Chapter 7

Patterns of Repetition: Iteration and Recursion

Repeating an action over and over is called repetition. *This Chapter introduces techniques and applications of repetition in programming. The Chapter is organized into three distinct parts:*

- *The first part, Sections 7.1-7.8, introduce the* while- *and* for-statements *for writing standard and classic patterns of repetition, called* iteration.
- *The second part, Section 7.9, applies repetition in a case study of designing and building an animation*
- *The third part, Sections 7.10-7.12, promotes another form of repetition,* recursive method invocation*, as a technique for solving problems in terms of repeatedly solving simpler subproblems.*

The first part of the Chapter is essential reading; the second is strongly recommended; and the third can be omitted on first reading, if desired.

After studying the Chapter, the reader should be able to identify when a programming problem should be solved by means of repetitive computation, apply the appropriate repetitive pattern, and write the pattern in Java.

7.1 Repetition

Some jobs must be solved by repeating some step over and over:

- When replacing a flat tire with a spare, you must "place a lug nut at the end of each bolt, and as long as the nut is loose, rotate it clockwise over and over until it is finally secure against the wheel."
- When searching for your favorite show on the television, you must, "while you haven't yet found your show, press the channel button repeatedly."

Both of these "algorithms" rely on *repetition* to achieve their goals.

Computers became popular partly because a computer program will unfailingly repeat a tedious task over and over. For example, perhaps you must know the decimal values of the fractions (*reciprocals*) 1/2, 1/3, 1/4, and so on, up to 1/20. A painful way of programming these calculations is manually repeating a division statement 19 times:

```
public static void main(String[] args)
{ System.out.println("1/2 = " + (1.0/2));
  System.out.println("1/3 = " + (1.0/3));
  System.out.println("1/4 = " + (1.0/4));
    ...
  System.out.println("1/20 = " + (1.0/20));
}
```

The computer readily computes the reciprocals, but we should find a simpler way to request the computations than the "copy and paste" coding technique above. A better solution is built with a control structure called a *while loop*.

7.2 While Loops

At the beginning of the previous section, we saw two algorithms for tightening a car's wheel and finding a television show. We can rewrite the two algorithms in "while-loop style," where we begin the algorithm with the word, "while," followed by a true-false test, followed by an action to perform as long as the test is true:

- `while ( lug nut still loose ) { rotate nut clockwise one turn; }`
- `while ( favorite TV show not found ) { press the channel button; }`

The phrase within the parentheses is the "test expression"; when the test expression is true, the statement within the set braces is performed once, then *the entire statement repeats* ("loops"). Eventually (we hope!) the test expression becomes false, and the while-loop ceases.

We can write while-loops in Java; the format is

```
while ( TEST ) { BODY }
```

where `TEST` is a `boolean`-typed expression and `BODY` is a sequence of (zero or more) statements.

With Java's while-loop, we can rewrite the method that prints the reciprocals from 1/2 to 1/20. The algorithm goes like this:

1. Set variable `denominator = 2`. The variable remembers which reciprocal to print next.
2. Do this: `while` (`denominator <= 20`) { print `1.0/denominator` *and* add one to `denominator`. }

Step 1 initializes the variable that acts as the *loop's counter*; Step 2 prints the reciprocals, one by one, as the loop's counter increases by ones. Here is how the algorithm is written in Java:

```
public static void main(String[] args)
{ int denominator = 2;
  while ( denominator <= 20 )
        { System.out.println("1/" + denominator + " = " + (1.0 / denominator));
          denominator = denominator + 1;
        }
}
```

The while-loop prints the fractional representations of 1/2, 1/3, ..., 1/20 and stops.

Here is a more precise explanation of how the while-loop, `while ( TEST ) { BODY }`, executes:

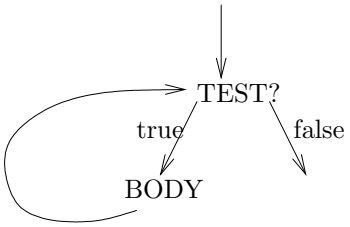
1. The `TEST` expression is computed.

2. If `TEST` computes to `true`, then the `BODY` executes and *the process repeats, restarting at Step 1.*

3. If `TEST` computes to `false`, then the `BODY` is ignored, and the loop terminates.

Repetition by means of a while-loop is called *iteration*, and one repetition of the loop's body is called *an iteration*; when the loop repeats its body over and over, we say that it *iterates*.

Here is the flowchart representation of a a while-loop—it is a graph with a cycle:



and indeed, this is the origin of the term, "loop."

Loops fall into two categories: *definite iteration*, where the number of the loop's iterations is known the moment the loop is started, and *indefinite iteration*, where it is not. Also, it is possible for a loop's iterations to be unbounded (that is, the loop never stops). We study all three behaviors in the examples in the sections that follow.

