第14章
Introduction to Java Programming, 8E

抽象类和接口

学习目标

- 设计和使用抽象类（14.2节）。
- 使用Calendar类和GregorianCalendar类处理日历（14.3节）。
- 使用接口指定对象共有的行动（14.4节）。
- 定义接口以及实现接口的类（14.4节）。
- 使用Comparable接口定义自然顺序（14.5节）。
- 使用ActionListener接口实现对象对动作事件的监听（14.6节）。
- 使用Cloneable接口使对象成为可克隆的（14.7节）。
- 探究抽象类和接口的相同点和不同点（14.8节）。
- 使用包装类（Byte、Short、Integer、Long、Float、Double、Character和Boolean）创建基本类型值的对象（14.9节）。
- 创建一个通用的排序方法（14.10节）。
- 使用基本类型与包装类类型之间的自动转化来简化程序设计（14.11节）。
- 使用BigInteger和BigDecimal类计算任意精度的大数字（14.12节）。
- 设计Rational类来定义Rational类型（14.13节）。

14.1 引言

你已经学习了如何编写简单的程序来创建和显示GUI组件。你能编写代码以响应像点击一个按钮这样的用户动作吗？如图14-1所示，当点击一个按钮时，控制台上就会显示一条消息。

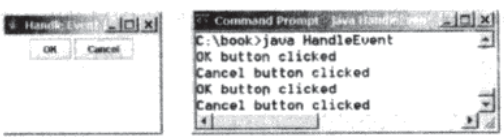


图14-1 程序响应点击按钮的行动事件

为了编写这样的代码，必须要了解接口。接口是为了定义多个类（特别是不相关的类）的共同行为。在讨论接口之前，我们介绍一个非常接近的相关主题：抽象类。

14.2 抽象类

在继承的层次结构中，随着每个新子类的出现，类会变得越来越明确和具体。如果从一个子类追溯到父类，类就会变得更通用、更加不明确。类的设计应该确保父类包含它的子类的共同特征。有时候，一个父类设计得非常抽象，以至于它都没有任何具体的实例。这样的类称为抽象类（abstract class）。

在第11章中，GeometricObject类定义成Circle类和Rectangle类的父类。GeometricObject类模拟了几何对象的共同特征。Circle类和Rectangle类都包含分别计算圆和矩形的面积和周长的方法getArea()和getPerimeter()。因为可以计算所有几何对象的面积和周长，所以最好在GeometricObject类中定义getArea()和getPerimeter()方法。但是，这些方法不能在GeometricObject类中实现，因为它们的实现取决于几何对象的具体类型。这样的方法称为抽象方法

(abstract method)，在方法头中使用abstract修饰符表示。在GeometricObject类中定义了这些方法后，GeometricObject就成为一个抽象类。在类头使用abstract修饰符表示该类为抽象类。在UML图形记号中，抽象类和抽象方法的名字用斜体表示，如图14-2所示。程序清单14-1给出了新GeometricObject类的源代码。

程序清单14-1 GeometricObject.java

```
1 public abstract class GeometricObject {
2   private String color = "white";
3   private boolean filled;
4   private java.util.Date dateCreated;
5
6   /** Construct a default geometric object */
7   protected GeometricObject() {
8     dateCreated = new java.util.Date();
9   }
10
11  /** Construct a geometric object with color and filled value */
12  protected GeometricObject(String color, boolean filled) {
13    dateCreated = new java.util.Date();
14    this.color = color;
15    this.filled = filled;
16  }
17
18  /** Return color */
19  public String getColor() {
20    return color;
21  }
22
23  /** Set a new color */
24  public void setColor(String color) {
25    this.color = color;
26  }
27
28  /** Return filled. Since filled is boolean,
29   *  the get method is named isFilled */
30  public boolean isFilled() {
31    return filled;
32  }
33
34  /** Set a new filled */
35  public void setFilled(boolean filled) {
36    this.filled = filled;
37  }
38
39  /** Get dateCreated */
40  public java.util.Date getDateCreated() {
41    return dateCreated;
42  }
43
44  /** Return a string representation of this object */
45  public String toString() {
46    return "created on " + dateCreated + "\ncolor: " + color +
47      " and filled: " + filled;
48  }
49
50  /** Abstract method getArea */
51  public abstract double getArea();
52
53  /** Abstract method getPerimeter */
54  public abstract double getPerimeter();
55 }
```

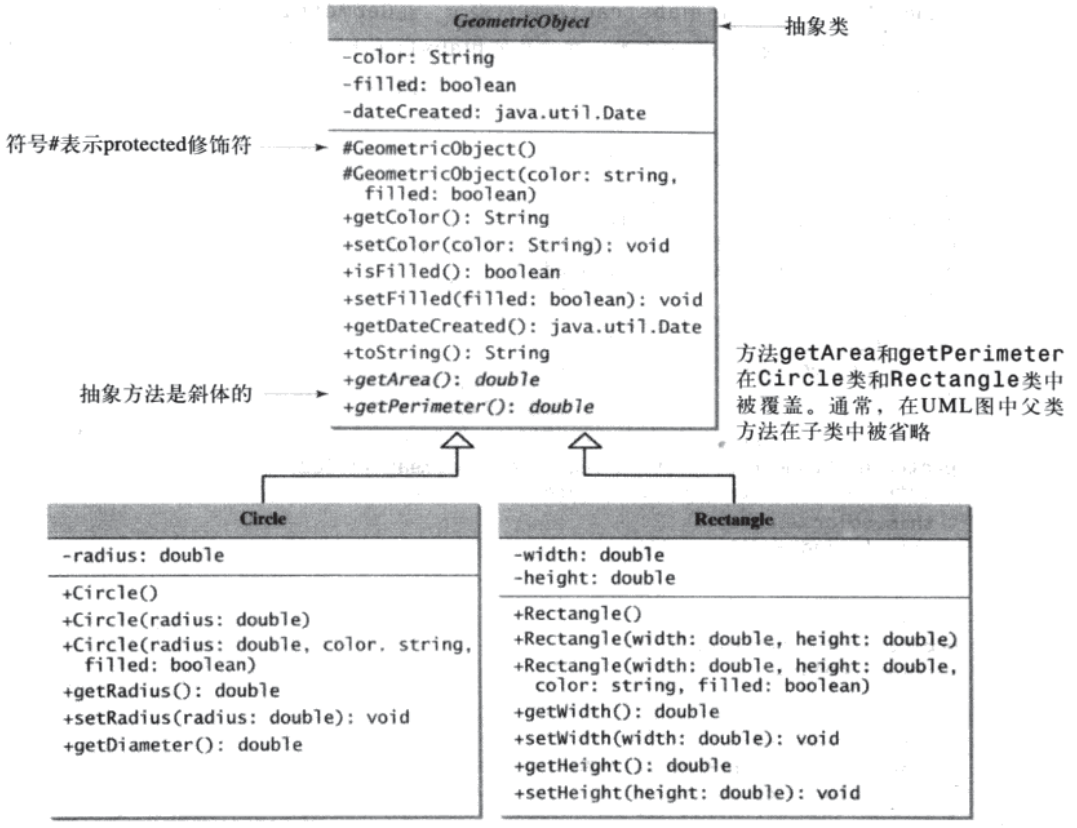


图14-2 新GeometricObject类包含抽象方法

抽象类和常规类很像，但是不能使用new操作符创建它的实例。抽象方法只有定义而没有实现。它的实现由子类提供。一个包含抽象方法的类必须声明为抽象类。

抽象类的构造方法定义为protected，因为它只被子类使用。创建一个具体子类的实例时，它的父类的构造方法被调用以初始化父类中定义的数据域。

抽象类GeometricObject为几何对象定义了共同特征（数据和方法），并且提供了正确的构造方法。因为不知道如何计算几何对象的面积和周长，所以，getArea和getPerimeter定义为抽象方法。这些方法在子类中实现。Circle类和Rectangle类的实现除了扩展本章定义的GeometricObject类之外，其他的都是同程序清单11-2和程序清单11-3一样的。

程序清单14-2 Circle.java

```
1 public class Circle extends GeometricObject {
2   // Same as lines 2-46 in Listing 11.2, so omitted
3 }
```

程序清单14-3 Rectangle.java

```
1 public class Rectangle extends GeometricObject {
2   // Same as lines 2-49 in Listing 11.3, so omitted
3 }
```

14.2.1 为什么要用抽象方法

你可能会疑惑在GeometricObject类中定义方法getArea和getPerimeter为抽象的而不是在每个子类中定义它们会有什么好处。下面的例子就能看出在GeometricObject中定义它们的好处。

程序清单14-4中的例子创建了两个几何对象：一个圆和一个矩形，调用equalArea方法来检查它们

的面积是否相同，然后调用displayGeometricObject方法来显示它们。

程序清单14-4 TestGeometricObject.java

```
1 public class TestGeometricObject {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create two geometric objects
5     GeometricObject geoObject1 = new Circle(5);
6     GeometricObject geoObject2 = new Rectangle(5, 3);
7
8     System.out.println("The two objects have the same area? " +
9       equalArea(geoObject1, geoObject2) );
10
11    // Display circle
12    displayGeometricObject(geoObject1);
13
14    // Display rectangle
15    displayGeometricObject(geoObject2);
16  }
17
18  /** A method for comparing the areas of two geometric objects */
19  public static boolean equalArea(GeometricObject object1,
20      GeometricObject object2) {
21    return object1.getArea() == object2.getArea();
22  }
23
24  /** A method for displaying a geometric object */
25  public static void displayGeometricObject(GeometricObject object) {
26    System.out.println();
27    System.out.println("The area is " + object.getArea());
28    System.out.println("The perimeter is " + object.getPerimeter());
29  }
30 }
```

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 14.0
The perimeter is 16.0
```

Circle类和Rectangle类中覆盖了定义在GeometricObject类中的getArea()和getPerimeter()方法。语句（第5～6行）：

