初的设计目的是为程序员构建一个通用的 GUI，使其能够在所有的平台上运行，但 Java 1.0 中基础类 AWT（抽象窗口工具箱）并没有达到这个要求，于是 Swing 出现了，它是 AWT 组件的增强组件，但是它并不能完全替代 AWT 组件，这两种组件需要同时出现在一个图形用户界面中。

13.1.1 Swing 特点

视频讲解：光盘\TM\lx\13\Swing 特点.exe

原来的 AWT 组件来自 java.awt 包，当含有 AWT 组件的 Java 应用程序在不同的平台上执行时，每个平台的 GUI 组件的显示会有所不同，但是在不同平台上运行使用 Swing 开发的应用程序时，就可以统一 GUI 组件的显示风格，因为 Swing 组件允许编程人员在跨平台时指定统一的外观和风格。

Swing 组件通常被称为“轻量级组件”，因为它完全由 Java 语言编写，而 Java 是不依赖于操作系统的语言，它可以在任何平台上运行；相反，依赖于本地平台的组件被称为“重量级组件”，如 AWT 组件就是依赖本地平台的窗口系统来决定组件的功能、外观和风格。Swing 主要具有以下特点：

☑ 轻量级组件。

☑ 可插入外观组件。

13.1.2 Swing 包

视频讲解：光盘\TM\lx\13\Swing 包.mp4

为了有效地使用 Swing 组件，必须了解 Swing 包的层次结构和继承关系，其中比较重要的类是 Component 类、Container 类和 JComponent 类。图 13.1 描述了这些类的层次和继承关系。

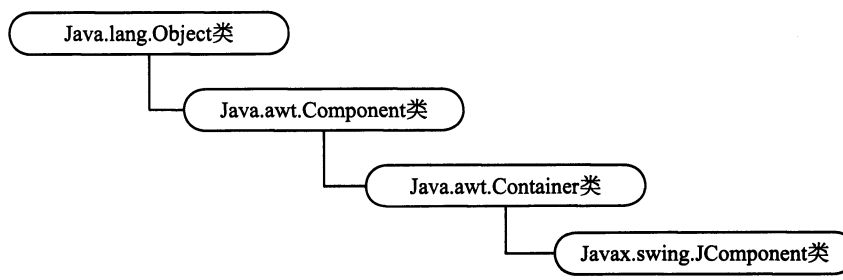

图 13.1 Swing 组件的类的层次和继承关系

在 Swing 组件中大多数 GUI 组件都是 Component 类的直接子类或间接子类，JComponent 类是 Swing 组件各种特性的存放位置，这些组件的特性包括设定组件边界、GUI 组件自动滚动等。

在 Swing 组件中最重要的父类是 Container 类，而 Container 类有两个最重要的子类，分别为 java.awt.Window 与 java.awt.Frame，除了以往的 AWT 类组件会继承这两个类之外，现在的 Swing 组件也扩展了这两个类。从图 13.1 中可以发现，顶层父类是 Component 类与 Container 类，所以 Java 关于窗口组件的编写，都与组件以及容器的概念相关联。

13.1.3　常用 Swing 组件概述

视频讲解：光盘\TM\lx\13\常用 Swing 组件概述.exe

下面给出基本 Swing 组件的概述，有关这些组件的内容将在后面详细讲解。表 13.1 列举了常用的 Swing 组件及其含义。

表 13.1　常用的 Swing 组件

组 件 名 称	定　义
JButton	代表 Swing 按钮，按钮可以带一些图片或文字
JCheckBox	代表 Swing 中的复选框组件
JComBox	代表下拉列表框，可以在下拉显示区域显示多个选项
JFrame	代表 Swing 的框架类
JDialog	代表 Swing 版本的对话框
JLabel	代表 Swing 中的标签组件
JRadioButton	代表 Swing 的单选按钮
JList	代表能够在用户界面中显示一系列条目的组件
JTextField	代表文本框
JPasswordField	代表密码框
JTextArea	代表 Swing 中的文本区域
JOptionPane	代表 Swing 中的一些对话框

13.2　常 用 窗 体

窗体作为 Swing 应用程序中组件的承载体，处于非常重要的位置。Swing 中常用的窗体包括 JFrame 和 JDialog，本节将着重讲解这两个窗体的使用方法。

13.2.1　JFrame 窗体

视频讲解：光盘\TM\lx\13\JFrame 窗体.exe

JFrame 窗体是一个容器，它是 Swing 程序中各个组件的载体，可以将 JFrame 看作是承载这些 Swing

组件的容器。在开发应用程序时可以通过继承 java.swing.JFrame 类创建一个窗体，在这个窗体中添加组件，同时为组件设置事件。由于该窗体继承了 JFrame 类，所以它拥有“最大化”“最小化”“关闭”等按钮。

下面将详细讲解 JFrame 窗体在 Java 应用程序中的使用方法。

JFrame 在程序中的语法格式如下：

```
JFrame jf=new JFrame(title);
Container container=jf.getContentPane();
```

☑ jf：JFrame 类的对象。

☑ container：Container 类的对象，可以使用 JFrame 对象调用 getContentPane()方法获取。

读者大致应该有这样一个概念，Swing 组件的窗体通常与组件和容器相关，所以在 JFrame 对象创建完成后，需要调用 getContentPane()方法将窗体转换为容器，然后在容器中添加组件或设置布局管理器。通常，这个容器用来包含和显示组件。如果需要将组件添加至容器，可以使用来自 Container 类的 add()方法进行设置。例如：

```
container.add(new JButton("按钮"));                    //JButton 按钮组件
```

在容器中添加组件后，也可以使用 Container 类的 remove()方法将这些组件从容器中删除。例如：

```
container.remove(new JButton("按钮"));
```

下面的实例中实现了 JFrame 对象创建一个窗体，并在其中添加一个组件。

【例 13.1】在项目中创建 Example1 类，该类继承 JFrame 类成为窗体类，在该类中创建标签组件，并添加到窗体界面中。（**实例位置：光盘\TM\sl\13.01**）

```
import java.awt.*;                                          //导入 awt 包
import javax.swing.*;                                       //导入 swing 包
public class Example1 extends JFrame {                     //定义一个类继承 JFrame 类
    public void CreateJFrame(String title) {                //定义一个 CreateJFrame()方法
        JFrame jf = new JFrame(title);                      //实例化一个 JFrame 对象
        Container container = jf.getContentPane();          //获取一个容器
        JLabel jl = new JLabel("这是一个 JFrame 窗体");      //创建一个 JLabel 标签
        //使标签上的文字居中
        jl.setHorizontalAlignment(SwingConstants.CENTER);
        container.add(jl);                                  //将标签添加到容器中
        container.setBackground(Color.white);               //设置容器的背景颜色
        jf.setVisible(true);                                //使窗体可视
        jf.setSize(200, 150);                               //设置窗体大小
        //设置窗体关闭方式
        jf.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
    public static void main(String args[]){                //在主方法中调用 CreateJFrame()方法
        new Example1().CreateJFrame("创建一个 JFrame 窗体");
    }
}
```

运行本实例程序，结果如图 13.2 所示。

图 13.2　创建 JFrame 窗体

在例 13.1 中，Example1 类继承了 JFrame 类，在 CreateJFrame()方法中实例化 JFrame 对象。JFrame 类的常用构造方法包括以下两种形式：

☑ public JFrame()。

☑ public JFrame(String title)。

JFrame 类中的两种构造方法分别为无参的构造方法与有参的构造方法，第 1 种形式的构造方法可以创建一个初始不可见、没有标题的新窗体；第 2 种形式的构造方法在实例化该 JFrame 对象时可以创建一个不可见但具有标题的窗体。可以使用 JFrame 对象调用 show()方法使窗体可见，但是该方法早已被新版 JDK 所弃用，通常使用 setVisible(true)方法使窗体可见。

同时可以使用 setSize(int x,int y)方法设置窗体大小，其中 x 与 y 变量分别代表窗体的宽与高。

创建窗体后，需要给予窗体一个关闭方式，可以调用 setDefaultCloseOperation()方法关闭窗体。Java 为窗体关闭提供了多种方式，常用的有以下 4 种：

☑ DO_NOTHING_ON_CLOSE。

☑ DISPOSE_ON_CLOSE。

☑ HIDE_ON_CLOSE。

☑ EXIT_ON_CLOSE。

这几种操作实质上是将一个 int 类型的常量封装在 javax.swing.WindowConstants 接口中。

第 1 种窗体退出方式代表什么都不做就将窗体关闭；第 2 种退出方式则代表任何注册监听程序对象后会自动隐藏并释放窗体；第 3 种方式表示隐藏窗口的默认窗口关闭；第 4 种退出方式表示退出应用程序默认窗口关闭。

13.2.2　JDialog 窗体

视频讲解：光盘\TM\lx\13\JDialog 窗体.exe

JDialog 窗体是 Swing 组件中的对话框，它继承了 AWT 组件中的 java.awt.Dialog 类。

JDialog 窗体的功能是从一个窗体中弹出另一个窗体，就像是在使用 IE 浏览器时弹出的确定对话框一样。JDialog 窗体实质上就是另一种类型的窗体，它与 JFrame 窗体类似，在使用时也需要调用 getContentPane()方法将窗体转换为容器，然后在容器中设置窗体的特性。

