11 Abstract Classes And Interfaces

Overview

This chapter presents some additional standard library classes from the `java.lang` package and an extended example illustrating polymorphism. You would help to study Sections 10.1-10.3 (Exceptions and the elements of input files) before reading this chapter. Arrays are used heavily starting in Section 11.7.

- Section 11.1 introduces a new software design problem, the Numeric class.
- Sections 11.2-11.4 define and illustrate all of the new language features for this chapter: abstract classes, interfaces, instanceof operator, final methods, and final classes. They also describe the standard wrapper classes Integer, Double, Long, etc. This is as far as you need to go to understand all remaining chapters.
- Sections 11.5-11.8 develop three different subclasses of the Numeric superclass and a class of objects representing arrays of Numeric values.
- Sections 11.9-11.11 are enrichment topics: a concept from theoretical computability, an elementary introduction to the use of threads, and some details on the Math and Character classes from the Sun standard library.

11.1 Analysis And Design Of The Mathematicians Software

You are hired by a think-tank of mathematicians to write software that they will use in their work. When you discuss with them the kinds of software they need, you find that they work with various kinds of numbers that the Java language does not have.

One kind of number they work with is fractional -- one whole number divided by another. A fraction such as 1/3 cannot be represented exactly using a double value. For some software situations, these mathematicians want the answer calculated as a fraction reduced to lowest terms, not as a decimal number approximation.

Another category of numbers that Java does not provide is complex numbers, and a third is very long integers of perhaps 40 to 50 digits. Many of the things you do with fractions you will also want to do with complex numbers and with very long integers. So you will want basically the same methods for each of the Complex and VeryLong classes that you have for the Fraction class (though the details of how the methods calculate will differ).

You decide to make all three classes subclasses of one superclass. You can begin by developing what is common to all three of the Fraction, Complex, and VeryLong subclasses, then add to the individual subclasses whatever extra operations they need.

In designing object classes, you consider what the objects can do (instance methods) and what they know (instance variables). You start with what they can do for you. After you have that figured out, you decide what the objects have to know in order to do it. That is, analysis and design starts by developing the object operations involved. You later use those operations to decide on the object attributes.

Analysis

You need to develop documentation describing the operations you want to have in this superclass. Part of getting the specifications clear is deciding how these methods are to be called by other methods. That is easily expressed as a method heading.

Your talks with the mathematicians reveal four kinds of operations that they want their Numeric objects to have:

1. They want to be able to add, subtract, and multiply two Numerics to get a new Numeric result.
2. They want to be able to test two Numerics to see which is larger or if they are equal.
3. They want to be able to convert from a standard String form (e.g., "2/3" for a fraction) to the object and back again, and also convert any Numeric to a decimal number equivalent or approximation.
4. They want several other operations, such as finding the largest or smallest of two Numerics, finding the square of a Numeric, adding numbers obtained from an input source to a running total, etc.

This discussion leads to the sketch in Listing 11.1. The bodies of the methods are the minimum needed for compiling, since they are irrelevant at this point; the sketch is design documentation, not implementation. The `toString` and `equals` methods override those standard methods from the Object class (the standard methods require that the parameter be of type Object). The `valueOf` method for Fractions returns the Fraction object corresponding to e.g. the String value "3/7"; the `valueOf` method for Complexes returns the Complex object corresponding to e.g. the String value "7.0 + 4.3i".

Listing 11.1 Documentation for the Numeric class

```
public class Numeric       // stubbed documentation
{
   /** Convert to the appropriate subclass. Return null if par is
    *  null or "".  Throw NumberFormatException if wrong form. */
   public Numeric valueOf (String par)          { return null; }

   /** Convert to string form. */
   public String toString()                     { return null; }

   /** Convert to a real-number equivalent. */
   public double doubleValue()                     { return 0; }

   /** Tell whether the executor equals the parameter. */
   public boolean equals (Object ob)            { return true; }

   /** Like String's compareTo: positive means ob is "larger". */
   public int compareTo (Object ob)               { return 0; }

   /** Return a new Numeric that is the sum of both.
    *  Return null if they are the same subtype but the answer
    *  is not computable.  Throw an Exception if par is null or
    *  the executor and par are different subtypes. */
   public Numeric add (Numeric par)             { return null; }

   /** Same as add, except return executor minus par. */
   public Numeric subtract (Numeric par)        { return null; }

   /** Same as add, except return executor times par. */
   public Numeric multiply (Numeric par)        { return null; }

   /** Return the square of the executor. */
   public Numeric square()                      { return null; }

   /** Read Numeric values from the BufferedReader source and add
    *  them to the executor, until null is seen or until a value
    *  cannot be added to the executor.  Return the total. */
   public Numeric addMore (BufferedReader source){ return null; }
}
```

Test data

You begin by writing a small test program that uses one of these special kinds of numbers. This helps you fix in your mind what you are trying to accomplish and how the methods will be used. It also gives you a way of later testing the correctness of the coding you develop. A simple example is to have the user enter a number of Fraction values; the program responds with the total of the values entered and the smallest of the values entered. Your expectation is that, if the program is executed using e.g.

```
java AddFractions 2/3 -9/10 3/4
```

then the output from the program will be a dialog box saying

```
total = 31/60; smallest = -9/10
```

A good design of such a program is as follows: First check that at least one value was entered on the command line. If so, assign the first value entered to a Numeric variable `total` and also to a Numeric variable `smallestSoFar`. To apply the `valueOf` method to the first String `args[0]`, you have to supply an **executor** (an object reference before the dot in ".valueOf") to show that it is the `valueOf` method from the Fraction class instead of some other `valueOf` method. Each subclass of Numeric will define a ZERO value for this purpose. You then have a loop that goes through the rest of the entries and (a) adds each one to `total`; (b) replaces `smallestSoFar` by the most recently seen value if that one is smaller.

Listing 11.2 contains a reasonable coding. Since the `valueOf` method can throw an Exception if a String value is not in the right format, you need a try/catch statement to intercept any Exception thrown inside the try-block and give an appropriate message.

Listing 11.2 Application program to test Fraction operations

```
import javax.swing.JOptionPane;

/** Add up all Fraction values given on the command line.
 *  Print the total and the smallest one entered. */

public class AddFractions
{
   public static void main (String[] args)
   {  String response = "No values entered.";               //1
      if (args != null && args.length > 0)                  //2
         try                                                //3
         {  Numeric total = Fraction.ZERO.valueOf (args[0]); //4
            Numeric smallestSoFar = total;                  //5
            for (int k = 1;  k < args.length;  k++)         //6
            {  Numeric data = total.valueOf (args[k]);      //7
               total = total.add (data);                    //8
               if (smallestSoFar.compareTo (data) > 0)      //9
                  smallestSoFar = data;                     //10
            }                                               //11
            response = "total = " + total.toString()        //12
                  + "; smallest = " + smallestSoFar.toString();
         }catch (RuntimeException e)                        //14
         {  response = "Some entry was not fractional.";    //15
         }                                                  //16
      JOptionPane.showMessageDialog (null, response);       //17
      System.exit (0);                                      //18
   }  //=====================
}
```

Exercise 11.1* Write the heading of a `divide` method to divide a Numeric executor by a whole-number value and the heading of a `before` method that tells whether a Numeric executor is less than another Numeric value.

11.2 Abstract Classes And Interfaces

At first you think you might declare Numeric as an interface, since it makes no sense to give most of these methods bodies in the Numeric class. But then you realize that some of the methods can have definitions that make sense for all three subclasses. An interface has only method headings, with no method bodies allowed. So you need something in between an interface (where all methods must be overridden) and a "concrete" superclass (all methods defined, so overriding is always optional). The recommended Java solution in this case is an **abstract class**.

You can put `abstract` in an instance method heading if you replace the method's body by a semicolon. This makes the class it is in an abstract class. You must also put `abstract` in the class heading to signal this. Any concrete class that extends the abstract class must have coding for each of the abstract methods. Reminder: A method call is **polymorphic** if at runtime the call could execute any of several different method definitions (i.e., codings).

You cannot have an abstract class method, because class methods cannot be overridden. You may have a constructor in an abstract class if you wish, but you cannot call it except by using `super` in a subclass. A reasonable abstract Numeric class is shown in Listing 11.3 (see next page), leaving eight methods to override in each subclass.

You might at some point want to read some Fractions in from the keyboard or from a disk file and add them to a Fraction object that already has a value. The command `y = x.addMore(someBufferedReaderObject)` would do this, using the `addMore` method from the Numeric class and a Fraction variable `x`. Since the executor is a Fraction object and `addMore` is an instance method, the runtime system uses the `valueOf` and `add` methods from the Fraction class. Reminder on disk files:

- `new BufferedReader (new FileReader (someString))` opens the file that has the given name and produces a reference to it.
- `someBufferedReader.readLine()` produces the next line in the file (with the `\n` removed), except it produces the null String when it reaches the end of the file.
- Both of these uses of BufferedReader can throw an IOException. IOException and BufferedReader are both in the `java.io` package.

Interfaces

The phrase `implements Comparable` in the heading for the Numeric class means that any Numeric object, or any object from a subclass of Numeric, can be used in any situation that requires a Comparable object. This is because the Numeric class contains the standard `compareTo` method. The `compareTo`, `add`, `subtract`, and `multiply` methods throw a ClassCastException or NullPointerException if the parameter is not a non-null object of the same subclass as the executor.

You may declare a variable or parameter to be of Comparable type, but you cannot use the phrase `new Comparable()` to create Comparable objects. Comparable is the name of an **interface**, not a class. When you put the word `interface` in a heading instead of `class`, it means that all you can have inside the definition of the interface are (a) instance method headings with a semicolon in place of the method body, and (b) final class variables. The former means that an interface has form without substance.

Listing 11.3 The abstract Numeric class

```
import java.io.BufferedReader;

public abstract class Numeric implements Comparable
{
   public abstract Numeric valueOf (String par);
   public abstract String toString();
   public abstract double doubleValue();
   public abstract boolean equals (Object ob);
   public abstract int compareTo (Object ob);
   public abstract Numeric add (Numeric par);
   public abstract Numeric subtract (Numeric par);
   public abstract Numeric multiply (Numeric par);


   /** Return the larger of the two Numeric values. Throw an
    *  Exception if they are not both non-null values of the
    *  same Numeric type.*/

   public static Numeric max (Numeric data, Numeric other)
   {  return (data.compareTo (other) > 0)  ?  data  :  other;
   }  //=====================


   /** Return the square of the executor. */

   public Numeric square()
   {  return this.multiply (this);
   }  //=====================


   /** Read Numeric values from the BufferedReader and add them
    *  to the executor, until null is seen or until a value
    *  cannot be added to the executor.  Return the total. */

   public Numeric addMore (BufferedReader source)
   {  Numeric total = this;
      try
      {  Numeric data = this.valueOf (source.readLine());
         while (data != null && total != null)
         {  total = total.add (data);
            data = this.valueOf (source.readLine());
         }
      }catch (Exception e)
      {   // no need for any statements here; just return total
      }
      return total;
   }  //=====================
}
```

You must compile a file containing an interface definition before you can use it. The Comparable interface in the standard library has a single method heading (and is already compiled for you). The complete compilable file `Comparable.java` is as follows:

```
package java.lang;
public interface Comparable
{  public int compareTo (Object ob);
}
```

The two primary reasons why you might have a class implement an interface instead of extend another class are as follows:

1. You will not or cannot give the logic (the body) of any of the methods in a superclass; you want each subclass to define all methods in the superclass. So you use an interface instead of a superclass.
2. You want your class to inherit from more than one class. This is not allowed in Java, because it can create confusion. A class may extend only one concrete or abstract class. But it may implement many interfaces. A general class heading is as follows:

```
class X extends P implements Q, R, S
```

The reason an interface cannot contain a method heading for a class method is that, if `someMethod` is a class method, the method definition to be used for `sam.someMethod()` is determined at compile-time from the class of the variable `sam`, not at run-time from the class of the object.