```
GeometricObject geoObject1 = new Circle(5);
GeometricObject geoObject2 = new Rectangle(5, 3);
```

创建了一个新圆和一个新矩形，并把它们赋值给变量geoObject1和geoObject2。这两个变量都是GeometricObject类型的。

当调用equalArea(geoObject1,geoObject2)时（第9行），由于geoObject1是一个圆，所以object1.getArea()使用的是Circle类定义的getArea()方法，而geoObject2是一个矩形，所以object2.getArea()使用的是Rectangle类的getArea()方法。

类似地，当调用displayGeometricObject(geoObject1)时（第12行），使用在Circle类中定义的getArea和getPerimeter方法，而当调用displayGeometricObject(geoObject2)（第15行）时，使用的是在Rectangle类中定义的getArea和getPerimeter方法。JVM在运行时根据对象的类型动态地决定调用哪一个方法。

注意 如果GeometricObject里没有定义getArea方法，就不能在该程序中定义equalArea方法来计算这两个几何对象的面积是否相同。所以，现在可以看出在GeometricObject中定义抽象方法的好处。

14.2.2 关于抽象类的几个关注点

下面是关于抽象类的值得注意的几点：

- 抽象方法不能包含在非抽象类中。如果抽象父类的子类不能实现所有的抽象方法，那么子类也必须定义为抽象的。换句话说，在抽象类扩展的非抽象子类中，必须实现所有的抽象方法。还要注意到，抽象方法是非静态的。
- 抽象类是不能使用new操作符来初始化的。但是，仍然可以定义它的构造方法，这个构造方法在它的子类的构造方法中调用。例如，GeometricObject类的构造方法在Circle类和Rectange类中调用。
- 包含抽象对象的类必须是抽象的。但是，可以定义一个不包含抽象方法的抽象类。在这种情况下，不能使用new操作符创建该类的实例。这种类是用来定义新子类的基类的。
- 即使子类的父类是具体的，这个子类也可以是抽象的。例如，Object类是具体的，但是它的子类如GeometricObject可以是抽象的。
- 子类可以覆盖父类的方法并将它定义为abstract。这是很少见的，但是它在当父类的方法实现在子类中变得不合法时是很有用的。在这种情况下，子类必须定义为abstract。
- 不能使用new操作符从一个抽象类创建一个实例，但是抽象类可以用作一种数据类型。因此，下面的语句是创建一个元素是GeometricObject类型的数组，这个语句是正确的：

```
GeometricObject[] objects = new GeometricObject[10];
```

然后可以创建一个GeometricObject的实例，并将它的引用赋值给数组，如下所示：

```
objects[0] = new Circle();
```

14.3 举例：日历类Calendar和公历类GregorianCalendar

一个java.util.Date的实例表示以毫秒为单位的特定时间段。java.util.Calendar是一个抽象的基类，可以提取出详细的日历信息，例如，年、月、日、小时、分钟和秒。Calendar类的子类可以实现特定的日历系统，例如，公历（Gregorian历）、阴历和犹太历。目前，Java支持公历类java.util.GregorianCalendar，如图14-3所示。Calendar类中的add方法是抽象的，因为它的实现依赖于某个具体的日历系统。

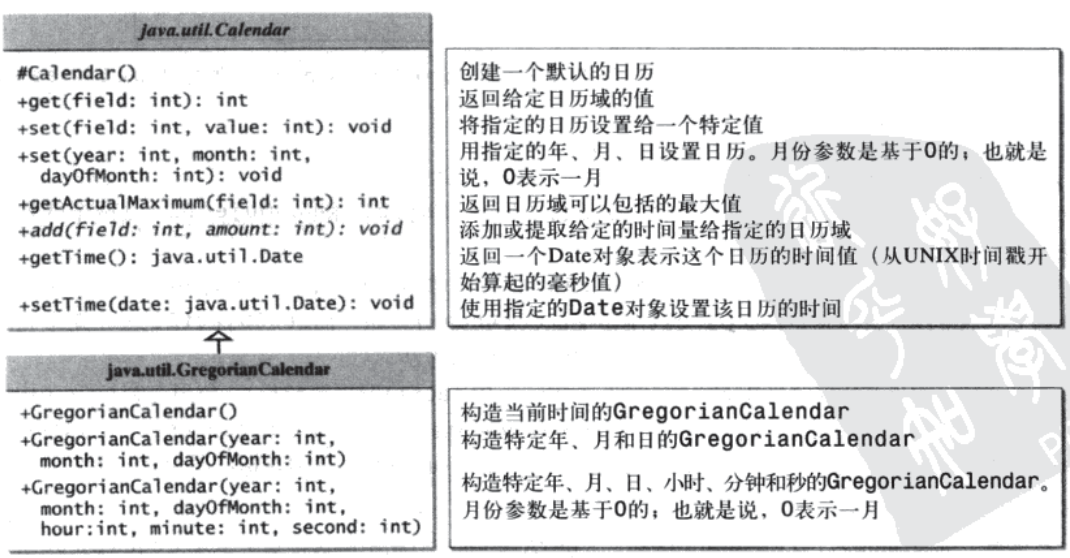


图14-3 抽象的Calendar类定义了各种日历的共同特点

可以使用new GregorianCalendar()利用当前时间构造一个默认的GregorianCalendar对象，可以使用new GregorianCalendar(year,month,date)利用指定的年year、月month和日date构造一个GregorianCalendar对象。参数month是基于0的，即0就是1月（January）。

在Calendar类中定义的get(int field)方法在从Calendar类中提取日期和时间信息方面是很有用的。日期和时间域都被定义为常量，如表14-1所示。

表14-1 Calendar类的域常量

常 量	说 明
YEAR	日历的年份
MONTH	日历的月份，0表示一月
DATE	日历的天
HOUR	日历的小时（12小时制）
HOUR_OF_DAY	日历的小时（24小时制）
MINUTE	日历的分钟
SECOND	日历的秒
DAY_OF_WEEK	一周的天数，1是星期日
DAY_OF_MONTH	和DATE一样
DAY_OF_YEAR	当前年的天数，1是一年的第一天
WEEK_OF_MONTH	当前月内的星期数，1是该月的第一个星期
WEEK_OF_YEAR	当前年内的星期数，1是该年的第一个星期
AM_PM	表明是上午还是下午（0表示上午，1表示下午）

程序清单14-5给出的例子显示当前时间的日期和时间信息。

程序清单14-5 TestCalendar.java

```
1 import java.util.*;
2
3 public class TestCalendar {
4   public static void main(String[] args) {
5     // Construct a Gregorian calendar for the current date and time
6     Calendar calendar = new GregorianCalendar();
7     System.out.println("Current time is " + new Date());
8     System.out.println("YEAR:\t" + calendar.get(Calendar.YEAR));
9     System.out.println("MONTH:\t" + calendar.get(Calendar.MONTH));
10    System.out.println("DATE:\t" + calendar.get(Calendar.DATE));
11    System.out.println("HOUR:\t" + calendar.get(Calendar.HOUR));
12    System.out.println("HOUR_OF_DAY:\t" +
13      calendar.get(Calendar.HOUR_OF_DAY));
14    System.out.println("MINUTE:\t" + calendar.get(Calendar.MINUTE));
15    System.out.println("SECOND:\t" + calendar.get(Calendar.SECOND));
16    System.out.println("DAY_OF_WEEK:\t" +
17      calendar.get(Calendar.DAY_OF_WEEK));
18    System.out.println("DAY_OF_MONTH:\t" +
19      calendar.get(Calendar.DAY_OF_MONTH));
20    System.out.println("DAY_OF_YEAR: " +
21      calendar.get(Calendar.DAY_OF_YEAR));
22    System.out.println("WEEK_OF_MONTH: " +
23      calendar.get(Calendar.WEEK_OF_MONTH));
24    System.out.println("WEEK_OF_YEAR: " +
25      calendar.get(Calendar.WEEK_OF_YEAR));
26    System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
27
28    // Construct a calendar for September 11, 2001
29    Calendar calendar1 = new GregorianCalendar(2001, 8, 11);
30    System.out.println("September 11, 2001 is a " +
31      dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)));
32  }
```

```
33
34   public static String dayNameOfWeek(int dayOfWeek) {
35     switch (dayOfWeek) {
36       case 1: return "Sunday";
37       case 2: return "Monday";
38       case 3: return "Tuesday";
39       case 4: return "Wednesday";
40       case 5: return "Thursday";
41       case 6: return "Friday";
42       case 7: return "Saturday";
43       default: return null;
44     }
45   }
46 }
```

```
Current time is Sun Sep 09 21:23:59 EDT 2007
YEAR:               2007
MONTH:              8
DATE:               9
HOUR:               9
HOUR_OF_DAY:        21
MINUTE:             23
SECOND:             59
DAY_OF_WEEK:        1
DAY_OF_MONTH:       9
DAY_OF_YEAR:        252
WEEK_OF_MONTH:      3
WEEK_OF_YEAR:       37
AM_PM:              1
September 11, 2001 is a Tuesday
```

Calendar类中定义的set(int field,value)方法用来设置一个域。例如，可以使用calendar.set(Calendar.DAY_OF_MONTH,1)将calendar设置为当月的第一天。

add(field,value)方法为某个特定域增加指定的量。例如，add(Calendar.DAY_OF_MONTH,5)给日历的当前时间加五天，而add(Calendar.DAY_OF_MONTH,-5)从日历的当前时间减去五天。

为了获得一个月中的天数，使用calendar.getActualMaximum(Calendar.DAY_OF_MONTH)方法。例如，如果是三月的calendar，那么这个方法将返回31。

可以通过调用canlendar.setTime(date)为calendar设置一个用Date对象表示的时间，通过调用calendar.getTime()获取时间。

14.4 接口

接口（interface）是一种与类相似的结构，只包含常量和抽象方法。接口在许多方面都与抽象类很相似，但是它的目的是指明多个对象的共同行为。例如，使用正确的接口，可以指明这些对象是可比较的、可食用的或可克隆的。

为了区分接口和类，Java采用下面的语法来定义接口：

```
修饰符  interface 接口名{
/**  常量声明  */
/**  方法签名 */
}
```

下面是一个接口的例子：

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

在Java中，接口被看作是一种特殊的类。就像常规类一样，每个接口都被编译为独立的字节码文件。与抽象类相似，不能使用new操作符创建接口的实例，但是大多数情况下，使用接口或多或少有点像使用抽象类。例如，可以使用接口作为引用变量的数据类型或类型转换的结果等。

现在，可以使用Edible接口来明确一个对象是否是可食用的。这需要使用implements关键字让对象的类实现这个接口来完成。例如，程序清单14-6中的Chicken类和Fruit类（第14、23行）实现Edible接口。类和接口之间的关系称为接口继承（interface inheritance）。因为接口继承和类继承本质上是相同的，所以我们将它们都简称为继承。

程序清单14-6 TestEdible.java

