Interlude: Integers And For-Loops

The next two chapters, Chapters Four and Five, explain how you store data values in objects using instance variables and how you store data values in classes using class variables. You use these language features in defining object classes that depend only on the standard library. You will need to know how to work with whole-number values. This Interlude collects this information in one place, so that the following two chapters can concentrate just on language features for defining your own classes from scratch.

You have previously seen that a variable declared as type **boolean** can store either of the values `true` and `false`. You may assign to a boolean variable the result of a call of a boolean method or the result of combining boolean values using any of the three operators `!` (meaning "not"), `&&` (meaning "and"), and `||` (meaning "or").

You may also declare a variable as type **int** (rather than e.g. String or boolean). This means that you can store a whole-number value in that variable, as long as it is in the range plus or minus 2,147,483,647. The reason for that limit is that the storage space set aside for these whole-number values is limited to 31 base-two digits along with a plus or minus sign, and 2^{31} is 2,147,483,648, just slightly over two billion.

Increment and decrement operators

The following method could be in a subclass of Vic (described in Chapters Two and Three). It tells how many slots the Vic has. First, the original position is recorded in `spot`. Then each time the first while-loop moves forward in the sequence of slots (by executing `moveOn()`), `count` is incremented by 1. When `seesSlot()` becomes false, the second while-loop backs up to the original position. Then `count` is returned as the answer to the `getNumSlots()` query. Reminder: The body of a while-statement must be enclosed in braces `{ }` unless it consists of only one statement:

```
public int getNumSlots()
{  String spot = getPosition();
   int count = 0;
   while (seesSlot())
   {  count++;
      moveOn();
   }
   while ( ! spot.equals (getPosition()))
      backUp();
   return count;
}  //======================
```

The `++` symbol is the **increment** operator; it adds 1 to the int variable to which it is appended. So the expression `count++` changes the value of count to be 1 more than its current value. Java also has a **decrement** operator, e.g., the expression `num--` would subtract 1 from the value stored in the `num` variable.

A method that returns an int value does so analogously to methods that return boolean values: You specify int in the heading as its return type and then have one or more statements consisting of `return` followed by an int value. As with any `return` statement, executing this statement immediately stops execution of the method.

You may also have int variables as parameters. The following method could be in a subclass of Turtle; a sample call of this method is `tina.makeSquare(40)`. The executor of the instance method (the object before the dot; `tina` in this case) turns 90 degrees and then draws a line `side` pixels long.

```
public void makeSquare (int side)
{  paint (90, side);
   paint (90, side);
   paint (90, side);
   paint (90, side);
}  //======================
```

Arithmetic operators for integers

A **binary operator** is a symbol which combines two values to obtain a new value. Those two values are the **operands** of that operator. For instance, `&&` and `||` are binary operators. By contrast, `!` is a **unary operator**: You apply it to only one value (operand) to obtain a new value. The words "binary" and "unary" come from the Latin for "two" and "one". Java provides five binary arithmetic operators for int values:

- **`x + y`** is the integer result of adding y to x;
- **`x - y`** is the integer result of subtracting y from x;
- **`x * y`** is the integer result of multiplying y times x;
- **`x / y`** is the integer quotient after dividing y into x; and
- **`x % y`** is the integer remainder after dividing y into x.

Since `x / y` is a whole number, the fractional part of the division is discarded. So `12 / 4` and `13 / 4` and `15 / 4` are all `3`. Since `x % y` is the remainder from the division, `12 % 4` is `0` and `13 % 4` is `1` and `15 % 4` is `3`. Negative numbers can be tricky: `(-13) / 4` is `-3` and so `(-13) % 4` is `-1`.

Caution If one operand of an arithmetic operator involves another arithmetic operator, you should usually put parentheses around that operand to make your meaning clear. The compiler applies normal algebra rules for clarifying an expression such as `x + y * z`, but it is safer to parenthesize. This book does so except for repeated additions.

Comparison operators for integers

Java has six binary operators which compare two numbers to obtain a true-false value:

- **`x > y`** means x is greater than y;
- **`x < y`** means x is less than y;
- **`x <= y`** means x is less than or equal to y;
- **`x >= y`** means x is greater than or equal to y;
- **`x == y`** means x equals y (note the DOUBLE equals-sign);
- **`x != y`** means x is NOT equal to y.

You could generalize the earlier `makeSquare` method to make any triangle, square, pentagon, or other regular polygon, by supplying a second int parameter that tells how many degrees to rotate each time (`turn` must evenly divide 360 for this to work right). In this method, the statement `total = total + turn;` means that you first calculate the sum of `total` and `turn` and then store the result in `total`:

```
public void makeRegularPolygon (int turn, int side)
{  paint (turn, side);
   int total = turn;
   while (total < 360)
   {  paint (turn, side);
      total = total + turn;
   }
}  //======================
```

Strings concatenated with ints and objects

String and int are two different types of values. Java is strongly typed, which means that there are strict limits on assigning a value of one type to a variable of another type. In particular, you cannot assign an int value to a String variable or return an int value when a String value is to be returned. Nor can you assign a String value to an int variable or return it from an int-returning method. However, the **concatenation operator** `+` will combine a String value and an int value into a String value by treating the int value as if it were a numeral, i.e., the string of digits as you would normally write them. So `"a" + 572` is the string `"a572"`, `32 + "cat"` is the String value `"32cat"`, and `"8" + "4"` is the String value `"84"`.

The following method call prints the value of the String parameter in the terminal window (often called the DOS window). The method call usually has a concatenation of a String and a numeric value as its parameter, to help in tracing the execution of the program:

```
System.out.println ("the value of x is " + x);
```

If a method is to return a String value and you have declared `int x` in that method, the statement `return x;` would not even compile, because you cannot return an int value when a String value is required. But concatenating the empty String `""` with the int value makes a String value expression, and it is legal to return that value from the method. So you could have `return "" + x;` as a statement in the method.

General principle When you "add" a String value to anything, you always get a String value. This definition of the plus sign makes coding in Java easier. The compiler expects that you made a mistake if you assigned a non-String value to a String variable or returned a non-String value from a String method. But if you explicitly add "" to a non-String value, you are both acknowledging and correcting the incompatibility.

Java allows variables to be declared as type `double` as well as `int`, which means they can store numbers with decimal points (such as 4.7 and -0.53). Chapter Six contains detailed discussion of the use of this type of number. We do not need it until then.

Language elements
A Statement can be: VariableName ++ ;
or: VariableName -- ;
You may use int as the declared type of a variable or as the return type of a method.
An int literal is a sequence of digits, optionally preceded by a negative sign.
The operators that combine two int values to get an int value are + - * / %
The operators that combine two int values to get a boolean value are > < == != >= <=
You may use a plus sign between a numeric value and a String value. This concatenates the numeral with the string of characters.

Loop controls

The `makeBigSquare()` method of the Turtle class (Section 1.4) contained just four statements, each being the command `paint(90,40)`. And the `makeHexagon()` method contained just six statements, each being the command `paint(60,30)`. The following expresses the same logic using while-statements:

```
public void makeBigSquare()
{  int k = 0;
   while (k < 4)
   {  paint (90, 40);
      k++;
   }
}  //====================
```

```
public void makeHexagon()
{  int k = 0;
   while (k < 6)
   {  paint (60, 30);
      k++;
   }
}  //====================
```

In each case, the while-statement tests the value of `k` to see if the loop should continue (`k < someValue`). Since `k` is the only variable in the continuation condition whose value is changed by the loop, it is the **loop control variable** in each case. The while-statement is preceded by an **initializer step** `int k = 0`, to assign the starting value of the loop control variable. And the last statement in each loop is an **update step** `k++`, whose purpose is to modify the value of the loop control variable.

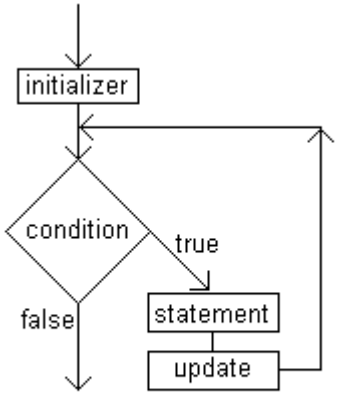
The for-statement

Java's for-statement allows us to bring these three intimately-connected parts of the logic together in one place, to make the overall structure clear. Those two earlier methods can be written as follows with exactly the same effect:

```
public void makeBigSquare()
{  for (int k = 0;  k < 4;  k++)
      paint (90, 40);
}  //======================

public void makeHexagon()
{  for (int k = 0;  k < 6;  k++)
      paint (60, 30);
}  //======================
```

In general, `for(initializer; condition; update){...}` is a loop that executes as long as the condition is true, with the initializing command executed just before the first iteration and the updating command executed at the end of each iteration. The two semicolons are required inside the parentheses. As with the other structured statements using `if` and `while`, you may omit the braces around the subordinate part of a for-statement if it is just one statement.



for (initializer; condition; update) statement

Figure 1 Flow-of-control for the for-statement

The `makeRegularPolygon` method shown earlier in this Interlude can be written using a for-statement as follows. It illustrates the fact that that the loop control variable does not always have to increment or decrement by 1 each time:

```
public void makeRegularPolygon (int turn, int side)
{  for (int total = 0;  total < 360;  total = total + turn)
      paint (turn, side);
}  //======================
```

You are not required to declare the loop control variable in the initializing step. But if you do declare a variable in the initializing step of a for-statement, the compiler does not allow you to mention it outside of the for-statement.

Progressing through a sequence of integers

The following method finds the first power of 2 that is greater than a given int value:

```
public int powerGreaterThan (int given)
{  int power;
   for (power = 1;  given >= 1;  power = 2 * power)
      given = given / 2;
   return power;
}  //======================
```

However, this is going too far. If you are undisciplined in the use of the for-statement, it loses its basic meaning of "continuation condition together with the update of the variable on which the continuation condition depends". It is not a coincidence that the preceding for-statements have an integer as the loop control variable. The common features in the earlier examples are as follows:

- The updating phrase moves the loop control variable one step further in a sequence of integers or the equivalent.
- The body of the for-statement does not change the position of the loop control variable in the sequence of integers.
- The continuation condition stops the loop when the loop control variable reaches the end of the sequence of integers, if not before.

Programming Style Some programmers use a for-statement wantonly, ignoring its basic meaning. This book restricts the use of the for-statement to situations with the three features above. It is also good style to declare the loop control variable in the for-statement heading where possible.

There are some cases in which the loop control variable is given its initial value several statements before the for-statement. You may leave out the initializer step in such a case, so that the first thing inside the parentheses of the for-statement heading is a semicolon. But the two semicolons are required in the heading of the for-statement even when the initializer step is missing. Note: You may use this even simpler rule: Use a for-statement when the updating is k++ or k--; use a while-statement for all other loops.

Caution When the body of a for-statement or while-statement is a single statement rather than something in braces, that single statement cannot be a variable declaration. Similarly, when one of the two subordinate parts of an if-statement is a single statement not within braces, the statement cannot be a variable declaration. For example, the compiler does not allow the phrase `if (x == y) int max = z;`.

The do-while statement

You have now seen two looping statements in Java, the while-statement and the for-statement. Java has one more, the **do-while statement**. The following coding does exactly what a while-statement would do, repeatedly executing the two statements, except that no test of the condition is made until after the first time through the loop:

```
do
{  paint (turn, side);
   total = total + turn;
}while (total < 360);
```

The need for the do-while statement rarely arises. We will not use it in this book except in case studies at the ends of Chapters Four and Five.

Programming Style The do-while statement is one case where you should put something on the line with the ending brace. Do it because, if the while-part were on the next line, it would deceive people into thinking it is the beginning of a new while statement instead of the end of a do-while statement. Deceiving people is not good style. You will further avoid confusion if you use braces around the body of the do-while statement even when it has only one statement in it.

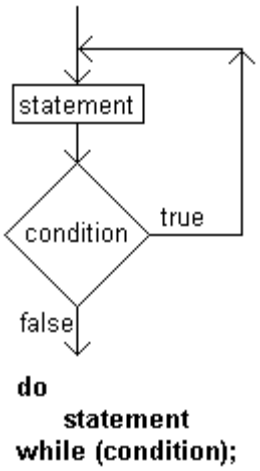


Figure 2 Flow-of-control for the do-while statement

Language elements

A Statement can be: for (Initializer ; Condition ; Update) Statement
or: for (Initializer ; Condition ; Update) { StatementGroup }
or: do { StatementGroup } while (Condition) ;
An Initializer can be: int VariableName = Expression
or: VariableName ++
or: VariableName --
or: VariableName = Expression
The Update part can be an assignment or a method call.
Any one of the three parts of a for-statement can be left out, but keep the two semicolons.
If the condition is left out, the loop continues until a return statement in its body is executed.

Exercise 1 If x is 23 and y is 5, what are `x / y`, `(x+1) / y`, and `(x+2) / y`?
Exercise 2 If x is 23 and y is 5, what are `x % y`, `(x+1) % y`, and `(x+2) % y`?
Exercise 3 How would you revise the `getNumSlots` method in this section to return the number of slots that contain CDs?
Exercise 4 Write a method `public void FiveCircles()` for a subclass of Turtle: The executor draw five circles, all with the same center, but with diameters of 60, 120, 180, 240, and 300. Use a for-statement.
Exercise 5 Write an action method for a subclass of Turtle that has the executor print the word "Hi" seven times, moving ahead 20 pixels after each word. Use a for-statement. Use a parameter for 7. Assume the Turtle is facing due east.
Exercise 6 How many times does each of the following print "Hi"?

(a)
```
for (int k = -4;  k <= 5;  k++)  System.out.println ("Hi");
```
(b)
```
for (int k = -2;  k < 8;   k++)  System.out.println ("Hi");
```
(c)
```
for (int k = 3;   k >= -6; k--)  System.out.println ("Hi");
```

Exercise 7 Rewrite the while-statement in the main method at the end of Section 3.1 as a do-while statement.

Answers to Exercises

1 23 / 5 is 4. 24 / 5 is 4. 25 / 5 is 5.
2 23 % 5 is 3. 24 % 5 is 4. 25 % 5 is 0.
3 Replace the count++ statement by: if (seesCD()) count++;
4
```
public void FiveCircles()
{   for (int radius = 30;  radius <= 150;  radius = radius + 30)
        swingAround (radius);
}
```
5
```
public void ManyHi (int numWords)
{   for (int k = 0;  k < numWords;  k++)
    {   say ("Hi");
        move (0, 20);
    }
}
```
6 (a) 10 times. (b) 10 times; (c) 10 times.
7
```
do
{   sequence.takeCD();
    sequence = new Vic();
}while (sequence.seesSlot());
```

4 Instance Variables

Overview

So far you have derived new classes that extend what objects can do. This chapter shows you how to derive new classes that extend what the objects know. These programs will be developed entirely with classes from the Sun standard library. So you will be able to run these programs without having the Vic or Turtle class or their analogs.

This chapter introduces a new context for learning about object-oriented software design. You are to develop several programs, each allowing the user to play a game against the computer. Some examples of such games are Checkers and Solitaire (although these particular games are not developed in this chapter). You only need study through Section 4.7 to understand the material in the rest of this book.