在应用程序中创建 JDialog 窗体需要实例化 JDialog 类，通常使用以下几个 JDialog 类的构造方法。

☑ public JDialog()：创建一个没有标题和父窗体的对话框。

☑ public JDialog(Frame f)：创建一个指定父窗体的对话框，但该窗体没有标题。

☑ public JDialog(Frame f,boolean model)：创建一个指定类型的对话框，并指定父窗体，但该窗体没有指定标题。

☑ public JDialog(Frame f,String title)：创建一个指定标题和父窗体的对话框。

☑ public JDialog(Frame f,String title,boolean model)：创建一个指定标题、窗体和模式的对话框。

下面来看一个实例，该实例主要实现单击 JFrame 窗体中的按钮后，弹出一个对话框窗体。

【例 13.2】 在项目中创建 MyJDialog 类，该类继承 JDialog 窗体，并在窗口中添加按钮，当用户单击该按钮后，将弹出一个对话框窗体。本实例关键代码如下：（实例位置：光盘\TM\sl\13.02）

```
class MyJDialog extends JDialog {                          //创建新类继承 JDialog 类
    public MyJDialog(MyFrame frame) {
        //实例化一个 JDialog 类对象，指定对话框的父窗体、窗体标题和类型
        super(frame, "第一个 JDialog 窗体", true);
        Container container = getContentPane();            //创建一个容器
        container.add(new JLabel("这是一个对话框"));        //在容器中添加标签
        setBounds(120, 120, 100, 100);                     //设置对话框窗体大小
    }
}
public class MyFrame extends JFrame {                      //创建新类
    public static void main(String args[]) {
        new MyFrame();                                     //实例化 MyJDialog 类对象
    }
    public MyFrame() {
        Container container = getContentPane();            //创建一个容器
        container.setLayout(null);
        JLabel jl = new JLabel("这是一个 JFrame 窗体");     //在窗体中设置标签
        //将标签的文字置于标签中间位置
        jl.setHorizontalAlignment(SwingConstants.CENTER);
        container.add(jl);
        JButton bl = new JButton("弹出对话框");             //定义一个按钮
        bl.setBounds(10, 10, 100, 21);
        bl.addActionListener(new ActionListener() {        //为按钮添加鼠标单击事件
            public void actionPerformed(ActionEvent e) {
                //使 MyJDialog 窗体可见
                new MyJDialog(MyFrame.this).setVisible(true);
            }
        });
        container.add(bl);                                 //将按钮添加到容器中
        …//省略部分代码
    }
}
```

运行本实例，结果如图 13.3 所示。

在本实例中，为了使对话框在父窗体弹出，定义了一个 JFrame 窗体，首先在该窗体中定义一个按钮，然后为此按钮添加一个鼠标单击监听事件（在这里使用了匿名内部类的形式，如果读者对这部分的代码实现有疑问，不妨回顾一下第 11 章中该部分的内容，而监听事件会在后续章节中进行讲解，在这里读者只需知道这部分代码是当用户单击该按钮后实现的某种功能即可），这里使用 new MyJDialog().setVisible(true)语句使对话框窗体可见，这样就实现了用户单击该按钮后弹出对话框的功能。

图 13.3　弹出 JDialog 窗体

在 MyJDialog 类中，由于它继承了 JDialog 类，所以可以在构造方法中使用 super 关键字调用 JDialog 构造方法。在这里使用了 public JDialog(Frame f,String title,boolean model)这种形式的构造方法，相应地设置了自定义的 JFrame 窗体以及对话框的标题和窗体类型。

在本实例代码中可以看到，JDialog 窗体与 JFrame 窗体形式基本相同，甚至在设置窗体的特性时调用的方法名称都基本相同，如设置窗体大小、窗体关闭状态等。

13.3　标签组件与图标

在 Swing 中显示文本或提示信息的方法是使用标签，它支持文本字符串和图标。在应用程序的用户界面中，一个简短的文本标签可以使用户知道这些组件的目的，所以标签在 Swing 中是比较常用的组件。本节将探讨 Swing 标签的用法、如何创建标签，以及如何在标签上放置文本与图标。

13.3.1　标签的使用

视频讲解：光盘\TM\lx\13\标签的使用.exe

标签由 JLabel 类定义，它的父类为 JComponent 类。

标签可以显示一行只读文本、一个图像或带图像的文本，它并不能产生任何类型的事件，只是简单地显示文本和图片，但是可以使用标签的特性指定标签上文本的对齐方式。

JLabel 类提供了多种构造方法，可以创建多种标签，如显示只有文本的标签、只有图标的标签或包含文本与图标的标签。JLabel 类常用的几个构造方法如下。

- ☑ public JLabel()：创建一个不带图标和文本的 JLabel 对象。
- ☑ public JLabel(Icon icon)：创建一个带图标的 JLabel 对象。
- ☑ public JLabel(Icon icon,int aligment)：创建一个带图标的 JLabel 对象，并设置图标水平对齐方式。
- ☑ public JLabel(String text,int aligment)：创建一个带文本的 JLabel 对象，并设置文字水平对齐方式。
- ☑ public JLabel(String text,Icon icon,int aligment)：创建一个带文本、带图标的 JLabel 对象，并设置标签内容的水平对齐方式。

在这里读者可以大致了解 JLabel 类的用法，13.4.2 小节中将结合图标的使用来举例说明 JLabel 类的具体用法。

13.3.2 图标的使用

视频讲解：光盘\TM\lx\13\图标的使用.mp4

Swing 中的图标可以放置在按钮、标签等组件上，用于描述组件的用途。图标可以用 Java 支持的图片文件类型进行创建，也可以使用 java.awt.Graphics 类提供的功能方法来创建。

1. 创建图标

在 Swing 中通过 Icon 接口来创建图标，可以在创建时给定图标的大小、颜色等特性。如果使用 Icon 接口，必须实现 Icon 接口中的 3 个方法：

- ☑ public int getIconHeight()。
- ☑ public int getIconWidth()。
- ☑ public void paintIcon(Component arg0, Graphics arg1, int arg2, int arg3)。

getIconHeigth()与 getIconWidth()方法用于获取图标的长与宽，paintIcon()方法用于实现在指定坐标位置画图。

下面列举一个实现 Icon 接口创建图标的例子。

【例 13.3】 在项目中创建实现 Icon 接口的 DrawIcon 类，该类实现自定义的图标类。本实例关键代码如下：（实例位置：光盘\TM\sl\13.03）

```
public class DrawIcon implements Icon {        //实现 Icon 接口
    private int width;                          //声明图标的宽
    private int height;                         //声明图标的长
    public int getIconHeight() {                //实现 getIconHeight()方法
        return this.height;
    }
    public int getIconWidth() {                 //实现 getIconWidth()方法
        return this.width;
    }
    public DrawIcon(int width, int height) {    //定义构造方法
        this.width = width;
        this.height = height;
    }
    //实现 paintIcon()方法
    public void paintIcon(Component arg0, Graphics arg1, int x, int y) {
        arg1.fillOval(x, y, width, height);     //绘制一个圆形
    }
    public static void main(String[] args) {
        DrawIcon icon = new DrawIcon(15, 15);
        //创建一个标签，并设置标签上的文字在标签正中间
        JLabel j = new JLabel("测试", icon, SwingConstants.CENTER);
        JFrame jf = new JFrame();               //创建一个 JFrame 窗口
        Container c = jf.getContentPane();
        …//省略部分代码
```

```
    }
}
```

运行本实例，结果如图 13.4 所示。

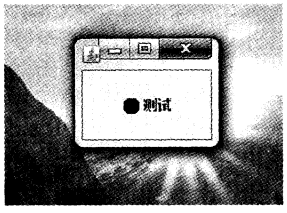

图 13.4　实现 Icon 接口创建图标

在本实例中，由于 DrawIcon 类继承了 Icon 接口，所以在该类中必须实现 Icon 接口中定义的所有方法，其中在实现 paintIcon()方法中使用 Graphics 类中的方法绘制一个圆形的图标，其余实现接口的方法为返回图标的长与宽。在 DrawIcon 类的构造方法中设置了图标的长与宽，这样如果需要在窗体中使用图标，就可以使用如下代码创建图标：

```
DrawIcon icon=new DrawIcon(15,15);
```

在前文中提到过，一般情况下会将图标放置在按钮或标签上，这里将图标放置在标签上，然后将标签添加到容器中，这样就实现了在窗体中使用图标的功能。

2．使用图片图标

Swing 中的图标除了可以绘制之外，还可以使用某个特定的图片创建。Swing 利用 javax.swing.ImageIcon 类根据现有图片创建图标，ImageIcon 类实现了 Icon 接口，同时 Java 支持多种图片格式。

ImageIcon 类有多个构造方法，下面是其中几个常用的构造方法。

- ☑ public ImageIcon()：该构造方法创建一个通用的 ImageIcon 对象，当真正需要设置图片时再使用 ImageIcon 对象调用 setImage(Image image)方法来操作。
- ☑ public ImageIcon(Image image)：可以直接从图片源创建图标。
- ☑ public ImageIcon(Image image,String description)：除了可以从图片源创建图标之外，还可以为这个图标添加简短的描述，但这个描述不会在图标上显示，可以使用 getDescription()方法获取这个描述。
- ☑ public ImageIcon(URL url)：该构造方法利用位于计算机网络上的图像文件创建图标。

下面来看一个创建图片图标的实例。

【例 13.4】在项目中创建继承 JFrame 类的 MyImageIcon 类，在类中创建 ImageIcon 类的实例对象，该对象使用现有图片创建图标对象，并应用到组件上。（**实例位置：光盘\TM\sl\13.04**）

```
public class MyImageIcon extends JFrame {
    public MyImageIcon() {
        Container container = getContentPane();
        //创建一个标签
        JLabel jl = new JLabel("这是一个 JFrame 窗体", JLabel.CENTER);
        //获取图片所在的 URL
        URL url = MyImageIcon.class.getResource("imageButton.jpg");
        Icon icon = new ImageIcon(url);                    //实例化 Icon 对象
        jl.setIcon(icon);                                  //为标签设置图片
        //设置文字放置在标签中间
        jl.setHorizontalAlignment(SwingConstants.CENTER);
        jl.setOpaque(true);                                //设置标签为不透明状态
        container.add(jl);                                 //将标签添加到容器中
```

```
        setSize(250, 100);                          //设置窗体大小
        setVisible(true);                           //使窗体可见
        //设置窗体关闭模式
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
    public static void main(String args[]) {
        new MyImageIcon();                          //实例化 MyImageIcon 对象
    }
}
```