```
1 public class TestEdible {
2   public static void main(String[] args) {
3     Object[] objects = {new Tiger(), new Chicken(), new Apple()};
4     for (int i = 0; i < objects.length; i++)
5       if (objects[i] instanceof Edible)
6         System.out.println(((Edible)objects[i]).howToEat());
7   }
8 }
9
10 class Animal {
11   // Data fields, constructors, and methods omitted here
12 }
13
14 class Chicken extends Animal implements Edible {
15   public String howToEat() {
16     return "Chicken: Fry it";
17   }
18 }
19
20 class Tiger extends Animal {
21 }
22
23 abstract class Fruit implements Edible {
24   // Data fields, constructors, and methods omitted here
25 }
26
27 class Apple extends Fruit {
28   public String howToEat() {
29     return "Apple: Make apple cider";
30   }
31 }
32
33 class Orange extends Fruit {
34   public String howToEat() {
35     return "Orange: Make orange juice";
36   }
37 }
```

```
Chicken: Fry it
Apple: Make apple cider
```

Chicken类扩展自Animal类，并实现Edible以表明小鸡是可食用的。当一个类实现接口时，该类实现了定义在接口中的所有带确切签名和返回类型的方法。Chicken类实现了howToEat方法（第15～17行）。

Fruit类实现Edible。因为它不实现howToEat方法，所以Fruit必须表示为abstract（第23行）。Fruit的具体子类必须实现hotToEat方法。Apple类和Orange类实现howToEat方法（第28、34行）。

main方法创建由Tiger、Chicken和Apple的三个对象构成的数组（第3行），如果这个元素是可食用的，调用howToEat方法（第6行）。

注意 由于接口中所有的数据域都是public final static而且所有的方法都是public abstract，所以Java允许忽略这些修饰符。因此，下面的接口定义是等价的：

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

等价于

```
public interface T {
  int K = 1;

  void p();
}
```

提示 接口内定义的常量可以使用语法“接口名.常量名”（例如，T.K）来访问。

14.5 举例：Comparable接口

假设要设计一个求两个相同类型对象中较大者的通用方法。这里的对象可以是两个学生、两个圆、两个矩形或者两个正方形。为了实现这个方法，这两个对象必须是可比较的。因此，这两个对象都该有的共同方法就是comparable（可比较的）。Java为此目的提供了Comparable接口。接口的定义如下所示：

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable {
  public int compareTo(Object o);
}
```

compareTo方法判断这个对象相对于给定对象o的顺序，并且当这个对象小于、等于或大于给定对象o时，分别返回负整数、0或正整数。

Java类库中的许多类（例如，String和Date）实现了Comparable接口以定义对象的自然顺序。如果你检查这些类的源代码，就会发现这些类中使用的关键字implements，如下所示：

```
public class String extends Object
    implements Comparable {
  // class body omitted
}
```

```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

这样，字符串就是可比较的，日期也是如此。假设s为String对象，d为Date对象，那么下面所有的表达式都为true：

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

从两个对象中找出最大者的通用方法max可以定义为如图a或图b所示：

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if (((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

b)

图a中的max方法比图b中的简单。在图b中的Max类中，o1被声明为Object，而（Comparable）o1告诉编译器将o1转换成Comparable，因此，可以从o1中调用compareTo方法。但是，图a中的Max类无须转换，因为o1被声明为Comparable。

图a中的max方法比图b中的更鲁棒。必须用两个可比较的对象调用max方法。假设用两个不可比较的对象调用方法max：

```
Max.max(anyObject1,anyObject2);
```

因为anyObject1不是Comparable的实例，所以编译器会发现使用图a中max方法时的错误。使用图b中的max方法，该行代码将顺利编译，但在运行时会出现ClassCastException异常，因为anyObject1不是Comparable的实例，并且不能转换成Comparable。

从现在起，假设使用的是图a中的max方法。因为字符串和日期都是可比较的，所以，可以使用max方法找出String类或Date类的两个实例中的较大者。考虑下面的例子：

```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

从max方法返回的值是Comparable类型。所以，需要将它显式地转换为String或Date类型。

不能使用max方法得到两个Rectangle实例中较大的一个，因为Rectangle类没有实现接口Comparable。然而，可以定义一个新的Rectangle类来实现Comparable。这个新类的实例是可比较的。将这个新类命名为ComparableRectangle，如程序清单14-7所示。

程序清单14-7 ComparableRectangle.java

```
1 public class ComparableRectangle extends Rectangle
2     implements Comparable {
3   /** Construct a ComparableRectangle with specified properties */
4   public ComparableRectangle(double width, double height) {
5     super(width, height);
6   }
7
8   /** Implement the compareTo method defined in Comparable */
9   public int compareTo(Object o) {
10    if (getArea() > ((ComparableRectangle)o).getArea())
11      return 1;
12    else if (getArea() < ((ComparableRectangle)o).getArea())
13      return -1;
14    else
15      return 0;
16  }
17 }
```

ComparableRectangle类扩展自Rectangle类并实现Comparable方法，如图14-4所示。关键字implements表示ComparableRectangle类继承Comparable接口的所有常量，并实现该接口的方法。compareTo方法比较两个矩形的面积。ComparableRectangle类的一个实例也是Rectangle、GeometricObject、Object和Comparable的实例。

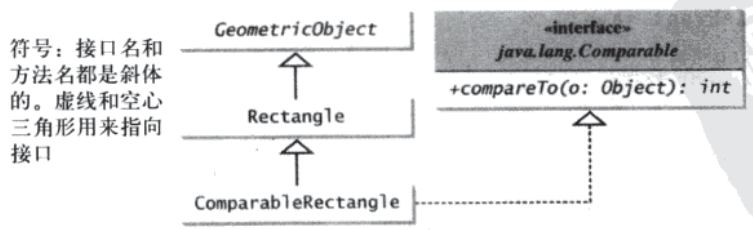


图14-4 ComparableRectangle类扩展Rectangle类并实现Comparable接口

现在，可以使用max方法找出两个ComparableRectangle对象中较大的一个。请看下面的例子：

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

接口提供通用程序设计的另一种形式。在这个例子中，如果不用接口，很难使用通用的max方法去求对象的最大值，因为必须使用多重继承才能同时继承Comparable和另一个类，例如Rectangle。

Object类包含equals方法，它的目的就是为了让Object类的子类来覆盖它，以比较对象的内容是否相同。假设Object类包含一个类似于Comparable接口中所定义的compareTo方法，那么新的max方法可以用来比较一组任意的对象。Object类中是否应该包含一个compareTo方法尚有争论。由于在Object类中没有定义compareTo方法，所以Java 中定义了Comparable接口，以便能够对两个Comparable接口的实例对象进行比较。强烈建议（尽管不要求）compareTo应该与equals保持一致。也就是说，对于两个对象o1和o2，应该确保当且仅当o1.equals(o2)为true时o1.compareTo(o2)==0成立。

14.6 举例：ActionListener接口

现在，你已经准备好编写一个小程序解决本章前言中提出的问题。程序在框架中显示两个按钮，如图14-1所示。为了响应对一个按钮的点击，需要编写代码来处理点击按钮动作。按钮就是动作来源的源对象（source object）。需要创建一个对象能够处理按钮上的动作事件。这个对象称为监听器（listener），如图14-5所示。

不是所有的对象都能成为动作事件的监听器。一个对象要成为源对象上动作事件的监听器，需要满足两个条件：

- 这个对象必须是ActionListener（事件监听器）接口的一个实例。该接口定义了所有动作监听器共有的动作。
- ActionListener对象listener必须使用方法source.addActionListener(listener)注册给源对象。

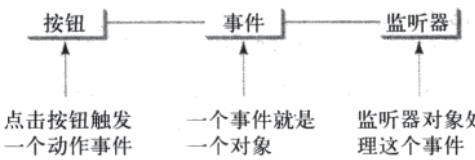


图14-5 监听器对象处理源对象触发的事件

ActionListener接口包含处理事件的actionPerformed方法。监听器必须覆盖该方法来响应事件。程序清单14-8给出在两个按钮上处理ActionEvent的代码。点击OK按钮时，就会显示消息“OK button clicked”（OK按钮被点击）。点击Cancel按钮时，就会显示消息“Cancel button clicked”（Cancel按钮被点击），如图14-1所示。

程序清单14-8 HandleEvent.java

```
1 import javax.swing.*;
2 import java.awt.event.*;
3
4 public class HandleEvent extends JFrame {
5   public HandleEvent() {
6     // Create two buttons
7     JButton jbtOK = new JButton("OK");
8     JButton jbtCancel = new JButton("Cancel");
9
10    // Create a panel to hold buttons
11    JPanel panel = new JPanel();
12    panel.add(jbtOK);
13    panel.add(jbtCancel);
14
15    add(panel); // Add panel to the frame
16
```

```
17      // Register listeners
18      OKListenerClass listener1 = new OKListenerClass();
19      CancelListenerClass listener2 = new CancelListenerClass();
20      jbtOK.addActionListener(listener1);
21      jbtCancel.addActionListener(listener2);
22    }
23
24    public static void main(String[] args) {
25      JFrame frame = new HandleEvent();
26      frame.setTitle("Handle Event");
27      frame.setSize(200, 150);
28      frame.setLocation(200, 100);
29      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30      frame.setVisible(true);
31    }
32  }
33
34  class OKListenerClass implements ActionListener {
35    public void actionPerformed(ActionEvent e) {
36      System.out.println("OK button clicked");
37    }
38  }
39
40  class CancelListenerClass implements ActionListener {
41    public void actionPerformed(ActionEvent e) {
42      System.out.println("Cancel button clicked");
43    }
44  }
```