- Section 4.1 develops the overall analysis and design for game programs.
- Section 4.2-4.5 describe language features you will need for these game programs: graphical input/output, instance variables, constructors, and whole numbers.
- Section 4.6 implements fully a game of guess-my-randomly-chosen-number.
- Sections 4.7-4.8 explain overloading, overriding, polymorphism, and precedence.
- Sections 4.9-4.10 implement fully the games of Nim and Mastermind, as further examples of analysis and design.
- Section 4.11 introduces the BlueJ development environment. This downloadable free software provides a very nice editor, debugger, and runtime environment.
- Section 4.12 shows you how you can use buttons and textfields in your programs.

4.1 Analysis And Design Of A Game Program

The computer games we develop will all have the same general structure. To get a fix on that structure, we begin by creating a **prototype**, a program that has the proper structure but plays a very trivial game. This gives us a solid foundation for the real games.

For this trivial game, the user tries to guess the secret word the computer has chosen. It is trivial because the secret word is not so secret -- it is "duck". The main application only needs to create a game-playing object and tell it to play. This straight-line logic is shown in Listing 4.1. For any of the more complicated games to be developed later, the main application would be the same except with "BasicGame" changed to "Mastermind" or "TicTacToe" or "Checkers" or whatever is appropriate.

Listing 4.1 A game-playing application program

```
class GameApp
{
   /** Play the basic game repeatedly until the user tires.  */

   public static void main (String[ ] args)
   {  BasicGame game;           // declare the game variable
      game = new BasicGame();   // create an object to put in it
      game.playManyGames();     // tell the object to play
      System.exit (0);          // terminate the GUI interface
   }  //=====================
}
```

A class that has only a `public static void main` method can omit **`public`** from the class heading if the only time you will use it is to execute it from the command line. Omitting "public" just means the class cannot be used by other software packages. It also means you can put that application class in a file with another (public) class.

The **`System.exit(0)`** command simply terminates the program. You have to have it in Listing 4.1 because the game-playing object will use a graphical interface. On some computer systems, the computer will lock up if you finish running a program that has a graphics interface but never execute the `System.exit(0)` command. You did not have to put this command in your Vic programs because it is already in the method that reacts to clicking the X-shaped window-closer icon in the top right corner of the window. System is a class in the Sun standard library, and `exit` is a class method in System.

Analysis and logic design

The way the computer plays many games is quite simple: First it plays a game. Then it asks the user if he/she wants to play another game. If the answer is no, the process stops, otherwise the computer plays another game and the cycle repeats. This general process applies to any of the several games we will develop. That is, once we have implemented it for a BasicGame, we can use that implementation for other games such as Checkers or Chess.

The progress of any one of these games is as follows: First, set up the initial state of the game (in Mastermind, choose a 3-digit random number; in TicTacToe, create an empty 3x3 board; in Checkers, create an 8x8 board with 12 red and 12 black pieces placed in the appropriate squares; etc). Second, ask the user for his/her first move in the game. Then see whether that move wins or loses the game (in almost all games, it will not).

If the user's move does not terminate the game, the computer makes its move in response or takes whatever action is appropriate, then the user takes another move or makes another choice. This cycle continues until the game is over. A reasonable plan for the logic design of the BasicGame is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for BasicGame's playOneGame

1. Ask the user for his/her first guess.
2. Repeat the following until the current guess is completely right...
    2a. Tell the user he/she is wrong, and give a hint.
    2b. Ask the user for his/her next guess.
3. Tell the user that he/she has finally gotten it right.

This logic can be implemented with the following coding for the `playOneGame` method. As usual, empty parentheses indicate a method that has no parameters. The obvious precondition on the five methods called here is that they can only be relied on to make sense when called by this `playOneGame` method in this order:

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

Asking the user whether another game is to be played, or asking for the next guess, requires a method for getting input from the user. And telling the user that the guess is right or wrong requires a method for displaying output to the user. The next section introduces methods from the Sun standard library that allow this.

4.2 Input And Output With JOptionPane Dialog Boxes

The various compiled classes are organized into categories called **packages**. You simply put `package X;` as the top line in a file to make the class in that file be in package X. If you do not specify a package for a class when you compile it, its package is the folder on the hard disk where its file is stored.

The JOptionPane class

The **JOptionPane** class is in the `javax.swing` package (technically, swing is the "subpackage" of the javax package). JOptionPane provides some components for a GUI (Graphical User Interface). A file containing a class that uses the JOptionPane class should have the **import directive** `import javax.swing.JOptionPane;` at the top of the file. This line directs the compiler to look in the Sun standard library package named `javax.swing` for the class. If you do not have the import directive, the compiler is not able to find JOptionPane.

The System and String classes are in the standard `java.lang` package, which the compiler automatically makes available to every class. If you want to use any class that is not in `java.lang` or in your current package, you have to give an import directive to tell the compiler where to find it. You need one import directive for each such class, except if you use several classes from the same package, e.g., `javax.swing`, you may have `import javax.swing.*;` as a shorthand for all import directives for classes from that same package. Reminder: `s.equals(t)` tests whether the String values stored in `s` and `t` have the same characters in the same order.

The showMessageDialog method

The JOptionPane class has three class methods that we will use (i.e., you call them with the class name in place of the executor object). One of these is the `showMessageDialog` method that displays a small rectangle containing a message. It does not ask for user input. An example of its use is as follows:

```
JOptionPane.showMessageDialog (null, "Hi there!");
```

The first parameter of the `showMessageDialog` method call is the window to display the **dialog box** in. When this window value is null, the value that indicates the absence of an object, the dialog box appears in the middle of the monitor screen. This is the only way we will use the `showMessageDialog` method.

The second parameter of `showMessageDialog` is the message to be displayed. In the preceding example, it is characters in quotes, called a **String literal**. The right side of Figure 4.1 shows an example of what such a message dialog box looks like.

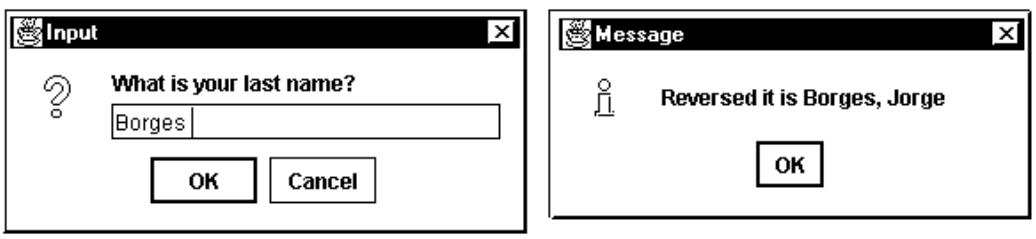


Figure 4.1 Screen shots of calls of showMessageDialog and showInputDialog

If you want to display several lines on this dialog box, you should use the `\n` symbol in the String literal. Each such use causes the start of a new line; `'\n'` is called the **newline** character. So "Hi \nthere \nCathy" displays those three words on three separate lines. The `showFinalStatus` method of the BasicGame class need only print out a message saying the user guessed the secret word right, so it could have this statement:

```
JOptionPane.showMessageDialog (null,
          "That was right.  \nCongratulations.");
```

The showInputDialog method

The `showInputDialog` method in JOptionPane displays a small rectangle containing a given string of characters (the only parameter). This string is a **prompt** for input from the user. Then it waits until the user types something in the empty text box and presses the OK button or the ENTER key. When the user does so, the method returns the string of characters the user typed. The left side of Figure 4.1 shows what this dialog box looks like (the vertical line is the cursor). For instance, the `askUsersNextChoice` method of the BasicGame class could consist of the following statement, storing the input in a String variable named `itsUsersWord`:

```
itsUsersWord = JOptionPane.showInputDialog
                ("Guess the secret word: ");
```

Listing 4.2 is an application program that illustrates the use of these two JOptionPane methods. It asks the user for the user's first name and last name, stores each in a String variable, then prints them out in reverse order with a comma between them (e.g., "Jones, Bill"). It needs the `System.exit(0)` because it uses JOptionPane's graphic interface. Figure 4.1 shows what you might see on the screen when the program runs. Note: This is one of the few application programs in this book that does not call an instance method.

The next-to-last statement of Listing 4.2 uses a plus sign between pairs of String values to indicate that the message is a single String value obtained by combining the four String values into one long String value. This is concatenation of string values.

Listing 4.2 An application program illustrating the use of JOptionPane

```
import javax.swing.JOptionPane;

class NameChange
{
   /** Ask the user for his/her first name, then for the last
    *  name, then print them out in the opposite order.  */

   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null,
              "Illustrate the use of JOptionPane methods");
      String firstName = JOptionPane.showInputDialog
                         ("What is your first name?");
      String lastName = JOptionPane.showInputDialog
                         ("What is your last name?");
      JOptionPane.showMessageDialog (null,
               "Reversed it is " + lastName + ", " + firstName);
      System.exit (0);   // needed when using JOptionPane
   }  //====================
}
```

The showConfirmDialog method

JOptionPane also has a method that asks a yes/no question (supplied as a String for the second parameter; the first parameter is again null). If the user clicks the YES option, the value returned is an int value named YES_OPTION, defined in the JOptionPane class. The `==` operator tests the value to see if it equals the YES_OPTION, so the following logic could be used for the `playManyGames` method of the BasicGame class:

```
playOneGame();
while (JOptionPane.showConfirmDialog (null, "again?")
               == JOptionPane.YES_OPTION)
   playOneGame();
```

As usual, we can omit the braces around the body of the while-statement when it only contains one statement. Note: Due to a glitch in this Sun standard library method, the user cannot just tab over to the NO or CANCEL option and then press the ENTER key; the user must actually click NO or CANCEL to terminate this loop.

The null value

The value **null** is a special value that can be assigned to any object variable. It indicates the absence of a reference to an actual object. For instance, if `sam` refers to some String object and you do not want it to do so anymore, execute the command `sam = null`. That erases the object reference in `sam`. `sam` then refers to no object at all, so of course you cannot execute `sam.equals(x)` or any other such method call -- it would not make sense to ask a nonexistent object to answer a question or to perform an action.

If the user clicks the OK button in response to the `showInputDialog` message without entering any value, it returns the empty String, a String with no characters, written as `""`. But if the user clicks the Cancel button or the closer icon in the upper-right corner, the method returns null. Your coding must allow for these two possibilities.

Indenting a Java program

Programming Style In Listing 4.2, four of the five statements do not fit on one line. It is good style in such cases to strongly indent the continuation of the line, so a reader does not think it is a separate statement. The line break should come after a comma or semicolon or before a left parenthesis or an operator (such as `+` and `>`).

In general, it is good programming style to indent your Java logic in the conventional way:
1. Indent one tab position for the heading of a method and two tabs for its body.
2. Indent one extra tab position for a statement subordinate to `if`, `while`, `do`, or `for`.
3. Indent two extra tab positions or more for a continuation of a program line onto the next line.
4. Indent almost nowhere else.

Language elements

You may put an import directive before the first class in a file. Use one of these two formats:

```
import  PackageDescription . ClassName  ;
import PackageDescription . * ;
```

The value null may be assigned to any object variable or otherwise used as an object value.
The \n sequence within quotes indicates that the start of a new output line occurs at that point.

Exercise 4.1* Write an application program that asks the user for two strings of characters and then tells the user whether they are equal to each other (print "equal" or "different"). But tell the user "Stop fooling around" if either entry is null.
Exercise 4.2* Draw the UML class diagram for Listing 4.2.

4.3 Declaring Instance Variables: A First Look At Encapsulation

The only way you have previously seen to store information is in variables declared within a method. Those variables "belong" to the method and only exist during a single execution of the method . But in order for a BasicGame object to do what it has to do, such as decide whether the current game should continue, it needs to know two things: What is the secret word and what is the user's most recent choice. Information is stored in variables, but a local variable is not suitable for this information.

Declaring instance variables

You need to store these two pieces of information as part of the BasicGame object. Java indicates this relation by having the variables declared outside of any method to distinguish them from local variables. You could name them `itsSecretWord` and `itsUsersWord`. They are called **instance variables**, because each instance (object) of the BasicGame class has its own separate values stored in these variables.

These two instance variables are declared as **private variables** in the BasicGame class. This means that methods outside of the BasicGame class cannot access them directly. It is legal to declare them as public instead, but we rarely do that with instance variables. Bugs in a program are usually easier to avoid if we prevent direct access by outsiders to the instance variables of objects. This is called **encapsulation**. Since this simplest of all games has the secret word "duck", and the user has made no guess when the game is first created, the BasicGame class contains the following two declarations:

```
private String itsSecretWord = "duck";
private String itsUsersWord = "none";
```

Each time a BasicGame object is created, it is given these two variables, where the initial value of the first is "duck" and the initial value of the second is "none". These declarations are exactly the same as for local variables except they have `private` at the beginning. Note: Some books define encapsulation more generally, to mean keeping data and objects together in one class so that outside classes can access variables only indirectly, through method calls that protect against inappropriate changes.

Using instance variables

Suppose `sam` refers to an instance of a class that has an instance variable `someVar`. Then `sam.someVar` is the `someVar` variable belonging to the object `sam` refers to. Within an instance method, `this.someVar` is the variable that belongs to the executor of the method, but you can write just plain `someVar` (`this` refers to that executor).

This is analogous to what you already know about instance methods: If `sam` refers to an instance of a class, then `sam.someMethod()` calls the `someMethod` belonging to the object `sam` refers to. Within an instance method, `this.someMethod()` calls the method that belongs to the executor, and so does just plain `someMethod()`.

Listing 4.3 (see next page) shows the complete BasicGame class. Some people prefer to put the instance variables first in the class, as shown, and others prefer to put them last in the class. The order of the declarations of instance variables and instance methods in a class generally has no effect on whether the class compiles or how it executes.

The logic of the methods for the basic game in Listing 4.3 was discussed earlier:

- `askUsersFirstChoice` stores the input in the game's own variable named `itsUsersWord`.

- `shouldContinue` is true whenever the two words are not the same word. The `equals` method for Strings tests whether two objects have the same contents. If the user clicked Cancel, the input is null, and the `equals` method returns `false` when the parameter is null.
- `askUsersNextChoice` gets the user's response on each turn other than the first. It does the same as `askUsersFirstChoice` for this game.
- The two methods that show the status simply print an appropriate message.