The phrase X **implements** Y is used only when Y is an interface; X should have every method heading that Y has, but with a method body for each one. That is, Y declares the methods and X defines them (unless X is an "abstract" class).

Early and late binding

When you call Numeric's `addMore` method in Listing 11.3 with a Fraction executor, most statements in that `addMore` method refers to a Fraction object. For instance, since `this` is a Fraction object, the first statement makes `total` refer to a Fraction object. Similarly, the runtime system uses the `valueOf` method from the Fraction class, which returns a reference to a Fraction object to be stored in `data`. If, however, you called the `addMore` method with a Complex executor, then most statements in that method would refer to a Complex object.

Each of the method calls in the method definitions of Listing 11.3 is polymorphic except perhaps `source.readLine()`. For instance, the `square` method calls the `multiply` method of the Fraction class if the executor is a Fraction object, but it calls the `multiply` method of the Complex class if the executor is a Complex object.

The runtime system decides which method is called for during execution of the program, depending on the class of the executor. This is called **late binding** (since it binds a method call to a method definition). When a class has no subclasses, the compiler can do the binding; that is **early binding**. Late binding is somewhat slower and more complex, but it gives you greater flexibility. And tests have shown that it is no slower than an efficiently implemented logic using if-statements or the equivalent. Late binding is possible only with instance methods, not class methods. That is why the `valueOf` method is an instance method, even though it does not really use its executor.

Language elements

The heading of a compilable unit can be: public abstract class ClassName
or: public interface InterfaceName

You may replace "public" by "public abstract" in a method heading if (a) it is in an abstract class, (b) it is a non-final instance method, and (c) you replace the body of the method by a semicolon.
If your class extends an abstract class and you want to construct objects of your class, your class must have non-abstract methods that override each abstract method in the abstract class.
All methods in an interface must be non-final instance methods with a semicolon in place of the body. They are by default "public abstract" methods. Field variables must be final class variables.
You may add the phrase "implements X" to your class heading if X is an interface and if each method heading in the interface appears in your class. Use the form "implements X, Y, Z" to have your class implement several interfaces.

Exercise 11.2 Write a class method named `min3` that could be added to the Numeric class: It finds the smallest of three Numeric parameters.
Exercise 11.3 (harder) Write a method `public Numeric smallest (BufferedReader source)` for Numeric: it finds the smallest of a list of Numeric values read in from `source`, analogous to `addMore`.
Exercise 11.4** Write a method `public boolean equalPair (BufferedReader source)` for Numeric: It reads in a list of Numeric values from a given BufferedReader source and then tells whether or not any two consecutive numbers were equal. Explain how the runtime system knows which method definition to use.
Reminder: Answers to unstarred exercises are at the end of the chapter.

11.3 More Examples Of Abstract Classes And Polymorphism

You may put the word **`final`** in a method heading, which means no subclass can override that method. This allows the compiler to apply early binding to calls of that method, which makes the program run a bit faster. For instance, you would probably want to make the Fraction methods mentioned in the previous section final to speed execution, since you do not see why anyone would want to extend the Fraction class.

Also, you may put the word `final` in a class heading, which means that the class cannot have subclasses. For instance, the developers of the standard String class made it a `final` class. So when Chapter Six created a StringInfo class of objects that would do what Strings do, only more, it could not make StringInfo a subclass of String. Instead, the StringInfo class had a String object as its only instance variable. That way, whenever you ask a StringInfo object to do something, it does it with its String instance variable. In other words, the StringInfo class uses composition rather than inheritance.

Abstract games

The BasicGame class in Listing 4.3 describes how to play a trivial game in which the user guesses the secret word, which is always "duck". Its true purpose is to serve as a parent class for interesting games such as Mastermind and Checkers. It would make more sense to make BasicGame an abstract class, thereby forcing programmers to override some of its methods. The seven methods in the BasicGame class are as follows:

```
public void playManyGames()   // calls playOneGame
public void playOneGame()     // calls the 5 below
public void askUsersFirstChoice()
public boolean shouldContinue()
public void showUpdatedStatus()
public void askUsersNextChoice()
public void showFinalStatus() // just says the player won
```

Of these seven methods, a subclass should not override the first two, since their logic is what is common to all games. A subclass may choose to override `showFinalStatus` from the BasicGame class (as did Nim in Listing 4.8) or may leave it as inherited (as did GuessNumber in Listing 4.6 and Mastermind in Listing 4.9). Every subclass of BasicGame should override the other four methods.

You can enforce these three rules as follows: Declare the first two methods as `final` (a method declared as `final` cannot be overridden in a subclass). Leave the `showFinalStatus` method as a normal "concrete" method, so overriding is optional. Declare the other four as abstract, which means that they must be overridden. That gives the compilable class shown in Listing 11.4 (see next page). If this definition replaces the BasicGame class in Chapter Four, then the three different game subclasses in Listings 4.6, 4.8, and 4.9 will all compile and run correctly as is.

Listing 11.4 The BasicGame class as an abstract class

```
public abstract class BasicGame
{
   public final void playManyGames()
   {  playOneGame();
      while (javax.swing.JOptionPane.showConfirmDialog (null,
                "again?") == javax.swing.JOptionPane.YES_OPTION)
         playOneGame();
   }  //====================

   public final void playOneGame()
   {  askUsersFirstChoice();
      while (shouldContinue())
      {  showUpdatedStatus();
         askUsersNextChoice();
      }
      showFinalStatus();
   }  //====================

   public abstract void askUsersFirstChoice();
   public abstract boolean shouldContinue();
   public abstract void showUpdatedStatus();
   public abstract void askUsersNextChoice();

   public void showFinalStatus()
   {  javax.swing.JOptionPane.showMessageDialog (null,
                     "That was right.  \nCongratulations.");
   }  //====================
}
```

Abstract Buttons

Sun's Swing classes include several different kinds of graphical components called buttons: JButton (the basic kind of button), JToggleButton (click it to change its state from "selected" to "deselected" or back again), and JMenuItem (one of several buttons on a menu). All have different appearances, but they have some behaviors in common, such as `someButton.setText(someString)` to change what the button says on it and `someButton.setMnemonic(someChar)` to say what character can be used as the "hot key".

The AbstractButton class in the Swing library contains the methods that are common to all of these kinds of buttons, including `setText` and `setMnemonic`. The advantage is that each of these common methods only has to be defined once, in the AbstractButton class, but a common method can be called for any of the three kinds of buttons.

Abstract Animals

A particular application may require you to keep track of fish and other inhabitants of the ocean. You might have a Shark class, a Tuna class, a Porpoise class, and many others. Any behaviors that are common to several classes of animals should be specified in a superclass and inherited by its subclasses. For instance, all animals eat. So you could have an Animal superclass that declares a method for eating. Animal should be an abstract class, since any concrete animal you create will be of a specific animal subclass. You could define the Animal class as follows:

```
public abstract class Animal
{  public abstract void eat (Object ob);
}
```

There are two kinds of animals, those that move around in the ocean and those that do not. It makes sense to abstract the behavior of swimming for a subclass of Animals. So you could have an abstract subclass of the Animal class as follows:

```
public abstract class Swimmer extends Animal
{  public abstract void swim();
}
```

Note that the Swimmer class does not have to code the `eat` method, since Swimmer is abstract. But any concrete class (i.e., a class that you want to create an instance of) that extends Swimmer must implement both `eat` and `swim`. For instance, a prototype for the Shark class could be as follows:

```
public class Shark extends Swimmer
{  public void swim()
   {  System.out.println ("shark is swimming");
   }
   public void eat (Object ob)
   {  System.out.println ("shark eats " + ob.toString());
   }
   public String toString()
   {  return "shark";
   }
}
```

Question What difference would it make if an abstract class X were simply declared as a regular class X with empty method bodies? Answer Only one difference: The compiler will not warn you if you create an object of type X or if you extend X with a class that forgets to implement one of the methods that should be overridden.

Question What difference would it make if an interface Y were simply declared as an abstract class Y with all of its methods marked as abstract? Answer Only one difference: You cannot write a class that inherits from Y and also from another class.

The instanceof operator

A class that overrides the `equals` method from the Object class must have its parameter be of Object type. If the parameter is of the same class as the executor, you test something such as the names or other instance variables to see if they are equal. If the parameter is not of the same class as the executor, then of course it is not equal to the executor. Do not throw an Exception in that case; just return `false`. The problem is, how do you ask an object whether it is of the right class without throwing a ClassCastException?

You will need the **instanceof** operator for this purpose: If `x` is an object variable and Y is a class, then `x instanceof Y` is true when `x` is not null and `x` is an object of class Y or of a subclass of Y; if Y is an interface, `x` must be of a class that implements Y.

The Time class in Listing 4.5 has two int instance variables `itsHour` and `itsMin` to specify the time of day. It should have an `equals` method that overrides the Object method. The `equals` method for Time could be defined as follows:

```
public boolean equals (Object ob)    // in the Time class
{  if ( ! (ob instanceof Time))
      return false;
   Time given = (Time) ob;
   return this.itsHour == given.itsHour
         && this.itsMin == given.itsMin;
}  //======================
```

The `instanceof` test guards against having the third line of this method throw a ClassCastException method; calling the `equals` method should never throw an Exception. Similarly, the Worker class in Listing 7.6 has an `equals` method whose parameter is a Worker. It would be good to have another `equals` method that overrides the standard Object method. The following logic calls on the existing Worker `equals` method to do most of the work (overloading the `equals` method name):

```
public boolean equals (Object ob)   // in the Worker class
{  return ob instanceof Worker && this.equals ((Worker) ob);
}  //======================
```

If a class of objects implements the Comparable interface, its objects may be passed to a Comparable parameter of some method. That method might use the `equals` method as well as the `compareTo` method, which is one good reason for having an `equals` method that overrides the Object method whenever you have a `compareTo` method. Note that both Time and Worker would naturally implement Comparable.

Another good reason to have an `equals` method with an Object parameter is that you might have an array of Animals or Numerics, where the objects in the array could be of several different subclasses. Then you could call the `equals` method with any one of those objects as the executor, knowing the runtime system will select the right definition of this polymorphic `equals` call according to the actual class of the executor.

Caution If you get the compiler message "class X must be declared as abstract", it may not be true. You might simply have forgotten to implement one of the abstract methods in an abstract superclass of X, or you might have forgotten to implement one of the methods required by an interface that X implements.

Technical Note You may be wondering why the operator `instanceof` does not capitalize the 'O' of "of". The Java convention is to do so for identifiers, i.e., names of variables, methods, and classes. But `instanceof` is none of those; it is a keyword.

Language elements
If you put "final" in a method heading, you cannot override it in a subclass.
If you put "final" in a class heading, you cannot make a subclass of it.
If Y is a class, x instanceof Y means x is an object in class Y or in a subclass of Y (so x != null).
The `equals` method in the Object class has an Object parameter. Any method you write that overrides that `equals` method should never to throw an Exception, so the first statement in such a method is usually if (! (ob instanceof Whatever)) return false;.

Exercise 11.5 Write an `equals` method for a Person class; have it override the Object `equals` method. Two Persons are equal if they have the same values of the instance variables `itsLastName` (a String) and `itsBirthYear` (an int).
Exercise 11.6 Write a prototype for the Tuna subclass of the Swimmer class. However, Tunas only eat things that swim, so make sure the `eat` method states that the Tuna is still hungry if the object it is given to eat is not a Swimmer.
Exercise 11.7* Modify the answer to the preceding exercise so that, if a Tuna is given a Shark to eat, then the Shark eats the Tuna instead.
Exercise 11.8* Override Object's `equals` method for the Worker class without calling on the other `equals` method. Just test directly whether two Worker objects have the same last name and the same first name (`itsFirstName` and `itsLastName` are Strings).
Exercise 11.9** Improve the Animal class and its subclasses to add an instance method `getWeight()` for all Animals and a corresponding instance variable set by a parameter of an Animal constructor. Also add an Animal instance variable `itsFuelReserves` that is to be updated by the weight of whatever the Animal eats. Then modify the Shark's `eat` method to call on the appropriate methods.