两个监听器对象在第34～44行定义。每个监听器类实现ActionListener来处理ActionEvent。对象listener1是OKListenerClass的一个实例（第18行），它注册给按钮jbtOK（第20行）。当点击OK按钮时，OKListenerClass中的actionPerformed(ActionEvent)方法（第36行）会被调用来处理该事件。对象listener2是CancelListenerClass的一个实例（第19行），将它在第21行注册给按钮jbtCancel。当点击Cancel按钮时，CancelListenerClass中的actionPerformed(ActionEvent)方法（第42行）会被调用来处理该事件。处理事件的方式将在第16章中进一步讨论。

14.7 举例：Cloneable接口

经常会出现需要创建一个对象拷贝的情况。需要使用clone方法并理解Cloneable接口，这些都是本节的主题。

接口包括常量和抽象方法，但是Cloneable接口是一个特殊情况。在java.lang包中的Cloneable接口的定义如下所示：

```
package java.lang;

public interface Cloneable {
}
```

这个接口是空的。一个带空体的接口称为标记接口（marker interface）。一个标记接口既不包括常量也不包括方法。它用来表示一个类拥有某些特定的属性。实现Cloneable接口的类标记为可克隆的，而且它的对象可以使用在Object类中定义的clone()方法克隆。

Java库中的很多类（例如，Date、Calendar和ArrayList）实现Cloneable。这样，这些类的实例可以被克隆。例如，下面的代码

```
1 Calendar calendar = new GregorianCalendar(2003, 2, 1);
2 Calendar calendar1 = calendar;
3 Calendar calendar2 = (Calendar)calendar.clone();
4 System.out.println("calendar == calendar1 is " +
5   (calendar == calendar1));
```

```
6 System.out.println("calendar == calendar2 is " +
7   (calendar == calendar2));
8 System.out.println("calendar.equals(calendar2) is " +
9   calendar.equals(calendar2));
```

显示

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```

在前面的代码中，第2行将calendar的引用复制给calendar1，所以calendar和calendar1都指向相同的Calendar对象。第3行创建一个新对象，它是calendar的克隆，然后将这个新对象的引用赋值给calendar2。calendar2和calendar是内容相同的不同对象。

可以使用clone方法克隆一个数组。例如，下面的代码

```
1 int[] list1 = {1, 2};
2 int[] list2 = list1.clone();
3 list1[0] = 7; list1[1] = 8;
4
5 System.out.println("list1 is " + list1[0] + ", " + list1[1]);
6 System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

显示

```
list1 is 7, 8
list2 is 1, 2
```

为了定义一个自定制类来实现Cloneable接口，这个类必须覆盖Object类中的clone()方法。程序清单14-9定义一个实现Cloneable和Comparable的名为House的类。

程序清单14-9 House.java

```
 1 public class House implements Cloneable, Comparable {
 2   private int id;
 3   private double area;
 4   private java.util.Date whenBuilt;
 5
 6   public House(int id, double area) {
 7     this.id = id;
 8     this.area = area;
 9     whenBuilt = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public double getArea() {
17     return area;
18   }
19
20   public java.util.Date getWhenBuilt() {
21     return whenBuilt;
22   }
23
24   /** Override the protected clone method defined in the Object
25     class, and strengthen its accessibility */
26   public Object clone() throws CloneNotSupportedException {
27     return super.clone();
28   }
29
30   /** Implement the compareTo method defined in Comparable */
31   public int compareTo(Object o) {
32     if (area > ((House)o).area)
33       return 1;
34     else if (area < ((House)o).area)
```

```
35       return -1;
36     else
37       return 0;
38   }
39 }
```

House类实现在Object类中定义的clone方法（第26～28行）。方法头是：

```
protected native Object clone() throws CloneNotSupportedException;
```

关键字native表明这个方法不是用Java写的，但它是JVM针对自身平台实现的。关键字protected限定方法只能在同一个包内或在其子类中访问。由于这个原因，House类必须覆盖该方法并将它的可视性修饰符改为public，这样，该方法就可以在任何一个包中使用。因为Object类中针对自身平台实现的clone方法完成了克隆对象的任务，所以，在House类中的clone方法只要调用super.clone()即可。在Object类中定义的clone方法会抛出CloneNotSupportedException异常。

House类实现定义在Comparable接口中的compareTo方法（第31～38行）。该方法比较两个房子的面积。

现在，可以创建一个House类的对象，然后从这个对象创建一个完全一样的拷贝，如下所示：

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

house1和house2是两个内容相同的不同对象。Object类中的clone方法将原始对象的每个数据域复制给目标对象。如果一个数据域是基本类型，复制的就是它的值。例如，area（double类型）的值从house1复制到house2。如果一个数据域是对象，复制的就是该域的引用。例如，域whenBuilt是Date类，所以，它的引用被复制给house2，如图14-6所示。因此，尽管house1==house2为假，但是house1.whenBuilt==house2.whenBuilt为真。这称为浅复制（shallow copy）而不是深复制（deep copy），这意味着如果数据域是对象类型，那么复制的是对象的引用，而不是它的内容。

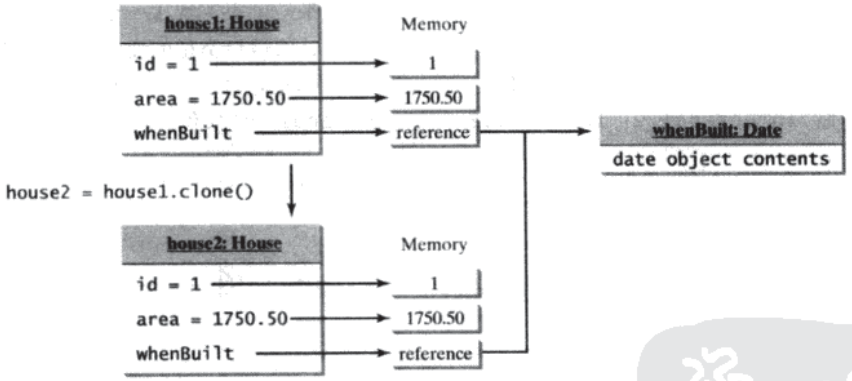


图14-6 clone方法复制基本类型域的值以及对象类型域的引用

如果希望完成深复制，可以在调用super.clone()之后用自定制的克隆操作来覆盖clone方法。参见练习题14.4。

警告 如果House没有覆盖clone()方法，程序就会收到一个语法错误，因为在java.lang.Object中clone()是被保护的。如果House不实现java.lang.Cloneable，调用House.java中的super.clone()会导致一个CloneNotSupportedException异常。这样，为了克隆一个对象，该对象的类必须覆盖clone()方法并实现Cloneable。

14.8 接口与抽象类

或多或少可以使用和抽象类一样的方式来使用接口，但是，定义一个接口与定义一个抽象类有所不同。表14-2总结出这些不同点。

表14-2 接口与抽象类

	变 量	构造方法	方 法
抽象类	无限制	子类通过构造方法链调用构造方法，抽象类不能用new操作符实例化	无限制
接口	所有的变量必须是 public static final	没有构造方法。接口不能用new操作符实例化	所有方法必须是公共的抽象实例方法

Java只允许为类的扩展做单一继承，但是允许使用接口做多重扩展。例如，

```
public class NewClass extends BaseClass
    implements Interface1, ..., InterfaceN {
  ...
}
```

利用关键字extends，接口可以继承其他接口。这样的接口称为子接口（subinterface）。例如，在下面代码中，NewInterface是Interface1，…，InterfaceN的子接口。

```
public interface NewInterface extends Interface1, ..., InterfaceN {
  // constants and abstract methods
}
```

一个类实现NewInterface必须实现在NewInterface，Interface1，…，InterfaceN中定义的抽象方法。接口可以扩展其他接口而不是类。一个类可以扩展它的父类同时实现多个接口。

所有的类共享同一个根类Object，但是接口没有共同的根。与类相似，接口也可以定义一种类型。一个接口类型的变量可以引用任何实现该接口的类的实例。如果一个类实现了一个接口，那么这个接口就类似于该类的一个父类。可以将接口当作一种数据类型使用，将接口类型的变量转换为它的子类，反过来也可以。例如，假设c是图14-7中Class2的一个实例，那么c也是Object、Class1、Interface1、Interface1_1、Interface1_2、Interface2_1和Interface2_2的实例。

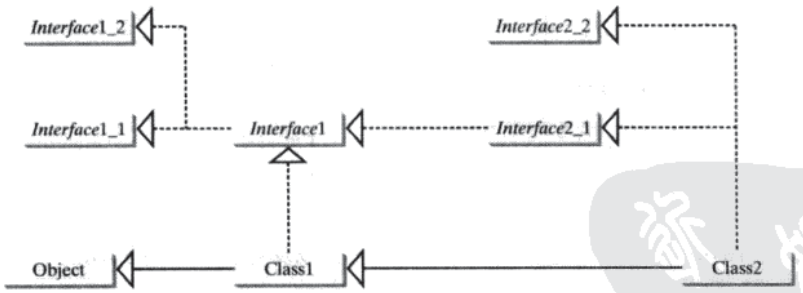


图14-7 Class1实现接口Interface1；Interface1扩展接口Interface1_1和Interface1_2。Class2扩展Class1并实现接口Interface2_1和Interface2_2

注意 类名是一个名词。接口名可以是形容词或名词。例如，java.lang.Comparable和java.awt.event.ActionListener都是接口。Comparable是一个形容词，而ActionListener是一个名词。

设计指南 抽象类和接口都是用来明确多个对象的共同特征的。那么该如何确定在什么情况下应该使用接口，什么情况下应该使用类呢？一般来说，详细描述父子关系的强是关系（strong

is-a relationship）应该用类建模。例如，因为公历是一种日历，所以，类java.util.GregorianCalendar和java.util.Calendar是用类继承建模的。弱是关系（weak is-a relationship）也称为类属关系（is-kind-of relationship），它表明对象拥有某种属性。弱是关系可以用接口来建模。例如，所有的字符串都是可比较的，因此，String类实现Comparable接口。

通常，推荐使用接口而非抽象类是因为接口可以定义不相关类共有的父类型。接口比类更加灵活。考虑Animal类。假设Animal类中定义了howToEat方法，如下所示：

```
abstract class Animal {
  public abstract String howToEat();
}
```

Animal的两个子类定义如下：

```
class Chicken extends Animal {
  public String howToEat() {
    return "Fry it";
  }
}

class Duck extends Animal {
  public String howToEat() {
    return "Roast it";
  }
}
```

假设给定这个继承体系结构，多态会让你在一个类型为Animal的变量中保存Chicken对象或Duck对象的引用，如下面代码所示：

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  animal.howToEat();
}
```