Exercises

1. What will these while-loops print?

   (a)
   ```
   String s = "";
   while ( s.length() != 5 )
         { s = s + 'a';
           System.out.println(s);
         }
   ```
   (Recall that the `length` operation returns the length of a string; for example,
   ```
   String s = "abcde";
   int i = s.length();
   ```
   assigns 5 to `i`.)

   (b)
   ```
   int countdown = 10;
   while ( countdown != 0 )
         { System.out.println(i);
           countdown = countdown - 1;
         }
   ```

(c)
```
int countdown = 5;
int countup = -1;
while ( countdown > countup )
      { countdown = countdown - 1;
        System.out.println(countdown + " " + countup);
        countup = countup + 1;
      }
```

(d)
```
while ( false )
      { System.out.println("false"); }
```

(e)
```
String s = "";
while ( s.length() < 6 )
      { s = s + "a";
        if ( (s.length() % 2) == 1 )
           { System.out.println(s); }
      }
```

2. Write a while-loop to print the characters, `'a'` to `'z'`, one per line. (Hint: recall that Java allows you to initialize a variable as `char c = 'a'` and to do arithmetic with it, e.g., `c = c + 1`.)

3. Write a while-loop to print all the divisors of 1000 that fall within the range 2 to 50. (Recall that an integer, `i`, *divides* another integer, `j`, if `j % i` equals 0.) (Hint: insert an if-statement in the body of the loop.)

7.3 Definite Iteration

A loop performs *definite iteration* when the number of the loop's iterations is known the moment the loop is started. The example in the previous section displayed definition iteration, since it was clear at the outset that the loop would repeat exactly 19 times. Definite-iteration loops arise often, and it is worth studying their development.

Say that we must construct a method that computes the average score of a student's exams. The method first asks the student to type the number of exams and then the method reads the exam scores, one by one, totals them, and computes the average score.

There is a numerical pattern to computing an average score; if the student has taken `N` exams, then the average score is calculated by this formula:

```
average = (exam1 + exam2 + ... + examN) / N
```

The ellipsis in the formula suggests that we should repeatedly sum the scores of the exams until all `N` scores are totalled; then we do the division. Here is an algorithm that uses several variables, along with a while-loop, to sum the exams:

1. Assume that variable `how_many` holds the quantity of test scores to be read.

2. Declare a variable `total_points = 0` to remember the total of all the exams, and declare variable, `count = 0`, to remember how many exam scores have been read already.

3. While `count` not yet equals `how_many`, do the following:

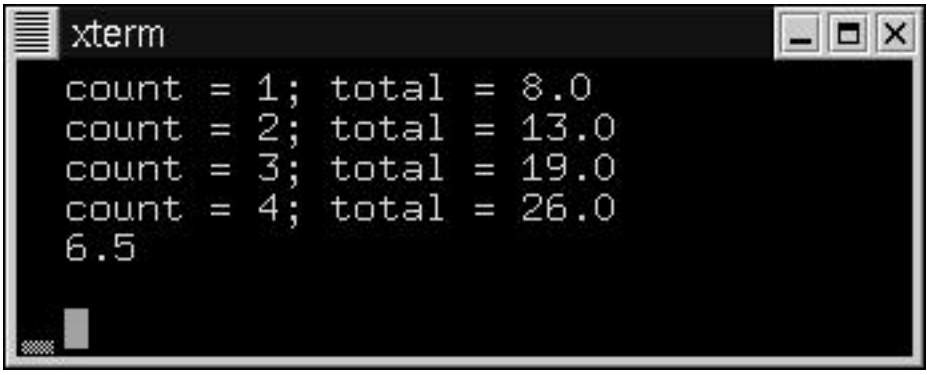
   - Ask the user for the next exam score and add it to `total_points`.
   - Increment `count` by 1.

4. Calculate the average score as `total_points / how_many`.

The above algorithm can be refined into this Java-like coding:

```
int how_many = HOW MANY SCORES TO READ;
double total_points = 0.0;
int count = 0;
while ( count != how_many )
      { int score = READ INTEGER FROM THE USER;
        total_points = total_points + score;
        count = count + 1;
      }
return (total_points / how_many);
```

The algorithm is presented as a Java method, `computeAverage`, in Figure 1. At each iteration of the loop, the method generates a dialog to read the next exam score, and a `println` statement helps us see the loop's progress. Say that the user wishes to average the scores, `8, 5, 6`, and `7`. The invocation, `System.out.println(computeAverage(4))`, produces this trace information:



By reading the printed trace information, we see that at the beginning (and the end) of each iteration, the value in `total_points` indeed equals the sum of all the exam scores read so far, that is,

```
total_points == exam_1 + exam_2 + ... + exam_count
```