Listing 4.3 The BasicGame class of objects

```
import javax.swing.JOptionPane;

public class BasicGame extends Object
{
   private String itsSecretWord = "duck";
   private String itsUsersWord = "none";


   public void playManyGames()
   {  playOneGame();
      while (JOptionPane.showConfirmDialog (null, "again?")
                     == JOptionPane.YES_OPTION)
         playOneGame();
   }  //=====================

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

   public void askUsersFirstChoice()
   {  itsUsersWord = JOptionPane.showInputDialog
                     ("Guess the secret word:");
   }  //=====================

   public boolean shouldContinue()
   {  return ! itsSecretWord.equals (itsUsersWord);
   }  //=====================

   public void showUpdatedStatus()
   {  JOptionPane.showMessageDialog (null,
                     "That was wrong.  Hint:  It quacks.");
   }  //=====================

   public void askUsersNextChoice()
   {  askUsersFirstChoice(); // no need to write the coding again
   }  //=====================

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

The **state** of an object is determined by its instance variables. They give the object its "personality". A method that modifies the executor's state and does nothing else is a **mutator** method (a special kind of action method, e.g., `askUsersNextChoice`). A method that merely accesses the executor's state is an **accessor** method (a special kind of query method, e.g., `shouldContinue`).

The Object class

The superclass for the BasicGame class is the Object class. **Object** is a class in the Sun standard library. It is in the `java.lang` package, which means you do not need an import directive to help the compiler find it. The Object class is the ultimate superclass of every class in Java; if you define any class without an `extends` phrase, the compiler supplies the **default extension** `extends Object`. So we could have left that phrase out of Listing 4.3 with no difference in effect. Chapters Four through Six put it in classes that have instance methods or instance variables, as a reminding that we are defining classes of objects.

The Object class contains a method named `toString`. It returns a String equivalent of the object. A plus sign between a String value and an object reference `x` causes the object reference to be interpreted as `x.toString()`, using that object's `toString` method. So if `now` is an object variable for which `now.toString()` has the value `"0730"`, then the phrase `"now = " + now` has the value `"now = 0730"`.

Exercise 4.3 Revise the `shouldContinue` method to change the secret word to "goose" for the second and later guesses if the first guess is not "duck".
Exercise 4.4 (harder) Revise the BasicGame logic so that the user gets it right after at most five guesses, as follows: Add an instance variable initialized to be the empty String at the start of each game. Concatenate "x" to it every time the user makes a guess. Then treat whatever answer the user gives as right when that instance variable has the value "xxxxx" (this illustrates how you can count without using numbers).
Exercise 4.5* Revise the BasicGame logic so that the user who takes more than three guesses to get it right will never succeed in guessing the right answer. Hint: See the preceding exercise.
Exercise 4.6* Explain the difference between an instance variable and a local variable.

4.4 Defining Constructors; Inheritance

When you create a new object, as in `sam = new SmartTurtle()` or `sue = new Vic()`, you do it by calling on a **constructor** method in that class. If you (or whoever developed the class) did not provide one or more constructor methods, the compiler provides one for you by default. It is as follows for the SmartTurtle class you saw in an earlier chapter:

```
public SmartTurtle()  // default constructor
{  super();
}  //======================
```

This **default constructor** (supplied by the compiler) is the method that `sam = new SmartTurtle()` calls, since the SmartTurtle class (Listing 1.3) does not explicitly define a constructor. The default constructor says you construct a new SmartTurtle object by doing nothing but calling the constructor method of the superclass of the SmartTurtle class. That is, of course, the Turtle class. The Turtle class provides an explicit constructor which the SmartTurtle constructor calls with the command `super()`.

The default constructor for the BasicGame class or any other class is the same as the above except of course the method heading is different, e.g.:

```
public BasicGame()  // default constructor
{  super();
}  //======================
```

`super()` calls the no-parameter constructor of the superclass, which for BasicGame is the Object class. That fills in the instance variables specified by the Object class. The Object class has a number of instance variables and methods that all objects need. One instance variable tells the class of the object, so the runtime system can inspect it to see what methods it is allowed to call. Another instance variable keeps track of information that the automatic garbage collection process needs. These Object instance variables are private, so you cannot access them in a program.

Explicitly defined constructors

When you define a class for which you want something more than what the default constructor gives you, you have to define the constructor(s) explicitly in the class. The first statement should always call `super()` (or its equivalent, discussed later). `super()` is only allowed for the first statement of a constructor.

Listing 4.3 specifies initial values for the two instance variables of a BasicGame. An alternative is to not have an assignment in the declaration, but to assign the value in the constructor itself. That is, the two lines declaring the instance variables in Listing 4.3 would be replaced by the following variable declarations and constructor:

```
private String itsSecretWord;
private String itsUsersWord;

public BasicGame()            // constructor
{  super();
   itsSecretWord = "duck";
   itsUsersWord = "none";
}  //======================
```

There is no reason to do so in this case, but usually it is necessary. For example, suppose you want a Vic object to be able to answer the question, "Was your first slot empty when you were created?" The answer is `true` if the Vic saw a slot but did not see a CD in that slot. You could test `someVic.firstWasEmpty()` at any time if you have the following instance variable and two methods in a subclass of Vic:

```
public class GoodVic extends Vic
{
   private boolean itsFirstWasEmpty;

   public GoodVic()           // constructor
   {  super();
      itsFirstWasEmpty = seesSlot() && ! seesCD();
   }  //======================

   public boolean firstWasEmpty()
   {  return itsFirstWasEmpty;
   }  //======================
}
```

Inheritance

An inconvenience with Vics is that you cannot return to the front of the sequence because you do not know how far back to go. That can easily be fixed if you define the following class of objects that know their starting position:

```
public class BasicVic extends Vic
{
   private String itsInitialPosition;

   public BasicVic()          // constructor
   {  super();
      itsInitialPosition = getPosition();
   }  //======================

   public void returnToStart()
   {  while ( ! itsInitialPosition.equals (getPosition()))
         backUp();
   }  //======================
}
```

Now if you change the heading on any subclass of Vic to say `extends BasicVic` instead of `extends Vic`, it inherits the `returnToStart` method. Then you can execute `sam.returnToStart()` for any object `sam` of that subclass, to have `sam` repositioned at the front of its sequence. Reminder: A subclass **inherits** all public methods and variables of the superclass it extends, i.e., an instance of the subclass can refer to them as if they were defined in the subclass.

Parameters of constructors

A constructor may have one or more parameters. For instance, you might want to define a class of objects that remember their first names and their last names and can provide them on request. The Person class in the upper part of Listing 4.4 (see next page) is such a class. You then must supply both names when you construct a new Person object, as in the following statements:

```
al = new Person ("Jorge", "Borges");
String fullName = al.getFirstName() + " " + al.getLastName();
```

The constructor in the Person class illustrates the most commonly used format for constructors, namely, the parameters supply the initial values of instance variables. You could call it the **natural constructor**, since the primary purpose of a constructor is to initialize all instance variables.

To develop software for a hospital, you might define Patient as a subclass of Person. Then the first statement in the Patient constructor is the `super` call of the constructor from its superclass, which is Person. Constructing a Person object requires supplying the first and last name. So Patient's `super` call must include those two parameters to initialize the private instance variables in the Person superclass. You cannot use plain `super()`, since the Person class does not have a constructor with no parameters. Therefore, part of the Patient class could be as shown in the lower part of Listing 4.4.

You may omit the call of `super`, in which case the compiler inserts `super()` (i.e., no parameters) in your constructor by **default**. So the `super` statement in the Person constructor is optional, but the explicit call of `super` in the Patient constructor is required because the insertion of `super()` would call a non-existent constructor with no parameters, which the compiler will not allow. This book writes the `super()` call at the beginning of each constructor in this chapter and the next, to remind you it is there.

Listing 4.4 The Person class of objects

```
public class Person extends Object
{
   private String itsFirstName;
   private String itsLastName;


   public Person (String first, String last)    // constructor
   {  super();
      itsFirstName = first;
      itsLastName = last;
   }  //======================


   public String getFirstName()
   {  return itsFirstName;
   }  //======================


   public String getLastName()
   {  return itsLastName;
   }  //======================
}
//############################################################


public class Patient extends Person
{
   private String itsDoctor;

   public Patient (String first, String last, String doc)
   {  super (first, last);
      itsDoctor = doc;
   }  //======================
}
```

Name shadowing

Caution A method is allowed to have a local variable or parameter with the same name as an instance variable. But it usually causes trouble. Within that method, the instance variable is **shadowed**: Use of the variable name is a reference to the locally declared variable, not to the instance variable. A common logic error is to "redeclare" an instance variable; e.g., to write the last statement of the Patient constructor as `String itsDoctor = doc`. That extra word String means you are declaring a new variable named `itsDoctor` local to the method. The compiler will not point out that it is surely not what you meant to do.

A related error occurs in the Patient constructor if you name the instance variable `doc`, the same name as the parameter, and so the last statement is `doc = doc`. This merely assigns the value of the parameter to the parameter, not to the instance variable. This form of assignment is never necessary and usually indicates a logic error in the program.

This book puts a prefix of "its" on almost all instance variables and never anywhere else. This hallmark makes both of the above-mentioned bugs less likely. Some people prefer instead the prefix "my", as in `myFirstName`, `myLastName`, `myDoctor`. If you do not do this sort of thing in your own definitions, at least obey the following safety principle: Never name a parameter or local variable the same as an instance variable.

Programming Style You should explicitly initialize each instance variable to a value either in its declaration or in every constructor. This makes programs clearer. You should make the initialization in the declaration rather than the constructor when possible. That will make it clear to the reader of the coding that the initial value does not depend on the constructor's parameters.

The reason that this is a style principle rather than a requirement is that, if you do not assign a value to an instance variable in its declaration or in some constructor, the runtime-system assigns one for you (null or false or zero). But you should not require the reader to remember this minor point of the Java language. Explicitly initialize.

The use of this for instance variables

If the body of instance method X contains a method call without saying which object is its executor, then its executor is by default the executor of X. The same principle applies to instance variables: If the body of instance method X mentions an instance variable without saying which object it belongs to, it is by default a use of the executor's instance variable. Since the keyword `this` can be used in an instance method to refer to the executor of the method, the statement in the Person's `getFirstName` method of Listing 4.4 could be written as `return this.itsFirstName;` to do the same thing.

A constructor method is technically neither a class method nor an instance method; it is a method of constructing objects of the class. Within a constructor definition, the use of an instance variable without saying which object it belongs to is by default a use of the instance variable of the object being constructed (the same is true of instance methods). The pronoun `this` refers to the object being constructed. So the last two statements of the Person constructor in Listing 4.4 could be written as follows without changing the constructor's effect:

```
this.itsFirstName = first;
this.itsLastName = last;
```

Caution Do not assign a value to `this`, as in `this = whatever;` it is never necessary, and the compiler will refuse to accept it anyway.

Language elements

Instance variables are declared outside of any method. You may use one of these two formats:

```
private Type VariableName = Expression;
private Type VariableName ;
```

A Declaration may be: public ClassName (Parameters) { StatementGroup }
A Statement may be: super (ExpressionsSeparatedByCommas) ;
Such a statement is only allowed as the first statement in a constructor.

Exercise 4.7 Write out the default constructor for the SmartTurtle class.
Exercise 4.8 Write a RedTurtle constructor for a subclass of Turtle, in which the turtle object always starts with the drawing color red and a due west heading.
Exercise 4.9 Define a NamedTurtle class: Objects are given a name when constructed (via a parameter) and they tell you what the name is when you use the `getName` method in an expression such as `sam.say ("I am " + sam.getName())`.
Exercise 4.10* Define a Terrapin subclass of the NamedTurtle class in the preceding exercise. A Terrapin is given its name and its best friend (another Turtle object) when constructed, and can tell you its name and who its best friend is when asked.

4.5 Integer Instance Variables

You can store numeric information in an instance variable of an object. For example, if you want each Person object to remember its birth year, you could add an instance variable to the Person class of Listing 4.4 as follows:

```
int itsBirthYear;  // instance variable for Persons
```

Then the Person constructor could have a third parameter so that each construction of a Person object supplies the birth year, as in `sam = new Person ("Jorge", "Borges", 1899)`. A representation of this Person object is in Figure 4.2.

```
public Person (String first, String last, int year)
{  super();
   itsFirstName = first;
   itsLastName = last;
   itsBirthYear = year;
}  //======================
```

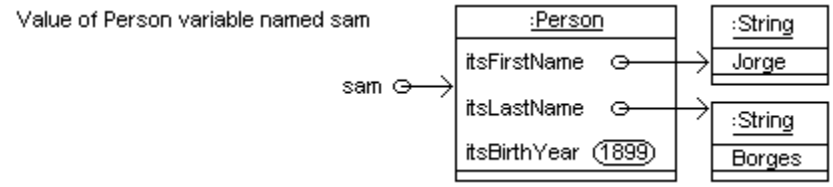


Figure 4.2 UML object diagram for a Person object

You would also want to add to the Person class a method to allow people to ask a Person object for its birth year with a method call such as `sam.getBirthYear()`:

```
public int getBirthYear()
{  return itsBirthYear;
}  //======================
```

A Turtle object needs to keep track of its current position (conventionally measured from the top left corner of the frame), and its current heading (conventionally measured in degrees counterclockwise of a due east heading). So the instance variables might be declared and initialized as follows:

```
private int itsHeading = 0;  // due east
private int itsX = 380;  // in the center of 760 pixels wide
private int itsY = 300;  // in the center of 600 pixels tall
```

The Time class of objects

In some programming situations you work with clock times and you need to compute how many hours and minutes there are between two clock times. For instance, you may need to compute that 10:15 in the morning is 2 hours and 45 minutes after 7:30 in the morning. You may also need to solve problems such as finding out what clock time is 2 hours and 45 minutes after 7:30 in the morning. When you add in the complexities of accounting for afternoon versus morning, you can see that encapsulating these calculations in a class of Time objects will simplify things greatly.

Listing 4.5 defines a class of Time objects that can track the time of day, measured in hours and minutes since midnight. You can then have statements such as the following:

```
now = new Time (7, 30);  // 7:30 in the morning
wait = new Time (2, 45);  // 2 hours 45 minutes
later = now.add (wait);   // produces 10:15 in the morning
```

This class makes Time an abstraction, so that the development of logic that works with clock times is easier.

Listing 4.5 The Time class of objects and its driver (can be in the same file)

```
public class Time extends Object
{
   private int itsHour;
   private int itsMin;

   /** Create an object for the given hour and minute.  If min
    *  is negative, adjust the values to make 0 <= min < 60. */

   public Time (int hour, int min)            // constructor
   {  super();
      itsHour = hour;
      for (itsMin = min;  itsMin < 0;  itsMin = itsMin + 60)
         itsHour--;
   }  //======================

   /** Return the time expressed in military time. */

   public String toString()
   {  if (itsHour < 10)
         return ("0" + itsHour) + itsMin;
      else
         return ("" + itsHour) + itsMin;
   }  //======================