运行本实例，结果如图 13.5 所示。

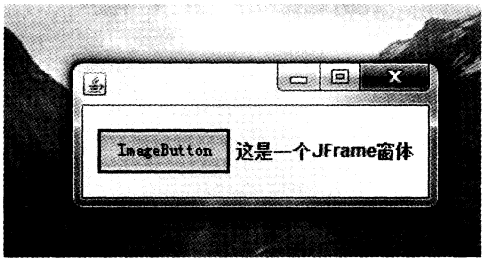

图 13.5　使用图片创建图标

注意

java.lang.Class 类中的 getResource()方法可以获取资源文件的 URL 路径。例 13.4 中该方法的参数是 imageButton.jpg，这个路径是相对于 MyImageIcon 类文件的，所以可将 imageButton.jpg 图片文件与 MyImageIcon 类文件放在同一个文件夹下。

在本实例中，首先使用 public JLabel(String text,int aligment)构造方法创建一个 JLabel 对象，然后调用 setIcon()方法为标签设置图标。当然，读者也可以选择在初始化 JLabel 对象时为标签指定图标，这时需要获取一个 Icon 实例。

13.4　常用布局管理器

在 Swing 中，每个组件在容器中都有一个具体的位置和大小，而在容器中摆放各种组件时很难判断其具体位置和大小。布局管理器提供了 Swing 组件安排、展示在容器中的方法及基本的布局功能。使用布局管理器较程序员直接在容器中控制 Swing 组件的位置和大小方便得多，可以有效地处理整个窗体的布局。Swing 提供的常用布局管理器包括流布局管理器、边界布局管理器和网格布局管理器。本节将探讨 Swing 中常用的布局管理器。

13.4.1　绝对布局

视频讲解：光盘\TM\lx\13\绝对布局.exe

在 Swing 中，除了使用布局管理器之外还可以使用绝对布局。绝对布局，就是硬性指定组件在容器中的位置和大小，可以使用绝对坐标的方式来指定组件的位置。

使用绝对布局的步骤如下：

（1）使用 Container.setLayout(null)方法取消布局管理器。

（2）使用 Component.setBounds()方法设置每个组件的大小与位置。

下面来看一个绝对布局的例子。

【例 13.5】 在项目中创建继承 JFrame 窗体组件的 AbsolutePosition 类，设置布局管理器为 null，即使用绝对定位的布局方式，创建两个按钮组件，将按钮分别定位在不同的窗体位置上。（**实例位置：光盘\TM\sl\13.05**）

```
public class AbsolutePosition extends JFrame {
    public AbsolutePosition() {
        setTitle("本窗体使用绝对布局");              //设置该窗体的标题
        setLayout(null);                            //使该窗体取消布局管理器设置
        setBounds(0, 0, 200, 150);                  //绝对定位窗体的位置与大小
        Container c = getContentPane();             //创建容器对象
        JButton b1 = new JButton("按钮 1");          //创建按钮
        JButton b2 = new JButton("按钮 2");          //创建按钮
        b1.setBounds(10, 30, 80, 30);               //设置按钮的位置与大小
        b2.setBounds(60, 70, 100, 20);
        c.add(b1);                                  //将按钮添加到容器中
        c.add(b2);
        setVisible(true);                           //使窗体可见
        //设置窗体关闭方式
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new AbsolutePosition();
    }
}
```

运行本例，结果如图 13.6 所示。

图 13.6　绝对布局效果

在本实例中，窗体的大小、位置以及窗体内组件的大小与位置都被进行绝对布局操作。绝对布局使用 setBounds(int x,int y,int width,int height)方法进行设置，如果使窗体对象调用 setBounds()方法，它的参数 x 与 y 分别代表这个窗体在整个屏幕上出现的位置，width 与 height 则代表这个窗体的宽与长；如果使窗体内的组件调用 setBounds()方法，参数 x 与 y 则代表这个组件在整个窗体摆放的位置，width 与 height 则代表这

个组件的大小。

需要注意的是，在使用绝对布局之前需要调用 setLayout(null)方法告知编译器，这里不再使用布局管理器。

13.4.2 流布局管理器

视频讲解：光盘\TM\lx\13\流布局管理器.exe

流（FlowLayout）布局管理器是最基本的布局管理器，在整个容器中的布局正如其名，像“流”一样从左到右摆放组件，直到占据了这一行的所有空间，然后再向下移动一行。默认情况下，组件在每一行都是居中排列的，但是通过设置也可以更改组件在每一行上的排列位置。

FlowLayout 类中具有以下常用的构造方法：

☑ public FlowLayout()。

☑ public FlowLayout(int alignment)。

☑ public FlowLayout(int alignment,int horizGap,int vertGap)。

构造方法中的 alignment 参数表示使用流布局管理器后组件在每一行的具体摆放位置。它可以被赋予以下 3 个值之一：

☑ FlowLayout.LEFT=0。

☑ FlowLayout.CENTER=1。

☑ FlowLayout.RIGHT=2。

上述 3 个值分别代表容器使用流布局管理器后组件在每一行中的摆放位置。例如，将 alignment 设置为 0 时，每一行的组件将被指定按照左对齐排列；而将 alignment 设置为 2 时，每一行的组件将被指定为按照右对齐排列。

在 public FlowLayout(int alignment,int horizGap,int vertGap)构造方法中还存在 horizGap 与 vertGap 两个参数，这两个参数分别以像素为单位指定组件之间的水平间隔与垂直间隔。

下面是一个流布局管理器的例子。在此例中，首先将容器的布局管理器设置为 FlowLayout，然后在窗体上摆放组件。

【例 13.6】 在项目中创建 FlowLayoutPosition 类，该类继承 JFrame 类成为窗体组件。设置该窗体的布局管理器为 FlowLayout 布局管理器的实例对象。（**实例位置：光盘\TM\sl\13.06**）

```
public class FlowLayoutPosition extends JFrame {
    public FlowLayoutPosition() {
        setTitle("本窗体使用流布局管理器");          //设置窗体标题
        Container c = getContentPane();
        //设置窗体使用流布局管理器，使组件右对齐，并且设置组件之间的水平间隔与垂直间隔
        setLayout(new FlowLayout(2, 10, 10));
        for (int i = 0; i < 10; i++) {                  //在容器中循环添加 10 个按钮
            c.add(new JButton("button" + i));
        }
        setSize(300, 200);                              //设置窗体大小
```

```
        setVisible(true);                          //设置窗体可见
        //设置窗体关闭方式
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    }
    public static void main(String[] args) {
        new FlowLayoutPosition();
    }
}
```

运行本实例，结果如图 13.7 所示。

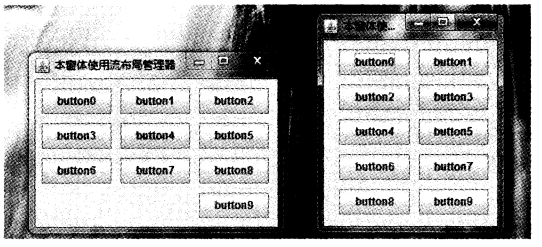

图 13.7　在应用程序使用流布局管理器

从本实例的运行结果中可以看出，如果改变整个窗体的大小，其中组件的摆放位置也会相应地发生变化，这正好验证了使用流布局管理器时组件从左到右摆放，当组件填满一行后，将自动换行，直到所有的组件都摆放在容器中为止。

13.4.3　边界布局管理器

视频讲解：光盘\TM\lx\13\边界布局管理器.exe

在默认不指定窗体布局的情况下，Swing 组件的布局模式是边界（BorderLayout）布局管理器。例如，在例 13.3 中，容器中只添加了一个标签组件，在运行结果中可以看到这个标签被放置在窗体中间，并且整个组件占据了窗体的所有空间，实质上在这个容器中默认使用了边界布局管理器。

但是边界布局管理器的功能不止如此，边界布局管理器还可以将容器划分为东、南、西、北、中 5 个区域，可以将组件加入到这 5 个区域中。容器调用 Container 类的 add()方法添加组件时可以设置此组件在边界布局管理器中的区域，区域的控制可以由 BorderLayout 类中的成员变量来决定，这些成员变量的具体含义如表 13.2 所示。

表 13.2　BorderLayout 类的主要成员变量

成 员 变 量	含　义
BorderLayout.NORTH	在容器中添加组件时，组件置于顶端
BorderLayout.SOUTH	在容器中添加组件时，组件置于底端
BorderLayout.EAST	在容器中添加组件时，组件置于右端
BorderLayout.WEST	在容器中添加组件时，组件置于左端
BorderLayout.CENTER	在容器中添加组件时，组件置于中间开始填充，直到与其他组件边界连接

下面举一个在容器中设置边界布局管理器的例子，分别在容器的东、南、西、北、中区域添加 5 个按钮。

【例 13.7】 在项目中创建 BorderLayoutPosition 类，该类继承 JFrame 类成为窗体组件，设置该窗体的布局管理器使用 BorderLayout 类的实例对象。（**实例位置：光盘\TM\sl\13.07**）

```
public class BorderLayoutPosition extends JFrame {
    //定义组件摆放位置的数组
    String[] border = { BorderLayout.CENTER, BorderLayout.NORTH,
            BorderLayout.SOUTH, BorderLayout.WEST, BorderLayout.EAST };
    String[] buttonName = { "center button", "north button",
            "south button", "west button", "east button" };
    public BorderLayoutPosition() {
        setTitle("这个窗体使用边界布局管理器");
        Container c = getContentPane();              //定义一个容器
        setLayout(new BorderLayout());               //设置容器为边界布局管理器
        for (int i = 0; i < border.length; i++) {
            //在容器中添加按钮，并设置按钮布局
            c.add(border[i], new JButton(buttonName[i]));
        }
        setSize(350, 200);                           //设置窗体大小
        setVisible(true);                            //设置窗体可见
        //设置窗体关闭方式
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    }
    public static void main(String[] args) {
        new BorderLayoutPosition();
    }
}
```