Figure 7.1: while-loop to compute average score

```
import javax.swing.*;

/** computeAverage computes the average of test scores submitted by a user
  * @param how_many - the quantity of test scores to read; must be nonnegative
  * @return the average of the test scores */
public double computeAverage(int how_many)
{ double total_points = 0.0;  // the total of all test scores
  int count = 0;  // the quantity of tests read so far
  while ( count != how_many )
        // at each iteration: total_points == exam_1 + exam_2 + ... + exam_count
        { // ask the user for the next exam score:
          String input = JOptionPane.showInputDialog("Type next exam score:");
          int score = new Integer(input).intValue();
          // add it to the total:
          total_points = total_points + score;
          count = count + 1;
          // print a summary of the progress:
          System.out.println("count = " + count + "; total = " + total_points);
        }
        // at conclusion: total_points == exam_1 + exam_2 + ... + exam_how_many
  return (total_points / how_many);
}
```

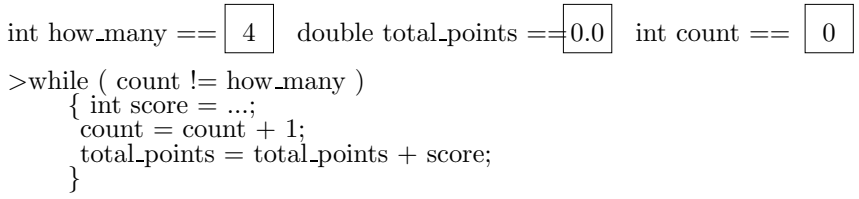
This crucial fact is called the loop's *invariant property* or *invariant*, for short. The invariant explains what the loop has accomplished with its iterations so far. Because of their value in helping us understand a loop's secrets, we will state invariant properties for the loops we study.

Because of the loop's invariant property, we know that when the loop stops with `count` equal to `how_many`, then it must be the case that `total_points` holds all the total points for all exams. From here, it is a small step to compute the average.

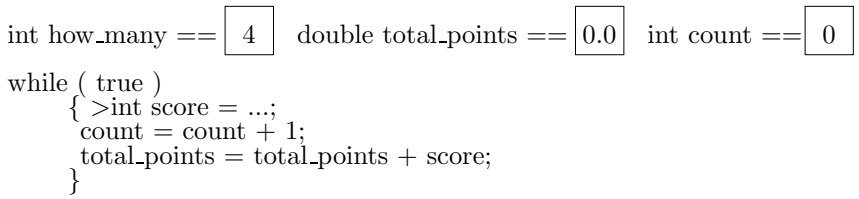
In the example, variables `count` and `total_points` play crucial roles in remembering the loop's progress—the former acts as the loop's counter, and the latter remembers a running total.

It is absolutely crucial to understand the details of loop execution, so we examine part of the execution trace of the example. Say that the method has been invoked as `computeAverage(4)`; once variables `count` and `total_points` are initialized, we have

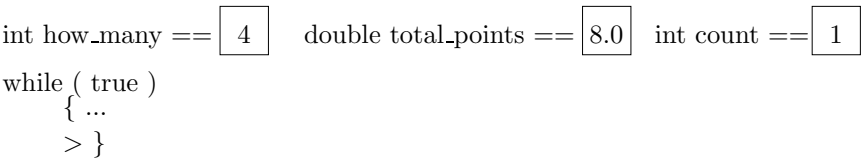
the following configuration:



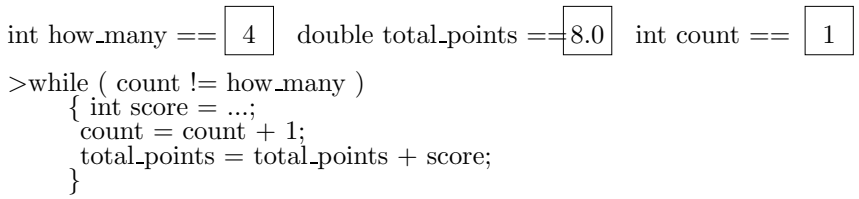
The loop's test evaluates to true, so execution moves into the body:



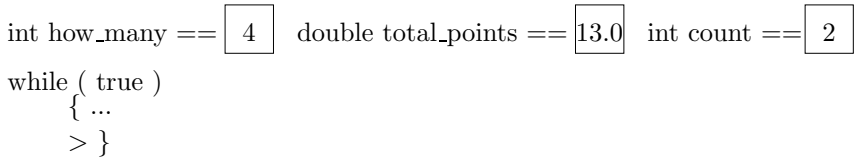
The user types `8` as the first exam score; the loop's body revises the variables' values:



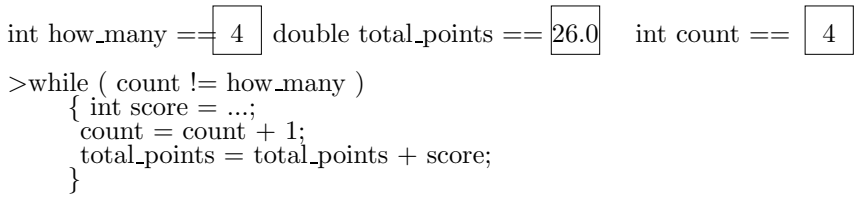
At this point, the while-loop "restarts"—the control marker, `>`, returns to the beginning of the loop, and the process repeats. Of course, the variables *retain their updated values*:



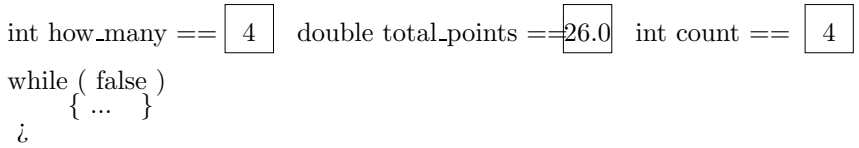
Again, the test is evaluated and the result is `true`, so the body of the loop is entered for yet another repetition, and the second score, 5, is read:



The loop repeats twice more, reading the last two scores, and we reach the configuration where `count`'s cell holds 4:



The test evaluates to `false`, and the loop is abandoned:



This loop is an example of definite iteration, because once the loop is started, the number of iterations is completely decided; here, it is `how_many` iterations.

Definite iteration loops almost always use

- a loop counter that remembers how many iterations are completed,
- a test expression that examines the loop counter to see if all the iterations are completed,
- a statement that increments the loop counter to record the iteration.

For a loop counter, `count`, the pattern of definite iteration often looks like this:

```
int count = INITIAL VALUE;
while ( TEST ON count )
      { EXECUTE LOOP BODY;
        INCREMENT count;
      }
```

We have seen this pattern used twice already.

Occasionally, `count` is incremented at the beginning of the loop body:

```
int count = INITIAL VALUE;
while ( TEST ON count )
      { INCREMENT count;
        EXECUTE LOOP BODY;
      }
```

Exercises

1. Implement an application that computes upon exam averages:

   (a) First, place the `computeAverage` method in a new class you write, called `class ExamStatistics`. Next, write a `main` method whose algorithm goes like this:

      i. construct a new `ExamStatistics` object;

      ii. construct a dialog that asks the user to enter a positive number for the number of exams

      iii. invoke the `computeAverage` method in the `ExamStatistics` object

      iv. construct a dialog that shows the result returned by `computeAverage`.

   (b) Modify `computeAverage` so that if its argument is a nonpositive integer, the method shows a dialog that announces an error and returns 0.0 as its answer.

   (c) Next, write a method, `computeMaxScore`:

   ```
   /** computeMaxScore reads a sequence test scores submitted by a user and
     * returns the highest score read
     * @param how_many - the quantity of test scores to read; must be nonnegative
     * @return the maximum test score */
   public double computeMaxScore(int how_many)
   ```

   Add this method to `class ExamStatistics` and modify the `main` method you just wrote to invoke `computeMaxScore` instead.

   (d) Write this method and include it within `class ExamStatistics`:

   ```
   /** computeBetterAverage computes the average of test scores submitted
     *  by a user _but discards_ the lowest score in the computation.
     * @param how_many - the quantity of test scores to read; must be > 1
     * @return the average of the best (how_many - 1) test scores */
   ```

2. Write an execution trace for this example:

   ```
   int t = 4;
   int count = 2;
   while ( count <= 4 )
         { t = t * 2;
           count = count + 1;
         }
   ```

3. Use the pattern for definite iteration to write a loop that displays this output, all on one line: `88 77 66 55 44 33 22 11`

4. Write a `main` method that does the following:

   (a) uses a loop to ask the user to type four words, one word at a time, e.g.,

   ```
   I
   like
   my
   dog
   ```

   (b) shows a dialog that displays the words listed in reverse order on one line, e.g.,

   ```
   dog my like I
   ```

5. Many standard mathematical definitions are computed with definite-iteration loops. For each of the definitions that follow, write a method that computes the definition:

   (a) The *summation* of a nonnegative integer, `i`, is defined as follows:

   ```
   summation(i) = 0 + 1 + 2 + ... + i
   ```

   For example, `summation(4)` is `0 + 1 + 2 + 3 + 4 = 10`. So, write a method that computes summation whose header line reads like this:

   ```
   public int summation(int i)
   ```

   (b) The *iterated product* of two nonnegative integers, `a` and `b`, goes

   ```
   product(a, b) =  a * (a+1) * (a+2) * ... * b
   ```

   (Note: if `b > a` holds true, then define `product(a, b) = 1`.) For example, `product(3, 6)` is `3 * 4 * 5 * 6 = 360`.

   (c) A famous variation on iterated product is `factorial`; for a nonnegative integer, `m`, its factorial, `m!`, is defined as follows:

   ```
   0! = 1
   n! = 1 * 2 * ... * n,  for positive  n
   ```

   For example, `5!` is `1 * 1 * 2 * 3 * 4 * 5 = 120`. (Important: Because the values of `m!` grow large quickly as `m` grows, use the Java data type `long` ("long integer") instead of `int` for the argument and answer of this function.)