   /** Return the result of adding this Time to that Time. */

   public Time add (Time that)
   {}  // left as an exercise
}
//#################################################################


import javax.swing.JOptionPane;

class TimeTester
{
   public static void main (String[] args)
   {  Time t1 = new Time (13, 25);
      Time t2 = new Time (8, -150);
      JOptionPane.showMessageDialog (null, "1 " + t1.toString());
      JOptionPane.showMessageDialog (null, "2 " + t2.toString());
      Time t3 = t1.add (t2);
      JOptionPane.showMessageDialog (null, "3 " + t3.toString());
      t1 = t2.add (t3);
      JOptionPane.showMessageDialog (null, "1 " + t1.toString());
      System.exit (0);
   }  //======================
}
```

The Time class has two int instance variables for the hours and the minutes, and the constructor is the natural one that initializes the two instance variables with the values of the two parameters. It makes a reasonable adjustment for a negative number of minutes. The `toString` method returns the time expressed as a String value indicating the usual military time. For instance, 14 hours and 15 minutes is expressed as "1415" but 7 hours and 30 minutes is expressed as "0730". A defect is corrected in the exercises.

Time's `toString` method concatenates the "0" with the digits of `itsHour` to get a String value, then concatenates the result with the digits of `itsMin`. The order of the operations is important; `"0" + (itsHour + itsMin)` would have a quite different result, e.g., "042" if `itsHour` were 12 and `itsMin` were 30.

Driver programs

Before you use a class of objects in a program having several classes, you should test out the methods in that class separately. A **driver program** is an application program whose only purpose is to test the methods of a class thoroughly. The TimeTester class in the lower part of Listing 4.5 does this fairly well, by creating three Time objects and using both the `toString` method and the `add` method several times.

This driver program would be an even better test if it could accept input from the user that gives the hour and minute values for one or both of `t1` and `t2`. However, you do not as yet have any way to get a number from the user; `showInputDialog` only gets a string of characters, which is not at all the same thing. This situation will be rectified in the next section.

Programming Style Always put a space on each side of an operator (including + = <) and also directly after a comma or semicolon; it makes your program much easier to read. The space between a method name and its parentheses is optional -- some people prefer it and some do not.

Exercise 4.11 If x is 23 and y is 5, what are `"x" + (x + y)` and `("x" + x) + y`?
Exercise 4.12 Write a Person method `public int getAge (int currentYear)`: The executor tells the current age of the Person, given the current year. But it returns zero if the current year is before the Person's birth year.
Exercise 4.13 If a Time object `t4` has 13 for `itsHour` and 5 for `itsMin`, what is `t4.toString()`? What should it be in military time?
Exercise 4.14 (harder) How would you revise Time's `toString` method to avoid the problem indicated by the preceding exercise?
Exercise 4.15 (harder) Write the Time method `public Time add (Time that)`: The executor returns a new Time object that is the sum of the two, e.g., 0740 add 1430 is 2210. If the sum is more than 2359, drop the extra 24 hours, e.g., 1300 add 1400 is 300.
Exercise 4.16* Write a Time method `public int timeInMinutes()`: The executor tells the total number of minutes that have passed since midnight.
Exercise 4.17* Write a Time method `public Time subtract (Time that)`: The executor returns a new Time object that is itself minus the parameter, e.g., 0720 subtract 1430 is 1650. If the difference is negative, add an extra 24 hours.
Exercise 4.18* If you wanted Time objects to have a third attribute, the name of the day of the week, then (a) What instance variable declaration would you add? (b) What change would you make in the constructor? (c) What instance method would you have that tells the caller the object's day of the week? Write and compile the revised class.
Exercise 4.19* Revise the Time constructor to properly adjust for `itsMin` larger than 59, adding to `itsHour` as needed. Then repeatedly add or subtract 24 from `itsHour` until the value is in the range 0 to 23 (discard the excess; we do not care what day it is).
Exercise 4.20** Revise the Time constructor as stated in the preceding exercise, but do not have any looping statements anywhere in the constructor. Hint: Use the % operator.

4.6 Making Random Choices

We next develop a more interesting game than a BasicGame. The game object starts by picking a secret number at random in the range from 1 to 100, inclusive. It then asks the user to guess the number. If the user gets it right, of course the game is over. Otherwise the game object tells the user whether the guess was higher than the secret number or lower, and the game continues with more guessing.

Random integers

One thing this game needs is the ability to choose an integer value at random from a certain range of values. Fortunately, the Sun standard library has a **Random** class in the `java.util` package that provides objects that can do this. The phrase `new Random()` creates a random number generator that you may assign to a variable of Random type, called perhaps `randy`.

A Random object has an instance method named `nextInt` with a positive int value as the parameter. It returns an int value chosen at random in the range from 0 up to but not including that parameter value:

`Random randy = new Random()` creates a random-number generator.
`randy.nextInt(6)` returns one of 0, 1, 2, 3, 4, or 5, with equal likelihood.
`randy.nextInt(2)` returns either 0 or 1, with equal likelihood.
`randy.nextInt(100)` returns one of 0, 1, 2, ..., 99, with equal likelihood.

In general, `nextInt(n)` returns one of the first `n` non-negative int values, chosen with equal likelihood. These numbers are not truly random, since computers hardly ever do anything at random. But the sequence of numbers you get by repeated calls of the method is close enough for most purposes.

Since the number-guessing game wants one of the 100 int values in the range 1 to 100 inclusive, you need to call `randy.nextInt(100)` to get one of 100 different int values, then add 1 to the result to get one of the 100 different int values beginning with 1. If instead you needed a random int value in the range 20 to 30 inclusive, a total of 11 possibilities, you would call `randy.nextInt(11)` and then add 20 to the result.

The general principle is that the expression `min + randy.nextInt(num)` gives one of `num` consecutive int values with the smallest being `min`. If, however, you want a multiple of ten chosen at random from 30, 40, 50, ..., 150, you could use the expression `30 + 10 * randy.nextInt(13)`, which clearly gives one of the 13 possible multiples of 10. And if you want to get the result of rolling two dice, you could use the following:

```
int die1 = 1 + randy.nextInt (6);
int die2 = 1 + randy.nextInt (6);
JOptionPane.showMessageDialog (null,
      ("The 2 dice show " + die1) + die2);
```

The GuessNumber game

It will be quite convenient to have the GuessNumber class be a subclass of BasicGame. That means we can use whatever methods of the BasicGame class we want. For instance, the `playManyGames` and `playOneGame` methods can be used unchanged for a GuessNumber game. That is, we have the advantage of **reusable software** by making GuessNumber a subclass of BasicGame. We can play the game with this one statement: `new GuessNumber().playManyGames();`.

A GuessNumber object should apparently have two instance variables, perhaps named `itsSecretNumber` and `itsUsersNumber` (parallel to the instance variables in Listing 4.3). The `askUsersFirstChoice` method should begin by having a random number generator assign to `itsSecretNumber` a value from 1 to 100 inclusive. That means you should have the random number generator as an instance variable as well. This initialization of the secret number cannot be done in the constructor, since each time the game is played, the secret number must be re-initialized.

This logic is shown in the top part of Listing 4.6 (see next page). It has the import directive for JOptionPane but not for Random. You do not need to have an import directive if you explicitly name the package that Random is in every time you mention the word Random. This is the "fully qualified name" of the Random class. The alternative to what is shown is to have `import java.util.Random;` as a second line before the class heading and then simply use an unadorned Random in two places.

The `askUsersFirstChoice` method does the same as for any later choice, except that it first initializes `itsSecretNumber`. So it can call the `askUsersNextChoice` method to get the input. There is no reason to write the same coding twice.

The `shouldContinue` method returns `true` whenever `itsSecretNumber` is not equal to `itsUsersNumber`. The test of equality of numbers is always made with the operator `==` or `!=`. The `equals` method can only be used with objects, and int values are not objects.

The rest of Listing 4.6 should be clear except for `askUsersNextChoice`, which is discussed next. Figure 4.3 gives the UML class diagram.

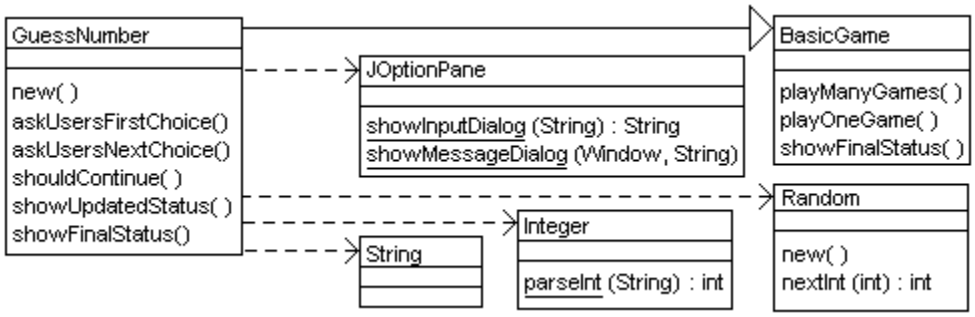


Figure 4.3 UML class diagram for GuessNumber

Caution A test for equality of Strings, as in the `shouldContinue` method of the earlier Listing 4.3, is always made with the `equals` method. Do not use `==` or `!=` to compare strings except in a comparison with null.

The use of the parseInt method for askUsersNextChoice

The `showInputDialog` method returns a String value (which can be null if the user clicked the Cancel button). A String value is not an int value, so you cannot assign it to the int variable `itsUsersNumber`. You have to convert the string of characters, which is a numeral, to a number. There is a difference: The number 88 can be added to or subtracted from other numbers; the numeral "88" is just keystrokes or marks on the screen. Some people like to remember the difference this way: Half of the number 88 is 44, but half of the numeral "88" is "8" (or even "oo" if you cut it in half horizontally rather than vertically). Or perhaps this way: Pro football players have big numerals on their jerseys and big numbers in their bank accounts.

Listing 4.6 The GuessNumber class

```
import javax.swing.JOptionPane;

public class GuessNumber extends BasicGame
{
   private java.util.Random randy;
   private int itsSecretNumber;
   private int itsUsersNumber;


   public GuessNumber()
   {  super();
      randy = new java.util.Random();
   }  //======================

   public void askUsersFirstChoice()
   {  itsSecretNumber = 1 + randy.nextInt (100);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  String s = JOptionPane.showInputDialog
                    ("Guess my number from 1 to 100:");
      if (s != null && ! s.equals (""))
         itsUsersNumber = Integer.parseInt (s);
      else
         itsUsersNumber = -1;  // just to have a value there
   }  //======================

   public boolean shouldContinue()
   {  return itsUsersNumber != itsSecretNumber;
   }  //======================

   public void showUpdatedStatus()
   {  if (itsUsersNumber > itsSecretNumber)
         JOptionPane.showMessageDialog (null, "Too high");
      else
         JOptionPane.showMessageDialog (null, "Too low");
   }  //======================

   // inherited from BasicGame:
   //    playManyGames
   //    playOneGame
   //    showFinalStatus
}
```

Fortunately, the Sun standard library has a class named Integer that can help convert a numeral to a number. **Integer** (in the `java.lang` package) contains a class method:

```
Integer.parseInt (someString)
```

This method returns the int value you get when you interpret the keystrokes as digits of a number. If the user enters letters instead of digits or otherwise provides a badly-formed numeral, the program can crash. This is a violation of the robustness principle: All application programs should be written so that they cannot crash. But you may ignore this possibility until you see the advanced techniques in Chapter Six to avoid such crashes by checking that the String is a well-formed numeral. Until then, all user input that is supposed to be numeric will be assumed to be so, unless it is null or `""`.

Parse means to analyze a string of characters into its component parts. For instance, when you make sense of a Roman numeral, you are not "translating" it to an ordinary number, you are "parsing" it.

So the `askUsersNextChoice` method in Listing 4.6 gets the String value input using JOptionPane's class method and then has it parsed to an int value using Integer's class method. However, a null value or empty String returned by `showInputDialog` is taken as -1. Note that the order of the operands of `&&` is crucial; you cannot ask the null value if it equals the empty String. With the order given, if `s` is null, short-circuit evaluation will avoid testing the call of `equals`.

You have now seen all three varieties of object design:

1. Use existing objects and their methods (the Vic class in Listing 3.3).
2. Use existing objects but add new methods (the GuessNumber class in Listing 4.6).
3. Invent new objects to do the job (the Time class in Listing 4.5).

Language elements
You may use == or != between two object values (but rarely do so unless one is null).

Exercise 4.21 Write an expression that gives a random int value from -5 to 5, inclusive. Also write an expression that gives a random even number from 30 to 50, inclusive.
Exercise 4.22 Revise the GuessNumber game to have the secret number be in the range from 200 to 300, inclusive.
Exercise 4.23 What is the most number of guesses it would take for a really smart person to win this GuessNumber game? Explain your answer.
Exercise 4.24 (harder) Revise the GuessNumber game to tell the user "close enough; you win" when the guess is either 1 or 2 off from the right answer.
Exercise 4.25 (harder) Write a method that asks the user for a sequence of int values and, when `showInputDialog` returns null, announces the smallest of them.
Exercise 4.26* Write an application program that asks the user for two int values and then announces whether one evenly divides the other (using `%`).
Exercise 4.27* Revise the GuessNumber game to say "you're hot" if within 2 of the right answer, "you're warm" if within 6 of the right answer, and "too high" or "too low" otherwise.
Exercise 4.28* Revise the GuessNumber game to have the program lie one-third of the time that the guess is too high; it says it is too low. But it never lies about the guess being too low, nor does it lie twice in a row.
Exercise 4.29* Find the best strategy for getting the right answer in the fewest guesses for the revision of GuessNumber described in the preceding exercise.

4.7 Overloading, Overriding, And Polymorphism

The **signature** of a method is its name followed by the types of its parameters in parentheses (leave out the return type and the names of its parameters). For instance, the signatures of the three JOptionPane methods described in Section 4.2 are as follows (a Component is a graphical object such as a Window):

```
showMessageDialog (Component, String)
showInputDialog (String)
showConfirmDialog (Component, String)
```

and the signatures of the methods of the Time class in Listing 4.5 are as follows:

```
Time (int, int)
toString()
add (Time)
```

Overloading of method names

Java lets you have several methods in the same class with the same name if they have different signatures. This is **overloading** of method names. This is quite common for constructors; you may have as many as you want as long as the compiler can distinguish them based on the parameter structure. For instance, you might want a second Time constructor that has a single int value as input, being the total number of minutes to be converted to hours plus minutes. So you could add to Listing 4.5 a constructor with the following heading:

```
public Time (int totalMinutes)
```

You might also want the Time class to have a second `toString` method, one with a String parameter telling which time zone to use, so it could have this heading:

```
public String toString (String timeZone)
```

In the email metaphor of Section 1.6, a Time object that gets the message with subject line `toString` first checks the body of the message. If it contains a String, that is an indication of the time zone. If the body of the message is blank, it returns military time.

Overriding of method definitions

A class may have the same method heading as a method in its superclass. For instance, both the BasicGame class and the GuessNumber class have a method with the heading `public boolean shouldContinue()`. The Java language rule in such a case is, if `x` is declared as a GuessNumber object somewhere, then `x.shouldContinue()` calls the method in GuessNumber, not the one inherited from BasicGame. The new definition of `shouldContinue()` **overrides** the original definition of `shouldContinue()` for GuessNumber objects, since it has the same signature (name and parameter pattern). This principle applies to instance methods, not class methods.

To illustrate the overriding of method definitions, we have just one more listing involving a subclass of Vic (the last in this book, because you are probably getting tired of Vics by now). This subclass is named BigVic. Each BigVic object keeps its own record of the number of filled slots and the number of empty slots it has. These are two int variables named `itsNumFilled` and `itsNumEmpty`. They are instance variables, because each instance (object) of the BigVic class has its own.

If a program says `sue = new BigVic()`, then `sue's` object has the two instance variables `itsNumFilled` and `itsNumEmpty` attached to it, in addition to whatever all Vic objects have. A call of `sue.getNumFilled()` has `sue` look up the value of `sue.itsNumFilled` and report it back. This is true even if the variable `sue` is declared as a simple Vic variable. If the program also says `sam = new BigVic()`, then `sam`'s object has its own two variables which are completely distinct from `sue`'s.

When `sam` refers to a BigVic object, a main method that calls `sam.putCD()` cannot simply execute the original `putCD()` method in the Vic class, because that would make `sam`'s counts wrong. So the BigVic class must have its own `putCD()` method that makes the two counts right. The method call `sam.putCD()` will execute the `putCD()` in the BigVic class, not the original one inherited from the Vic class. In other words, the new definition of `putCD()` overrides the original definition of `putCD()` for BigVic objects, since it has the same signature.

Caution You cannot have two methods defined in the same class with the same signature. If they have the same name, they must have a different parameter structure (overloading the name). It is not enough to be different in the part of the method heading that comes before the method name.

Calling a method that has been overridden

Part of the job of the new `putCD` method in BigVic is to execute the original `putCD` method in the Vic class. But if you write `putCD()` as one of the statements in the definition of the new `putCD` method, the executor would execute the new BigVic method, not the old Vic method, which would cause no end of difficulty.

Listing 4.7 contains the BigVic class other than the constructor (left as an exercise). The method call `super.putCD()` tells the runtime system to execute the `putCD` method from the superclass, i.e., from the class BigVic extends. That will be the `putCD` method defined in the Vic class (since the intermediate Looper superclass does not override that definition of `putCD()`).

Listing 4.7 Four methods in the BigVic class

```
public class BigVic extends Looper
{
   private int itsNumFilled;     // count this's filled slots
   private int itsNumEmpty;  // count this's non-filled slots


   public BigVic()                        // constructor
   {  // left as an exercise
   }  //======================

   /** Tell how many of the executor's slots are filled. */

   public int getNumFilled()
   {  return itsNumFilled;
   }  //======================

   /** Tell how many of the executor's slots are empty. */

   public int getNumEmpty()
   {  return itsNumEmpty;
   }  //======================

   /** Do the original putCD(), but also update the counters. */

   public void putCD()
   {  if ( ! seesCD() && stackHasCD())
      {  itsNumFilled++;
         itsNumEmpty--;
         super.putCD();
      }
   }  //======================

   /** Do the original takeCD(), but also update the counters. */

   public void takeCD()
   {  if (seesCD())
      {  itsNumFilled--;
         itsNumEmpty++;
         super.takeCD();
      }
   }  //======================
}
```

Similarly, `super.takeCD()` inside BigVic's `takeCD` method will execute the `takeCD` method from the original Vic class. In either case, the BigVic object also corrects the counts for the number of filled slots and empty slots. Note: `super.someMethod` is only allowed in a subclass of the class that contains `someMethod`.

Polymorphism

If you execute `sue = new BigVic()` and then execute `sue.fillSlots()`, that contains a call of `putCD()` (`fillSlots` is defined for Loopers in the earlier Listing 3.4, and every BigVic object is a Looper object as well). That call changes two counters, `sue.itsNumFilled` and `sue.itsNumEmpty`, in the BigVic object. But calls of `fillSlots()` with an ordinary Looper object do not change any counters. The command `putCD()` within the definition of `fillSlots` has different effects depending on the class of the object that is executing the command. So we say that the `putCD()` statement in Listing 3.4 is **polymorphic** (meaning it has more than one form).

Similarly, when a method executes `game = new GuessNumber()` and then executes `game.playOneGame()`, that causes a call of `shouldContinue()`. This is because `playOneGame` is defined for BasicGames in the earlier Listing 4.3, and every GuessNumber object is a BasicGame as well. That call tests `itsUsersNumber`. But calls of `shouldContinue()` with an ordinary BasicGame object test `itsUsersWord` instead. The command `shouldContinue()` within the definition of `playOneGame` has different effects depending on the class of the object that is executing the command. So we say that the `shouldContinue()` condition in Listing 4.3 is polymorphic. **Polymorphism** is the execution of polymorphic method calls.

Stickies

Think of polymorphism this way: Each object has a sticky-note on its side saying what class it belongs to. When the runtime system executes `this.shouldContinue()` for a GuessNumber object, it checks out the sticky-note on the side, sees that `this` is a GuessNumber object, and so executes the `shouldContinue` method defined in the GuessNumber class. If it found that the sticky-note said it was a BasicGame object, it would execute the `shouldContinue` method defined in the BasicGame class.

How does the sticky get there, you ask? The `super` call in the constructor puts the sticky on the object. The `super` call has to be the first statement in the constructor so that the rest of its statements can use `this` (implicitly or explicitly). (Note: In case you had not guessed by now, this is a metaphor. There really is no sticky-note in RAM, just an extra instance variable defined in the Object class and initialized by the `super` call.)

The equals method for Strings

The Object class defines a method named `equals`: `x.equals(y)` returns an answer of `true` or `false`, depending on whether `x` and `y` refer to exactly the same object. The String class overrides that definition with its own `equals` method: `x.equals(y)` for two String references returns `true` if the String objects contain exactly the same characters in the same order, even if they are two different String objects. In effect, it tests whether two boxes have the same contents, even if they are different boxes.

Caution It is almost always bad, and often uncompilable, to override a method in a superclass that does not have exactly the same method heading in all respects. For instance, Object's `equals` method has the heading `public boolean equals (Object o)`, so any `equals` method you define should also begin with `public boolean`.

Language elements
Two methods may have the same name if they are in different classes or have different signatures. The following kinds of method calls are allowed inside an instance method and will call MethodName in the superclass using the same executor:

```
super . MethodName ( )
super . MethodName ( ExpressionsSeparatedByCommas )
```

Exercise 4.30 Create a Liar subclass of the Person class in Listing 4.4: When you ask a Liar for its first name, half the time it says it is Darryl even if that is a lie.
Exercise 4.31 (harder) What changes would you make in Listing 4.7 to use an instance variable named `itsNumSlots` (the total number of slots) but not the instance variable `itsNumEmpty`, and still do the same things with the same method calls?
Exercise 4.32* Write out the signature of each method called in Listings 1.10 and 1.11.
Exercise 4.33* Write out the full Time constructor with the heading `public Time (int totalMinutes)`. Be sure to allow for negative inputs.
Exercise 4.34** Write out the BigVic constructor that Listing 4.7 needs.

4.8 The Rules Of Precedence For Operators

Sometimes you have to put parentheses around parts of an expression to have it mean what you want it to mean. For instance, as you learned in algebra class, `x + y * z` needs parentheses if you mean `(x + y) * z` but not if you mean `x + (y * z)`. This is because multiplication takes precedence over addition.

This section gives all the rules of precedence you need, if you choose to minimize the number of parentheses you use. This thorough discussion requires talking about a few things you will not see until the next two chapters. So just accept for now that it can be useful to put `(int)` or `(Time)` in front of an expression. You can use the name of any type of value in those parentheses, so the generic form is `(someType)`.

Accept also that you can put not only parentheses `( )` but also brackets `[ ]` around an expression to get something useful. Both of these are "grouping symbols." For this section only, we call a matched pair of grouping symbols plus their contents "groupers".

The **algebra rules of precedence** in Java are as follows:

1. For arithmetic expressions, multiply and divide operations are done first, then add and subtract. The % symbol counts as a divide operation. So `3+5 * 8` has the value 43, not 64.

2. The compiler works left-to-right in a sequence of multiply and divide operations. So `10 / 2*5` is 25, not 1.

3. The compiler also works left-to-right in a sequence of add and subtract operations. So `10 - 2+5` is 13, not 3. When one of the two added values is a String, the result is a string of characters, not a numeric value. So when you take into account the left-to-right evaluation, you see that the value of `3 + 4 + "x" + 3 + 4` is `"7x34"`.

4. You need to put parentheses around an expression formed with the addition, subtraction, multiplication, and division signs + - * / % if you negate it (as in `-(x + y)`) or you apply `(someType)` to it or you want to override the usual algebra rules of precedence.

The **general rules of precedence** in Java are as follows:

1. You never need to put parentheses around an expression that only involves words connected by dots and/or followed by groupers, `++` or `--`.

2. You never need to put parentheses around `! Whatever` or `- Whatever` if the latter is the negation operator (as in `y > -x`; we are not talking about subtraction here). You need to put parentheses just around the Whatever part if that part consists of an operator applied to two or more operands and the operator is not a dot or grouper.

3. You only need to put parentheses around `(someType) Whatever` when that expression is directly followed by a dot or grouper. You need to put parentheses just around the Whatever part if that part consists of an operator applied to two or more operands and the operator is not a dot or grouper.

4. You need to put parentheses around an expression formed with `||`, such as `Stuff || MoreStuff`, if you use `&&` to combine that or-expression with another value.

5. You should always put parentheses around an expression formed with `?:` except when it is to be assigned or returned (you will see `?:` expressions in Chapter Six).

6. You should also use parentheses liberally in some quite rare situations which are not used in this book: if you use shift or logical or bitwise operators, if you test for equality between two boolean expressions, or if you use the value of an assignment operation in a statement.

Caution You would not put a space in the middle of the word String or boolean and expect it to compile, would you? In Java, `++` and `<=` and `==` are all words, and the compiler will not like it if you put a space in the middle of one of those words. Also, the compiler does not understand coding containing multi-comparisons, as in `0 < x < y`. You must express this as `0 < x && x < y`.

1	2	3	4	5	6	7	8	9	10	11
( ) [ ] .	++ --	! (type) - +	* / %	+ -	< <= > >= instanceof	== !=	&&	||	?:	= += -= *= /= %=

Figure 4.4 Precedence of operators: The highest precedence is towards the left. The + and - at level 3 are not add and subtract; they apply to one operand.

Exercise 4.35 Neither of the following expressions will compile correctly. Add correcting parentheses to fix each one: (a) `! y + 2 < 3 && z.isTall()`
(b) `(Boss) person.getJob()`.
Exercise 4.36 (harder) There are eight possible assignments of `true` and `false` to the boolean variables `b`, `c`, and `d`. Which assignments give `(b || c) && d` a different value from `b || (c && d)`?
Exercise 4.37* Same question as the preceding exercise, but for the two expressions `!b && (c || !d)` and `!(b || !c && d)`.

Part B Enrichment And Reinforcement

4.9 Analysis And Design Example: The Game Of Nim

The game of Nim starts with a pile of stones, perhaps 20 to 40 in all. Two players take turns removing 1 to 5 stones from the pile (although the upper limit might be set at some other small number, such as 3 or 4). The player who takes the last stone wins the game.

For instance, if the pile starts with 23 stones, the first player might take 4, leaving 19, and then the second player could take 3, leaving 16. If the first player then takes 2, leaving 14, the second player could take 4, leaving 10. Then the first player might take 1, leaving 9, and the second player could take 3, leaving 6. Then the second player will easily win the game. The accompanying design block is a reasonable plan for this program.

STRUCTURED NATURAL LANGUAGE DESIGN for Nim
1. Choose at random an initial number of stones for the pile, 20 to 40.
2. Choose at random the maximum number to take each turn, 3 to 5.
3. Ask the user how many he/she wants to take for the first turn.
4. Repeat the following until the pile has no more than the maximum one may take...
    4a. Choose the number the computer takes and tell the user.
    4b. Ask the user how many he/she wants to take for the next turn.
5. Tell the user who won and why.

Object design for Nim

The BasicGame logic applies here, so Nim should be a subclass of BasicGame. That is, BasicGame objects can be "retrained" to perform acceptably the tasks that this situation requires. The `playManyGames` and `playOneGame` logic can stand without change, so you have five methods to develop.

A single game seems to have just two attributes, the number of stones currently in the pile and the maximum number of stones one may take on each turn. Store this information in two instance variables named `itsNumLeft` and `itsMaxToTake`. The `askUsersFirstChoice` method should initialize these two values, the former as a random number in the range from 20 to 40 and the latter as a random number in the range from 3 to 5. This implies that a Nim object also needs a random number generator. The Nim object should keep the generator around to use in deciding its next move, so the generator should be an instance variable rather than a local variable of one method. Once the `askUsersFirstChoice` method has chosen the initial values, it asks for the user's next choice. These declarations are in the top part of Listing 4.8 (see next page).

Implementing `askUsersFirstChoice` is a bit more complicated than the other methods. The general idea should be to ask for input, repeating as needed until the user supplies a permissible choice, then subtract that number from `itsNumLeft`. When you rework this basic logic in detail, you could come up with the logic in the accompanying design block, for which the coding is in the middle part of Listing 4.8.

STRUCTURED NATURAL LANGUAGE DESIGN for askUsersNextChoice
1. Do the following until the `choice` is in the range from 1 to `maxToTake`...
    1a. Get input from the user using showInputDialog (so it is a String or else null).
    1b. If the input contains one or more characters then...
        1ba. Convert the string of characters received into an integer `choice`.
2. Subtract `choice` from `itsNumLeft` to get the new value of `itsNumLeft`.

Listing 4.8 The Nim class of objects

```
import javax.swing.JOptionPane;

public class Nim extends BasicGame
{
   private java.util.Random randy = new java.util.Random();
   private int itsNumLeft;
   private int itsMaxToTake;


   public void askUsersFirstChoice()
   {  itsNumLeft = 20 + randy.nextInt (21);
      itsMaxToTake = 3 + randy.nextInt (3);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  int choice = 0;
      do
      {  String s = JOptionPane.showInputDialog
               (itsNumLeft + " left. Take 1 to " + itsMaxToTake);
         if (s != null && ! s.equals (""))
            choice = Integer.parseInt (s);
      }while (choice < 1 || choice > itsMaxToTake);
      itsNumLeft = itsNumLeft - choice;
   }  //======================

   public boolean shouldContinue()
   {  return itsNumLeft > itsMaxToTake;
   }  //======================

   public void showFinalStatus()
   {  if (itsNumLeft == 0)
         super.showFinalStatus();
      else
         JOptionPane.showMessageDialog (null, "I take "
                + itsNumLeft + " and so I win.");
   }  //======================

   public void showUpdatedStatus()
   {  int move = itsNumLeft % (itsMaxToTake + 1);
      if (move == 0)
         move = 1 + randy.nextInt (itsMaxToTake);
      itsNumLeft = itsNumLeft - move;
      JOptionPane.showMessageDialog (null, "I take " + move
                + ", leaving " + itsNumLeft);
   }  //======================
}
```

The shouldContinue and showFinalStatus methods

The `shouldContinue` method is the easiest to develop next: If there are no stones left, the game is over and the human player wins. Otherwise, if the computer opponent can take all remaining stones and win, it should do so, which also makes the game over. Otherwise the game should continue. You may conclude, therefore, that the game should continue if and only if `itsNumLeft` is greater than `itsMaxToTake`.

The logic for the `showFinalStatus` method is implicit in the preceding analysis. It is basically an if-statement: If there are no stones left, the program should announce that the user wins. This can be done by using `super` to call on the `showFinalStatus` method in the BasicGame superclass. But if there are a few stones left, it will not be more than `itsMaxToTake`, so the computer opponent announces that it takes the remaining stones and wins. See the middle part of Listing 4.8 for the coding of these two methods.

The showUpdatedStatus method

The hard part of the `showUpdatedStatus` method is to decide what move the computer player is to make. This is a problem of strategy rather than a programming problem. Extensive thought leads to the conclusion that the computer opponent can always win if it leaves a multiple of six stones each time (assuming the maximum allowed is five; the general formula is `itsMaxToTake+1`). So the computer's move will always be to calculate the remainder left after `itsNumLeft` is divided by `itsMaxToTake+1` and take that many stones.

However, the human player might have left a multiple of six stones (or whatever `itsMaxToTake+1` is). In that case, the computer opponent must choose some number of stones and hope that, on the next move, the human player does not leave a multiple of six (or whatever). A random choice from 1 to `itsMaxToTake` is appropriate here. The program then subtracts that many stones from `itsNumLeft` and announces the result.

Note how easily the preceding two paragraphs lead to the correct coding for `showUpdatedStatus`, as shown in the lower part of Listing 4.8. The most important principle for developing the correct logic for a method is to write it out in English first (or whatever natural language you are most comfortable with), normally with a structured organization. Read it over, make sure there are no errors in it, then write it down in Java.

Debugging The primary information you need for debugging a program is the intermediate values of key variables. Each variable in the Nim class is printed right after it is assigned a value with one exception. You can get that information by temporarily inserting the following as the last statement of `askUsersNextChoice`:

```
System.out.println (itsNumLeft + "=num; choice=" + choice);
```

Note on terminology If X extends Y, we often say that X "is derived" from Y, that X is a "child" of Y, and that Y is a "parent" of X. Since BasicGame is the parent class of Nim and GuessNumber, that makes those two "sibling" classes.

Exercise 4.38 Revise Listing 4.8 so that an illegal choice by the user is changed to be a guess of 1. Be sure to inform the user of this adjustment each time it happens.
Exercise 4.39 (harder) Revise Listing 4.8 so that the user gets to decide who goes first, once informed of `itsNumLeft` and `itsMaxToTake`.
Exercise 4.40* What happens if `itsMaxToTake` is 5, the human player leaves 6, the computer takes 4, and then the human player tries to take 3? Fix the inconsistency.
Exercise 4.41* Revise Listing 4.8 so that the rule is that a player can take up to half of the remaining stones, but always at least 1, and the size of the initial pile of stones is from 50 to 100. Have the computer take about a third of the remaining stones each time.
Exercise 4.42** For the game rules of the preceding exercise, give the computer a strategy that guarantees it wins if the human player does not make the one right choice every time.
Exercise 4.43** Revise Listing 4.8 so that the `showFinalStatus` method announces how many games so far the human player has won and how many the program has won.

4.10 Analysis And Design Example: The Game Of Mastermind

Suppose you decide to write a program to play the game of Mastermind. The program is to choose a three-digit whole number (000 to 999) and the human player is to guess the number. Each time the user announces a guess, the program is to tell whether all three digits are right. If the user's choice is not completely right, the program is to tell how many digits of the guess were in the right position. It is also to tell how many other digits of the guess were in the wrong position in the number. Then it has the user guess again.

Analysis and logic design

To make sure you have the concept right, you try it out with some sample data, e.g., if the program's chosen number is 123 and you guess 325, the program is to say you have one digit in the right place (the 2) and one digit in the wrong place (the 3). You realize that you are not sure how to count the number wrong when duplicate digits are involved, e.g., how many digits are wrong in the guess 225 if the program's chosen number is 123? So you check with your client and find out that the first 2 is considered to be in the wrong position. That is, the game should announce 1 in the right position and 1 in the wrong position. After several more trials, you are ready to work out the logic design. A reasonable plan is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for Mastermind

1. Choose a secret three-digit whole number.
2. Ask the user for his/her first guess.
3. Repeat the following until the current guess is completely right...
    3a. Tell the user how many digits are right and how many are wrong.
    3b. Ask the user for his/her next guess.
4. Tell the user he/she has finally gotten it right.

Object design and initialization

This logic has the same structure as the BasicGame logic, so you can make Mastermind a subclass of BasicGame. The inherited `playManyGames`, `playOneGame`, and `showFinalStatus` methods can be used unchanged.

The construction `game = new Mastermind()` should create the game-playing object and `askUsersFirstChoice` should choose a secret three-digit number at random. Since you will have to compare individual digits of the guess, it is probably more convenient to pick three separate one-digit numbers at random. So you could have the following initializations of instance variables (the digits are in the range 0 to 9 inclusive):

```
randy = new Random();
itsFirst = randy.nextInt (10);
itsSec   = randy.nextInt (10);
itsThird = randy.nextInt (10);
```

The user's input will be a three-digit integer. It will be easiest to store the three digits in three separate variables, so you can compare them with the three digits of the secret number. Since one instance method gets these three values and another instance method looks at them to see if they match, you must store the three digits of the user's guess in instance variables. These three variables could be named `usersFirst`, `usersSec`, and `usersThird`. The upper part of Listing 4.9 (see next page) declares the instance variables, the constructor, and the rather obvious `askUsersFirstChoice` method (which has most of its work done by the `askUsersNextChoice` method).

Listing 4.9 The Mastermind class of objects, one method postponed

```
import javax.swing.JOptionPane;
import java.util.Random;

public class Mastermind extends BasicGame
{
   private Random randy;
   private int itsFirst;    // 100's digit of secret number
   private int itsSec;      //  10's digit of secret number
   private int itsThird;    //   1's digit of secret number
   private int usersFirst;  // 100's digit of user's guess
   private int usersSec;    //  10's digit of user's guess
   private int usersThird;  //   1's digit of user's guess


   public Mastermind()
   {  super();
      randy = new Random();
   }  //======================

   public void askUsersFirstChoice()
   {  itsFirst = randy.nextInt (10);
      itsSec   = randy.nextInt (10);
      itsThird = randy.nextInt (10);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  String s;
      do
      {  s = JOptionPane.showInputDialog
                     ("Enter a three-digit integer:");
      }while (s == null || s.equals (""));
      int guess = Integer.parseInt (s);
      usersThird = guess % 10;
      usersFirst = guess / 100;
      usersSec = (guess / 10) % 10;
   }  //======================

   public boolean shouldContinue()
   {  return usersFirst != itsFirst || usersSec != itsSec
                                    || usersThird != itsThird;
   }  //======================
}
```

The askUsersNextChoice method

First you have to get a string of characters as input and convert it to an integer value stored in `guess` (except if the input string has no characters, you get another input). Now you need to separate out the digits. Taking the remainder from dividing `guess` by 10 clearly gives the last digit of `guess`. Taking the quotient from dividing `guess` by 100 clearly gives the first digit of `guess` (assuming the user has not made a mistake and entered more than 999). But how do you get the middle digit?

If you take the quotient from dividing `guess` by 10, you suppress the last digit, so the last digit of that quotient must be the next-to-last digit of the original `guess`. In other words, the computation `(guess / 10) % 10` gives the middle digit of `guess`. The coding for this `askUsersNextChoice` method is in the middle part of Listing 4.9.

The shouldContinue method

If each one of the user's digits matches the corresponding secret digit, the game is over. So if any one of the three user's digits is not the same as the corresponding secret digit, the game should continue. That gives the obvious coding in the bottom of Listing 4.9.

The showUpdatedStatus method

The `showUpdatedStatus` method has to calculate the number of user's digits in the right position and the number of user's digits that are in the wrong position. You can simply go through the user's digits one at a time and for each one, count it as right if it matches the corresponding secret digit, otherwise see if it matches either of the other two secret digits (in which case it counts as being in the wrong position).

The easiest way is to initialize two counters `numRight` and `numWrong` to 0 and increment the appropriate one when you see that a particular user's digit matches some secret digit. Then print a message telling the user the result. The coding is in Listing 4.10.

Listing 4.10 The Mastermind class of objects, part 2

```
public void showUpdatedStatus()     // in Mastermind
{  numRight = 0;
   numWrong = 0;

   if (usersFirst == itsFirst)
      numRight++;
   else if (usersFirst == itsSec || usersFirst == itsThird)
      numWrong++;

   if (usersSec == itsSec)
      numRight++;
   else if (usersSec == itsFirst || usersSec == itsThird)
      numWrong++;

   if (usersThird == itsThird)
      numRight++;
   else if (usersThird == itsSec || usersThird == itsFirst)
      numWrong++;

   JOptionPane.showMessageDialog (null, " You have "
         + numRight + " in the right place and "
         + numWrong + " in the wrong place.");
}  //=====================
```

Exercise 4.44 Under what circumstances can the user get one or more digits right if the user chooses a negative number for the guess? Which digits are they?
Exercise 4.45 Explain what happens when the user chooses a positive four-digit or longer number for the guess.
Exercise 4.46** Revise the Mastermind program so that, after each game played, it tells the user the number of games played to date and the average number of guesses required to get the right answer.
Exercise 4.47** Rewrite the Mastermind program so that `askUsersNextChoice` calculates `numRight` and `numWrong`. Have `shouldContinue` and `showUpdatedStatus` use those calculated values.

4.11 Using BlueJ With Its Debugger

You can download for free the BlueJ IDE (this acronym stands for "Integrated Development Environment"). BlueJ is an environment in which you can compile, execute, and analyze Java classes. It uses true Java, since you have to first install a recent official version of Java (from e.g. java.sun.com/products) in order to use BlueJ. The primary purposes of the BlueJ environment are to help you debug your programs and to let you manipulate objects one step at a time (execute a single method, see the result, execute another method, etc.).

Go to the URL www.bluej.org and click on the icon for downloading the latest version. Store it in a folder named bluej. You will then have a file with a name something like bluej-115.jar. Open the bluej folder and click on that jar file. The installer will ask you for the name of the folder where your current version of Java JDK is stored, which you should supply (e.g., jdk1.3.1_01). You can then enter the BlueJ environment by clicking on the icon specified in the installation instructions (probably bluej or bluej.bat).

Checking out an example

Click on the Project option, then click Open Project. Double-click on the examples sub-folder in the bluej folder. Choose one of the examples listed there to try out. We illustrate the process with the Hello example:

- Click on the Hello sub-folder of examples and then click `open`. This loads the Hello project into BlueJ. You will see a class box named Hello in a central **workarea**. It is cross-hatched to indicate that it is not yet compiled.
- Click on Compile. This compiles all the classes showing in the workarea. The cross-hatching disappears from the Hello class box to indicate this.
- Right-click the Hello class box. You will see a list of the methods in the Hello class that can be called without using an instance of the Hello class. There are but two: a constructor `new Hello()` and `public static void main(String[] args)`.
- Click on the main method. It allows you to enter the `args` parameter value, which you may ignore (we rarely use command-line arguments). Then click OK to execute the main method. A terminal window pops up with the output "Hello world", which is all this main method produces.
- Read the source code for the Hello class, which you can view by clicking on the "Open Editor" option within the Hello class box. The source code is shown below.
- Close the editor and click on the constructor `new Hello()`. You are asked for the name of the Hello object you are constructing; enter `sam` and click OK. You will see an object box (rounded corners, "sam:" at the top) to represent this Hello object.
- Right-click on the object box to see a list of the instance methods you can call for this Hello object. Click on the only one shown, namely, `public void go()`. That will execute `sam.go()`, which causes a second "Hello, world" to appear in the terminal window.

```
class Hello
{
    public void go()
    {   System.out.println("Hello, world");
    }

    public static void main(String[] args)
    {   Hello hi = new Hello();
        hi.go();
    }
}
```

In general, you can execute any class method or constructor by right-clicking on a class box and then clicking the method name. You can execute any instance method by right-clicking on an object box (which represents an instance of the class) and then clicking the method name. You can also inspect the current values of all class variables (described in Chapter Five) and instance variables by clicking on the Inspect option for an object box. Figure 4.5 shows what the BlueJ display looks like at this point.

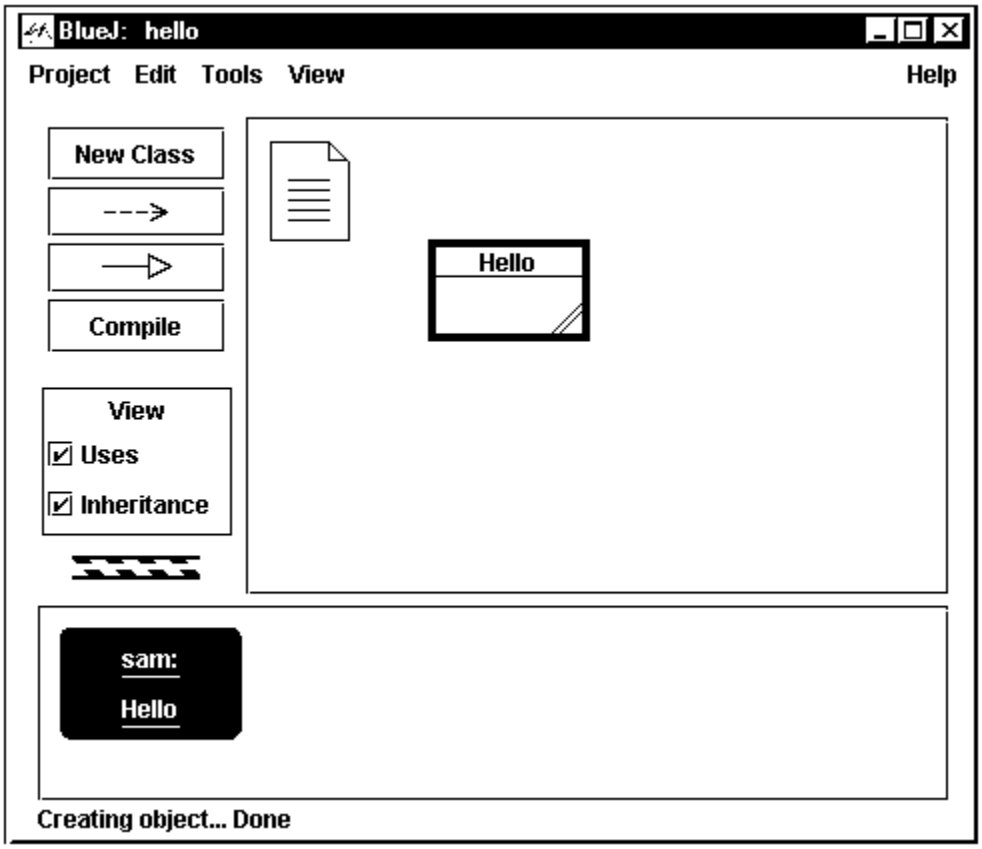


Figure 4.5 The BlueJ display for the Hello class with an instance of Hello

Note If your attempt to run BlueJ causes the message "out of environment space" on a Windows system, it may help to click on the Properties icon of your DOS window, select the Memory option, and set the initial environment to 4096.

Adding a new class to the project

Click New Class and give it the name Adder. A class box with that name will appear in the workarea. Right-click it and choose Open Editor to create code for it. You will see a minimal coding for the class, with an instance variable `x` and a method named `sampleMethod`. Replace the body of that method to have the following coding:

```
public int sampleMethod (int y)
{   x = x + y;
    return x;
}
```

Close the editor window, compile the Adder class (click Compile), then construct an Adder object named `adder_1` (right-click the Adder class box and click on `new Adder()`). Right-click on `adder_1` and then click Inspect to see that the current value of the instance variable `x` is 0. Figure 4.6 shows what the display looks like now.

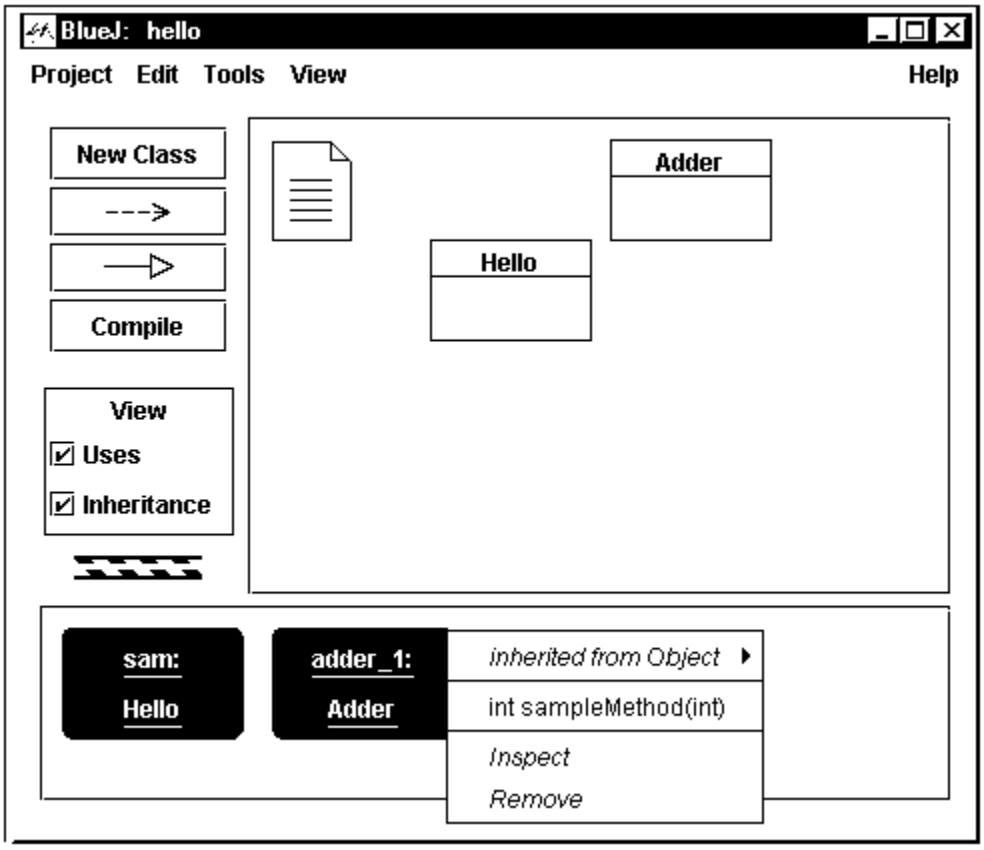


Figure 4.6 The BlueJ display with the Adder class and an Adder instance

Right-click on `adder_1` and then click `int sampleMethod(int y)` to execute that method. You have to enter the value of the parameter; type 5 and then click OK. BlueJ will then tell you that the result of the method call `sampleMethod(5)` is 5.

Repeat with parameter value 3; BlueJ will tell that the result of the method call `sampleMethod(3)` is 8. Inspect `adder_1` to see that the instance variable `x` now has the value 8. Make sure you understand why these things happen before you go on. Play around with it all a bit more.

Using BlueJ with existing classes

Put the following classes from this chapter in a folder named `games`: GameApp, BasicGame, and GuessNumber. Each should be in a file named like Whatever.java. Click on the Project menu choice and select Open Non-BlueJ Project. Navigate to the folder that contains the games folder, click on `games`, then click on Open in BlueJ. All three classes will appear in the workarea. Click on Compile to compile them all.

You have created a BlueJ project unit to manage these three classes, so you can now create instances of the class and try them out. For instance, once you create an instance of GuessNumber, you can right-click on it, choose `void playOneGame()` from the list of methods inherited from BasicGame, and thereby play a game. During the game, right-click the GuessNumber object from time to time and click Inspect so that you can see the current values of the instance variables. Note: If you use Turtles from Chapter One, and the first command you give a Turtle is to sleep 5 seconds (5000 ms), that gives you time to click on the Turtle's graphics window to bring it forward so you can see what it draws. Or you can simply move the bluej window out of the way (this works for Vics too).

Debugging

The basic debugging process is to set a **breakpoint** in your coding where you want to look closely at what is happening. You do this by choosing Open Editor for some class and then clicking in the far left column, next to some statement in the coding. A stop sign icon should appear there. That indicates the breakpoint you have set.

Now if you execute the coding, execution pauses when it comes to that stop sign (just before executing that marked statement) and a debugger window appears. This window shows the current values of all variables local to the method. You may also inspect the current state of all the objects involved in the execution.

You may click the Step option to advance the execution by one statement at a time. Inspect the values of variables at each point of interest. When you have seen all you want to see, click on the stop sign to remove it, then click Continue in the debugger window. Execution will then continue normally.

Figure 4.7 shows how the debugger window looks. You may also bring up the debugger window on a program while it is running by clicking on the turning barber-pole icon. The Call Sequence window shows what method calls are currently active. In this example, you are currently executing the `askUsersNextChoice` method, which was called from the `askUsersFirstChoice` method, which was called from the `playOneGame` method, which was called by clicking on the method call in the GuessNumber object box.

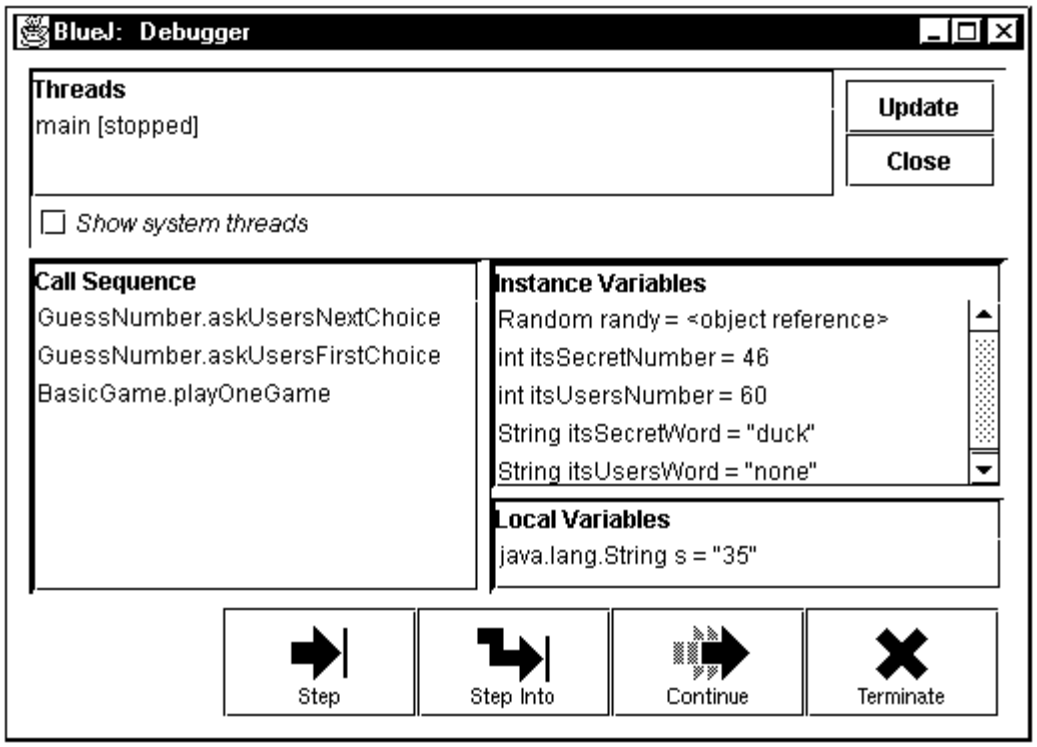


Figure 4.7 A debugger window for BlueJ

This has been a very short introduction to how to use BlueJ. You will understand it much better if you read the complete 30-page tutorial available at www.bluej.org (click on Documentation) and try it out with the example programs and with your own programs.

4.12 A First Look At Event-Handling: Buttons And Textfields

Chapter Ten goes into details on event-handling in Java. But you may wish to use buttons and textfields well before you are ready to study the subject at some length. This section shows you how to do this using an author-defined class named BabyFrame (the name reminds you that you are not directly using the standard event-handling methods).

Listing 4.11 illustrates the use of this class, along with the Sun standard JLabel library class from the javax.swing package. This listing is all you need to have software that accepts a temperature in Fahrenheit degrees and translates it to Centigrade degrees, as long as you first compile the BabyFrame class provided on this book's website.

A JLabel displays text to the user. The JLabel class has a constructor that gives the text that initially appears on the label (lines 1 and 4). You may change this text by using the `setText` instance method of the JLabel class (line 10). The label added in line 4 is not used elsewhere in the coding, so it is not given a name. The label added in line 6 is used in line 10, so it is given the name `itsLabel` in line 1.

Listing 4.11 An application program illustrating the use of textfields and labels

```
import javax.swing.JLabel;

   /** Calculate the centigrade equivalent of any Fahrenheit
    *  temperature that the user enters in a textfield.  */

public class CentigradeFrame extends BabyFrame
{
   private Jlabel itsLabel = new JLabel ("none");           //1


   public CentigradeFrame (String title)
   {  super (title);                                        //2
      this.setSize (250, 110);    // width, height          //3
      this.add (new JLabel ("What is the Fahrenheit temp?"));//4
      this.add (new BabyField_1 (15));                      //5
      this.add (itsLabel);                                  //6
      this.setVisible (true);                               //7
   }  //=====================


   /** React to ENTER in textfield #1. */

   public void fieldAction_1 (BabyField_1 source)
   {  int temp = Integer.parseInt (source.getText());       //8
      int centi = (temp - 32) * 5 / 9;                      //9
      itsLabel.setText ("In centigrade that is " + centi);  //10
   }  //=====================
}
//#############################################################


class CentigradeMain
{
   public static void main (String[ ] args)
   {  new CentigradeFrame ("Simple event handler");        //11
   }  //=====================
}
```

Textfields in a BabyFrame

The CentigradeFrame class is a subclass of BabyFrame. A BabyFrame object recognizes the BabyField_1 class, which defines a class of textfield objects where the user can give input. The constructor for a BabyField_1 requires you to say how many characters wide the textfield will be (line 5). When you have a BabyField_1 kind of object, you should have a method with exactly the heading shown above line 8. That method will be executed each time the user presses the ENTER key while typing within that textfield.

If your program needs more than one textfield for data entry, you also have available the BabyField_2 class, the BabyField_3 class, and the BabyField_4 class by inheritance from the BabyFrame class. You should have a fieldAction_2 method to react to the ENTER key in the BabyField_2 textfield, and similarly for numbers 3 and 4. In this limited software, you cannot have more than four textfields within your frame.

When the ENTER key is pressed and the fieldAction_1 method is executed, it is passed the textfield as a parameter. This is so you can retrieve the text that the user typed with the `getText` instance method (line 8). You may also change the text that is within the text field using `source.setText (someString)`, but that is not needed in this case.

BabyFrame methods

A method creates a BabyFrame object by calling a constructor (line 11) whose first statement must be `super(someString)` (line 2). This creates a rectangular window with `someString` as the title at the top of the frame. Figure 4.8 shows how this window looks for the coding in Listing 4.11. On the left you see how it looks when the frame is created. On the right you see how it looks after the user enters 77. Note that the text shown on `itsLabel` (at the bottom) has changed.

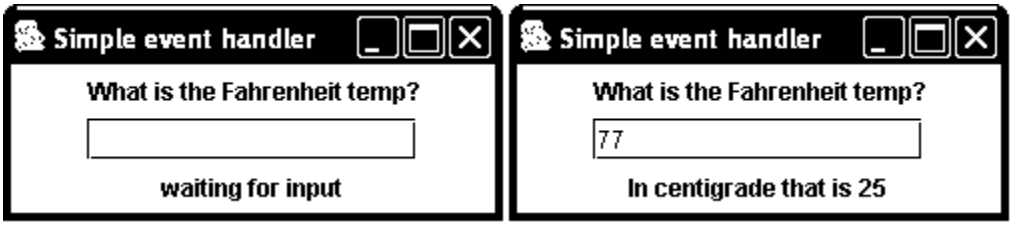


Figure 4.8 Display for CentigradeFrame, before and after entering 77

The constructor for a BabyFrame object should set the width and height of the frame in pixels (line 3). The statement `setSize(250, 110)` makes the frame be 250 pixels wide and 110 pixels tall. The constructor should then add any components you want to have (lines 4, 5, and 6). In this case, it adds a label, then a textfield, then a label. The order in which they are added determines the order in which they appear, top to bottom. The last statement in the constructor should always be `setVisible(true)`, because otherwise the frame will not show the components that have been added.

If the frame is wide enough, several components will be put on one line, left-to-right, before going on to fill in the next line. For instance, in this case, a width of 500 would have put the first label and the text field on the same line with `itsLabel` below them. A width of 750 would have put all three components on the same line. This "book-reading order" is called FlowLayout.

Buttons and TextAreas in a BabyFrame

A BabyFrame object also recognizes the BabyButton_1 class, which defines a class of button objects that the user can click. The constructor for a BabyButton_1 requires you to say what text is to appear on the button. When you have a BabyButton_1 kind of object, you should have a method with exactly the following heading. That method will be executed each time the user clicks that button:

```
public void buttonAction_1()
```

If your program needs more than one button, you also have available the BabyButton_2 class, the BabyButton_3 class, and so on up through the BabyField_6 class by inheritance from the BabyFrame class. You should have a fieldAction_2 method to react to a click of the BabyButton_2, and similarly for numbers 3, 4, 5, and 6. In this limited software, you cannot have more than six buttons within your frame.

A BabyFrame object can also create a BabyTextArea. This is a rectangular area that lets you print many lines on the screen at once. By contrast, a JLabel's text can only be one line. The constructor tells, first, the number of rows of text you want to be able to see at one time, and second, the number of characters wide the textarea is. For instance, the following allows for 8 lines with at least 20 characters per line:

```
itsOutputArea = new BabyTextArea (8, 20);
```

When you want to add a line of output to a BabyTextArea, you call on the `say` instance method, as in the following:

```
itsOutputArea.say ("The overall result is " + x);
```

Example: revision of the game of Nim

Listing 14.12 (see next page) is a revision of the game of Nim in the earlier Listing 4.8. It does almost exactly the same thing except that it has more user-friendly GUI components. Instead of frequently flashing little windows on the screen, it leaves the frame on the screen with all output to date shown in the textarea. The textarea even has scrollbars so the user can see everything that has happened so far.

You should closely compare this coding with that in Listing 4.8 to see the similarities as well as the differences. This NimFrame class requires copies of the last three methods of that Listing, but with the JOptionPane statements replaced by statements calling on `itsOutput.say (someString)`. Note particularly that Listing 4.8 has three loops (including the inherited `playManyGames`) but Listing 4.12 needs none.

Exercise 4.48* Revise Listing 4.11 so that it asks for a number of inches and gives the equivalent number of centimeters, assuming 2.52 centimeters per inch.
Exercise 4.49* Revise Listing 4.12 so that it prints the message "Illegal input; try again" when a number is entered in the textfield that does not lead to calling `shouldContinue`.
Exercise 4.50* What happens if the user enters a number to take after the game of Nim is over? Revise the coding so that nothing happens in such a case.
Exercise 4.51** Revise Listing 4.12 to add another button labeled "Repeat same game". When the user clicks this button, the game starts again with the same values for `itsNumLeft` and `itsMaxToTake` as the previous game.
Exercise 4.52** Revise the simpler GuessNumber game (Listing 4.6) analogously to Listing 4.12.

Listing 4.12 The game of Nim, revised for event-handling

```
public class NimFrame extends BabyFrame  // based on Listing 4.8
{
   private java.util.Random randy = new java.util.Random();
   private int itsNumLeft;
   private int itsMaxToTake;
   private javax.swing.JLabel itsPrompt
                = new javax.swing.JLabel ("Take 1 to 00");
   private BabyTextArea itsOutput = new BabyTextArea (6, 42);

   public NimFrame (String title)
   {  super (title);
      setSize (500, 250);
      add (new BabyButton_1 ("start new game"));
      add (itsPrompt);
      add (new BabyField_1 (4));
      add (itsOutput);
      setVisible (true);  // otherwise the frame will not appear
      buttonAction_1(); // start the software by starting a game
   }  //======================

   /** Initialize 2 variables and wait for user's first choice. */

   public void buttonAction_1()
   {  itsNumLeft = 20 + randy.nextInt (21);
      itsMaxToTake = 3 + randy.nextInt (3);
      itsOutput.say ("We start with " + itsNumLeft);
      itsPrompt.setText ("Take 1 to " + itsMaxToTake);
   }  //======================

   /** This method executes when the user types a number in the
       textfield. If it is legal, the computer makes its move. */

   public void fieldAction_1 (BabyField_1 source)
   {  int choice = 0;
      String s = source.getText();
      if (s != null && ! s.equals (""))
         choice = Integer.parseInt (s);
      if (choice >= 1 && choice <= itsMaxToTake)
      {  itsNumLeft = itsNumLeft - choice;
         if (shouldContinue())
            showUpdatedStatus();
         else
            showFinalStatus();
      }
   }  //======================

   // showUpdatedStatus, showFinalStatus, and shouldContinue
   // are the same as in Listing 4.8, except "itsOutput.say("
   // replaces "JOptionPane.showMessageDialog(null,".
}
//#################################################################

class NimMain
{
   public static void main (String[ ] args)
   {  new NimFrame ("The Game of Nim");
   }  //======================
}
```

4.13 Review Of Chapter Four

Listing 4.3, Listing 4.5, and Listing 4.7 illustrate almost all Java language features introduced in this chapter. This review includes the preceding Interlude.

About the Java language:

- `package X;` as the top line of a file puts the classes in that file in **package** X.
- The **import directive** `import javax.swing.JOptionPane` goes in a compilable file before any class definitions if you use JOptionPane in some class in the file. Similarly, you must have `import java.util.Random` if you use Random. However, you may instead give the full name (`java.util.Random` or `javax.swing.JOptionPane`) at each mention of the class.
- If you want to import several classes from the same package, e.g., from `java.util`, use the import directive `import java.util.*;`. Classes from the `java.lang` package (e.g. System, String, Object, Integer) do not need a directive.
- A non-public class can be compiled in a file with a public class. This should only be done for classes that contain only a main method. Imports go at the top of the file.
- You may assign the value **null** to any object variable. It signals the absence of any actual object reference stored in the variable.
- The result of evaluating a plus sign between two strings of characters or between a string and a number is the first string of characters followed directly by the second. If one is a number, it is converted to a decimal numeral (a string of characters) before the **concatenation**. The **empty String** `""` is the String value whose length is zero.
- The **newline** character `\n` within the quotes of a **String literal** starts a new line.
- You may declare a variable in a class outside of any method. Then each object of the class has its own value for the variable if it is an **instance variable** (which, as you will see in the next chapter, requires that it not be declared using `static`).
- If one class extends another, then each public method and public variable of the superclass is indirectly in the subclass. If a class does not explicitly extend any class, then it makes the **default extension** of the **Object** class.
- Within the body of an instance method or constructor, the use of an instance variable without saying which object it belongs to signals that it belongs to the executor of the instance method or to the object being constructed, respectively. The use of `this` inside a constructor is a reference to the object that is being constructed.
- A class must have one or more **constructors**, to say what happens when an instance of the class is created. If you do not explicitly define a constructor, the compiler provides the **default constructor** that has no parameters and no statements other than `super()`.
- If the declaration of an instance variable assigns it a value, that takes effect when the constructor is called.
- For any **int** variable `x`, `x++` **increments** `x` and `x--` **decrements** `x`.
- The six comparison operators are `>` `<` `>=` `<=` `==` `!=` .
- The five operators for int values are `+` `-` `*` `/` `%` (the % is for remainder).
- The **signature** of a method is its name followed by parentheses containing the types of its parameters (not their names). Different methods can have the same name if they have a different parameter pattern (signature) or are in different classes. This is **overloading of a method name** if they are in the same class.
- If a subclass defines an instance method with the same signature as an instance method in its superclass, that is **overriding of a method definition**. Suppose the signature is `doStuff(int)` Then `super.doStuff(3)` calls on the coding in the superclass method.
- One of the two **rules of precedence** people get wrong most is that one should put parentheses around the whole `(someType) Whatever` expression when followed by a bracket `[` or dot. The other one is that `!` takes precedence over `&&` which in turn takes precedence over `||`. Section 4.8 says more on precedence.

- Figure 4.8 describes the remaining new language features. The Initializer part of a for-statement can assign a value to the **loop control variable** (the only variable mentioned in the Condition of the for-statement whose value changes during execution of the loop). The Initializer part may be the declaration and assignment of the loop control variable if that variable is not used outside of the for-statement.

`public ClassName (`*`ParameterList`*`)`
`{ StatementGroup`
`}`
	declaration of constructor method. The parameters are optional. The first statement should be super(...)

`super (`*`ArgumentList`*`);`	**statement** calling the superclass's constructor. Only allowed as the first statement in the constructor

`for (`*`Initializer`*` ; Condition ;`
`Update)`
`Statement`
	statement that executes the Initializer part first, then repeats test-Condition-do-Statement-do-Update, quitting when the Condition is false

`do`
`{ Statement`
`}while (Condition);`
	statement that repeats do-Statement-test-Condition, quitting when the Condition is false

`private Type VariableName` `= Expression;`	**declaration** of instance variable outside any method. Initializing is optional
`super.MethodName (`*`ArgumentList`*`);`	**statement** that calls the method of that name in the superclass

Figure 4.8 Declarations and statements added in Chapter Four

About some key Sun standard library methods:

- `System.exit(0)` terminates an execution of a program immediately. A program that has a graphical user interface should explicitly execute this statement when it is done, because otherwise some computer systems lock up. The 0 in the parentheses indicates a normal termination.
- `System.out.println(someString)` displays the value in the parentheses in the terminal window. The argument may also be a numeric value.
- `someObject.equals(otherObject)` tests whether the two objects are identical.
- `someObject.toString()` is a String form of the object reference.
- `JOptionPane.showMessageDialog(null, messageString)` displays the `messageString` on the screen until the user clicks OK or closes the window.
- `JOptionPane.showInputDialog(promptString)` displays the `promptString` and waits for input. It returns the String input, except it returns null if the user clicks the Cancel button.
- `JOptionPane.showConfirmDialog(null, messageString)` displays the `messageString` on the screen with three buttons labeled Yes, No, and Cancel, until the user clicks one of the buttons or closes the window. It returns the int value `JOptionPane.YES_OPTION` if the user clicked the Yes button.
- `new Random()` constructs a Random object.
- `someRandom.nextInt(limitInt)` returns an int value chosen randomly with equal likelihood from zero up to but not including `limitInt`.
- `Integer.parseInt(someString)` returns the int value that the String value represents, assuming that it is not badly-formed with letters or other defects.
- General note: In these review sections, we normally specify Sun standard library methods as **generic method calls**, so you see how they are called: The executor of an instance method is named "`some`" followed by the executor's class (e.g., `someRandom`), and the parameters are specified the same way (e.g., `someString`) except that parameters may be numeric or boolean or the like (e.g. `someInt`) and "`some`" may be replaced by a more descriptive word (e.g., `limitInt`).

Other vocabulary to remember:

- A **natural constructor** is a constructor that assigns the values of its parameters to all or almost all of the instance variables of the class.
- **Encapsulation** means primarily preventing outside classes from changing instance variables directly (i.e., without calling a method in the class). It includes declaring instance variables as `private`. The purpose is abstraction: sending a message to an object to have a complex task done rather than doing the task directly.
- A local variable or parameter **shadows** an instance variable when they have the same name. The name refers to the local variable instead of to the instance variable.
- A **binary operator** is a symbol that combines two values to obtain a new value. `&&` and `+` are binary operators. But `!` is a **unary operator**: You apply it to only one value to obtain a new value.
- Software is **reusable** if it is designed to be used in several different programming situations. A **driver program** is an application program whose only purpose is to test the methods of a class thoroughly. It is rarely needed when you use BlueJ.
- A method call is **polymorphic** if it calls an instance method and the executor could be from any of two or more classes that have differing definitions of the method being called. **Polymorphism** is the execution of polymorphic method calls.

Answers to Selected Exercises

4.3
```
public boolean shouldContinue()
{   if (itsSecretWord.equals (itsUsersWord))
        return false;  // since the game should NOT continue
    itsSecretWord = "goose";  // so this will apply on all future calls
    return true;
}
```

4.4
Add this declaration outside of any method: String itsMark = "";
Add this statement to askUsersFirstChoice: itsMark = "x";
Add this statement to askUsersNextChoice: itsMark = itsMark + "x";
Add this statement to showFinalStatus: itsMark = "";
Replace the statement in the body of shouldContinue by this statement:
```
return ! (itsSecretWord.equals (itsUsersWord) || itsMark.equals ("xxxxx"));
```

4.7
```
public SmartTurtle()
{   super();
}
```

4.8
```
public RedTurtle()
{   super();
    switchTo (RED);
    move (-180, 0);
}
```

4.9
```
public class NamedTurtle extends Turtle
{   private String itsName;
    public NamedTurtle (String name)
    {   super();
        itsName = name;
    }
    public String getName()
    {   return itsName;
    }
}
```

4.11 "x" + (x + y) is "x28" and ("x" + x) + y is "x235".

4.12
```
public int getAge (int currentYear)
{   if (currentYear < itsBirthYear)
        return 0;
    else
        return currentYear - itsBirthYear;
}
```

4.13 t4.toString() is "135", but it should be "1305".

4.14 Insert the following at the beginning of the toString() method body:
```
if (itsMin < 10)
    {  if (itsHour < 10)      return ("0" + itsHour) + "0" + itsMin;
       else                   return ("" + itsHour) + "0" + itsMin;
    }
else
```

4.15
```
public Time add (Time that)
{   Time valueToReturn = new Time (this.itsHour + that.itsHour, this.itsMin + that.itsMin);
    if (valueToReturn.itsMin >= 60)
    {   valueToReturn.itsMin = valueToReturn.itsMin - 60;
        valueToReturn.itsHour++;
    }
    valueToReturn.itsHour = valueToReturn.itsHour % 24;
    return valueToReturn;
}
```

4.21 -5 + randy.nextInt (11) and 2 * (15 + randy.nextInt (11)), assuming Random randy.

4.22 Replace the first statement of askUsersFirstChoice by:
itsSecretNumber = 200 + randy.nextInt (101);
Replace the string literal in askUsersChoice by: "Guess my number from 200 to 300:".

4.23 Guess 50. If you are wrong, you know there are at most 50 possible right answers.
Guess 25 or 75, depending on whether the answer was "too high" or "too low".
If you are wrong, you know there are at most 25 possible right answers. Continue guessing in the middle of the range of possible right answers, reducing the number of possible right answers to 12, 6, 3, and 1 in that order. The seventh guess will then get it right.

4.24 Replace the statement in the body of shouldContinue by:
return itsUsersNumber < itsSecretNumber - 2 || itsUsersNumber > itsSecretNumber + 2;
Add a showFinalStatus method to GuessNumber with this statement in its body:

```
if (itsUsersNumber != itsSecretNumber)
    JOptionPane.showMessageDialog (null, "close enough; you win");
else
    JOptionPane.showMessageDialog (null, "That was right.  \nCongratulations.");
```

4.25
```
public void findSmallest()
{   JOptionPane.showMessageDialog (null, "Finding the smallest of a group of integers");
    String prompt = "Enter an integer.  Click Cancel when done";
    String s = JOptionPane.showInputDialog (prompt);
    if (s != null)
    {   int smallestSoFar = Integer.parseInt (s);
        s = JOptionPane.showInputDialog (prompt);
        while (s != null)
        {   int input = Integer.parseInt (s);
            if (smallestSoFar > input)
                smallestSoFar = input;  // replace the value in smallestSoFar by the one in input
            s = JOptionPane.showInputDialog (prompt);
        }
        JOptionPane.showMessageDialog (null, "The smallest was " + smallestSoFar);
    }
}
```

4.30
```
public class Liar extends Person
{   public Liar (String first, String last)
    {   super (first, last);
    }
    public String getFirstName()
    {   if (new java.util.Random().nextInt (2) == 1)    //   == 0 would work as well
            return "Darryl";
        else
            return super.getFirstName();  // since you cannot mention itsFirstName here
    }
}
```

4.31 Replace itsNumEmpty by itsNumSlots in the declaration of the instance variables.
The statement in the body of getNumEmpty should be: return itsNumSlots - itsNumFilled.
Omit the statements that mention itsNumEmpty in putCD and takeCD.

4.35 (a) could be either ! (y + 2 < 3) && z.isTall() or ! (y + 2 < 3 && z.isTall())
(b) is ((Boss) person).getJob()

4.36 In the two cases when b is true and d is false and c is either true or false, (b || c) && d is false but b || (c && d) is true. In the other six cases they match.

4.38 Delete "do {" and the line beginning "}while". Put the following just before the last statement:

```
if (choice < 1 || choice > itsMaxToTake)
{   choice = 1;
    JOptionPane.showMessageDialog ("Illegal; changed to 1.");
}
```

4.39 Replace the last statement of askUsersFirstChoice by the following:

```
if (JOptionPane.showConfirmDialog (null, "Taking up to " + itsMaxToTake + " and starting with"
                + itsNumLeft + ".  Will you go first?") == JOptionPane.YES_OPTION)
    askUsersNextChoice();
```

4.44 A negative guess will still get a digit right if the guess's digit is 0 and the secret digit is also 0. This can happen at any or all of the three digits.

4.45 usersFirst will be wrong, since it will be 10 or more. The other two digits will be right or wrong according to whether they would have been without the extra digits.