11.4 Double, Integer, And Other Wrapper Classes

Quite a few of the programs the mathematicians need involve whole numbers that are extremely large, up to 40 to 50 digits (somewhat over one septillion of septillions). The problem is that the Java language does not provide for whole numbers that big. The only primitive numeric types in Java are byte, short, int, long, double, and float.

The six primitive number types

A **byte of storage** is 8 on-off switches, or 8 high/low voltages, or 8 binary digits (1's and 0's). So a byte can store any of 256 different values (since the eighth power of 2 is 256).

You may define a variable as **byte x**, which means that x stores a number in the range from -128 to 127. Note that there are exactly 256 different values in that range, so it is stored in one byte of space in RAM.

You may define a variable as **short x**, which means that x stores a number in the range from -32,768 to 32,767. Note that there are exactly 65,536 different values in that range, which is the second power of 256. So it is stored in two bytes of space in RAM. This is 16 bits, since one byte is 8 bits.

You may define a variable as **int x**, which means that x stores a number in the range of plus or minus just over 2 billion. The reason is the fourth power of 256 is just over 4 billion. So an int value takes up four bytes of space in RAM, which is 32 bits.

You may define a variable as **long x**, which means that x stores a number in the range of plus or minus just over 8 billion billion. The reason is that the eighth power of 256 is just over 16 billion billion (a number with 19 digits). So a long value takes up eight bytes of space in RAM. This is 64 bits, since one byte is 8 bits. You can indicate that an integer value in your program is of the long type with a trailing capital L, as in 47L or 1000000L.

Figure 11.1 Difference in sizes of whole-number values

You may define a variable as **double x**, a value with 64 bits, which takes up eight bytes of storage. It has only about 15 decimal digits of accuracy; the rest of the storage is used for the minus sign (if there is one) and to note the power of sixteen it is multiplied by (scientific notation, but with a base of 16 rather than 10).

You may define a variable as **float x**, a value with 32 bits, which takes up four bytes of storage. It is a decimal number with about 7 decimal digits of accuracy, since it is stored in scientific notation with a base of 16.

You may write a double value in **computerized scientific notation** within a Java class, e.g., 3.724E20 is 3.724 times 10-to-the-twentieth-power, and 5.6E-15 is 5.6 times 10-to-the-negative-fifteenth-power. The `toString()` method of the Double class produces this scientific notation unless the number is at least 0.001 and less than 10 million.

Wrapper classes for primitives

The Sun standard library offers eight special **wrapper classes**, one for each of the eight primitive types. Their names are **Double**, **Integer**, **Float**, **Long**, **Short**, **Byte**, **Character**, and **Boolean**. These classes give you an easy way to convert one of the primitive types of values to an object and back again. They are all in the `java.lang` package, which means that they are automatically imported into every class; you do not have to have an explicit import directive.

One use of these classes is for polymorphic processing of several different kinds of objects. Since each of these eight wrapper classes inherits from the Object class, you can apply the `equals` method and the `toString` method to any of their objects. Each of these wrapper classes overrides these two methods to provide the obvious meaning to them. For instance, you might have an array of Objects which can be any of the eight kinds of values. You might then search for an Object equal to a target Object with this logic:

```
for (int k = 0;  k < item.length && item[k] != null;  k++)
{  if (item[k].equals (target))
      System.out.println (item[k].toString());
}
```

Numeric wrappers

The Double class has the class method `parseDouble` for converting a String value to double value. It can throw a NumberFormatException if the numeral in the string of characters is badly formed. Similarly, the Integer class has the `parseInt` class method and the Long class has the `parseLong` class method. All of these can throw a NumberFormatException. The `parseDouble` method allows spaces before or after the numeral, but `parseInt` does not.

Each of the six numeric wrapper classes has a constructor with a parameter of the corresponding primitive type and a constructor that has a String parameter. So you can create a new object of one of these wrapper types using e.g. `new Double(4.7)` or `new Integer("45")`. The latter is equivalent to `new Integer (Integer.parseInt("45"))`.

All six of the numeric wrapper classes are subclasses of the abstract **Number** class in the `java.lang` package. This chapter would have used Number rather than Numeric except that the Number class does not have many of the methods that the clients wanted; Number only has six parameterless instance methods for converting to each of the six primitive types: `doubleValue`, `intValue`, `longValue`, `byteValue`, `shortValue`, and `floatValue`. Each of the six numeric wrapper classes overrides those six methods to convert the object to any of the six numeric primitive types.

All wrapper classes except Boolean implement the Comparable interface. That is, they have the standard `compareTo` method required to be Comparable. And they each have final class variables MAX_VALUE and MIN_VALUE with the appropriate meaning.

It is occasionally useful to "break up" an 8-byte long value stored in e.g. a long variable `sam` into two 4-byte int values. If you define `long n = 1L + Integer.MAX_VALUE`, then `(int)(sam / n)` gives the first 4 bytes of `sam`'s value and `(int)(sam % n)` gives the last 4 bytes of `sam`'s value. Read the Sun documentation about `Double.doubleToLongBits(double)` and `Double.longBitsToDouble(long)` to see how to "re-interpret" an 8-byte double value as a long value and vice versa.

Integer methods for different number bases

The **binary form** of a number is a sequence of 1's and 0's. Its value is the sum of several numbers: 1 if the last digit is 1, 2 if the next-to-last digit is 1, 4 if the third-to-last digit is 1, 8 if the fourth-to-last digit is 1, etc. That is, a 1 at any place has twice the value of a 1 at the place to its right. For instance, "1111" represents `1*8 + 1*4 + 1*2 + 1` which is 15 in base 10.

The **octal form** of a number is a sequence of digits in the range 0 to 7 inclusive. Its value is the sum of several numbers: n if the last digit is n, 8*n if the next-to-last digit is n, 64*n if the third-to-last digit is n, 512*n if the fourth-to-last digit is n, etc. That is, a digit at any place indicates 8 times the value of the same digit at the place to its right. For instance, "2537" represents `2*512 + 5*64 + 3*8 + 7` which is 1375 in base 10.

The **hexadecimal form** of a number is a sequence of digits in the range 0 to F inclusive (we run out of ordinary digits after 9, so we use A=10, B=11, C=12, D=13, E=14, and F=15). Its value is the sum of several numbers: n if the last digit is n, 16*n if the next-to-last digit is n, 256*n if the third-to-last digit is n, 4096*n if the fourth-to-last digit is n, etc. That is, a digit at any place indicates 16 times the value of the same digit at the place to its right. For instance, "B4E" can be thought of as (11)(4)(14), which represents `11*16*16 + 4*16 + 14`, which is `2816 + 64 + 14`, which is 2894 in base 10.

`Integer.toString (n, 2)` gives the binary form of the int value `n`. If you replace the 2 by 8, 16, or 10, you get the octal, hexadecimal, or decimal form of the int value `n`, respectively. Conversely, `Integer.parseInt (someString, 2)` returns the int value corresponding to the string of binary digits (and similarly for 8, 16, and 10). Some examples of the use of the overloaded `toString` method are as follows:

```
Integer.toString (43, 2) is "101011", i.e., 32 + 8 + 2 + 1.
Integer.toString (43, 8) is "53", i.e., 5 * 8 + 3.
Integer.toString (43, 16) is "2B", i.e., 2 * 16 + 11.
```

Useful Boolean and Character methods

The phrase `new Boolean(someString)` gives the Boolean object `Boolean.TRUE` if `someString` is the word `TRUE` (in any combination of upper- and lowercase), otherwise it gives `Boolean.FALSE`. And `new Boolean(someBoolean)` gives the wrapper object for the primitive value.

If `x` is a Boolean object, then `x.booleanValue()` gives the boolean primitive value `true` or `false` as appropriate. Boolean does not implement the Comparable interface, but it does have instance methods `equals` and `toString` that override the corresponding methods in the Object class.

The only constructor for the Character class has a char parameter, as in `new Character('B')`. If `x` and `y` are Character objects, then `x.compareTo(y)`, `x.equals(y)`, and `x.toString()` all have the standard meanings. `x.charValue()` is the char equivalent of `x`.

Language elements
You may declare variables of type byte, short, or float. The primary reason to do so is to save space. Most people do not use these types except for an array of thousands of values. This book does not use these three types outside of this section.

Exercise 11.10 Convert each of these numbers to binary, octal, and hexadecimal: 5, 11, 260.
Exercise 11.11 Convert each of these hexadecimal numbers to decimal: F, 5D, ACE.

Part B Enrichment And Reinforcement

11.5 Implementing The Fraction Class

A Fraction object is a virtual fraction representing e.g. 2/3. Since you already know the eight operations you want (described in the upper part of Listing 11.3), you can move on to deciding about the instance variables. You will also find it useful to have a final class variable representing ZERO.

The concrete form of a Fraction

After mulling the situation over for a while, you decide that the best concrete form of a Fraction object is two int instance variables representing the numerator and the denominator of the fraction, reduced to lowest terms. The logic for addition, multiplication, and other operations can be worked out from there. Figure 11.2 shows a representation of a Fraction object. It is a good idea to write down the internal invariant separately (the condition that each Fraction method will make sure is true about its instance variables when the method exits):

Internal invariant for Fractions A Fraction has two int instance variables `itsUpper` and `itsLower`. `itsLower` is positive and the two values have no integer divisor over 1 in common. This pair of int values represents the quotient `itsUpper/itsLower`.

Figure 11.2 A Fraction variable and its object

The Fraction constructor

You will need a constructor that creates a Fraction object from two given int values that will be the numerator and the denominator. You could just set the two instance variables of the Fraction to those two values, except for three problems your logic must manage:

1. If the denominator is zero, there is no such number. A reasonable response is to just create the fraction 0/1.
2. If the denominator is negative, then multiply both the upper and lower parts of the fraction by -1 before proceeding, since the denominator is supposed to be positive.
3. You have to reduce the fraction to lowest terms.

You may have several other methods that require reducing a fraction to lowest terms. So that reducing logic should be in a private method. The command `this.reduceToLowestTerms()` in the constructor has the object being constructed execute the `reduceToLowestTerms` method. That is, `this` refers to the object being constructed within constructor methods and to the executor within ordinary methods. The coding for this constructor is in the upper part of Listing 11.5 (see next page).

The valueOf method

For the `valueOf` method, you return null if the String value is null or the empty string. Otherwise you apply the `Integer.parseInt` method to the parts before and after the slash to get `itsUpper` and `itsLower` parts. This may throw a NumberFormatException. You may then call the Fraction constructor to check for zeros, negatives, or values that should be reduced. The coding for this method is in the lower part of Listing 11.5.

Listing 11.5 The Fraction class, partial listing

```
public class Fraction extends Numeric
{
   /** A constant representing ZERO. */

   public static final Fraction ZERO = new Fraction (0, 1);
   ////////////////////////////////
   private int itsUpper;  // the numerator of the fraction
   private int itsLower;  // the denominator of the fraction


   /** Construct a Fraction from the given two integers. */

   public Fraction (int numerator, int denominator)
   {  if (numerator == 0 || denominator == 0)
      {  itsUpper = 0;
         itsLower = 1;
      }
      else if (denominator < 0)
      {  itsUpper = - numerator;
         itsLower = - denominator;
         this.reduceToLowestTerms();
      }
      else
      {  itsUpper = numerator;
         itsLower = denominator;
         this.reduceToLowestTerms();
      }
   }  //=====================


   /** The parameter should be two ints separated by '/', e.g.
    *  "2/3".  Return null if par is null or "".  Otherwise
    *  throw a NumberFormatException if the wrong form. */

   public Numeric valueOf (String par)
   {  if (par == null || par.length() == 0)
         return null;
      int k = 0;
      while (k < par.length() - 1 && par.charAt (k) != '/')
         k++;
      return new Fraction
                  (Integer.parseInt (par.substring (0, k)),
                   Integer.parseInt (par.substring (k + 1)));
   }  //=====================
}
```

The toString and equals methods

The `toString` method simply returns the two int values with a slash between them, so that just takes one statement to implement. The `equals` method is more difficult. It has an Object as the given parameter, not a Fraction. This is required to have it override the `equals` method in the Object class. So you cannot refer to `par.itsUpper` in the body of `equals`. Such an expression requires that `par` be a Fraction variable.