运行本实例，结果如图 13.8 所示。

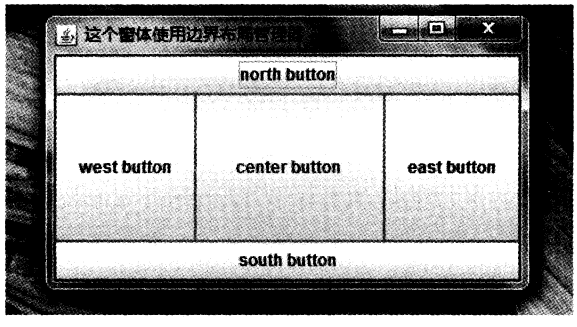

图 13.8 在应用程序中使用边界布局管理器

在本实例中将布局以及组件名称分别放置在数组中，然后设置容器使用边界布局管理器，最后在循环中将按钮添加至容器中，并设置组件布局。add()方法提供在容器中添加组件的功能，并同时设置组件的摆放位置。

13.4.4　网格布局管理器

视频讲解：光盘\TM\lx\13\网格布局管理器.exe

网格（GridLayout）布局管理器将容器划分为网格，所以组件可以按行和列进行排列。在网格布局管理器中，每一个组件的大小都相同，并且网格中空格的个数由网格的行数和列数决定，如一个两行两列的网格能产生 4 个大小相等的网格。组件从网格的左上角开始，按照从左到右、从上到下的顺序加入到网格中，而且每一个组件都会填满整个网格，改变窗体的大小，组件的大小也会随之改变。

网格布局管理器主要有以下两个常用的构造方法。

☑ public GridLayout(int rows,int columns)。

☑ public GridLayout(int rows,int columns,int horizGap,int vertGap)。

在上述构造方法中，rows 与 columns 参数代表网格的行数与列数，这两个参数只有一个参数可以为 0，代表一行或一列可以排列任意多个组件；参数 horizGap 与 vertGap 指定网格之间的距离，其中 horizGap 参数指定网格之间的水平距离，vertGap 参数指定网格之间的垂直距离。

下面来看一个在应用程序中使用网格布局管理器的例子。

【例 13.8】 在项目中创建 GridLayoutPosition 类，该类继承 JFrame 类成为窗体组件，设置该窗体使用网格布局管理器。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.08**）

```
package com.lzw;
import java.awt.*;
import javax.swing.*;
public class GridLayoutPosition extends JFrame {
    public GridLayoutPosition() {
        Container c = getContentPane();
        //设置容器使用网格布局管理器，设置 7 行 3 列的网格
        setLayout(new GridLayout(7, 3, 5, 5));
        for (int i = 0; i < 20; i++) {
            c.add(new JButton("button" + i));          //循环添加按钮
        }
        setSize(300, 300);
        setTitle("这是一个使用网格布局管理器的窗体");
        setVisible(true);
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new GridLayoutPosition();
    }
}
```

运行本实例，结果如图 13.9 所示。

从本实例的运行结果中可以看出，组件在窗体中的布局呈现出一个 3 行 7 列的网格，并且添加到该布局中的组件被放置在网格中。如果尝试改变窗体的大小，将发现其中的组件大小也会发生相应的改变。

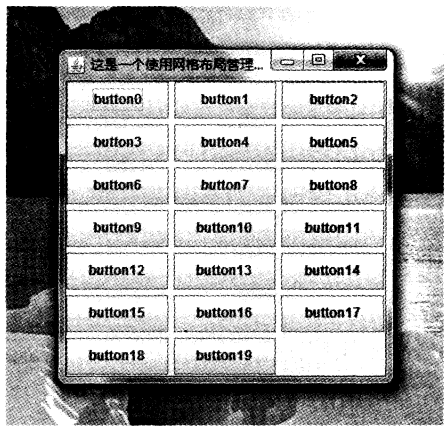

图 13.9　在应用程序中使用网格布局管理器

13.5　常 用 面 板

面板也是一个 Swing 容器，它可以作为容器容纳其他组件，但它也必须被添加到其他容器中。Swing 中常用的面板包括 JPanel 面板以及 JScrollPane 面板。下面着重讲解 Swing 中的常用面板。

13.5.1　JPanel 面板

视频讲解：光盘\TM\lx\13\JPanel 面板.mp4

JPanel 面板可以聚集一些组件来布局。读者首先应该明确的是面板也是一种容器，因为它也继承自 java.awt.Container 类。

例 13.9 给出一个小程序，在窗体中使用了 4 个面板，然后在每个面板中设置布局管理器，最后分别在 4 个面板中放置一些组件。

【例 13.9】 在项目中创建 JPanelTest 类，该类继承 JFrame 类成为窗体组件，在该类中创建 4 个 JPanel 面板组件，并将它们添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.09**）

```
package com.lzw;
import java.awt.*;
import javax.swing.*;
public class JPanelTest extends JFrame {
    public JPanelTest() {
        Container c = getContentPane();
        //将整个容器设置为 2 行 1 列的网格布局
        c.setLayout(new GridLayout(2,1,10,10));
        //初始化一个面板，设置 1 行 3 列的网格布局
        JPanel p1 = new JPanel(new GridLayout(1, 3, 10, 10));
        JPanel p2 = new JPanel(new GridLayout(1, 2, 10, 10));
        JPanel p3 = new JPanel(new GridLayout(1, 2, 10, 10));
```

```
        JPanel p4 = new JPanel(new GridLayout(2, 1, 10, 10));
        p1.add(new JButton("1"));                    //在面板中添加按钮
        …//省略部分代码
        c.add(p1);                                   //在容器中添加面板
        c.add(p2);
        c.add(p3);
        c.add(p4);
        …//省略部分代码
    }
    public static void main(String[] args) {
        new JPanelTest();
    }
}
```

运行本实例，结果如图 13.10 所示。

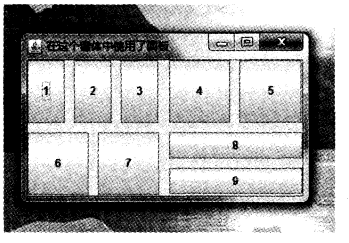

图 13.10　在应用程序中使用面板

在本实例中，首先设置整个窗体的布局为 2 行 1 列的网格布局，然后先后定义 4 个面板，分别为 4 个面板设置网格布局，当然，行列数会有所不同，将按钮放置在每个面板中，最后将面板添加至容器中。

13.5.2　JScrollPane 面板

视频讲解：光盘\TM\lx\13\JScrollPane 面板.exe

在设置界面时，可能会遇到在一个较小的容器窗体中显示一个较大部分的内容的情况，这时可以使用 JScrollPane 面板。JScrollPane 面板是带滚动条的面板，它也是一种容器，但是 JScrollPane 只能放置一个组件，并且不可以使用布局管理器。如果需要在 JScrollPane 面板中放置多个组件，需要将多个组件放置在 JPanel 面板上，然后将 JPanel 面板作为一个整体组件添加在 JScrollPane 组件上。

下面列举一个 JScrollPane 面板的例子。

【例 13.10】 在项目中创建 JScrollPaneTest 类，该类继承 JFrame 类成为窗体组件，在类中创建 JScrollPane 滚动面板组件，该滚动面板组件包含 JTextArea 文本域组件。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.10**）

```
import javax.swing.*;                                //包含 swing 包
public class JScrollPaneTest extends JFrame {
    public JScrollPaneTest() {
        Container c = getContentPane();              //创建容器
```

```
        JTextArea ta = new JTextArea(20, 50);          //创建文本区域组件
        JScrollPane sp = new JScrollPane(ta);          //创建 JScrollPane 面板对象
        c.add(sp);                                     //将该面板添加到该容器中
        setTitle("带滚动条的文字编译器");
        setSize(200, 200);
        setVisible(true);
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    }
    public static void main(String[] args) {
        new JScrollPaneTest();
    }
}
```

运行本实例，结果如图 13.11 所示。

从本实例的运行结果中可以看出，在窗体中创建一个带滚动条的文字编译器，首先需要初始化编译器（在 Swing 中编译器类为 JTextArea 类），并在初始化时指定编译器的大小完成（如果读者对编译器的概念有些困惑，可以参见后续章节）。当创建带滚动条的面板时，需将编译器加入面板中，最后将带滚动条的编译器放置在容器中即可。

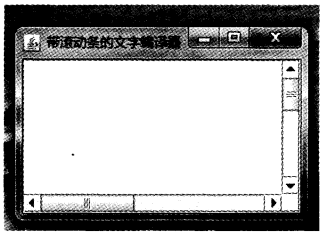

图 13.11　在应用程序中使用 JScrollPane 面板

13.6　按 钮 组 件

按钮在 Swing 中是较为常见的组件，用于触发特定动作。Swing 中提供多种按钮，包括提交按钮、复选框、单选按钮等，这些按钮都是从 AbstractButton 类中继承而来的，本节将着重讲解这些按钮的应用。

13.6.1　提交按钮组件

视频讲解：光盘\TM\lx\13\提交按钮组件.exe

Swing 中的提交按钮（JButton）由 JButton 对象表示，其构造方法主要有以下几种形式：

- ☑ public JButton()。
- ☑ public JButton(String text)。
- ☑ public JButton(Icon icon)。
- ☑ public JButton(String text,Icon icon)。