JVM会基于调用方法时所用的确切对象来动态地决定调用哪个howToEat方法。

可以定义Animal的一个子类。但是，这里有个限制条件。该子类必须是另一种动物（例如，Turkey）。

接口就无此限制。接口比类带来更多的灵活性，因为不用使每件东西都适用于同一类型的类。可以定义接口中的howToEat()方法，然后把它当做其他类的公用父类型。例如，

```
public static void main(String[] args) {
  Edible stuff = new Chicken();
  eat(stuff);

  stuff = new Duck();
  eat(stuff);

  stuff = new Broccoli();
  eat(stuff);
}

public static void eat(Edible stuff) {
  stuff.howToEat();
}
```

```
interface Edible {
  public String howToEat();
}

class Chicken implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Duck implements Edible {
  public String howToEat() {
    return "Roast it";
  }
}

class Broccoli implements Edible {
  public String howToEat() {
    return "Stir-fry it";
  }
}
```

为了定义表示可食用对象的一个类，只需让该类实现Edible接口即可。现在，这个类就成为Edible类型的子类型。任何Edible对象都可以被传递来调用eat方法。

14.9 将基本数据类型值作为对象处理

出于对性能的考虑，在Java中，基本数据类型不作为对象使用。因为处理对象需要额外的系统开销，所以，如果将基本数据类型当做对象，就会给语言性能带来负面影响。然而，许多Java中的方法需要将对象作为参数。例如，在ArrayList类中的add(object)方法向ArrayList中添加一个对象。Java提供了一个方便的办法，将基本数据类型并入对象或包装成对象（例如，将int包装成Integer类，将double包装成Double类）。对应的类称为*包装类*（wrapper class）。通过使用包装对象而不是基本数据类型变量，就可以利用通用程序设计。

Java为基本数据类型提供了Boolean、Character、Double、Float、Byte、Short、Integer和Long等包装类。这些包装类都打包在java.lang包里。它们的继承层次体系结构如图14-8所示。

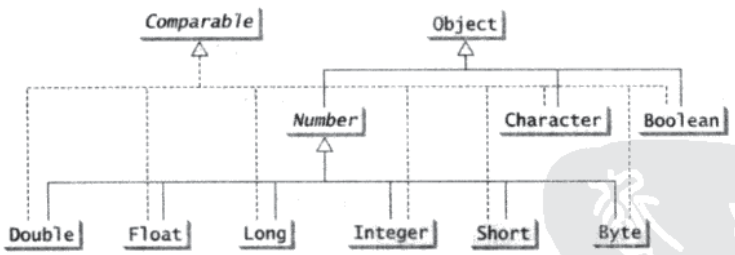


图14-8 Number类是Double、Float、Long、Integer、Short和Byte类的抽象父类

注意 大多数基本类型的包装类的名称与对应的基本数据类型名称一样，但第一个字母要大写。Integer和Character例外。

每一个数值包装类都是从抽象类Number扩展而来的，类Number包含doubleValue()、floatValue()、intValue()、longValue()、shortValue()和byteValue()方法。这些方法将对象“转换”为基本类型值。每一个包装类覆盖了在Object类中定义的toString和equals方法。因为所有的包装类都实现Comparable接口，所以这些类中都会实现compareTo方法。

包装类相互之间都非常相似。在第9章中已经介绍了Character类。Boolean类包装了布尔值true

或false。本节使用Integer和Double类为例介绍数值包装类。Integer类和Double类的主要特征如图14-9所示。

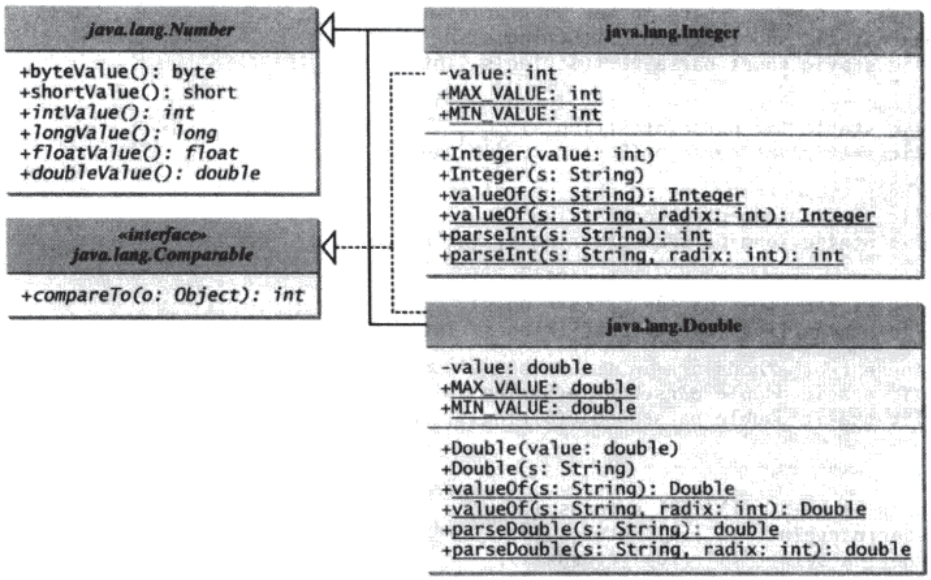


图14-9 包装类提供构造方法、常量和处理各种数据类型的转换方法

既可以用基本数据类型值也可以用表示数值的字符串来构造包装类——例如，new Double(5.0)、new Double("5.0")、new Integer(5)和new Integer("5")。

包装类没有无参构造方法。所有包装类的实例都是不可变的，这意味着一旦创建对象后，它们的内部值就不能再改变。

每一个数值包装类都有常量MAX_VALUE和MIN_VALUE。MAX_VALUE表示对应的基本数据类型的最大值。对于Byte、Short、Integer和Long，MIN_VALUE表示对应的基本类型byte、short、int和long的最小值。对Float和Double类而言，MIN_VALUE表示float型和double型的最小正值。下面的语句显示最大整数（2 147 483 647）、最小正浮点数（1.4E-45），以及双精度浮点数的最大值(1.79769313486231570e+308d)：

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

每个数值包装类都会实现在Number类中定义的抽象方法doubleValue()、floatValue()、intValue()、longValue()和shortValue()。这些方法返回包装对象的double、float、int、long或short值。

数值包装类有一个有用的静态方法valueOf(String s)。该方法创建一个新对象，并将它初始化为指定字符串表示的值。例如，

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

你已经使用过Integer类中的parseInt方法将一个数值字符串转换为一个int值，而且使用过Double类中的parseDouble方法将一个数值字符串转变为一个double值。每个数值包装类都有两个重载的方法，将数值字符串转换为正确的以10（十进制）或指定值为基数（例如，2为二进制，8为八进制，16为十六进制）的数值。这些方法如下所示：

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

例如，

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

Integer.parseInt("12",2)会引起一个运行时错误，因为12不是二进制数。

14.10 举例：对一个对象数组排序

这个例子给出一个静态通用方法，对一个可比较的对象数组进行排序。这些对象都是Comparable接口的实例，并且可以使用compareTo方法进行比较。对于任何对象，只要其所在的类都实现了Comparable接口，这个方法就能对这些对象构成的数组进行排序。

为了测试这个方法，程序对整数数组、双精度数数组、字符数组和字符串数组进行排序。程序如程序清单14-10所示。

程序清单14-10 GenericSort.java

```
1  public class GenericSort {
2    public static void main(String[] args) {
3      // Create an Integer array
4      Integer[] intArray = {new Integer(2), new Integer(4),
5        new Integer(3)};
6
7      // Create a Double array
8      Double[] doubleArray = {new Double(3.4), new Double(1.3),
9        new Double(-22.1)};
10
11     // Create a Character array
12     Character[] charArray = {new Character('a'),
13       new Character('J'), new Character('r')};
14
15     // Create a String array
16     String[] stringArray = {"Tom", "John", "Fred"};
17
18     // Sort the arrays
19     sort(intArray);
20     sort(doubleArray);
21     sort(charArray);
22     sort(stringArray);
23
```

```
24      // Display the sorted arrays
25      System.out.print("Sorted Integer objects: ");
26      printList(intArray);
27      System.out.print("Sorted Double objects: ");
28      printList(doubleArray);
29      System.out.print("Sorted Character objects: ");
30      printList(charArray);
31      System.out.print("Sorted String objects: ");
32      printList(stringArray);
33    }
34
35    /** Sort an array of comparable objects */
36    public static void sort(Comparable[] list) {
37      Comparable currentMin;
38      int currentMinIndex;
39
40      for (int i = 0; i < list.length - 1; i++) {
41        // Find the maximum in the list[0..i]
42        currentMin = list[i];
43        currentMinIndex = i;
44
45        for (int j = i + 1; j < list.length; j++) {
46          if (currentMin.compareTo(list[j]) > 0) {
47            currentMin = list[j];
48            currentMinIndex = j;
49          }
50        }
51
52        // Swap list[i] with list[currentMinIndex] if necessary;
53        if (currentMinIndex != i) {
54          list[currentMinIndex] = list[i];
55          list[i] = currentMin;
56        }
57      }
58    }
59
60    /** Print an array of objects */
61    public static void printList(Object[] list) {
62      for (int i = 0; i < list.length; i++)
63        System.out.print(list[i] + " ");
64      System.out.println();
65    }
66  }
```

```
Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Fred John Tom
```

sort方法的算法和6.10.1节中的算法是一样的。在6.10.1节中的sort方法对double值构成的数组进行排序。假定这些对象也是Comparable接口的实例，那么这个例子中的sort方法可以对任何对象类型的数组排序。这是通用程序设计（generic programming）的另一个例子。因为通用程序设计使一个方法可以对通用类型参数进行操作，所以这种方法可以在多种类型上重复使用。

Integer、Double、Character和String都实现了Comparable接口，所以这些类的对象都可以使用compareTo方法进行比较。sort方法使用compareTo方法判断对象在数组中的顺序。