No outside class should call Fraction's `equals` method unless the parameter does in fact refer to a Fraction object. You can test for this using the `instanceof` operator defined earlier: `ob instanceof Fraction` is `true` if ob refers to a Fraction object at runtime, otherwise it is `false`. Once you know that ob in fact refers to a Fraction object, you may refer to the `itsUpper` instance variable of that Fraction object with the phrase `((Fraction) ob).itsUpper`. This coding is in the upper part of Listing 11.6.

Listing 11.6 Three more methods in the Fraction class

```
/** Express the Fraction as a String, e.g., "2/3". */

public String toString()
{  return this.itsUpper + "/" + this.itsLower;
}  //=====================


/** Tell whether the two Fraction objects are equal. */

public boolean equals (Object ob)
{  return ob instanceof Fraction
         && ((Fraction) ob).itsUpper == this.itsUpper
         && ((Fraction) ob).itsLower == this.itsLower;
}  //=====================


/** Return the sum of the executor and par. */

public Numeric add (Numeric par)
{  Fraction that = (Fraction) par;    // for simplicity
   return new Fraction (this.itsUpper * that.itsLower
                      + this.itsLower * that.itsUpper,
                        this.itsLower * that.itsLower);
}  //=====================
```

A method that overrides the basic Object class's `equals` method is never to throw a NullPointerException or ClassCastException, so you need to use the `instanceof` operator to guard against these Exceptions.

The `(Fraction)` part of that phrase is a cast, just as `(int)` is. It says that ob can be treated as a reference to a Fraction object. However, if you use a phrase such as `(Fraction) ob` more than once or twice in some coding, it is clearer and more efficient if you assign the value to a Fraction variable and use that Fraction variable instead. This is illustrated in the `add` method discussed next.

The add method

The addition of one Fraction to another gives a new Fraction as a result. You should remember that you add two fractions by "cross-multiplying": The new `itsUpper` value is the first's `itsUpper` times the second's `itsLower`, added to the first's `itsLower` times the second's `itsUpper`. And the new `itsLower` value is the first's `itsLower` times the second's `itsLower`. Once you make this calculation, you can create a new Fraction object out of the two results and return it. This coding is in the lower part of Listing 11.6.

The reduceToLowestTerms method

The `reduceToLowestTerms` method should divide both `itsUpper` and `itsLower` by the same whole number wherever possible. You could proceed in this manner:

1. If both `itsUpper` and `itsLower` are divisible by 2, divide out the 2 and repeat.
2. If both `itsUpper` and `itsLower` are divisible by 3, divide out the 3 and repeat.
3. If both `itsUpper` and `itsLower` are divisible by 5, divide out the 5 and repeat.
4. If both `itsUpper` and `itsLower` are divisible by 7, divide out the 7 and repeat, etc.

After the first step, you have divided out 2 until one of the two numbers is odd. So it is sufficient to try only odd divisors thereafter. A little more thought shows that you do not have to try any divisor that is more than the smaller of `itsUpper` and `itsLower`. To calculate the smaller of `itsUpper` and `itsLower`, you have to use the absolute value of `itsUpper`, because it could be a negative number.

Dividing out a whole number is done in two places in the main logic of `reduceToLowestTerms`, so a separate private method is desirable. This method simply divides both parts of a Fraction by the parameter until it will not go evenly into one of them. Listing 11.7 has this `reduceToLowestTerms` method, as well as the obvious logic for the `doubleValue` method. The other three methods of the Fraction class are left as exercises. Figure 11.3 gives the UML class diagram for the whole Fraction class.

Listing 11.7 Additional methods in the Fraction class

```
/** Reduce the fraction to lowest terms.  Precondition:  the
 *  denominator is positive and the numerator is non-zero. */

private Fraction reduceToLowestTerms()
{  divideOut (2);
   int limit = Math.min (Math.abs (itsUpper), itsLower);
   for (int divider = 3;  divider <= limit;  divider += 2)
      divideOut (divider);
   return this;
}  //=====================


/** "Cancel" out divider as much as possible.  Precondition:
 *  itsUpper, itsLower, and divider are all non-zero. */

private void divideOut (int divider)
{  while (itsUpper % divider == 0 && itsLower % divider == 0)
   {  itsUpper /= divider;
      itsLower /= divider;
   }
}  //=====================


/** Return the approximate value as a double value. */

public double doubleValue()
{  return itsUpper * 1.0 / itsLower;
}  //=====================

// these three are stubbed and left as exercises
public int     compareTo (Object ob)            { return 0; }
public Numeric subtract (Numeric par)        { return null; }
public Numeric multiply (Numeric par)        { return null; }
```

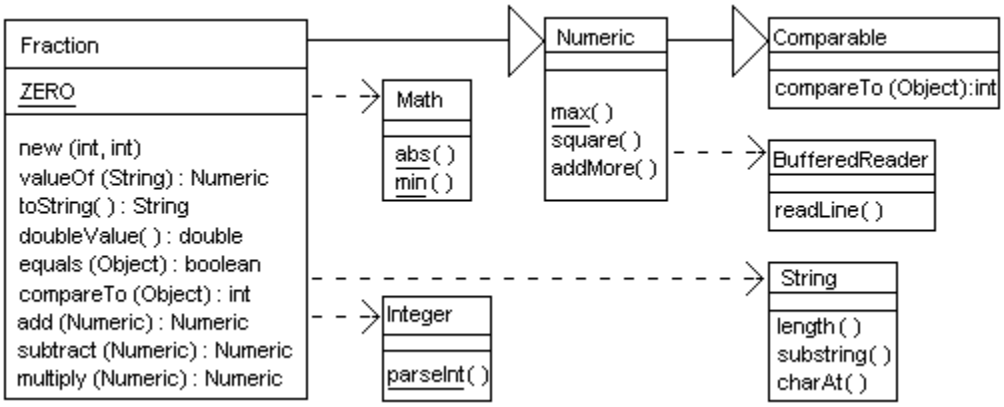

Figure 11.3 UML class diagram for the Fraction and Numeric classes

Exercise 11.12 Explain why you cannot replace `{itsUpper = 0; itsLower = 1;}` by `return ZERO;` in the logic for the Fraction constructor.
Exercise 11.13 In Listing 11.6, the `equals` method tests `ob instanceof Fraction` but the `add` method does not. Explain why it is not necessary.
Exercise 11.14 Write the Fraction method `public Numeric subtract (Numeric par)`, giving the difference of two Fractions (analogous to the `add` method).
Exercise 11.15 The String class has a method `indexOf(someChar)` that returns the index where the first instance of `someChar` is found in the String. It returns `-1` if `someChar` is not there. Use it appropriately to rewrite the `valueOf` method of the Fraction class.
Exercise 11.16 Write the Fraction method `public int compareTo (Object ob)`.
Exercise 11.17 Write a Fraction method `public Numeric divide (Numeric par)`, the result of dividing the executor by `par`. Return null if `par` represents zero.
Exercise 11.18* Write the Fraction method `public Numeric multiply (Numeric par)`, giving the product of two Fractions (analogous to the `add` method).
Exercise 11.19* How would you revise the `add` method to return `Fraction.ZERO` when a ClassCastException arises?

11.6 Implementing The Complex Class

Another category of software that the mathematicians need involves the use of complex numbers. These are numbers that have a real part and an imaginary part, such as `3 - 4i`. The `i` stands for the square root of negative 1.

The Complex class extends the Numeric class and defines objects that represent complex numbers. Since you already know what operations you want, you can move on to deciding about the instance variables. Clearly, each Complex object should have a real part and an imaginary part. The following is a rather obvious internal invariant.

Internal invariant for Complexes A Complex object has two double instance variables `itsReal` and `itsImag`. It represents the complex number `itsReal + itsImag * i`.

You need a constructor to create a Complex object from two given numbers for the real and imaginary parts in that order. And you need methods that let you add, subtract, multiply and compare Complex numbers, and to convert to and from Complex numbers. The Complex class in Listing 11.8 is a start (see next page). It has the constructor and four of the Numeric methods; the other four are left for exercises.

Listing 11.8 The Complex class, partial listing

```
public class Complex extends Numeric
{
   public static final Complex ZERO = new Complex (0, 0);
   ////////////////////////////////
   private final double itsReal;
   private final double itsImag;


   public Complex (double realPart, double imagPart)
   {  itsReal = realPart;
      itsImag = imagPart;
   }  //======================


   public String toString ()
   {  String operator = itsImag < 0  ?  " "  :  " +";
      return itsReal + operator + itsImag + "i";
   }  //======================


   public double doubleValue()
   {  return itsReal;
   }  //======================


   public int compareTo (Object ob)
   {  double diff = this.itsReal - ((Complex) ob).itsReal;
      if (diff == 0)
         return 0;
      else
         return (diff > 0)  ?  1  :  -1;
   }  //======================


   public Numeric add (Numeric par)
   {  Complex that = (Complex) par;
      return new Complex (this.itsReal + that.itsReal,
                          this.itsImag + that.itsImag);
   }  //======================

   // the following are left as exercises
   public Numeric valueOf (String par)          {return ZERO;}
   public boolean equals (Object ob)            {return true;}
   public Numeric subtract (Numeric par)        {return ZERO;}
   public Numeric multiply (Numeric par)        {return ZERO;}
}
```

The `ToString` method has to print a plus sign between the two numerals if the second number is not negative. But a negative number as a minus sign as part of its string form.

The `doubleValue` method returns the real part (non-imaginary part) of the complex number. For the imaginary part of `x`, a person could compute this expression:

```
x.subtract (new Complex (x.doubleValue(), 0)
```

The `compareTo` method in Listing 11.8 compares Complex numbers on the basis of their real parts. The one with the greater real part is considered larger. The `(Complex)`

cast is required, because otherwise the phrase `ob.itsReal` is not acceptable to the compiler -- Object objects in general do not have an `itsReal` instance variable.

The `add` method saves the trouble of making two casts by storing the cast value in a local variable of the Complex type and using it in the calculations. Both `compareTo` and `add` will throw a ClassCastException or a NullPointerException if the parameter is not a non-null Complex object.

You might be wondering how the group of mathematicians handle cases where they just want to use ordinary decimal numbers mixed in with the Fractions and the Complex class. The answer is, you need to develop a subclass of Numeric for decimal numbers as well. Until you get around to it, you can just use Complex objects with the imaginary part equal to zero. Figure 11.4 shows a representation of a Complex object.

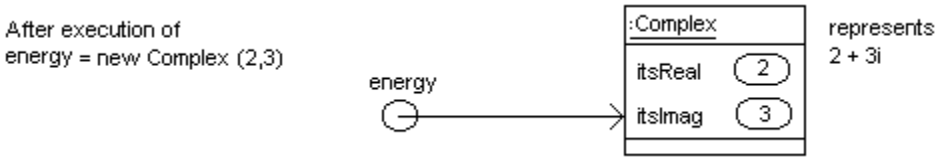

Figure 11.4 A Complex variable and its object

Exercise 11.20 Write the `equals` method of the Complex class, testing whether the two Complex values have the same real and imaginary parts.
Exercise 11.21 Write the `multiply` method of the Complex class.
Exercise 11.22 Revise the Complex `compareTo` method so that it accepts any Numeric object as the parameter and gives a reasonable answer.
Exercise 11.23 Revise the Complex `ToString` method to not print "+0i" and also to not print a zero real part when the imaginary part is nonzero.
Exercise 11.24* Write the `subtract` method of the Complex class.
Exercise 11.25* Write the `valueOf` method for the Complex class. Assume the input is of the same form that `toString` produces, but possibly with extra whitespace.