通过使用上述构造方法，在 Swing 按钮上不仅能显示文本标签，还可以显示图标。上述构造方法中的第一个构造方法可以生成不带任何文本组件的对象和图标，可以在以后使用相应方法为按钮设置指定的文本和图标；其他构造方法都在初始化时指定了按钮上显示的图标或文字。

下面来看一个例子，在设置的窗体中指定了一个同时带文字与图标的按钮。

【例 13.11】 在项目中新建 JButtonTest 类，该类继承 JFrame 类成为窗体组件，在该窗体中创建按钮组件，并为按钮设置图标，添加动作监听器。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.11**）

```
public class JButtonTest extends JFrame {
    public JButtonTest() {
        URL url = MyImageIcon.class.getResource("/imageButtoo.jpg");
        Icon icon = new ImageIcon(url);
        setLayout(new GridLayout(3, 2, 5, 5));          //设置网格布局管理器
        Container c = getContentPane();                 //创建容器
        for (int i = 0; i < 5; i++) {
            //创建按钮，同时设置按钮文字与图标
            JButton J = new JButton("button" + i, icon);
            c.add(J);                                   //在容器中添加按钮
            if (i % 2 == 0) {
                J.setEnabled(false);                    //设置其中一些按钮不可用
            }
        }
        JButton jb = new JButton();                     //实例化一个没有文字与图片的按钮
        jb.setMaximumSize(new Dimension(90, 30));       //设置按钮与图片相同大小
        jb.setIcon(icon);                               //为按钮设置图标
        jb.setHideActionText(true);
        jb.setToolTipText("图片按钮");                   //设置按钮提示为文字
        jb.setBorderPainted(false);                     //设置按钮边界不显示
        jb.addActionListener(new ActionListener() {     //为按钮添加监听事件
            public void actionPerformed(ActionEvent e) {
                //弹出确认对话框
                JOptionPane.showMessageDialog(null, "弹出对话框");
            }
        });
        c.add(jb);                                      //将按钮添加到容器中
        … //省略非关键代码
        }
    … //省略主方法
}
```

运行本实例，结果如图 13.12 所示。

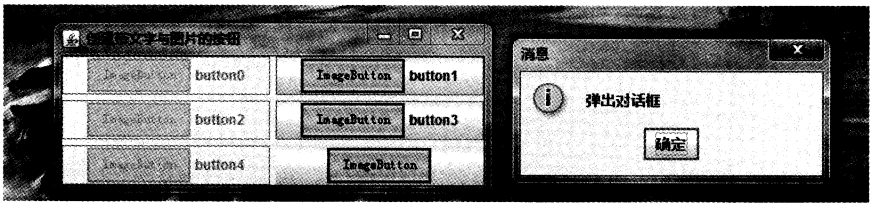

图 13.12　按钮组件的应用

在本实例中使用了两种方式创建按钮：第一种方式是在初始化按钮时赋予按钮图标与文字；另一种方式是首先创建一个没有定义图标和文字的按钮对象，然后使用 setIcon()方法为这个按钮定制一个图标，其中 setToolTipText()方法是为按钮设置提示文字，setBorderPainted()方法设置按钮边界是否显示。

最后为该按钮定制了一个鼠标单击事件，实现当用户单击该按钮时弹出提示对话框的功能。这里值得注意的一点是，使用 setMaximumSize()方法设置按钮的大小与图标的大小一致，该方法需要的参数类型为 Dimension 类对象，这样看上去此图片就如同按钮一样摆放在窗体中，同时也可以使用 setEnabled()方法设置按钮是否可用。

说明

上述这些设置按钮属性的方法多来自 JButton 的父类 AbstractButton 类，这里只是简单列举了几个常用的方法，读者如果有需要可以查询 Java API，使用自己需要的方法实现相应的功能。

13.6.2 单选按钮组件

视频讲解：光盘\TM\lx\13\单选按钮组件.exe

在默认情况下，单选按钮（JRadioButton）显示一个圆形图标，并且通常在该图标旁放置一些说明性文字，而在应用程序中，一般将多个单选按钮放置在按钮组中，使这些单选按钮表现出某种功能，当用户选中某个单选按钮后，按钮组中其他按钮将被自动取消。单选按钮是 Swing 组件中 JRadioButton 类的对象，该类是 JToggleButton 的子类，而 JToggleButton 类又是 AbstractButton 类的子类，所以控制单选按钮的诸多方法都是 AbstractButton 类中的方法。

1. 单选按钮

可以使用 JRadioButton 类中的构造方法创建单选按钮对象。JRadioButton 类的常用构造方法主要有以下几种形式。

- ☑ public JRadioButton()。
- ☑ public JRadioButton(Icon icon)。
- ☑ public JRadioButton(Icon icon,boolean selected)。
- ☑ public JRadioButton(String text)。
- ☑ public JRadioButton(String text,Icon icon)。
- ☑ public JRadioButton(String text,Icon icon,boolean selected)。

根据上述构造方法的形式，可以知道在初始化单选按钮时，可以同时设置单选按钮的图标、文字以及默认是否被选中等属性。

2. 按钮组

在 Swing 中存在一个 ButtonGroup 类，用于产生按钮组，如果希望将所有的单选按钮放置在按钮组中，需要实例化一个 JRadioButton 对象，并使用该对象调用 add()方法添加单选按钮。

【例 13.12】 在应用程序窗体中定义一个单选按钮组。

```
JRadioButton jr1 = new JRadioButton();
JRadioButton jr2 = new JRadioButton();
JRadioButton jr3 = new JRadioButton();
ButtonGroup group = new ButtonGroup();
```

```
group.add(jr1);
group.add(jr2);
group.add(jr3);
```

从上述代码中可以看出，单选按钮与提交按钮的用法基本类似，只是实例化单选按钮对象后需要将其添加至按钮组中。

13.6.3　复选框组件

视频讲解：光盘\TM\lx\13\复选框组件.exe

复选框（JCheckBox）在 Swing 组件中的使用也非常广泛，它具有一个方块图标，外加一段描述性文字。与单选按钮唯一不同的是，复选框可以进行多选设置，每一个复选框都提供“选中”与“不选中”两种状态。复选框用 JCheckBox 类的对象表示，它同样继承于 AbstractButton 类，所以复选框组件的属性设置也来源于 AbstractButton 类。

JCheckBox 的常用构造方法如下：

☑ public JCheckBox()。

☑ public JCheckBox(Icon icon,boolean checked)。

☑ public JCheckBox(String text,boolean checked)。

复选框与其他按钮设置基本相同，除了可以在初始化时设置图标之外，还可以设置复选框的文字是否被选中。

下面来看一个实例，在这个实例中笔者将滚动面板与复选框结合使用。

【例 13.13】 在项目中创建 CheckBoxTest 类，该类继承 JFrame 类成为窗体组件，在类中设置窗体使用边界布局管理器，为窗体添加多个复选框对象。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.12**）

```
import …;
public class CheckBoxTest extends JFrame{
    …//省略非关键代码
    public CheckBoxTest(){
        …//省略非关键代码
        c.setLayout(new BorderLayout());
        c.add(panel1, BorderLayout.NORTH);
        final JScrollPane scrollPane = new JScrollPane(jt);
        panel1.add(scrollPane);
        c.add(panel2, BorderLayout.SOUTH);
        panel2.add(jc1);
        jc1.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jt.append("复选框 1 被选中\n");
            }
        });
        …//省略其他复选框监听事件
    }
…//省略主方法
}
```

运行本实例，结果如图 13.13 所示。

本实例中的窗体使用了边界布局管理器，将编译器放置在面板中置于窗体的最北端，同时将 3 个复选框放置在面板中置于窗体的最南端（带滚动条的编译器在 13.6.2 节中已经讲解过，这里不再赘述）。使用 JCheckBox 类中的构造方法实例化 3 个复选框对象，将这 3 个复选框放置在面板中，分别为这 3 个复选框设置监听事件，当用户选中某个复选框时，相应文本框将显示相关内容，这里使用的是 JTextArea 类中的 append()方法为文本域添加文字。

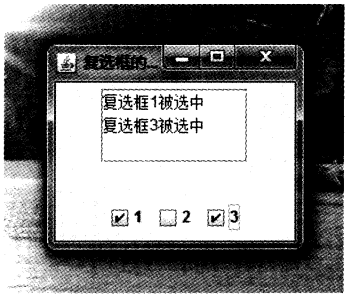

图 13.13　复选框的应用

13.7　列 表 组 件

Swing 中提供两种列表组件，分别为下拉列表框与列表框。下拉列表框与列表框都是带有一系列项目的组件，用户可以从中选择需要的项目。列表框较下拉列表框更直观，它将所有的项目罗列在列表框中；但下拉列表框较列表框更为便捷、美观，它将所有的项目隐藏起来，当用户选用其中的项目时才会显现出来。本节将详细讲解列表框与下拉列表框的应用。

13.7.1　下拉列表框组件

视频讲解：光盘\TM\lx\13\下拉列表框组件.exe

1．JComboBox 类

初次使用 Swing 中的下拉列表框时，会感觉到该类下拉列表框与 Windows 操作系统中的下拉列表框有一些相似，实质上两者并不完全相同，因为 Swing 中的下拉列表框不仅可以供用户从中选择项目，也提供编辑项目中内容的功能。