提示 在java.util.Arrays类中，Java提供一个对任意对象类型的数组进行排序的静态方法sort，假定数组中的元素都是可比较的。这样，就能使用下面的代码对这个例子中的数组进行排序：

```
java.util.Arrays.sort(intArray);
java.util.Arrays.sort(doubleArray);
java.util.Arrays.sort(charArray);
java.util.Arrays.sort(stringArray);
```

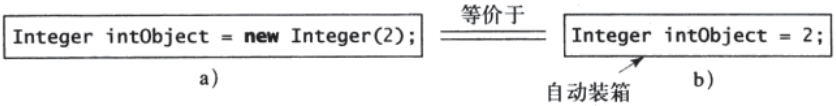
注意 数组是对象，一个数组是Object类的一个实例。此外，如果A是B的子类型，那么A[]的每一个实例都是B[]的实例。所以，下面语句的值都是true：

```
new int[10] instanceof Object
new Integer[10] instanceof Object
new Integer[10] instanceof Comparable[]
new Integer[10] instanceof Number[]
new Number[10] instanceof Object[]
```

警告 尽管int型的值可以赋给double型的变量，但是int[]和double[]是两个不兼容的类型。所以，不能把一个int[]数组赋值给double[]型变量或Object[]型变量。

14.11 基本类型和包装类类型之间的自动转换

Java允许基本类型和包装类类型之间进行自动转换。例如，可以用自动装箱将图a中的语句简化为图b中的语句：



将基本类型值转换为包装类对象的过程称为*装箱*（boxing），相反的过程称为*开箱*（unboxing）。如果一个基本类型值出现在需要对象的环境中，编译器会将基本类型值进行自动装箱；如果一个对象出现在需要基本类型值的环境中，编译器将对象进行自动开箱。考虑下面的例子：

```
1 Integer[] intArray = {1, 2, 3};
2 System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

在第一行中，基本类型值1、2和3被自动装箱成对象new Integer(1)、new Integer(2)和new Integer(3)。第二行中，对象intArray[0]、intArray[1]和intArray[2]被自动转换为int值，然后进行相加。

14.12 BigInteger和BigDecimal类

如果要进行非常大的数的计算或者高精度浮点值的计算，可以使用java.math包中的BigInteger类和BigDecimal类。它们都是不可变的。它们都扩展Number类且实现Comparable接口。long类型的最大整数值为long.MAX_VALUE（即9223372036854775807）。BigInteger的实例可以表示任意大小的整数。可以使用new BigInteger(String)和new BigDecimal(String)来创建BigInteger和BigDecimal的实例，使用add、subtract、multiple、divide和remainder方法完成算术运算，使用compareTo方法比较两个大数字。例如，下面的代码创建两个BigInteger对象并且将它们进行相乘：

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

它的输出为18446744073709551614。

对BigDecimal对象的精度没有限制。如果结果不能终止，那么divide方法会抛出ArithmeticException异常。但是，可以使用重载的divide(BigDecimal d,int scale, int roundingMode)方法来指定尺度和舍入方式来避免这个异常，这里的scale是指小数点后最小的整数位数。例如，下面的代码创建两个尺度为20、舍入方式为BigDecimal.ROUND_UP的BigDecimal对象。

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

输出为0.33333333333333333334。

注意 一个整数的阶乘可能会非常大。程序清单14-11给出可以返回任意整数阶乘的方法。

程序清单14-11 LargeFactorial.java

```
1 import java.math.*;
2
3 public class LargeFactorial {
4   public static void main(String[] args) {
5     System.out.println("50! is \n" + factorial(50));
6   }
7
8   public static BigInteger factorial(long n) {
9     BigInteger result = BigInteger.ONE;
10    for (int i = 1; i <= n; i++)
11      result = result.multiply(new BigInteger(i + ""));
12
13    return result;
14  }
15 }
```

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

BigInteger.ONE（第9行）是一个定义在BigInteger类中的常量。BigInteger.ONE和new BigInteger("1")是一样的。

调用multiply方法（第11行）获得一个新结果。

14.13 实例学习：Rational类

有理数有一个分子和分母，形式为a/b，这里的a是分子而b是分母。例如，1/3、3/4和10/4都是有理数。

有理数的分母不能为0，但是分子可以为0。每个整数i等价于一个有理数i/1。有理数用于涉及分数的准确计算中，例如，1/3=0.33333...。这个数字不能用double或float数据类型精确地表示为浮点形式。为了获取准确的结果，必须使用有理数。

Java提供了表示整数和浮点数的数据类型，但是没有提供表示有理数的数据类型。本节给出如何设计一个表示有理数的类。

因为有理数共享了很多整数和浮点数的通用特性，而且Number是数值包装类的根类，所以将Rational类定义为Number类的子类是合适的。因为有理数是可以比较的，所以Rational类应该也能实现Comparable接口。图14-10说明了Rational类以及它和Number类及Comparable接口的关系。

一个有理数包括一个分子和一个分母。有很多有理数是等价的，例如，1/3=2/6=3/9=4/12。1/3的分子和分母除了1之外没有公约数，所以，1/3称为最低形式。

为了将一个有理数约减为它的最低形式，需要找到分子和分母绝对值的最大公约数（GCD），然后将分子和分母都除以这个值。可以使用程序清单4-8中计算两个整数n和d的GCD方法。在Rational对象中的分子和分母都可以降为它们的最低形式。

通常，首先编写一个测试程序来创建两个Rational对象，然后测试它的方法。程序清单14-12是一个测试程序。

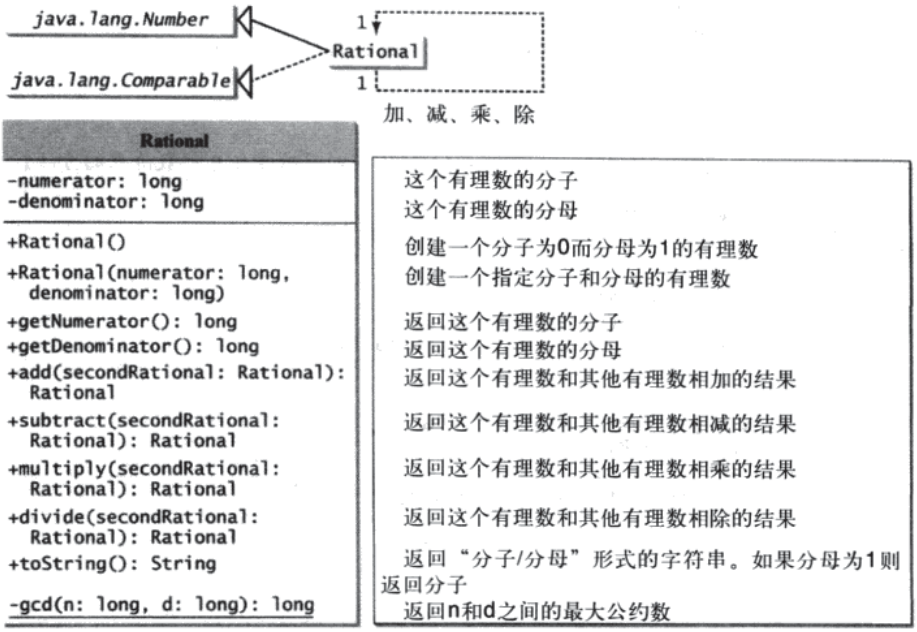


图14-10 Rational类的属性、构造方法和方法在UML中的图解

程序清单14-12 TestRationalClass.java

```
1 public class TestRationalClass {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create and initialize two rational numbers r1 and r2.
5     Rational r1 = new Rational(4, 2);
6     Rational r2 = new Rational(2, 3);
7
8     // Display results
9     System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
10    System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
11    System.out.println(r1 + " * " + r2 + " = " + r1.multiply(r2));
12    System.out.println(r1 + " / " + r2 + " = " + r1.divide(r2));
13    System.out.println(r2 + " is " + r2.doubleValue());
14  }
15 }
```

```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2/3 is 0.6666666666666666
```

main方法创建两个有理数：r1和r2（第5～6行），然后显示r1+r2、r1-r2、r1×r2和r1/r2的结果（第9～12行）。为了完成r1+r2，调用r1.add(r2)返回一个新的Rational对象。同样地，r1.subtract(r2)用以完成r1-r2，r1.multiply(r2)用以完成r1×r2，而r1.divide(r2)用以完成r1/r2。

doubleValue()方法显示r2的double值（第13行）。doubleValue()方法在java.lang.Number中定义并且在Rational中被覆盖。

注意，当使用加号（+）将一个字符串和一个对象进行链接时，使用的是来自toString()方法的这个对象的字符串表示同这个字符串进行链接。因此，r1+"+"+r2+"="+r1.add(r2)等价于

r1.toString()+"+"+r2.toString()+"="+r1.add(r2).toString()。

Rational类在程序清单14-13中实现。

程序清单14-13 Rational.java

```
1  public class Rational extends Number implements Comparable {
2    // Data fields for numerator and denominator
3    private long numerator = 0;
4    private long denominator = 1;
5
6    /** Construct a rational with default properties */
7    public Rational() {
8      this(0, 1);
9    }
10
11   /** Construct a rational with specified numerator and denominator */
12   public Rational(long numerator, long denominator) {
13     long gcd = gcd(numerator, denominator);
14     this.numerator = ((denominator > 0) ? 1 : -1) * numerator / gcd;
15     this.denominator = Math.abs(denominator) / gcd;
16   }
17
18   /** Find GCD of two numbers */
19   private static long gcd(long n, long d) {
20     long n1 = Math.abs(n);
21     long n2 = Math.abs(d);
22     int gcd = 1;
23
24     for (int k = 1; k <= n1 && k <= n2; k++) {
25       if (n1 % k == 0 && n2 % k == 0)
26         gcd = k;
27     }
28
29     return gcd;
30   }
31
32   /** Return numerator */
33   public long getNumerator() {
34     return numerator;
35   }
36
37   /** Return denominator */
38   public long getDenominator() {
39     return denominator;
40   }
41
42   /** Add a rational number to this rational */
43   public Rational add(Rational secondRational) {
44     long n = numerator * secondRational.getDenominator() +
45       denominator * secondRational.getNumerator();
46     long d = denominator * secondRational.getDenominator();
47     return new Rational(n, d);
48   }
49
50   /** Subtract a rational number from this rational */
51   public Rational subtract(Rational secondRational) {
52     long n = numerator * secondRational.getDenominator()
53       - denominator * secondRational.getNumerator();
54     long d = denominator * secondRational.getDenominator();
55     return new Rational(n, d);
56   }
57
58   /** Multiply a rational number to this rational */
59   public Rational multiply(Rational secondRational) {
60     long n = numerator * secondRational.getNumerator();
```