11.7 Implementing The VeryLong Class Using Arrays

You have to define a VeryLong object class for extremely large whole numbers, up to 40 or 50 digits. Since you already know what they can do (the methods in Numeric), you have to decide what they know (the private instance variables that store the value).

You could use three long values to store up to 54 digits, which is quite enough for your needs. However, problems arise when you try to perform multiplication. It is better to use an array of six int values, one for each group of nine digits of the number being stored. You can multiply two int values and temporarily store the exact result in a long variable; you have no easy way to store the exact result of multiplying two long values.

The internal invariant

You might decide to call the int array `itsItem`. `itsItem[0]` could contain the first (leftmost) nine digits of the VeryLong number, and the rest could go on from there. You would find it easiest to not allow negative VeryLong numbers. You need to check with your clients to make sure that this is acceptable to them.

Internal invariant for VeryLongs A VeryLong object is stored as six int values in `itsItem[0]...itsItem[5]`. Each must be non-negative and have no more than 9 digits. The whole-number value that the object represents is `itsItem[0] * 10`^{45} `+ itsItem[1] * 10`^{36} `+ ... + itsItem[4] * 10`^{9} `+itsItem[5]`.

This concrete description tells you all you need to know to be able to write methods that add, multiply, etc. with VeryLong numbers: The internal invariant will be true when your method begins execution; your job is to make sure it is true when the method finishes. A partial listing of the VeryLong class of numbers is in Listing 11.9.

For this class, it turns out to be more efficient to have two different forms of one billion, an int value and a long value (you want to minimize unnecessary casts). The 'L' appended to a string of digits indicates it is a long value; otherwise it is an int value. The 'L' is required if you have more than 10 digits; it is optional in Listing 11.9.

Listing 11.9 The VeryLong class, partial listing

```
public class VeryLong extends Numeric
{
   public static final VeryLong ZERO = new VeryLong (0, 0, 0);
   private static final int BILLION = 1000000000;
   private static final long LONGBILL = 1000000000L;
   private static final int MAX = 6;
   ////////////////////////////////
   private final int[] itsItem = new int[MAX];  // all zeros


   public VeryLong (long left, long mid, long right)
   {  this.itsItem[0] = (int) (left / LONGBILL);
      this.itsItem[1] = (int) (left % LONGBILL);
      this.itsItem[2] = (int) (mid / LONGBILL);
      this.itsItem[3] = (int) (mid % LONGBILL);
      this.itsItem[4] = (int) (right / LONGBILL);
      this.itsItem[5] = (int) (right % LONGBILL);
   }  //======================


   public String toString ()
   {  String s = "" + itsItem[0];
      for (int k = 1;  k < MAX;  k++)
         s += "," + ((BILLION + itsItem[k]) + "").substring(1);
      return s;
   }  //======================


   public Numeric add (Numeric par)
   {  VeryLong that = (VeryLong) par;
      VeryLong result = new VeryLong();
      for (int k = 0;  k < MAX;  k++)
         result.itsItem[k] = this.itsItem[k] + that.itsItem[k];
      for (int k = MAX - 1;  k > 0;  k--)
      {  if (result.itsItem[k] >= BILLION)
         {  result.itsItem[k] -= BILLION;
            result.itsItem[k - 1]++;
         }
      }
      return result.itsItem[0] >= BILLION  ?  null  :  result;
   }  //======================


   private VeryLong()     // only used by other VeryLong methods
   {
   }  //======================
}
```

The constructor and the toString method

How will people supply a numeric form of a very long number to be made into a VeryLong object? A reasonable way is to have them break the number up into three 18-digit parts, left to right, and supply those as three long values. So `new VeryLong (0L, 217L, 333333333222222222L)` would give the number 217,333,333,333,222,222,222. The constructor only needs to split each of the three long values into two 9-digit parts and store them in the appropriate six components of the array.

What if one of the three parameters is negative or has nineteen digits? Just as with Fractions, you should create the equivalent of ZERO when one of the parameters is unacceptable this way. This adjustment is left as an exercise.

The `toString` method should put the commas in the written form of the number. Without them, the numeral would be too hard to read. Grouping digits by threes would give up to 18 groups, which is probably not helpful. So the client agrees that groups of nine digits is better. If the `itsItem` array contains e.g. {0, 0, 0, 37, 12345, 1234567}, you have to supply the missing zeros to make it 37,000012345,001234567.

A simple trick adds the right number of leading zeros: Add a billion to the up-to-9-digit number, convert it to a string of characters, then throw away the initial 1. You do not need to do this for the first part, `itsItem[0]`, but you do for the rest of the components. These methods are in the top part of Listing 11.9.

The add method

To add two VeryLong numbers, you first create a VeryLong object in which to store the `result`. You then add up corresponding components in the two things being added to get the same component of the `result`. But what if one of the sums goes over nine digits? You carry the 1. That is, you subtract a billion from that component of the `result` and add 1 to its next component. But if the leftmost component goes over nine digits, the sum is too big to store, so you are to return null according to the specifications. The accompanying design block expresses this algorithm in Structured English.

DESIGN for the add method in the VeryLong class

1. Name the two values to be added `this` and `that`.
2. Create a VeryLong object to store the result of the addition. Call it `result`.
3. For each of the six possible indexes do the following...
   Add the current components of `this` and `that` to get the corresponding component of `result`.
4. For each components of `result` except one, starting from the rightmost digits of the number and working towards the left, do the following...
   If the current component has more than nine digits then...
   a. Subtract a billion to reduce it to nine or fewer digits.
   b. Add 1 to the component to its left.
5. If the leftmost component of result has more than nine digits then...
   Return null, since the answer cannot be expressed using six components.
   Otherwise...
   Return the `result`.

The rest of the VeryLong methods are left as exercises, except that the `multiply` method is sufficiently complicated that it is a major programming project. Note that no public method in the Numeric class or any of its subclasses allows an outside class to change the value of a Numeric object once it is created: Objects from Numeric and its subclasses are **immutable**. This is similar to the String class, in that no method in the String class allows you to change a String object once you create it.

You could have a subclass of VeryLong that allows negative numbers, as follows:

```
public class SignedNumber extends VeryLong
{  private boolean isNegative;
   public SignedNumber (long left, long mid, long right)
   {  super (Math.abs (left), mid, right);
      isNegative = left < 0;
   }
//... lots more is required here in the same vein
```

Exercise 11.26 Modify the VeryLong constructor to create the equivalent of ZERO when any one of the parameters is negative or has more than eighteen digits.
Exercise 11.27 (harder) Write the `doubleValue` method for the VeryLong class.
Exercise 11.28 (harder) Write the `equals` method for the VeryLong class.
Exercise 11.29 (harder) Write a simplified `valueOf` method for the VeryLong class, for which you have a precondition that the parameter is a string of 1 to 54 digits.
Exercise 11.30* Write the full `valueOf` method for the VeryLong class. Allow the input to contain commas among the digits. Use the preceding exercise to get started.
Exercise 11.31* Write a Real subclass of Numeric for ordinary numbers with one double instance variable. This lets the clients mix in ordinary numbers with the special ones.
Exercise 11.32* Write the `subtract` method for the VeryLong class.
Exercise 11.33* The Complex instance variables are final but the Fraction instance variables are not, even though both are immutable classes. Explain why.
Exercise 11.34** Write the `compareTo` method for the VeryLong class.
Exercise 11.35** The VeryLong `toString` method produces leading zeros when the leftmost one or more components of `itsItem` are zero. Revise it to fix this problem.

11.8 Implementing The NumericArray Class With Null-Terminated Arrays

These mathematicians often deal with numbers in big bunches. They may read in a bunch of numbers from a file, then calculate the average of the whole bunch, find the smallest and the largest, insert a value in order, etc. For this, you decide to store a lot of Numeric objects in an array of Numeric values. Call it the **NumericArray** class.

Once you define an array, you cannot change its size. So you need to make it big enough for the largest number of values you expect. But then the array is generally only partially filled. So you have to have some way of noting the end of the array. One way is to keep track of the size. Another way is to put the null value in all the components after the end of the actual Numeric values in the array, as shown in Figure 11.5.

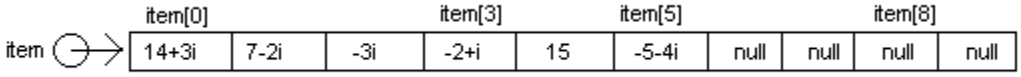

Figure 11.5 picture of a null-terminated array with six Complex values

For instance, if the array has size 1000 and currently contains only 73 values, then those 73 values will be stored in components 0 through 72 and the null value will be stored in each of components 73...1000. We do not allow a full array. A precondition for these **null-terminated arrays** is that they contain at least one instance of null.

Precondition for NumericArrays For the array parameter `item`, there is some integer `n` such that `0 <= n < item.length` and `item[k]` is not null when `k < n` and `item[k]` is null otherwise. The non-null values are the values on the conceptual list in order, with the first at index 0.

The find method

To illustrate how to work with such a null-terminated array, consider the problem of searching through the array to find whether a particular non-null value is there. You process each value of an int variable `k` from 0 on up, until `item[k]` is null. At each array value before that point, you compare it with the target value. If they are equal, you return `true`. If you reach the point where `item[k]==null`, you know the target value is not in the array, so you return `false`. This logic is in the upper part of Listing 11.10.

Listing 11.10 The NumericArray class

```
public class NumericArray
{
   /** Tell whether target is a non-null value in the array. */

   public static boolean find (Object[ ] item, Object target)
   {  for (int k = 0;  item[k] != null;  k++)
      {  if (item[k].equals (target))
            return true;
      }
      return false;
   }  //=====================

   /** Return the sum of the values in the array; return null
    *  if the sum is not computable.  Precondition:
    *  All non-null values are of the same Numeric type. */

   public static Numeric sum (Numeric[ ] item)
   {  if (item[0] == null)
         return null;
      Numeric total = item[0];
      for (int k = 1;  item[k] != null && total != null;  k++)
         total = total.add (item[k]);
      return total;
   }  //=====================

   /** Print each non-null value on the screen. */

   public static void display (Object[ ] item)
   {  for (int k = 0;  item[k] != null;  k++)
         System.out.println (item[k].toString());
   }  //=====================

   /** Return the array of values read from the source, to a
    *  maximum of limit values.  Precondition: limit >= 0 and
    *  data is the same Numeric subtype as all input values. */

   public static Numeric[ ] getInput (Numeric data, int limit,
                                      java.io.BufferedReader source)
                              throws java.io.IOException
   {  Numeric[ ] item = new Numeric [limit + 1];  // all nulls
      for (int k = 0;  k < limit;  k++)
      {  item[k] = data.valueOf (source.readLine());
         if (item[k] == null)
            return item;
      }
      return item;
   }  //=====================
}
```

What if the calling method passed in a target value of null? The condition `item[k].equals(target)` returns `false` when you test whether a non-null value equals a null value. So each time through the loop in the `find` method, the if-condition is false. Eventually `find` returns `false`.

This `find` method has been written with Object in place of Numeric. The reason is that only the `equals` method is used in the logic, and every Object has an `equals` method. Making the parameter the Object type allows the method to be used in more situations.

The `find` method goes in the NumericArray class because that is where you need it. But as a general principle, you should make your methods apply more generally when nothing is lost by it. You are allowed to assign a `Numeric[]` array value to an `Object[]` array variable, though not vice versa. The runtime system chooses the right `equals` method for each Object (another use of polymorphism).

The sum and display methods

Listing 11.10 contains some other useful class methods for the NumericArray class. The `sum` method finds the sum of all the non-null values in the array. It throws a ClassCastException if they are not all of the same Numeric type. It uses a logic you have seen before: Initialize the `total` to the first value in the array. Then add the second value to it, then the third value to that, etc. The `total.add(item[k])` expression is polymorphic: The runtime system chooses the `add` method in the subclass of Numeric that `total` belongs to. That requires that each item be of the same Numeric subtype.