下拉列表框是一个带条状的显示区，它具有下拉功能。在下拉列表框的右方存在一个倒三角形的按钮，当用户单击该按钮时，下拉列表框中的项目将会以列表形式显示出来。

Swing 中的下拉列表框使用 JComboBox 类对象来表示，它是 javax.swing.JComponent 类的子类。它的常用构造方法如下：

- ☑ public JComboBox()。
- ☑ public JComboBox(ComboBoxModel dataModel)。
- ☑ public JComboBox(Object[] arrayData)。
- ☑ public JComboBox(Vector vector)。

在初始化下拉列表框时，可以选择同时指定下拉列表框中的项目内容，也可以在程序中使用其他方法设置下拉列表框中的内容，下拉列表框中的内容可以被封装在 ComboBoxModel 类型、数组或 Vector 类型中。

2．JComboBox 模型

在开发程序中，一般将下拉列表框中的项目封装为 ComboBoxModel 的情况比较多。ComboBoxModel 为接口，它代表一般模型，可以自定义一个类实现该接口，然后在初始化 JComboBox 对象时向上转型为 ComboBoxModel 接口类型，但是必须实现以下两种方法：

☑ public void setSelectedItem(Object item)。

☑ public Object getSelectedItem()。

其中，setSelectedItem()方法用于设置下拉列表框中的选中项，getSelectedItem()方法用于返回下拉列表框中的选中项，有了这两个方法，就可以轻松地对下拉列表框中的项目进行操作。

自定义这个类除了实现该接口之外，还可以继承 AbstractListModel 类，在该类中也有两个操作下拉列表框的重要方法。

☑ getSize()：返回列表的长度。

☑ getElementAt(int index)：返回指定索引处的值。

下面来看一个使用 JComboBox 模型的实例。

【例 13.14】 在项目中创建 JComboBoxModelTest 类，使该类继承 JFrame 类成为窗体组件，在类中创建下拉列表框，并添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.13**）

```
import …;
public class JComboBoxModelTest extends JFrame{
    JComboBox<String> jc = new JComboBox<>(new MyComboBox());      //此处应用了 JDK 7 的新特性
    JLabel jl=new JLabel("请选择证件:");
    public JComboBoxModelTest(){
        …//省略非关键代码
        cp.setLayout(new FlowLayout());
        cp.add(jl);
        cp.add(jc);
    }
    …//省略主方法
}
class MyComboBox extends AbstractListModel<String> implements ComboBoxModel<String> {
    String selecteditem=null;
    String[] test={"身份证","军人证","学生证","工作证"};
    public Object getElementAt(int index){          //根据索引返回值
        return test[index];
    }
    public int getSize(){                           //返回下拉列表框中项目的数目
        return test.length;
    }
    public void setSelectedItem(Object item){       //设置下拉列表框项目
        selecteditem=(String)item;
    }
    public Object getSelectedItem(){                //获取下拉列表框中的项目
        return selecteditem;
    }
    …//省略非关键代码
}
```

运行本实例，结果如图 13.14 所示。

在本实例中，笔者自定义了一个实现 ComboBoxModel 接口并继承 AbstractListModel 类的类，这样这个类就可以实现或重写该接口与该类中的重要方法，同时在定义下拉列表框时，只要将该类向上转型为 ComboBoxModel 接口即可。

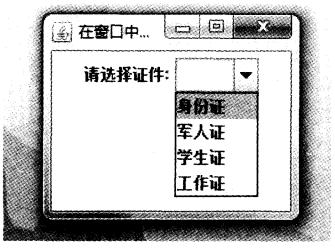

图 13.14　下拉列表框的应用

13.7.2　列表框组件

视频讲解：光盘\TM\lx\13\列表框组件.exe

列表框（JList）与下拉列表框的区别不仅表现在外观上，当激活下拉列表框时，还会出现下拉列表框中的内容；但列表框只是在窗体上占据固定的大小，如果需要列表框具有滚动效果，可以将列表框放入滚动面板中。用户在选择列表框中的某一项时，按住 Shift 键并选择列表框中的其他项目，则当前选项和其他项目之间的选项将全部被选中；也可以按住 Ctrl 键并单击列表框中的单个项目，这样可以使列表框中被单击的项目反复切换非选择状态或选择状态。

Swing 中使用 JList 类对象来表示列表框，下面列举几个常用的构造方法。

- ☑ public void JList()。
- ☑ public void JList(Object[] listData)。
- ☑ public void JList(Vector listData)。
- ☑ public void JList(ListModel dataModel)。

在上述构造方法中，存在一个没有参数的构造方法，可以通过在初始化列表框后使用 setListData() 方法对列表框进行设置，也可以在初始化的过程中对列表框中的项目进行设置。设置的方式有 3 种类型，包括数组、Vector 类型和 ListModel 模型。

当使用数组作为构造方法的参数时，首先需要创建列表项目的数组，然后再利用构造方法来初始化列表框。

【例 13.15】 使用数组作为初始化列表框的参数。

```
String[] contents={"列表 1","列表 2","列表 3","列表 4"};
JList jl=new JList(contents);
```

如果使用上述构造方法中的第 3 个构造方法，将 Vector 类型的数据作为初始化 JList 组件的参数，通常可以使用例 13.16 中的代码。

【例 13.16】 使用 Vector 类型数据作为初始化列表框的参数。

```
Vector contents=new Vector();
JList jl=new JList(contents);
contents.add("列表 1");
contents.add("列表 2");
contents.add("列表 3");
contents.add("列表 4");
```

如果使用 ListModel 模型为参数，需要创建 ListModel 对象。ListModel 是 Swing 包中的一个接口，它提供了获取列表框属性的方法。但是在通常情况下，为了使用户不完全实现 ListModel 接口中的方法，通常自定义一个类继承实现该接口的抽象类 AbstractListModel。在这个类中提供了 getElementAt()与 getSize()方法，其中 getElementAt()方法代表根据项目的索引获取列表框中的值，而 getSize()方法用于获取列表框中的项目个数。例 13.17 描述了使用第 4 种构造方法初始化列表框的基本方法。

【例 13.17】 在项目中创建 JListTest 类，使该类继承 JFrame 类成为窗体组件，在该类中创建列表框，并添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.14**）

```
public class JListTest extends JFrame{
    public JListTest(){
        Container cp=getContentPane();
        cp.setLayout(null);
        JList<String> jl = new JList<>(new MyListModel()); //此处应用了 JDK7 的新特性
        JScrollPane js=new JScrollPane(jl);
        js.setBounds(10, 10, 100, 100);
        cp.add(js);
        …//省略非关键代码
    }
    …//省略主方法
}
class MyListModel extends AbstractListModel<String> {        //继承抽象类 AbstractListModel
    //设置列表框内容
private String[] contents={"列表 1","列表 2","列表 3","列表 4","列表 5","列表 6"};
    public String getElementAt(int x){                        //重写 getElementAt()方法
        if(x<contents.length)
            return contents[x++];
        else
            return null;
    }
    public int getSize() {                                    //重写 getSize()方法
        return contents.length;
    }
}
```

运行本实例，结果如图 13.15 所示。

图 13.15　列表框的使用

除了可以使用例 13.17 中的方式创建列表框之外，还可以使用 DefaultListModel 类创建列表框，该类扩展了 AbstractListModel 类，所以也可以通过 DefaultListModel 对象向上转型为 ListModel 接口初始化列表框，同时 DefaultListModel 类提供 addElement()方法实现将内容添加至列表框中。

【例 13.18】 使用 DefaultListModel 类创建列表框。

```
final String[] flavors={"列表 1","列表 2","列表 3","列表 4","列表 5","列表 6"};
final DefaultListModel iItems=new DefaultListModel();
final JList lst=new JList(iItems);              //实例化 JList 对象
for(int i=0;i<4;i++){
```

```
        iItems.addElement(flavors[i]);          //为模型添加内容
    }
```

13.8 文本组件

文本组件在实际项目开发中使用最为广泛，尤其是文本框与密码框组件。通过文本组件可以很轻松地处理单行文字、多行文字、口令字段。本节将探讨文本组件的定义以及使用。

13.8.1 文本框组件

视频讲解：光盘\TM\lx\13\文本框组件.exe

文本框（JTextField）用来显示或编辑一个单行文本，在 Swing 中通过 javax.swing.JTextField 类对象创建，该类继承了 javax.swing.text.JTextComponent 类。下面列举了一些创建文本框常用的构造方法。

- ☑ public JTextField()。
- ☑ public JTextField(String text)。
- ☑ public JTextField(int fieldwidth)。
- ☑ public JTextField(String text,int fieldwidth)。
- ☑ public JTextField(Document docModel,String text,int fieldWidth)。

从上述构造方法可以看出，定义 JTextField 组件很简单，可以通过在初始化文本框时设置文本框的默认文字、文本框的长度等实现。

下面来看一个关于文本框的实例。

【例 13.19】 在项目中创建 JTextFieldTest 类，使该类继承 JFrame 类成为窗体组件，在该类中创建文本框和按钮组件，并添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.15**）

```
import …;
public class JTextFieldTest extends JFrame{
    public JTextFieldTest(){
        …//省略非关键代码
        final JTextField jt=new JTextField("aaa",20);
        final JButton jb=new JButton("清除");
        …//省略非关键代码
        jt.addActionListener(new ActionListener(){      //为文本框添加事件
            public void actionPerformed(ActionEvent arg0) {
                jt.setText("触发事件");                  //设置文本框中的值
            }
        });
        jb.addActionListener(new ActionListener(){      //为按钮添加事件
            public void actionPerformed(ActionEvent arg0) {
                jt.setText("");                          //将文本框置空
                jt.requestFocus();                       //焦点回到文本框
```

```
            }
        });
    }
    …//省略主方法
}
```