   (d) If you enjoy using sines and cosines, then use the `factorial` method in the previous exercise to implement these classic methods:

i.
```
/** sine  calculates the sine value of its argument, using the formula
  *   sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!) + ... - (x^n/19!)
  * @param x - the value, in radians, whose sine is desired
  *   (i.e., sine(0)=0, sine(pi/2)=1, sine(pi)=0, sine(3pi/2)=-1, etc.)
  * @return the sine as calculated by the formula */
public double sine(double x)
```
(Note: use `Math.pow(a,b)` to compute a^b.) Compare the answers your method produces to the ones produced by the method, `Math.sin(...)`.

ii.
```
/** cosine  calculates the cosine value of its parameter, using the formula
  *  cosin(x) = 1 - (x^2/2!) + (x^4/4!) - (x^6/6!) + ... - (x^20/20!)
  * @param x - the value, in radians, whose cosine is desired
  * @return the cosine as calculated by the formula */
public double cosine(double x)
```

7.3.1 Definite-Iteration Example: Painting a Bulls-Eye

The definite-iteration pattern for loops can help us to draw interesting graphical patterns. As an example, perhaps we want to paint a "bulls-eye" pattern with `n` alternating red and white rings, where `n`'s value is given by the program's user. When `n` is 7, we see:

Assuming that the `size` of the bulls-eye is also set by the user, we can use the definite iteration pattern to paint each stripe, one at a time, from the outermost to the innermost. Three variables will be needed to control the painting:

- `count`, which remembers how many circles have been drawn;
- `color`, which remembers the color of the next circle to paint;
- `diameter`, which remembers the diameter of the next circle to paint.

Why do we require these three variables? Obviously, to paint one ring, we must know the ring's diameter and color. If we wish to paint multiple rings, we must remember how many rings we have painted already, so that we know when it is time to stop.

The three variables are used in the painting algorithm:

1. Set `count` equal to `0`, set `color` equal to `red`, and set `diameter` equal to the bulls-eye's `size`.

2. While `count` not yet equals `rings` (the total number of rings desired), do the following:

   - In the center of the graphics window, paint a filled circle of the appropriate `diameter` with the current `color`.
   - Revise all variables: increment `count` by one, make the `diameter` smaller (so that the next ring is painted smaller than this one), and reset the `color` (if it's `red`, make it white; if it's `white`, make it `red`).

These are the basic steps, although the details for calculating the rings' exact positions are omitted for the moment.

Perhaps we write the algorithm as a method, so that the method can be used at will to paint bulls-eyes at whatever position, number of rings, and diameter that we desire: Figure 2 displays the Java method that is parameterized on these arguments. The method in Figure 2 is used as a helper method in the class that displays the graphics window—see Figure 3. To use `class BullsEyeWriter`, we would construct a new object such as

```
new BullsEyeWriter(7, 140)
```

which draws a 7-ring bulls-eye of diameter 140 pixels.

Exercises

1. Construct this object:

   ```
   new BullsEyeWriter(21, 200)
   ```

   Explain why it is important that the bulls-eye's circles are painted from the largest to the smallest; explain why the innermost ring (a circle, actually) has a width that is almost twice as large as all the other rings. (Hint: insert a `System.out.println(diameter)` statement into the `paintBullsEye` method to monitor the drawing of the rings.)

2. Write this Java method:

Figure 7.2: method that paints a bulls-eye

```
/** paintBullsEye paints a bulls-eye
    * @param x_position - of the upper left corner of the bulls-eye
    * @param y_position - of the upper left corner of the bulls-eye
    * @param rings - the number of rings in the bulls-eye
    * @size - the bulls-eye's diameter
    * @param g - the graphics pen */
public void paintBullsEye(int x_position, int y_position,
                          int rings, int size, Graphics g)
{ int count = 0;                  // no rings painted just yet
  int diameter = size;            // diameter of next ring to paint
  int ring_width = size / rings;  // set width for each ring
  Color color = Color.red;
  while ( count != rings )
        // invariant: have painted  count  rings so far
        { // calculate upper left corner of ring to paint, centering the
          //  the ring within the bulls-eye by dividing by 2:
          int new_x_position = x_position + ((ring_width * count)/2);
          int new_y_position = y_position + ((ring_width * count)/2);
          // paint the ring:
          g.setColor(color);
          g.fillOval(new_x_position, new_y_position, diameter, diameter);
          // increment variables:
          count = count + 1;
          diameter = diameter - ring_width;
          if ( color == Color.red )
               { color = Color.white; }
          else { color = Color.red; }
        }
}
```