Further thought indicates you need to allow for the possibility that the result of adding two values may be null, which occurs when the sum is not computable. The accompanying design block records the logic in detail. Remember, a primary purpose of the design in Structured English is to verify that all possibilities have been handled properly before you attempt to translate the logic to Java.

DESIGN for the sum method in the NumericArray class

1. If the given list contains no values at all, i.e., the first component is null, then...
   Return null.
2. Create a Numeric object to store the result of the addition. Call it `total`.
   Initialize it to be the first value in the list (at index 0).
3. For each additional value in the list do the following...
   Add the next value in the list to `total` (but stop if the result becomes null).
4. Return the `total` as the answer.

The `display` method prints every value in the array. Since the only method called in the body of the `display` method is the `toString` method, which every Object has, the `display` method is written more generally to handle any array of Objects whatsoever, even from different classes. The runtime system chooses the right `toString` method for each Object (another use of polymorphism).

The getInput method

The `getInput` method returns a new array containing all the values read in. It requires a Numeric data value to do its job, even though it totally ignores the value supplied. It would normally be the `ZERO` of a subclass of Numeric. The only purpose of that data value is to act as the executor of the `valueOf` method, so that the runtime system knows which of the subclass methods to use at that point (another use of polymorphism).

For the `getInput` method, every non-null value that is read in must be stored in the array. What if the `limit` is ten and the source contains ten or more Numeric values? That tenth value has to be put into the array as well. Since the array has to have a null value after all the useable Numeric values, it has to have at least eleven components. That is why the array size is made larger than the given limit.

A program using a NumericArray

Every method in the NumericArray class is a class method. This is because the object you are working with, an array of Numeric values, is a parameter rather than an executor. So this is a utilities class analogous to the Math class.

Listing 11.11 illustrates how simple programs can be that involve arrays of Fractions, with just the methods in Listing 11.10. The program reads in up to 100 Fraction values from a file named "numeric.dat". Then it prints all of the values as fractions reduced to lowest terms. Finally, it prints out the sum of the values as a fraction reduced to lowest terms. Figure 11.6 is the UML diagram for this AddFractions class.

Listing 11.11 An application program using Fractions and NumericArrays

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;

public class AddFromFile
{
   /** Read in up to 100 Fraction values from a file,
    *  then display them all on the screen with their sum. */

   public static void main (String[ ] args) throws IOException
   {  Numeric [ ] values = NumericArray.getInput
                  (Fraction.ZERO, 100, new BufferedReader
                        (new FileReader ("numeric.dat")));
      NumericArray.display (values);
      System.out.println ("Their sum is "
                        + NumericArray.sum (values));
   }  //====================
}
```

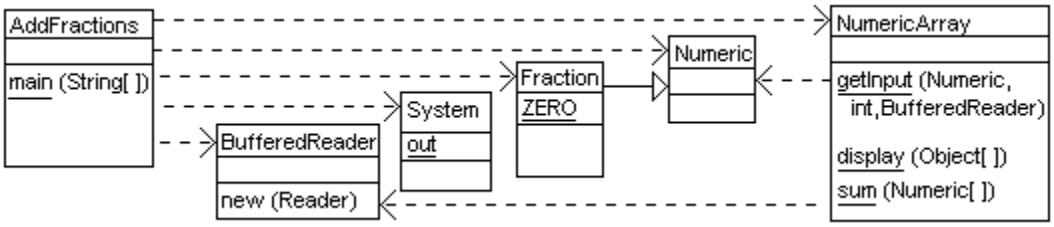

Figure 11.6 UML class diagram for the AddFractions class

An alternative for storing a large number of Numeric values is to use a NumericList object having two instance variables, a **partially-filled array** of Numerics `itsItem` and an int `itsSize` that tells how many components at the front of the array have useable Numeric values. Then the `display` method of the earlier Listing 11.10 would be expressed as follows:

```
private Numeric[] itsItem;
private int itsSize;
public void display()   // for NumericList
{  for (int k = 0;  k < itsSize;  k++)
      System.out.println (itsItem[k]);
}  //======================
```

Exercise 11.36 Write a NumericArray method to find the sum of the `doubleValue` values of the objects in the array, even when the objects are of various subclasses of Numeric.

Exercise 11.37 Write a NumericArray method `public static int size (Numeric[ ] item)` to find the number of non-null values in the null-terminated array.

Exercise 11.38 Write a NumericArray method `public static boolean allSmall (Numeric[ ] item, Numeric par)` to tell whether every value in the array is smaller than `par`. Precondition: All values are comparable to the non-null `par`.

Exercise 11.39 (harder) Write a NumericArray method `public static int indexSmallest (Numeric[ ] item)` to find the index where the smallest value is stored. Precondition: The array has at least one value, and all values are comparable to each other.

Exercise 11.40 (harder) Write a NumericArray method `public static void delete (Numeric[ ] item, int n)` to delete the value at index `n` and keep the rest of the values in the same order they were originally in. No effect if there is no value at index n.

Exercise 11.41 (harder) Write a NumericArray method `public static Numeric[] getComplexes (Numeric[ ] item)` to return a new null-terminated array containing just the values in the given array that are Complex objects.

Exercise 11.42* Rewrite the NumericArray `getInput` method efficiently to have the condition of the for-statement be `k < limit && data != null`.

Exercise 11.43* Write a NumericArray method `public static double fractional (Numeric[ ] item)` to find the fraction of values that are Fraction objects (e.g., return 0.25 if a quarter are Fractions). Return zero if there are no values in the array.

Exercise 11.44* Write a NumericArray method `public static Numeric maximum (Numeric[ ] item)` to find the largest value in the array. Crash-guard against all NullPointerExceptions. Return null if the array contains zero values. Precondition: Any two values in the array are comparable to each other.

Exercise 11.45* Write a NumericArray method with the heading `public static void insert (Numeric[ ] item, Numeric given)` to insert the given value in the array and keep it in ascending order. Precondition: The values in the array when the method is called are already in ascending order, the array has enough room, and any two values are comparable to each other. Restriction: Go to the end of the array and work backwards.

Exercise 11.46* Revise the `getInput` method in Listing 11.10 to catch any Exception and return the array as it stands at that time.

Exercise 11.47* Revise the entire Listing 11.10 so that it has one instance variable of type `Numeric[ ]`. Remove the `item` parameter of the first three methods, since it will be in the executor. Only the `getInput` method should be a class method; it should return a NumericArray object.

Exercise 11.48** Write a NumericArray method `public static Numeric[] reverse (Numeric[ ] item)` to return a null-terminated array containing the same values but in the opposite order.

Exercise 11.49** Write a NumericArray method `public static boolean isNulled (Object[] item)` that tells whether the parameter is in fact a null-terminated array. This method has no preconditions at all. Hint: The first statement should test `item[item.length-1] != null`.

*11.9 Too Many Problems, Not Enough Solutions (*Enrichment)*

Problem: Tell whether a given positive integer is odd.
Solution: The following boolean class method answers the question:

```
public static boolean isOdd (int num)
{  return num % 2 == 1;
}  //=====================
```

Problem: Tell whether a given positive integer is a prime.
Solution: The following boolean class method answers the question:

```
public static boolean isPrime (int num)
{  for (int k = 2;  k <= num / 2;  k++)
   {  if (num % k == 0)
         return false;
   }
   return num > 1;
}  //=====================
```

In general, a **decision problem** is of the form "Tell whether a given positive integer has a certain property." A **solution** to a decision problem is a boolean class method with one integer parameter, that returns `true` for parameters that have the property and `false` for those that do not. The two methods above are therefore solutions to the stated decision problems. Another example is the following:

Problem: Tell whether a given positive integer is the sum of two odd primes. For instance, `6 = 3+3, 8 = 5+3, 10 = 5+5, 12 = 7+5`, so those happen to all be the sum of two odd primes.
Solution: The following boolean method answers the question:

```
public static boolean isGoldbach (int num)
{  for (int k = 3;  k <= num / 2;  k += 2)
   {  if (isPrime (k) && isPrime (num - k))
         return true;
   }
   return false;
}  //=====================
```

Counting without limit

It would be nice if every decision problem had a solution. Of course, that cannot be true if there are more decision problems than there are solutions to decision problems. Let us see how many there are of each.

Every Java function is written as a sequence of characters. Each such character can be stored in one byte of storage, which is 8 bits. The (extremely long) sequence of bits you get this way can be interpreted as a single number base 2. Call that the **Java integer** of the function. So every solution to a decision problem has a Java integer, and different solutions have different Java integers. Therefore, we may conclude:

Deduction 1: The number of solutions to decision problems is no more than the number of positive integers.

For any given decision problem testing for a certain property, you can visualize a corresponding base 2 number as follows:

1. Write a decimal point (correction, make that a binary point).
2. Write 1 for the 1st digit after the binary point if 1 has the property; write 0 if it does not.
3. Write 1 for the 2nd digit after the binary point if 2 has the property; write 0 if it does not.
4. Write 1 for the 3rd digit after the binary point if 3 has the property; write 0 if it does not.
5. Write 1 for the 4th digit after the binary point if 4 has the property; write 0 if it does not, etc.

In general, the n^{th} digit of the number is 1 if the property is true for n and is 0 if the property is false for n. Call this the **property number** of the decision problem. For instance, the decision problem solved by `isOdd` has 0.101010101... as its property number (which is 2/3; check this by adding up 1/2 + 1/8 + 1/32 + 1/128 ad infinitum).

So every number between 0 and 1, written in base 2, is the property number of some decision problem, and different numbers between 0 and 1 have different decision problems (since if the numbers differ in even one place, say the 17^{th}, then for the two corresponding decision problems, 17 has one property but not the other). Therefore, we may conclude:

Deduction 2: The number of decision problems is no less than the number of real numbers between 0 and 1.

Put those two deductions together with the mathematical fact that the number of real numbers between 0 and 1 is far greater than the number of positive integers to get:

Deduction 3: Almost all decision problems have no solution in Java.

This is a result from the **Theory of Computability**, which you will learn more about in advanced courses in computer science. You will learn the reasoning behind the fact that, for any attempt to match up the positive integers one-to-one with the real numbers between 0 and 1, you will leave over 99% of the real numbers without a match. So over 99% of all decision problems have no Java method that solves them. In a sense, over 99% of all sets of yes-no questions about positive integers have no answers.

Exercise 11.50* (Essay Question) Which has more, the set of odd numbers or the set of primes? Why?

*11.10 Threads: Producers And Consumers (*Enrichment)*

Situation #1: A portion of a program displays a continually changing scene on the monitor, but as soon as the user clicks a button or moves the mouse or presses a key, the scene disappears and the program reacts to the user's action. An example of this is a screensaver.

Situation #2: A portion of a program is waiting for part of a web page to download. It cannot continue with what it is doing until the download completes. So it checks every tenth of a second or so and, when it sees that the download is still going on, it lets another portion of the program do something useful for the next tenth of a second.

Situation #3: A process performs a long series of calculations to eventually produce a result which it deposits in a variable. A second process is waiting for that result. When it appears, the second process uses it as the starting point for its own long series of calculations. Meanwhile, the first process is working on producing a second result. Each time the second process (the **consumer**) needs a new result, it waits for it to appear and then uses it. Each time the first process (the **producer**) computes a new result, it checks that the previous result has been taken and, when it has, deposits the new result.

In all of these situations, it would be extremely useful to have two or more independent computer chips, each executing its own method and interacting with the other chips as needed. That is much simpler than trying to keep track of where you are in each of several processes and switching back and forth between them.

Concurrent execution

Java provides an equivalent of this called Threads. Each of several threads of execution run their own methods "simultaneously". What really happens (unless your program actually has more than one computer chip available for its use) is that the one chip switches back and forth between several different **threads of execution**, doing each one for such a short period of time that it may appear to the human observer that all are executing simultaneously. We say they run **concurrently** rather than simultaneously.

Each thread is given a small period of time, called a **quantum**, during which it executes part of what it is supposed to do. If the quantum expires before the thread finishes what it is doing, the thread's action is suspended and another thread is given a quantum. When the thread receives another quantum, it takes up what it was doing at the point where it left off. This continues in round-robin fashion among all operating threads.

The graphical user interface is one thread of execution. If that thread is executing a screensaver kind of method, repeatedly making changes in the screen display, it is not available to listen for a button click or other action by the user. So the user would be clicking with no effect. The screensaver action might continue forever.

If, however, the main thread of execution performs statements that create a new Thread object `process` to execute the screensaver method, the main thread can then go back to listening for some event to happen. When it detects a button click, it can then react by sending a message to the `process` object to stop what it is doing.

The Runnable interface

The **Runnable** interface specifies a method with the heading `public void run()`. When a class `implements Runnable`, its `run` method can control a single thread of execution. If you have a Thread variable named `process` that is to execute the `run` method in a ScreenSaver class, you can create a new thread of execution and have it start execution with this coding:

```
Runnable action = new ScreenSaver();
Thread process = new Thread (action);
process.start();
```

The Thread constructor creates a new thread of execution that will use the `run` method of the parameter. When you call the `start` method for a Thread object, the object initializes the concurrent thread of execution and then calls the `run` method specified. If you later wish to stop execution of that thread, execute the following statement:

```
process.interrupt();
```

This statement notifies the process thread that some other thread wants it to terminate, but it does not force the termination. That is up to the `process` thread. It can find out whether it has received an interrupt request by testing the following condition:

```
Thread.interrupted()
```

Listing 11.12 contains an example of the use of these language features. It omits the messy details of how the screen display changes during execution of the screensaver, since that is not relevant to the overall concurrency logic. The `actionPerformed` method would be in some class that can refer to the `startButton` and the `process`. The Thread class and Runnable interface are in the `java.lang` package, so they can be used in a class without having import directives.

Listing 11.12 The ScreenSaver class and the method that uses it

```
// reaction to a click of either startButton or stopButton

   Thread process;
   Button startButton, stopButton;


   public void actionPerformed (ActionEvent e)
   {  if (e.getSource() == startButton)
      {  process = new Thread (new ScreenSaver());
         process.start();
      }
      else
         process.interrupt();
   }  //=====================



public class ScreenSaver implements Runnable
{
   public void run()
   {  while ( ! Thread.interrupted())
         changeTheScenerySome();
   }  //=====================


   private void changeTheScenerySome()
   {  // make a small change in the display on the monitor
   }  //=====================
}
```

The producer-consumer situation

If you have two Runnable objects called perhaps `producer` and `consumer`, you may create a Thread object for each and start both their `run` methods executing concurrently as follows:

```
Thread pro = new Thread (producer);
Thread con = new Thread (consumer);
pro.start();
con.start();
```

The `start` method for Thread objects sets up the concurrent process and then executes the `run` method of the given Runnable object (`producer` or `consumer` in this case). Think of the objects being produced as pies. The producer is continually producing pies and depositing them in the place where the consumer can find them, pausing only if the consumer gets behind in eating them. And the consumer is continuing consuming pies, pausing only if the producer gets behind in baking them.

The Producer and Consumer classes could be designed as shown in Listing 11.13. The messy details of the actual eating and baking are left unstated. The coding uses the conventional `for(;;)` notation to create an infinite loop.

Listing 11.13 The Producer and Consumer classes

```
public class Producer implements Runnable
{
   public void run()
   {  for (;;)
      {  Object dessert = produce();
         while (Resource.hasUneatenPie())
         {  }                          // wait until unoccupied
         Resource.setPie (dessert);    // mark it occupied
      }
   }  //=====================


   public Object produce()
   {  // extensive action required to produce a pie
   }  //=====================
}
//#################################################################


public class Consumer implements Runnable
{
   public void run()
   {  for (;;)
      {  while ( ! Resource.hasUneatenPie())
         {  }                              // wait until occupied
         consume (Resource.getPie());      // mark it unoccupied
      }
   }  //=====================


   public void consume (Object dessert)
   {  // extensive action required to consume a pie
   }  //=====================
}
```

These two classes presume the existence of the Resource class which serves as a pie depository. The Resource class has three class methods:

- `setPie(Object)` deposits the given pie. Only one can be there at a time.
- `getPie()` returns the pie currently on deposit; it returns null if there is no pie.
- `hasUneatenPie()` tells whether a pie is currently on deposit.

The Resource class could be implemented with the following two private class variables. Then the `setPie` method could set `ready` to `true` and assign its parameter value to `pie`, and the `hasUneatenPie` method could simply return the value of `ready`:

```
private static boolean ready = false;
private static Object pie = null;
```

Sleeping threads

The producer and consumer use a busywait to wait for something to happen, i.e., they execute statements that do nothing useful while waiting. This ties up the processor unnecessarily. A substantially better method is to have a thread of execution execute the following statement to free up the processor for n milliseconds:

```
Thread.sleep (n);
```

This method call can throw an InterruptedException that must be acknowledged (i.e., it is not a RuntimeException). So the method call should normally be used only within a try/catch statement. You could replace the no-action pair of braces in the body of the Consumer's `while` statement by the following statement, if you define the ThreadOp utility class shown in Listing 11.14:

```
if (ThreadOp.pause (50))  // true only when interrupted
   return;
```

Listing 11.14 The ThreadOp class

```
public class ThreadOp
{
   public static boolean pause (int millis)
   {  try
      {  Thread.sleep (millis);
         return Thread.interrupted();
      }catch (InterruptedException e)
      {  return true;
      }
   }  //====================
}
```

That statement relinquishes the processor for 50 milliseconds (0.050 seconds), then checks to see whether an interrupt signal was sent. If so, the `return` statement stops the execution of the `run` method. Otherwise the method checks to see if a new pie is available (`Resource.hasUneatenPie()`). If so, it gets one more pie, eats it, and then returns to its waiting state. Note that it does not allow itself to be interrupted in the middle of eating a pie; that would waste the effort spent in partially processing its data, to say nothing of wasting a chunk of a perfectly good pie. The interrupt request does not force termination, it only suggests it.

The producer requires a more complex response to a request to stop. It would be a shame to waste the pie it is waiting to place in the depository. On the other hand, perhaps the consumer has also been interrupted and will therefore never get around to retrieving the pie currently on deposit. So let's say the producer waits for 200 more milliseconds to see if that pie is taken and, if so, deposits its new pie before it terminates. Thus the no-action pair of braces in the body of the Producer's `while` statement could reasonably be replaced by the following statement:

```
if (ThreadOp.pause (50))  // true only when interrupted
{  ThreadOp.pause (200);
   if ( ! Resource.hasUneatenPie())
      Resource.setPie (dessert);
   return;
}
```

Note that the producer does not pay attention to whether another interrupt is sent during that 200 millisecond pause, since it plans to terminate in any case. Note also that, if either object wished to ignore any interrupt request, but still use the `sleep` method, it could simply replace the no-action pair of braces in the body of its `while` statement by the following. This discards the returned boolean value:

```
   ThreadOp.pause (50);
```

Synchronization

If the Resource class method `getPie` is coded as follows, the logic could fail to produce the desired result:

```
public static Object getPie()
{  ready = false;
   return pie;
}  //=====================
```

The problem is that a consumer may execute `getPie` when a chocolate meringue pie is on deposit and a producer is waiting to deposit a coconut creme pie. The consumer executes `ready = false` in `getPie`. Then before it can execute `return pie`, the producer may test `Resource.hasUneatenPie()`, which now returns `false`, so the producer executes `setPie`, thereby depositing the coconut creme pie before the chocolate meringue pie has been taken. That means that the consumer gets coconut creme, which is far inferior to chocolate meringue. The chocolate meringue is totally wasted.

A solution is to reverse the order of operations in the `getPie` method, so the consumer first takes the `pie` out of the depository and then sets the boolean `ready` to `false`:

```
public static Object getPie()
{  Object valueToReturn = pie;
   ready = false;
   return valueToReturn;
}  //=====================
```

This solution works if there is only one consumer. However, if there were several consumers, then as soon as a pie became available, two of them could try executing `getPie` at the same time, which could produce a food fight.

Java provides a solution for this messy problem. A class method that has the word `synchronized` in its heading can only be executed by one thread at a time. That is, when a Consumer object begins execution of the method, it is given a **lock** on the method. When another Consumer object tries to execute that same method, it is locked out; it must wait until the first Consumer object completes the method. Now, with some revisions of the coding given in this section, the program can manage several producers and several consumers properly.

Technical notes

The recommended order of the words in a method heading is shown by the following legal method heading. All those that come before `void` are optional. `native` means that the method is written in another language than Java and thus its body is to be found elsewhere than at this point:

```
public static final synchronized native void test()
```

Java has three rarely-used declaration modifiers: A field variable can be declared as `volatile`, which forces frequent synchronization, or as `transient`, which affects whether it is saved when an object is written to permanent storage, or as `strictfp`, which puts strictures on floating-point computations.

Exercise 11.51* Write the Resource method `public static void setPie (Object ob)` to avoid synchronization problems when there is only one producer.

*11.11 More On Math And Character (*Sun Library)*

This section briefly describes all Math methods other than those discussed in Chapter Six (`sqrt(x)`, `abs(x)`, `log(x)`, `exp(x)`, `min(x,y)`, `max(x,y)`, `pow(x,y)`, and `random()`) plus some additional methods from the Character wrapper class.

Math rounding methods

Math has five methods that round off a value in some way. `x` denotes a double value:

- `ceil(x)` returns a double that is the next higher whole-number value, except it returns `x` itself if `x` is a whole number. So `ceil(4.2)` is 5.0, `ceil(-4.2)` is -4.0, and `ceil(32.0)` is 32.0.
- `floor(x)` returns a double that is the next lower whole-number value, except it returns `x` itself if `x` is a whole number. So `floor(4.2)` is 4.0, `floor(-4.2)` is -5.0, and `floor(32.0)` is 32.0.
- `rint(x)` returns a double that is the closest whole-number value, except it returns an even number in case of a tie. So `rint(4.2)` is 4.0, `rint(4.7)` is 5.0, `rint(4.5)` is 4.0, and `rint(5.5)` is 6.0.
- `round(x)` returns the long value equivalent of `rint(x)`. It returns Long.MAX_VALUE or Long.MIN_VALUE if it would be otherwise out of range.
- `round(someFloat)` for a float parameter returns the int equivalent of `rint(someFloat)`. It returns Integer.MAX_VALUE or Integer.MIN_VALUE if it would otherwise be out of range.

Math trigonometric methods

The nine trigonometric methods all take double arguments and produce a double result. The angles are measured in radians, so `x = 3.14159` is about 180 degrees. For instance, `cos(Math.PI / 6.0)` is the cosine of 30 degrees, which is 0.5.

- `cos(x)` returns the cosine of `x`.
- `sin(x)` returns the sine of `x`.
- `tan(x)` returns the tangent of `x`.
- `acos(x)` returns the angle whose cosine is `x`, ranging from 0.0 through PI.
- `asin(x)` returns the angle whose sine is `x`, ranging from -PI/2 through PI/2.
- `atan(x)` returns the angle whose tangent is `x`, ranging from -PI/2 through PI/2.
- `atan2(x,y)` returns the angle whose tangent is `y/x`, ranging from -PI/2 through PI/2.
- `toDegrees(x)` returns the angle in degrees equivalent to `x` radians, which is equal to `x * 180.0 / Math.PI`.
- `toRadians(x)` returns the angle in radians equivalent to `x` degrees, for instance, `cos(toRadians(30))` is the cosine of 30 degrees.

The one remaining Math method is `IEEEremainder(x, y)`, which returns the remainder of `x` divided by `y` as specified by the IEEE 754 standard.

Character methods

The following class methods, which have char values for parameters and return a boolean value, can be quite handy. Unicode values outside the range 0 to 255 are not considered here:

- `Character.isDigit(someChar)` tells whether it is '0' through '9'.
- `Character.isLowerCase(someChar)` tells whether it is 'a' through 'z'.
- `Character.isUpperCase(someChar)` tells whether it is 'A' through 'Z'.
- `Character.isWhiteSpace(someChar)` tells whether `c` has Unicode 9-13 or 28-32.
- `Character.isLetter(someChar)` tells whether it is a letter, either lowercase or uppercase.
- `Character.isLetterOrDigit(someChar)` tells whether it is either a letter or a digit.

The following methods have a char parameter and return a char value:

- `Character.toLowerCase(someChar)` returns the lowercase equivalent of a capital letter, and returns the unchanged parameter otherwise.
- `Character.toUpperCase(someChar)` returns the capital letter equivalent of a lowercase letter, and returns the unchanged parameter otherwise.

11.12 Review Of Chapter Eleven

About the Java language:

- A method may be declared as **final**, which means it cannot be overridden. The phrase **final class** means that the class cannot have any subclasses.
- Declaring a class as **abstract** means (a) you may replace the body of any non-final instance method by a semicolon if you declare that method as abstract; (b) every non-abstract class that extends it must implement all of the abstract methods. An abstract class may have instance and class variables, constructors, and instance and class methods. Class methods in an abstract class cannot be abstract.
- The heading `public interface X` means X cannot contain anything but non-final instance method headings (with a semicolon in place of the body) and final class variables. A non-abstract object class with the heading `class Y implements X` must define all methods in that **interface**. A class may implement many interfaces (use `implement X,U,Z`) but may subclass only one class.
- The compiler binds a method call to a particular method definition when it can, namely, for a class method or for a final instance method. This is **early binding**, which reduces execution time compare with **late binding** (done at runtime).
- The four primitive integer types are long (8 bytes), int (4 bytes), **short** (2 bytes), and **byte** (1 byte). One byte of storage is 8 bits, so there are 256 different possible values for one byte. The two primitive decimal number types are double (8 bytes, 15 decimal digits) and **float** (4 bytes, 7 decimal digits). The remaining two **primitive types** are boolean and char.
- The operator **instanceof** can be used between an object variable and a class name; it yields a boolean value. `x instanceof Y` is true when `x` is not `null` and is an object of class Y or of a subclass of Y or (if Y is an interface) of a class that implements Y. The `!` operator takes precedence over the `instanceof` operator, so a phrase of the form `! X instanceof Y` is never correct; use parentheses.

About the six Number subclasses:

- The six Number **wrapper classes** Double, Float, Integer, Long, Short, and Byte are in `java.lang` and are Comparable. The twelve methods given below for the Long class apply to the other five Number wrapper classes with the obvious changes:
- `Long.parseLong(someString)` returns a long value or throws a NumberFormatException.
- `new Long(someString)` creates a Long object parsed from the String. It throws a NumberFormatException if the string is badly-formed.
- `new Long(primitive)` creates a Long object from a given long value.
- `someLongObject.longValue()` returns the long equivalent.
- `someLongObject.doubleValue()` returns the double equivalent
- `someLongObject.intValue()` returns the int equivalent.
- `someLongObject.floatValue()` returns the float equivalent.
- `someLongObject.shortValue()` returns the short equivalent.
- `someLongObject.byteValue()` returns the byte equivalent.
- `someLongObject.toString()` overrides the Object method; it returns the String equivalent.
- `someLongObject.equals(someObject)` tells whether the corresponding primitive values are equal.
- `someLongObject.compareTo(someObject)` returns an int with the usual meaning: positive if the executor is larger, negative if it is smaller.
- `Integer.toString(n,16)` gives the **hexadecimal** (base 16) form of the int value n. Use 2 or 8 in place of 16 for **binary** (base 2) or **octal** (base 8).
- `Integer.parseInt(someString, 16)` produces the int equivalent of the hexadecimal digits in `someString`. Use 2 or 8 for binary or octal, respectively.

About the java.lang.Character class:

- `new Character(someChar)` gives the object equivalent of the primitive value of the parameter.
- `someCharacter.charValue()` converts back from object to primitive value.
- `someCharacter.toString()` returns the String equivalent of the char value.
- `someCharacter.equals(someObject)` overrides the Object `equals`; it tells whether `someObject` represents the same char value.
- `someCharacter.compareTo(someObject)` returns an int with the usual meaning: positive if the executor is larger, negative if it is smaller.

About the java.lang.Boolean class:

- `new Boolean(primitive)` gives the object equivalent of the primitive boolean value given as a parameter.
- `new Boolean(someString)` yields `Boolean.TRUE` if the parameter is "true", ignoring case, otherwise it yields `Boolean.FALSE`.
- `someBooleanObject.booleanValue()` returns the primitive form of the Boolean object.
- `someBooleanObject.toString()` returns "true" or "false" as appropriate.
- `someBooleanObject.equals(someObject)` overrides the Object `equals`; it tells whether `someObject` represents the same true or false value.

Answers to Selected Exercises

11.2
```
public static Numeric min3 (Numeric one, Numeric two, Numeric three)   // in Numeric
{   if (one.compareTo (two) < 0)
        return one.compareTo (three) < 0  ?  one  :  three;
    else
        return two.compareTo (three) < 0  ?  two  :  three;
}
```
An alternative coding for the body of this method in 2 lines is:
```
Numeric smaller = (one.compareTo (two) < 0)  ?  one  :  two;
return smaller.compareTo (three) < 0  ?  smaller  :  three;
```
11.3
```
public Numeric smallest (BufferedReader source)
{   try
    {   Numeric valueToReturn = valueOf (source.readLine());
        if (valueToReturn == null)
            return null;
        Numeric data = valueOf (source.readLine());
        while (data != null)
        {   if (data.compareTo (valueToReturn) < 0)
                valueToReturn = data;
            data = valueOf (source.readLine());
        }
        return valueToReturn;
    }catch (Exception e)
    {   return null;
    }
}
```
11.5
```
public boolean equals (Object ob)
{   if (! (ob instanceof Person))
        return false;
    Person given = (Person) ob;
    return this.itsBirthYear == given.itsBirthYear
            && this.itsLastName.equals (given.itsLastName);
}
```

11.6
```
public class Tuna extends Swimmer
{   public void swim()
    {   System.out.println ("tuna is swimming");
    }
    public void eat (Object ob)
    {   if (ob instanceof Swimmer)
            System.out.println ("tuna eats " + ob.toString());
        else
            System.out.println ("tuna is still hungry");
    }
    public String toString()
    {   return "tuna";
    }
}
```
11.10 5 = 4 + 1, 11 = 8 + 2 + 1, 260 = 256 + 4. So:
In binary; 101, 1011, 100000100.
In octal: 5, 13, 404 (the last since 256 = 4 * 8 * 8).
In hexadecimal: 5, B, 104 (the last since 256 = 16 * 16).

11.11 F is 15. 5D is 5*16 + 13 = 93.
ACE is 10 * 256 + 12 * 16 + 14 = 2560 + 192 + 14 = 2766.

11.12 ZERO is defined using the constructor. That definition will be applied by the constructor when the program begins. How can the constructor return a value that does not yet exist? Besides, a constructor does not have a return type.

11.13 The specification for the equals method is that it not throw an Exception. But the Numeric add, subtract, and multiply methods are to throw an Exception if the parameter is not of the right type.

11.14 For subtract, simply replace the plus sign in the add method by the minus sign.

11.15
```
public Numeric valueOf (String par)
{   if (par == null)
        return null;
    int k = par.indexOf ('/');
    return (k == -1)  ?  null  :  new Fraction (Integer.parseInt (par.substring (0, k)),
                                                Integer.parseInt (par.substring (k + 1)));
}
```
11.16
```
public int compareTo (Object ob)
//  You can return the numerator of the result of subtracting this minus ob:
{   return this.itsUpper * ((Fraction) ob).itsLower
        - this.itsLower * ((Fraction) ob).itsUpper;
}
```
11.17
```
public Numeric divide (Numeric par)
{   Fraction that = (Fraction) par;
    return (that.itsUpper == 0)  ?  null
        :  new Fraction (this.itsUpper * that.itsLower, this.itsLower * that.itsUpper);
}
```
11.20
```
public boolean equals (Object ob)
{   return ob instanceof Complex && ((Complex) ob).itsReal == this.itsReal
                                 && ((Complex) ob).itsImag == this.itsImag;
}
```
11.21
```
public Numeric multiply (Numeric par)
{   Complex that = (Complex) par;
    return new Complex (this.itsReal * that.itsReal - this.itsImag * that.itsImag,
                        this.itsReal * that.itsImag + this.itsImag * that.itsReal);
}
```
11.22 Change the (Complex) cast to a (Numeric) cast. The compiler accepts it because Numeric declares doubleValue. The runtime system will then choose the right subclass's doubleValue method.

11.23 Insert the following before the return statement in the toString method:
```
if (itsImag * itsReal == 0)
    return (itsImag == 0)  ?  itsReal + “” : itsImag + “i”;
else
```
11.26 You could insert the following as the first statements:
```
long max = LONGBILL * LONGBILL - 1;
if (left < 0 || left > max || mid < 0 || mid > max || right < 0 || right > max)
{   left = 0L;
    mid = 0L;
    right = 0L;
}
```
11.27
```
public double doubleValue()
{   double total = itsItem[0];
    for (int k = 1;  k < MAX;  k++)
        total = total * BILLION + itsItem[k];
    return total;
}
```

```
11.28   public boolean equals (Object ob)
        {   if (! (ob instanceof VeryLong))
                return false;
            VeryLong that = (VeryLong) ob;
            for (int k = 0;  k < MAX;  k++)
            {   if (this.itsItem[k] != that.itsItem[k])
                    return false;
            }
            return true;
        }
11.29   public Numeric valueOf (String par)      // for VeryLong, in a simplified form for a first approximation
        {   VeryLong valueToReturn = new VeryLong();   // initially all zeros
            int k = MAX - 1;
            for ( ;  par.length() > 9;  k--)
            {   int firstDigit = par.length() - 9;
                valueToReturn.itsItem[k] = Integer.parseInt (par.substring (firstDigit));
                par = par.substring (0, firstDigit);
            }
            itsItem[k] = Integer.parseInt (par);
            return valueToReturn;
        }
11.36   public static double sumValues (Numeric[ ] item)
        {   double valueToReturn = 0;
            for (int k = 0;  item[k] != null;  k++)
                valueToReturn += item[k].doubleValue();
            return valueToReturn;
        }
11.37   public static int size (Object [ ] item)  // better if Object, not Numeric
        {   int k = 0;
            while (item[k] != null)
                k++;
            return k;
        }
11.38   public static boolean allSmall (Numeric[ ] item, Numeric par)
        {   for (int k = 0;  item[k] != null;  k++)
            {   if (item[k].compareTo (par) >= 0)
                    return false;
            }
            return true;
        }
11.39   public static int indexSmallest (Numeric[ ] item)
        {   int valueToReturn = 0;
            for (int k = 1;  item[k] != null;  k++)
            {   if (item[k].compareTo (item[valueToReturn]) < 0)
                    valueToReturn = k;
            }
            return valueToReturn;
        }
11.40   public static void delete (Object[ ] item, int n)  // best if Object, not Numeric
        {   if (n >= 0 && n < item.length)  // no need to verify it is non-null
            {   for ( ;  item[n] != null;  n++)
                    item[n] = item[n + 1];
            }
        }
11.41   public static Numeric[] getComplexes (Numeric[ ] item)
        {   Numeric[ ] valueToReturn = new Numeric [item.length];  // all initially null
            int next = 0;
            for (int k = 0;  item[k] != null;  k++)
            {   if (item[k] instanceof Complex)
                {   valueToReturn[next] = item[k];
                    next++;
                }
            }
            return valueToReturn;
        }
```