运行本实例，结果如图 13.16 所示。

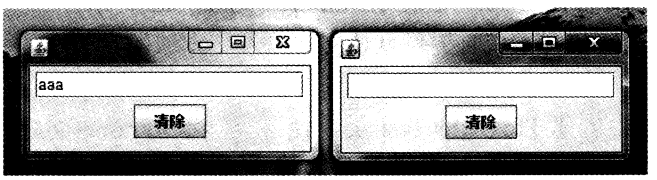

图 13.16　按钮控制文本框中的值

在本实例的窗体中主要设置一个文本框和一个按钮，然后分别为文本框和按钮设置事件，当用户将光标焦点落于文本框中并按下 Enter 键时，文本框将执行 actionPerformed()方法中设置的操作。同时还为按钮添加了相应的事件，当用户单击“清除”按钮时，文本框中的字符串将被清除。

13.8.2　密码框组件

视频讲解：光盘\TM\lx\13\密码框组件.exe

密码框（JPasswordField）与文本框的定义与用法基本相同，唯一不同的是密码框将用户输入的字符串以某种符号进行加密。密码框对象是通过 javax.swing.JPasswordField 类来创建的，JPasswordField 类的构造方法与 JTextField 类的构造方法非常相似。下面列举几个常用的构造方法。

- ☑ public JPasswordField()。
- ☑ public JPasswordFiled(String text)。
- ☑ public JPasswordField(int fieldwidth)。
- ☑ public JPasswordField(String text,int fieldwidth)。
- ☑ public JPasswordField(Document docModel,String text,int fieldWidth)。

【例 13.20】 在程序中定义密码框。

```
JPasswordField jp=new JPasswordField();
jp.setEchoChar('#');                    //设置回显字符
```

在 JPasswordField 类中提供一个 setEchoChar()方法，可以改变密码框的回显字符。

13.8.3　文本域组件

视频讲解：光盘\TM\lx\13\文本域组件.exe

在 13.6.2 小节中曾讲述过文本域（JTextArea）这一组件，其使用方法也非常简单。它在程序中接受用户的多行文字输入。

Swing 中任何一个文本区域都是 JTextArea 类型的对象。JTextArea 常用的构造方法如下：

☑ public JTextArea()。

☑ public JTextArea(String text)。

☑ public JTextArea(int rows,int columns)。

☑ public JTextArea(Document doc)。

☑ public JTextArea(Document doc,String Text,int rows,int columns)。

上述构造方法可以在初始化文本域时提供默认文本以及文本域的长与宽。

下面来看一个实例。

【例 13.21】 在项目中创建 JTextAreaTest 类，使该类继承 JFrame 类成为窗体组件，在该类中创建 JTextArea 组件的实例，并添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.16**）

```
public class JTextAreaTest extends JFrame{
    public JTextAreaTest(){
        setSize(200,100);
        setTitle("定义自动换行的文本域");
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        Container cp=getContentPane();
        JTextArea jt=new JTextArea("文本域",6,6);
        jt.setLineWrap(true);//可以自动换行
        cp.add(jt);
        setVisible(true);
    }
    public static void main(String[] args) {
        new JTextAreaTest();
    }
}
```

运行本实例，结果如图 13.17 所示。

图 13.17 定义文本域

JTextArea 类中存在一个 setLineWrap()方法，该方法用于设置文本域是否可以自动换行，如果将该方法的参数设置为 true，文本域将自动换行，否则不自动换行。

13.9 常用事件监听器

前文中一直在讲解组件，这些组件本身并不带有任何功能。例如，在窗体中定义一个按钮，当用户单击该按钮时，虽然按钮可以凹凸显示，但在窗体中并没有实现任何功能。这时需要为按钮添加特

定事件监听器，该监听器负责处理用户单击按钮后实现的功能。本节将着重讲解 Swing 中常用的两个事件监听器，即动作事件监听器与焦点事件监听器。

13.9.1　监听事件简介

视频讲解：光盘\TM\lx\13\监听事件简介.exe

在 Swing 事件模型中由 3 个分离的对象完成对事件的处理，分别为事件源、事件以及监听程序。事件源触发一个事件，它被一个或多个“监听器”接收，监听器负责处理事件。

所谓事件监听器，实质上就是一个“实现特定类型监听器接口”的类对象。具体地说，事件几乎都以对象来表示，它是某种事件类的对象，事件源（如按钮）会在用户做出相应的动作（如按钮被按下）时产生事件对象，如动作事件对应 ActionEvent 类对象，同时要编写一个监听器的类必须实现相应的接口，如 ActionEvent 类对应的是 ActionListener 接口，需要获取某个事件对象就必须实现相应的接口，同时需要将接口中的方法一一实现。最后事件源（按钮）调用相应的方法加载这个“实现特定类型监听器接口”的类对象，所有的事件源都具有 addXXXListener()和 removeXXXListener()方法（其中“XXX”表示监听事件类型），这样就可以为组件添加或移除相应的事件监听器。

13.9.2　动作事件监听器

视频讲解：光盘\TM\lx\13\动作事件监听器.exe

动作事件（ActionEvent）监听器是 Swing 中比较常用的事件监听器，很多组件的动作都会使用它监听，如按钮被单击。表 13.3 描述了动作事件监听器的接口与事件源。

表 13.3　动作事件监听器

事 件 名 称	事　件　源	监 听 接 口	添加或删除相应类型监听器的方法
ActionEvent	JButton、JList、JTextField 等	ActionListener	addActionListener()、removeActionListener()

下面以单击按钮事件为例来说明动作事件监听器，当用户单击按钮时，将触发动作事件。例 13.22 演示了按钮被按下时产生的事件处理。

【例 13.22】 在项目中创建 SimpleEvent 类，使该类继承 JFrame 类成为窗体组件，在类中创建按钮组件，为按钮组件添加动作监听器，然后将按钮组件添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.17**）

```
public class SimpleEvent extends JFrame{
    private JButton jb=new JButton("我是按钮，单击我");
    public SimpleEvent(){
        setLayout(null);
        …//省略非关键代码
        cp.add(jb);
        jb.setBounds(10, 10,100,30);
        //为按钮添加一个实现 ActionListener 接口的对象
```

```
        jb.addActionListener(new jbAction());
    }
    //定义内部类实现 ActionListener 接口
    class jbAction implements ActionListener{
        //重写 actionPerformed()方法
        public void actionPerformed(ActionEvent arg0) {
            jb.setText("我被单击了");
        }
    }
    …//省略主方法
}
```

运行本实例，结果如图 13.18 所示。

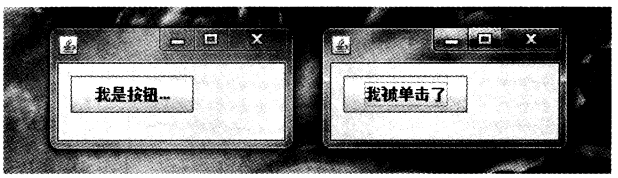

图 13.18　动作事件的应用

在本实例中，为按钮设置了动作监听器。由于获取事件监听时需要获取实现 ActionListener 接口的对象，所以笔者定义了一个内部类 jbAction 实现 ActionListener 接口，同时在该内部类中实现了 actionPerformed()方法，也就是在 actionPerformed()方法中定义当用户单击该按钮后实现怎样的功能。

也许有的读者会产生这样的疑问，难道一定要使用内部类来完成事件监听吗？或许可以使用 SimpleEvent 类实现 ActionListener 接口，或者在获取其他事件的同时实现其他接口，如例 13.23 中使用的方法。

【例 13.23】 在 SimpleEvent 类中，不使用内部类实现事件监听。本实例关键代码如下：

```
//实现 ActionListener 接口
public class SimpleEvent extends JFrame implements ActionListener{
    private JButton jb=new JButton("我是按钮，单击我");
    public SimpleEvent(){
        …//省略非关键代码
        cp.add(jb);
        jb.addActionListener(this); //添加本类对象
    }
    //重写 actionPerformed()方法
    public void actionPerformed(ActionEvent arg0){
        jb.setText("我被单击了");
    }
    …//省略主方法
}
```

显然，上述代码在编译器中不会报错。如果再定义一个按钮对象 jb2，并为该按钮也设置一个监听事件，这个监听事件与按钮对象 jb 不同，所以也需要重写 actionPerformed()方法，那么可以在同一个类中重写两次 actionPerformed()方法吗？这样是不可以的。所以为事件源做监听事件时，使用内部类的方式来解决这个问题。

说明

一般情况下，为事件源做监听事件应使用匿名内部类形式，如果读者对这方面的知识不熟悉，可以参看第 11 章的内容。

13.9.3　焦点事件监听器

视频讲解：光盘\TM\lx\13\焦点事件监听器.exe

焦点事件（FocusEvent）监听器在实际项目开发中应用也比较广泛，如将光标焦点离开一个文本框时需要弹出一个对话框，或者将焦点返回给该文本框等。焦点事件监听器的相关内容如表 13.4 所示。

表 13.4　焦点事件监听器

事 件 名 称	事　件　源	监 听 接 口	添加或删除相应类型监听器的方法
FocusEvent	Component 以及派生类	FocusListener	addFocusListener()、removeFocusListener()

下面来看一个焦点事件的实例，当用户将焦点离开文本框时，将弹出相应对话框。

【例 13.24】 在项目中创建 FocusEventTest 类，使该类继承 JFrame 类成为窗体组件，在类中创建文本框组件，并为文本框添加焦点事件监听器，将文本框组件添加到窗体中。本实例关键代码如下：（**实例位置：光盘\TM\sl\13.18**）

```
public class FocusEventTest extends JFrame{
    public FocusEventTest() {
        …//省略非关键代码
        JTextField jt=new JTextField("请单击其他文本框",10);    //创建一个文本框
        JTextField jt2=new JTextField("请单击我",10);           //创建另外一个文本框
        cp.add(jt);
        cp.add(jt2);
        jt.addFocusListener(new FocusListener(){
            //组件失去焦点时调用的方法
            public void focusLost(FocusEvent arg0) {
                JOptionPane.showMessageDialog(null, "文本框失去焦点");
            }
            //组件获取焦点时调用的方法
            public void focusGained(FocusEvent arg0) {
            }
        });

    }
    …//省略主方法
}
```