```
61      long d = denominator * secondRational.getDenominator();
62      return new Rational(n, d);
63    }
64
65    /** Divide a rational number from this rational */
66    public Rational divide(Rational secondRational) {
67      long n = numerator * secondRational.getDenominator();
68      long d = denominator * secondRational.numerator;
69      return new Rational(n, d);
70    }
71
72    /** Override the toString() method */
73    public String toString() {
74      if (denominator == 1)
75        return numerator + "";
76      else
77        return numerator + "/" + denominator;
78    }
79
80    /** Override the equals method in the Object class */
81    public boolean equals(Object parm1) {
82      if ((this.subtract((Rational)(parm1))).getNumerator() == 0)
83        return true;
84      else
85        return false;
86    }
87
88    /** Implement the abstract intValue method in java.lang.Number */
89    public int intValue() {
90      return (int)doubleValue();
91    }
92
93    /** Implement the abstract floatValue method in java.lang.Number */
94    public float floatValue() {
95      return (float)doubleValue();
96    }
97
98    /** Implement the doubleValue method in java.lang.Number */
99    public double doubleValue() {
100     return numerator * 1.0 / denominator;
101   }
102
103   /** Implement the abstract longValue method in java.lang.Number */
104   public long longValue() {
105     return (long)doubleValue();
106   }
107
108   /** Implement the compareTo method in java.lang.Comparable */
109   public int compareTo(Object o) {
110     if ((this.subtract((Rational)o)).getNumerator() > 0)
111       return 1;
112     else if ((this.subtract((Rational)o)).getNumerator() < 0)
113       return -1;
114     else
115       return 0;
116   }
117 }
```

有理数封装在Rational对象中。在机器内部，一个有理数表示为它的最低形式（第13行），分子决定有理数的符号（第14行）。分母总是正数（第15行）。

方法gcd()（Rational类中的第19～30行）是私有的，它是不能被用户使用的。gcd()方法只能在Rational类的内部使用。gcd()方法也是静态的，因为它不依赖于任何一个特定的Rational对象。

方法abs(x)（Rational类中的第20～21行）在Math类中定义，并返回x的绝对值。

两个Rational对象可以相互作用来完成加、减、乘、除操作。这些方法返回一个新的Rational对象（第43～70行）。

Object类中的toString方法和equals方法在Rational类中被覆盖（第73～91行）。toString()方法以numerator/denominator（分子/分母）的形式返回一个Rational对象的字符串表示，如果分母为1就将它简化为numerator。如果该有理数和另一个有理数相同，那么方法equals(Object other)返回值为真。

Number类中的抽象方法intValue、longValue、floatValue和doubleValue在Rational类中实现（第88～106行）。这些方法为有理数返回int、float和double值。

Comparable接口中的compareTo(Object other)方法在Rational类中实现（第109～116行），对这个有理数和另一个有理数进行比较。

提示 在Rational类中提供了属性numerator（分子）和denominator（分母）的get方法，但是没有提供set方法，因此，一旦创建Rational对象，那么它的内容就不能改变。Rational类是不可变的。String类和基本类型值的包装类也都是不可变的。

提示 可以使用两个变量表示分子和分母。也可以使用两个整数构成的数组表示分子和分母。参见练习题14.18。尽管有理数的内部表示改变，但是Rational类中的公共方法的签名是不变的。这是一个解释类的数据域应该保持私有，以确保将类的实现和类的使用分隔开的很好的例子。

Rational类有很严格的限定。这很容易溢出。例如，下面的代码将显示不正确的结果，因为分母太大了。

```
public class Test {
  public static void main(String[] args) {
    Rational r1 = new Rational(1, 123456789);
    Rational r2 = new Rational(1, 123456789);
    Rational r3 = new Rational(1, 123456789);
    System.out.println("r1 * r2 * r3 is " +
      r1.multiply(r2.multiply(r3)));
  }
}
```

```
r1 * r2 * r3 is -1/2204193661661244627
```

为了解决这个问题，可以使用BigInteger表示分子和分母实现Rational类（参见练习题14.19）。

关键术语

abstract class（抽象类）
abstract method（抽象方法）
deep copy（深复制）
interface（接口）
marker interface（标记接口）
multiple inheritance（多重继承）
subinterface（子接口）
shallow copy（浅复制）
single inheritance（单一继承）
wrapper class（包装类）

本章小结

- 抽象类和常规类一样，都有数据和方法，但是不能用new操作符创建抽象类的实例。
- 非抽象类中不能包含抽象方法。如果抽象类的子类没有实现所有被继承的父类抽象方法，就必须将子类也定义为抽象类。
- 包含抽象方法的类必须是抽象类。但是，抽象类可以不包含抽象的方法。
- 即使父类是具体的，子类也可以是抽象的。

- 接口是一种与类相似的结构，只包含常量和抽象方法。接口在许多方面与抽象类很相近，但抽象类除了包含常量和抽象方法外，还可以包含变量和具体方法。
- 在Java中，接口被认为是一种特殊的类。就像常规类一样，每个接口都被编译为独立的字节码文件。
- 接口`java.lang.Comparable`定义了`compareTo`方法。Java类库中的许多类都实现了`Comparable`。
- 接口`java.lang.Cloneable`是一个标记接口。实现`Cloneable`接口的类的对象是可克隆的。
- 一个类仅能继承一个父类，但一个类却可以实现一个或多个接口。
- 一个接口可以扩展一个或多个接口。
- 许多Java方法要求使用对象作为参数。Java提供了一个便捷的办法，将基本数据类型合并或包装到一个对象中（例如，包装`int`值到`Integer`类中，包装`double`值到`Double`类中）。对应的类称作*包装类*。使用包装对象而不是基本数据类型的变量，将有助于通用程序设计。
- Java可以根据上下文自动地将基本类型值转换为对应的包装对象，反之亦然。
- `BigInteger`类在计算和处理任意大小的整数方面是很有用的。`BigDecimal`类可以用作计算和处理带任意精度的浮点数。

复习题

14.2节

14.1 在下面类的定义中，哪个定义了一个合法的抽象类？

```
class A {
  abstract void unfinished() {
  }
}
```

a)

```
public class abstract A {
  abstract void unfinished();
}
```

b)

```
class A {
  abstract void unfinished();
}
```

c)

```
abstract class A {
  protected void unfinished();
}
```

d)

```
abstract class A {
  abstract void unfinished();
}
```

e)

```
abstract class A {
  abstract int unfinished();
}
```

f)

14.2 `getArea`方法和`getPerimeter`方法可以从`GeometricObject`类中删除。在`GeometricObject`类中将这两个方法定义为抽象方法的好处是什么？

14.3 下面说法为真还是为假？除了不能使用`new`操作符创建抽象类的实例之外，一个抽象类可以像非抽象类一样使用。

14.4～14.6节

14.4 下面哪个是正确的接口？

```
interface A {
  void print() { };
}
```

a)

```
abstract interface A extends I1, I2 {
  abstract void print() { };
}
```

b)

```
abstract interface A {
  print();
}
```

c)

```
interface A {
  void print();
}
```

d)

14.5 下面说法为真还是为假？如果一个类实现了`Comparable`，那么该类的对象可以调用`compareTo`方法。

14.6 在14.5节中定义了两个max方法。解释一下，为什么签名为max(Comparable,Comparable)的max优于签名为max(Object,Object)的max。

14.7 可以在类中定义compareTo方法而不实现Comparable接口。实现Comparable接口的好处是什么？

14.8 下面说法为真还是为假？如果一个类实现java.awt.event.ActionListener，那么类的对象可以调用actionPerformed方法。

14.7～14.8节

14.9 如果一个对象的类没有实现java.lang.Cloneable，那么可以调用clone()方法来克隆这个对象吗？Date类实现Cloneable吗？

14.10 如果House类（在程序清单14-9中定义）没有覆盖clone()方法，或者如果House类没有实现java.lang.Cloneable，会发生什么？

14.11 给出下面代码的输出结果：

```
java.util.Date date = new java.util.Date();
java.util.Date date1 = date;
java.util.Date date2 = (java.util.Date)(date.clone());
System.out.println(date == date1);
System.out.println(date == date2);
System.out.println(date.equals(date2));
```

14.12 给出下面代码的输出结果：

```
java.util.ArrayList list = new java.util.ArrayList();
list.add("New York");
java.util.ArrayList list1 = list;
java.util.ArrayList list2 = (java.util.ArrayList)(list.clone());
list.add("Atlanta");
System.out.println(list == list1);
System.out.println(list == list2);
System.out.println("list is " + list);
System.out.println("list1 is " + list1);
System.out.println("list2.get(0) is " + list2.get(0));
System.out.println("list2.size() is " + list2.size());
```

14.13 下面的代码有什么错误？

```
public class Test {
  public static void main(String[] args) {
    GeometricObject x = new Circle(3);
    GeometricObject y = x.clone();
    System.out.println(x == y);
  }
}
```

14.14 给出一个例子显示接口比抽象类有优势。

14.9节

14.15 描述基本类型包装类。为什么需要这些包装类？

14.16 下面的每条语句都能编译吗？

```
Integer i = new Integer("23");
Integer i = new Integer(23);
Integer i = Integer.valueOf("23");
Integer i = Integer.parseInt("23", 8);
Double d = new Double();
Double d = Double.valueOf("23.45");
int i = (Integer.valueOf("23")).intValue();
double d = (Double.valueOf("23.4")).doubleValue();
int i = (Double.valueOf("23.4")).intValue();
String s = (Double.valueOf("23.4")).toString();
```

14.17 如何将一个整数转换为一个字符串？如何将一个数值字符串转换为一个整数？如何将一个double数转换为一个字符串？如何将一个数值字符串转换为一个double数？

14.18 为什么下面两行代码可以编译，但会导致运行时错误？

```
Number numberRef = new Integer(0);
Double doubleRef = (Double)numberRef;
```

14.19 为什么下面两行代码可以编译，但会导致运行时错误？