Figure 7.3: a panel that displays a bulls-eye

```
import javax.swing.*;
import java.awt.*;
/** BullsEyeWriter paints a bulls-eye on a panel */
public class BullsEyeWriter extends JPanel
{ private int rings;  // how many rings appear in the bulls-eye
  private int size;   // the size of the completed bulls-eye
  private int panel_width;  // width of the graphics panel
  private int offset = 20;  // where to start painting the bulls-eye

  /** Constructor BullsEyeWriter constructs the panel and frames it.
    * @param number_of_rings - how many rings in the bulls-eye
    * @param total_size - the diameter of the bulls-eye  */
  public BullsEyeWriter(int number_of_rings, int total_size)
  { rings = number_of_rings;
    size = total_size;
    panel_width = size + (2 * offset);
    // construct frame for this panel:
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("Bulls-Eye");
    my_frame.setSize(panel_width, panel_width);
    my_frame.setVisible(true);
  }

  /** paintComponent  draws the bulls-eye
    * @param g - the graphics pen that does the drawing */
  public void paintComponent(Graphics g)
  { g.setColor(Color.yellow); // paint background yellow:
    g.fillRect(0, 0, panel_width, panel_width);
    paintBullsEye(offset, offset, rings, size, g);
  }

  ... paintBullsEye appears here ...
}
```

```
/** paintSquares paints n squares across the left-to-right diagonal
  *  of a graphics window
    * @param x_position - of the upper left corner of the first square
    * @param y_position - of the upper left corner of the first square
    * @param n - the number of squares to paint
    * @size - the width of each square
    * @param g - the graphics pen */
private void paintsquares(int x_position, int y_position,
                          int n, int size, Graphics g)
```

3. Figure 9 of Chapter 5 contains a helper method, `paintAnEgg`, that paints an egg on a graphics window. Use this method to write a method, `paintBullsEyeOfEggs`, that paints a "bulls-eye" of `n` centered eggs in a graphics window.

7.4 Nontermination

Recall the `computeAverage` method in Figure 1, which read a sequence of exam scores, summed them, and computed their average. What happens when we invoke the method with a negative integer, e.g., `computeAverage(-1)`? Indeed, the invocation requests exam scores forever, summing the scores without ceasing, and we will see an indefinite produced,

```
count = 1; total = 8
count = 2; total = 13
count = 3; total = 19
count = 4; total = 26
count = 5; total = 30
  ...
```

assuming that the user is patient enough to submit a never-ending sequence of scores!

The loop inside `computeAverage` iterates indefinitely because its test expression is always true. Such behavior is called *nontermination*, or more crudely, *infinite looping* or just "looping." A nonterminating loop prevents the remaining statements in the program from executing, and in this case, the method from computing the average and returning an answer.

Although the method's header comment tells us to supply only nonnegative parameters to `computeAverage`, the method might defend itself with a conditional statement that checks the parameter's value:

```
/** computeAverage computes the average of test scores submitted by a user
  * @param how_many - the quantity of test scores to read; must be nonnegative
  * @return the average of the test scores
  * @throw RuntimeException, if  how_many  is negative  */
public double computeAverage(int how_many)
```

```
{ if ( how_many <= 0 )
     { throw new RuntimeException("computeAverage error: negative quantity"); }
  double total_points = 0.0;  // the total of all test scores
  int count = 0;  // the quantity of tests read so far
  while ( count != how_many )
        {  ... see Figure 1 for the loop's body ... }
  return (total_points / how_many);
}
```

The initial conditional statement throws an exception to prevent looping.

Often, nontermination is an unwanted behavior, and unfortunately there is no mechanical technique that can verify that a given loop must terminate, but the section titled "Loop Termination," included as an optional section at the end of the Chapter, presents a technique that helps one prove loop termination in many cases.

If one of your Java applications appears to have infinite looping, you can terminate it: When using an IDE, select the *Stop* or *Stop Program* button; when using the JDK, press the *control* and *c* keys simultaneously.

Finally, we should note that a while-loop's test can sometimes be written cleverly so that looping becomes impossible. Consider again the loop in Figure 1, and say that we rewrite the loop's test so that the Figure reads as follows:

```
public double modifiedComputeAverage(int how_many)
{ double total_points = 0.0;
  int count = 0;
  while ( count < how_many )
        { String input = JOptionPane.showInputDialog("Type next exam score:");
          int score = new Integer(input).intValue();
          total_points = total_points + score;
          count = count + 1;
          System.out.println("count = " + count + "; total = " + total_points);
        }
  return (total_points / how_many);
}
```

Now, if `how_many` receives a nonpositive value, then the loop's test fails immediately, and the loop executes zero iterations.

But the simple alteration is imperfect, because it allows the method to continue its execution with an improper value for `how_many` and compute an erroneous result: If `how_many` holds a negative number, then the method computes that the exam average is 0.0, which is nonsensical, and if `how_many` holds zero, the result is even more surprising—try `modifiedComputeAverage(0)` and see.

Perhaps looping is unwanted, but *under no conditions do we want a method that returns a wrong answer*, either. For this reason, you should take care when altering a while-loop's test to "ensure" termination; the alteration might cause a wrong answer to be computed, travel to another part of the program, and do serious damage.