运行本实例，结果如图 13.19 所示。

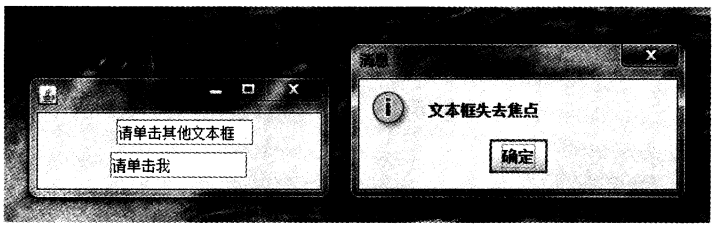

图 13.19　焦点事件的应用

在本实例中，为文本框组件添加了焦点事件监听器。这个监听需要实现 FocusListener 接口。在该接口中定义了两个方法，分别为 focusLost()与 focusGained()方法，其中 focusLost()方法是在组件失去焦点时调用的，而 focusGained()方法是在组件获取焦点时调用的。由于本实例需要实现在文本框失去焦点时弹出相应对话框的功能，所以重写 focusLost()方法，同时在为文本框添加监听时使用了匿名内部类的形式，将实现 FocusListener 接口对象传递给 addFocusListener()方法。

13.10　小　　结

本章主要讲解了 Swing 的基本要素，包括各种常用的组件、窗体、布局、事件监听器等。通过对本章的学习，读者应该能够开发带 GUI 界面的应用程序窗体，灵活运用各种组件完善窗体的功能，并实现组件的各种事件处理。

13.11　实践与练习

1．尝试开发一个窗体，如图 13.20 所示。（**答案位置：光盘\TM\sl\13.19**）

图 13.20　窗体

2．尝试创建一个窗体，选择合适的布局管理器，并在窗体中设置一个下拉列表框，初始状态下拉列表框中没有项目，并设置一个按钮，为按钮设置动作事件监听器，当用户单击该按钮时，下拉列表框中相应添加数组中的内容。（**答案位置：光盘\TM\sl\13.20**）

3．尝试开发一个登录窗体，包括用户名、密码以及提交按钮和重置按钮，当用户输入用户名 mr、密码 mrsoft 时，弹出登录成功提示对话框。（**答案位置：光盘\TM\sl\13.21**）

第14章

集合类

（视频讲解：13分钟）

集合可以看作是一个容器，如红色的衣服可以看作是一个集合，所有 Java 类的书也可以看作是一个集合。对于集合中的各个对象很容易将其存放到集合中，也很容易将其从集合中取出来，还可以将其按照一定的顺序进行摆放。Java 中提供了不同的集合类，这些类具有不同的存储对象的方式，并提供了相应的方法以方便用户对集合进行遍历、添加、删除以及查找指定的对象。学习 Java 语言一定要学会使用集合。本章将介绍 Java 中的各种集合类。

通过阅读本章，您可以：

- 了解集合类的概念
- 掌握 Collection 接口
- 掌握 List 集合
- 掌握 Set 集合
- 掌握 Map 集合

14.1 集合类概述

视频讲解：光盘\TM\lx\14\集合类概述.exe

java.util 包中提供了一些集合类，这些集合类又被称为容器。提到容器不难想到数组，集合类与数组的不同之处是，数组的长度是固定的，集合的长度是可变的；数组用来存放基本类型的数据，集合用来存放对象的引用。常用的集合有 List 集合、Set 集合和 Map 集合，其中 List 与 Set 继承了 Collection 接口，各接口还提供了不同的实现类。上述集合类的继承关系如图 14.1 所示。

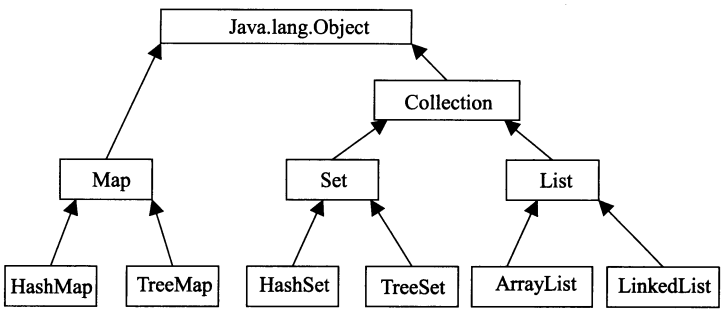

图 14.1 常用集合类的继承关系

14.2 Collection 接口

视频讲解：光盘\TM\lx\14\Collection 接口.exe

Collection 接口是层次结构中的根接口。构成 Collection 的单位称为元素。Collection 接口通常不能直接使用，但该接口提供了添加元素、删除元素、管理数据的方法。由于 List 接口与 Set 接口都继承了 Collection 接口，因此这些方法对 List 集合与 Set 集合是通用的。常用方法如表 14.1 所示。

表 14.1 Collection 接口的常用方法

方 法	功 能 描 述
add(E e)	将指定的对象添加到该集合中
remove(Object o)	将指定的对象从该集合中移除
isEmpty()	返回 boolean 值，用于判断当前集合是否为空
iterator()	返回在此 Collection 的元素上进行迭代的迭代器。用于遍历集合中的对象
size()	返回 int 型值，获取该集合中元素的个数

如何遍历集合中的每个元素呢？通常遍历集合，都是通过迭代器（Iterator）来实现。Collection 接口中的 iterator()方法可返回在此 Collection 进行迭代的迭代器。下面的实例就是典型的遍历集合的方法。

【例 14.1】 在项目中创建类 Muster，在主方法中实例化集合对象，并向集合中添加元素，最后将集合中的对象以 String 形式输出。（**实例位置：光盘\TM\sl\14.01**）

```
import java.util.*;                                    //导入 java.util 包，其他实例都要添加该语句
public class Muster {                                  //创建类 Muster
    public static void main(String args[]) {
        Collection<String> list = new ArrayList<>();   //实例化集合类对象
        list.add("a");                                 //向集合添加数据
        list.add("b");
        list.add("c");
        Iterator<String> it = list.iterator();         //创建迭代器
        while (it.hasNext()) {                         //判断是否有下一个元素
            String str = (String) it.next();           //获取集合中元素
            System.out.println(str);
        }
    }
}
```

运行结果如图 14.2 所示。

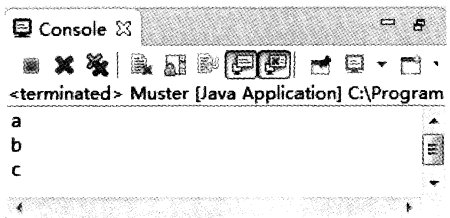

图 14.2　例 14.1 的运行结果

注意

Iterator 的 next()方法返回的是 Object。

14.3　List 集合

List 集合包括 List 接口以及 List 接口的所有实现类。List 集合中的元素允许重复，各元素的顺序就是对象插入的顺序。类似 Java 数组，用户可通过使用索引（元素在集合中的位置）来访问集合中的元素。

14.3.1　List 接口

视频讲解：光盘\TM\lx\14\List 接口.exe

List 接口继承了 Collection 接口，因此包含 Collection 中的所有方法。此外，List 接口还定义了以下两个非常重要的方法。

☑ get(int index)：获得指定索引位置的元素。

☑ set(int index,Object obj)：将集合中指定索引位置的对象修改为指定的对象。

14.3.2 List 接口的实现类

视频讲解：光盘\TM\lx\14\List 接口的实现类.exe

List 接口的常用实现类有 ArrayList 与 LinkedList。

☑ ArrayList 类实现了可变的数组，允许保存所有元素，包括 null，并可以根据索引位置对集合进行快速的随机访问；缺点是向指定的索引位置插入对象或删除对象的速度较慢。

☑ LinkedList 类采用链表结构保存对象。这种结构的优点是便于向集合中插入和删除对象，需要向集合中插入、删除对象时，使用 LinkedList 类实现的 List 集合的效率较高；但对于随机访问集合中的对象，使用 LinkedList 类实现 List 集合的效率较低。

使用 List 集合时通常声明为 List 类型，可通过不同的实现类来实例化集合。

【例 14.2】 分别通过 ArrayList、LinkedList 类实例化 List 集合，代码如下：

```
List<E> list = new ArrayList<>();
List<E> list2 = new LinkedList<>();
```

在上面的代码中，E 可以是合法的 Java 数据类型。例如，如果集合中的元素为字符串类型，那么 E 可以修改为 String。

【例 14.3】 在项目中创建类 Gather，在主方法中创建集合对象，通过 Math 类的 random()方法随机获取集合中的某个元素，然后移除数组中索引位置是“2”的元素，最后遍历数组。（**实例位置：光盘\TM\sl\14.02**）

```
public class Gather {                                    //创建类 Gather
    public static void main(String[] args) {             //主方法
        List<String> list = new ArrayList<>();           //创建集合对象
        list.add("a");                                   //向集合添加元素
        list.add("b");
        list.add("c");
        int i = (int) (Math.random()*list.size());       //获得 0~2 之间的随机数
        System.out.println("随机获取数组中的元素：" + list.get(i));
        list.remove(2);                                  //将指定索引位置的元素从集合中移除
        System.out.println("将索引是'2'的元素从数组移除后，数组中的元素是：");
        for (int j = 0; j < list.size(); j++) {          //循环遍历集合
            System.out.println(list.get(j));
        }
    }
}
```

运行结果如图 14.3 所示。Math 类的 random()方法可获得一个 0.0~1.0 之间的随机数。