```
Number[] numberArray = new Integer[2];
numberArray[0] = new Double(1.5);
```

14.20 下面的代码有什么错误？

```
public class Test {
  public static void main(String[] args) {
    Number x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(new Integer(4)));
  }
}
```

14.21 下面的代码有什么错误？

```
public class Test {
  public static void main(String[] args) {
    Number x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println((Integer)x.compareTo(new Integer(4)));
  }
}
```

14.22 下面代码的输出是什么？

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

14.10～14.12节

14.23 什么是自动装箱和自动开箱？下面的语句正确吗？

```
Number x = 3;
Integer x = 3;
Double x = 3;
Double x = 3.0;
int x = new Integer(3);
int x = new Integer(3) + new Integer(4);
double y = 3.4;
y.intValue();

JOptionPane.showMessageDialog(null, 45.5);
```

14.24 可以将new int[10]、new String[100]、new Object[50]或者new Calendar[20]赋值给一个Object[]类型的变量吗？

14.25 下面代码的输出是什么？

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    x.add(y);
    System.out.println(x);
  }
}
```

综合题

14.26 定义下面的术语：抽象类、接口。抽象类和接口的相同之处和不同之处是什么？

14.27 判断下面语句的对和错：

（1）抽象类可以有使用该抽象类的构造方法创建的实例。

（2）抽象类可以扩展。

（3）接口被编译为独立的字节码文件。

（4）非抽象父类的子类不能是抽象类。

（5）子类不能覆盖父类中的具体方法将其声明成抽象方法。

（6）抽象方法必须是非静态的。

（7）接口可以有静态方法。

（8）接口可以扩展一个或多个接口。

（9）接口可以扩展抽象类。

（10）抽象类可以扩展接口。

编程练习题

14.1～14.7节

*14.1 （将GeometricObject类变成可比较的）修改GeometricObject类以实现Comparable接口，并且在GeometricObject类中定义一个静态的求两个GeometricObject对象中较大者的max方法。画出UML图并实现这个新的GeometricObject类。编写一个测试程序，使用max方法求两个圆中的较大者和两个矩形中的较大者。

*14.2 （ComparableCircle类）创建名为ComparableCircle的类，它扩展Circle类并实现Comparable接口。画出UML图并实现compareTo方法，使其根据面积比较两个圆。编写一个测试程序求出ComparableCircle对象的两个实例中的较大者。

*14.3 （可着色接口Colorable）设计一个名为Colorable的接口，其中有名为howToColor()的void方法。可着色对象的每个类必须实现Colorable接口。设计一个扩展GeometricObject类并实现Colorable接口的名为Square的类。实现howToColor方法，显示消息"Color all four sides"（给所有的四条边着色）。

画出包含Colorable、Square和GeometricObject的UML图。编写一个测试程序，创建有五个GeometricObject对象的数组。对于数组中的每个对象而言，如果对象是可着色的，那就调用howToColor方法。

*14.4 （修改House类）改写程序清单14-9中的House类，对数据域whenBuilt进行深复制。

*14.5 （将Circle类变成可比较的）改写程序清单14-2中的Circle类，它扩展GeometricObject类并实现Comparable接口。覆盖Object类中的equals方法。当两个Circle对象半径相等时，则这两个Circle对象是相同的。画出包括Circle、GeometricObject和Comparable的UML图。

*14.6 （将Rectangle类变成可比较的）改写程序清单14-3的Rectangle类，它扩展GeometricObject类并实现Comparable接口。覆盖Object类中的equals方法。当两个Rectangle对象面积相同时，则这两个对象是相同的。画出包括Rectangle、GeometricObject和Comparable的UML图。

*14.7 （八边形Octagon类）编写一个名为Octagon的类，它扩展GeometricObject类并实现Comparable和Cloneable接口。假设八边形八条边的边长都相等。它的面积可以使用下面的公式计算：

$$面积 = (2 + 4/\sqrt{2}) \times 边长 \times 边长$$

画出包括Octagon、GeometricObject、Comparable和Cloneable的UML图。编写一个测试程序，创建一个边长值为5的Octagon对象，然后显示它的面积和周长。使用clone方法创建一个新对象，并使用compareTo方法比较这两个对象。

*14.8 （求几何对象的面积之和）编写一个方法，求数组中所有几何对象的面积之和。方法签名如下：

```
public static double sumArea(GeometricObject[] a)
```

编写测试程序，创建四个对象（两个圆和两个矩形）的数组，然后使用sumArea方法求它们的总面积。

*14.9 （找出最大的对象）编写一个方法，返回对象数组中最大的对象。方法签名如下：

```
public static Object max(Comparable[] a)
```

所有对象都是Comparable接口的实例。对象在数组中的顺序是由compareTo方法决定的。

编写测试程序，创建一个由10个字符串构成的数组、一个由10个整数构成的数组和一个由10个日期构成的数组，找出数组中最大的字符串、整数和日期。

**14.10 （显示日历）改写程序清单5-12中的PrintCalendar类，使用Calendar类和GregorianCalendar类显示指定月份的日历。程序从命令行接收月份和年份。例如，

```
java Exercise14_10 1 2010
```

这将显示如图14-11所示的日历。

也可以不输入年份运行这个程序。在这种情况下，年份为当前年。如果运行程序时没有指定月份和年份，月份就是当前月。

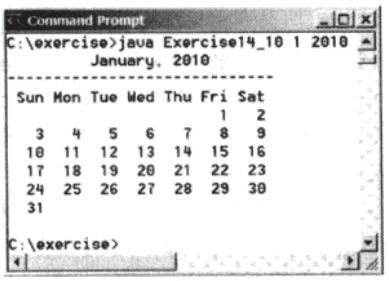


图14-11 程序显示2010年1月的日历

14.12节

**14.11 （被5或6整除）找出能被5或6整除的前10个数字（大于long.MAX_VALUE）。

**14.12 （被2或3整除）找出能被2或3整除的前10个数字，这些数字都有十进制数50。

**14.13 （平方数）找出大于long.MAX_VALUE的前10个平方数。平方数是指形式为n^2的数。

**14.14 （大素数）编写程序找出五个大于long.MAX_VALUE的素数。

**14.15 （Mersenne素数）如果一个素数可以写成2^p-1的形式，那么该素数就称为Mersenne素数，其中的p是一个正整数。编写程序找出$p \leqslant 100$的所有Mersenne素数，然后显示如下所示的输出。（必须使用BigInteger来存储数字，因为它太大了不能用long来存储。程序可能得花几个小时来完成。）

```
p         2^p - 1

2           3
3           7
5           31
...
```

14.13节

**14.16 （近似e）练习题4.26使用下面数列近似计算e：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots + \frac{1}{i!}$$

为了得到更好的精确度，在计算中使用25位精度的BigDecimal。编写程序显示当i=100，200，...，1000时e的值。

14.17 （使用Rational类）编写程序，使用Rational类计算下面的和数列：

$$\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\cdots+\frac{98}{99}+\frac{99}{100}$$

你将会发现输出是不正确的，因为整数溢出（太大了）。为了解决这个问题，参见练习题14.19。

*14.18 （演示封装的好处）使用新的分子分母的内部表达改写14.13节中的Rational类。创建有两个整数的数组，如下所示：

```
private long[] r = new long[2];
```

使用r[0]表示分子，使用r[1]表示分母。在Rational类中的方法签名没有改变，因此，无须重新编译，前一个Rational类的客户端应用程序可以继续使用这个新的Rational类。

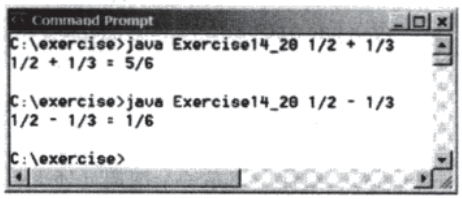


图14-12 程序从命令行读取三个参数（操作数1、运算符、操作数2），然后显示这个表达式以及这个算术运算的结果

**14.19 （在Rational类中使用BigInteger）使用BigInteger表示分子和分母，重新设计和实现14.13节的Rational类。

*14.20 （创建一个有理数的计算器）编写一个类似于程序清单9-5的程序。这里不使用整数，而是使用有理数，如图14-12所示。

需要使用在9.2.6节中介绍的String类中的split方法来获取分子字符串和分母字符串，并使用Integer.parseInt方法将字符串转换为整数。

*14.21 （数学方面：Complex类）一个复数是一个形式为$a+bi$的数，这里的a和b都是实数，i是-1的平方根。数字a和b分别称为复数的实部和虚部。可以使用下面的公式完成复数的加、减、乘、除：

$$a+bi+c+di=(a+c)+(b+d)i$$
$$a+bi-(c+di)=(a-c)+(b-d)i$$
$$(a+bi)*(c+di)=(ac-bd)+(bc+ad)i$$
$$(a+bi)/(c+di)=(ac+bd)/(c^2+d^2)+(bc-ad)i/(c^2+d^2)$$

还可以使用下面的公式得到复数的绝对值：

$$|a+bi|=\sqrt{a^2+b^2}$$

设计一个名为Complex的复数来表示复数以及完成复数运算的add、substract、multiply、divide和abs方法，并且覆盖toString方法以返回一个表示复数的字符串。方法toString返回字符串a+bi。如果b是0，那么它只返回a。

提供三个构造方法Complex(a,b)、Complex(a)和Complex()。Complex()创建数字0的Complex对象，而Complex(a)创建一个b为0的Complex对象。还提供getRealPart()和getImaginaryPart()方法以分别返回复数的实部和虚部。

编写一个测试程序，提示用户输入两个复数，然后显示它们做加、减、乘、除之后的结果。下面是一个运行示例：

```
Enter the first complex number: 3.5 5.5 Enter
Enter the second complex number: -3.5 1 Enter
3.5 + 5.5i + -3.5 + 1.0i = 0.0 + 6.5i
3.5 + 5.5i - -3.5 + 1.0i = 7.0 + 4.5i
3.5 + 5.5i * -3.5 + 1.0i = -17.75 + -15.75i
3.5 + 5.5i / -3.5 + 1.0i = -0.5094 + -1.7i
|3.5 + 5.5i| = 6.519202405202649
```