Exercises

1. Write this method, which is is designed to execute forever:

```
/** printReciprocals displays the decimal values of the fractions,
  *  1/2, 1/3, 1/4, ..., one at a time:  When invoked,
  *  it shows a message dialog with the information,  "1/2 = 0.5",
  *   and when the user presses OK, it shows another message dialog
  *   that says, "1/3 = 0.3333333333", and when the user presses OK,
  *   it shows another message dialog, "1/4 = 0.25", and so on....  */
public void printReciprocals()
```

   Now, write an application that invokes it. Test the application for as long as you have the patience to do so.

2. Reconsider the `paintBullsEye` method in Figure 2. What happens if the method is asked to paint zero rings? A negative number of rings? Revise the method so that it will always terminate.

   Study the invariant property for the loop in the `computeAverage` method in Figure 1. When the loop in that method concludes, we know that `total_points == exam_1 + exam_2 + ...  + exam_how_many` holds true.

   Now, review `modifiedComputeAverage`. Is the loop invariant the same as that in Figure 1? Can we conclude the same result when the loop terminates as we did in Figure 1? What *can* we conclude about the value of `total_points` when the loop in `modifiedComputeAverage` terminates?

7.5 Indefinite Iteration: Input Processing

Many programs are designed to interact with a user indefinitely; for example, the bank-account manager application in Chapter 6 processed as many account transactions as its user chose to enter. The appplication did this by sending a message to restart itself after processing each input request, but we see in this Section that a while-loop is a simpler technique for repetitive input processing.

We can build on the exam-average method in Figure 1 to illustrate the technique. Say that we modify the method so that it does not know how many exam scores it must read. Instead, the user enters exam scores, one by one, into input dialogs until the user terminates the process by pressing the `Cancel` button. The method handles this behavior with a loop that terminates when `Cancel` is pressed.

A first attempt at the revised method's algorithm might go,

1. while (the user has not yet pressed `Cancel`), do the following:

   - Generate a dialog to read the next exam score;

- If the user pressed `Cancel`, then note this *and do no further computation within the loop*;
- else the user entered an exam score, so add it to the total and add one to the count of exams read.

2. Compute the average of the exams.

To write the loop's test expression, we will use a boolean variable, `processing`, to remember whether the user has pressed the `Cancel` button, signalling the desire to quit. This extra variable gives us a clever way of writing the loop:

```
boolean processing = true;
while ( processing )
      { generate a dialog to read the next exam score;
        if ( the user pressed Cancel )
             { processing = false; }  // time to terminate the loop!
        else { add the score to the total and increment the count; }
      }
```

Figure 4 displays the modified method from Figure 1.

There is a standard way to process a user's input with a while-loop: A sequence of *input transactions* are submitted, one at a time, and each input transaction is processed by one iteration of the loop. The loop terminates when there are no more input transactions. Here is the pattern:

```
boolean processing = true;
while ( processing )
      { READ AN INPUT TRANSACTION;
        if ( THE TRANSACTION INDICATES THAT THE LOOP SHOULD STOP )
             { processing = false; }
        else { PROCESS THE TRANSACTION; }
      }
```

This pattern was used in Figure 4 and can be profitably used for most input-processing applications. The loop is an example of *indefinite iteration*, because when the loop starts, it is not decided how many iterations will occur until termination. Note that indefinite iteration is different from nontermination—assuming that there are finitely many input transactions, the loop *will* terminate!

Exercises

1. Explain what this method does:

```
public static void main(String[] args)
{ boolean processing = true;
```

Figure 7.4: processing input transactions

```
import javax.swing.*;

/** computeAverage computes the average of test scores submitted by a user.
  * The user terminates the submissions by pressing Cancel.
  * @return the average of the test scores
  * @throw RuntimeException if the user presses Cancel immediately */
public double computeAverage()
{ double total_points = 0.0;  // the total of all test scores
  int count = 0;  // the quantity of tests read so far
  boolean processing = true;
  while ( processing )
        // at each iteration: total_points == exam_1 + exam_2 + ... + exam_count
        { String input = JOptionPane.showInputDialog
                          ("Type next exam score (or press Cancel to quit):");
          if ( input == null )  // was Cancel pressed?
               { processing = false; }  // time to terminate the loop
          else { int score = new Integer(input).intValue();
                 total_points = total_points + score;
                 count = count + 1;
               }
        }
  if ( count == 0 )  // did the user Cancel immediately?
     { throw new RuntimeException("computeAverage error: no input supplied"); }

  return (total_points / count);
}
```

```
    int total = 0;
    while ( processing )
          { String s = JOptionPane.showInputDialog("Type an int:");
            int i = new Integer(s);
            if ( i < 0 )
                 { processing = false; }
            else { total = total + i; }
          }
    JOptionPane.showMessageDialog(null, total);
  }
```

2. Write an application that invokes the method in Figure 4 to compute an average of some exam scores. The application displays the average in a message dialog.

3. Write an application that reads as many lines of text as a user chooses to type.

The user types the lines, one at a time, into input dialogs. When the user types presses `Cancel` *or* when the user presses just the *Enter* key by itself (with no text typed), the program prints in the command window all the lines the user has typed and halts. (Hint: You can save complete lines of text in a string variable as follows:

```
String s = "";
  ...
s = s + a_line_of_text + "\n";
```

Recall that the `\n` is the *newline* character.)

4. Revise the bank-account manager of Chapter 6 so that it uses a while-loop to process its input transactions. (Hint: Revise `processTransactions` in `class AccountController` in Figure 16, Chapter 6.)

7.5.1 Indefinite Iteration: Searching

In the real world, searching for a missing object is an indefinite activity, because we do not know when we will find the object. Programs can also go "searching"—for example, a program might search a computerized telephone directory for a person's telephone number. A small but good example of computerized searching is finding a specific character in a string. Searches use an important pattern of loop, so we develop the character-search example in detail.

Recall these two useful operations on strings from Table 5, Chapter 3:

- The `length` operation returns the length of a string; for example,

  ```
  String s = "abcde";
  int i = s.length();
  ```

  assigns 5 to `i`. The length of an empty string, `""`, is 0.

- We use the `charAt` operation to extract a character from a string:

  ```
  String s = "abcde";
  char c = s.charAt(3);
  ```

  assigns `'d'` to `c`. (Recall that a string's characters are indexed by 0, 1, 2, and so on.)

Now we consider how to search a string, `s`, for a character, `c`: We examine the string's characters, one by one from left to right, until we find an occurrence of `c` *or* we reach the end of `s` (meaning `c` is not found). We use an integer variable, `index`, to remember which character of `s` we should examine next:

1. Set `index` to 0.
2. While (`c` is not yet found, and there are still unexamined characters within `s`), do the following:
   (a) If `s.charAt(index)` is `c`, then we have found the character at position `index` and can terminate the search.
   (b) Otherwise, increment `index`.

The while-loop suggested by the algorithm looks like this:

```
int index = 0;
boolean found = false;  // remembers whether  c  has been found in  s
while ( !found  &&  index < s.length() )
        { if ( s.charAt(index) == c )
               { found = true; }
          else { index = index + 1; }
        }
```

The loop's test consists of two boolean expressions, combined by conjunction (`&&`), and the loop terminates when either of the conjuncts becomes false.

This algorithm is realized in Figure 5.

The loop's invariant property tells us how the loop operates: As long as `found` is `false`, `c` is not found in the searched prefix of `s`; when `found` becomes `true`, the search has succeeded.

The loop might terminate one of two ways:

- `!found` becomes `false`, that is, variable `found` has value `true`. By Clause (1) of the invariant, we know that `c` is found at position `index` in `s`.
- `!found` stays `true`, but `index < s.length()` becomes `false`. By Clause (2) of the invariant, we know that `c` is not in the entire length of string `s`.

These facts are crucial to returning the correct answer at the end of the function; study the above until you understand it thoroughly.

A string is a kind of "collection" or "set" of characters. In the general case, one searches a set of items with the following pattern of while-loop:

```
boolean item_found = false;
DETERMINE THE FIRST ‘‘ITEM’’ TO EXAMINE FROM THE ‘‘SET’’;
while ( !item_found  &&  ITEMS REMAIN IN THE SET TO SEARCH )
      { EXAMINE AN ITEM;
        if ( THE ITEM IS THE DESIRED ONE )
             { item_found = true; }
        else { DETERMINE THE NEXT ITEM TO EXAMINE FROM THE SET; }
      }
```

Figure 7.5: searching for a character in a string

```
/** findChar locates the leftmost occurrence of a character in a string.
  * @param c - the character to be found
  * @param s - the string to be searched
  * @return the index of the leftmost occurrence of  c  in  s;
  *  return -1 if  c  does not occur in  s  */
public int findChar(char c, String s)
{ boolean found = false;  // remembers if  c  has been found in  s
  int index = 0;          // where to look within  s  for  c
  while ( !found  &&  index < s.length() )
        // invariant:
        // (1) found == false  means  c  is not any of chars 0..(index-1) in  s
        // (2) found == true  means  c  is  s.charAt(index)
        { if ( s.charAt(index) == c )
               { found = true; }
          else { index = index + 1; }
        }
  if ( !found )  // did the loop fail to find  c  in all of  s?
     { index = -1; }
  return index;
}
```

The loop can terminate in two possible ways:

- `!item_found` evaluates to `false`. This means the search succeeded, and the desired item is the last `ITEM` examined.

- `ITEMS REMAIN` evaluates to `false`. This means the search examined the entire set and failed to locate the desired item.

This pattern was used to write the method in Figure 5.

Here is another use of the searching pattern—determining whether an integer is a prime number. Recall that an integer 2 or larger is a *prime* if it is divisible (with no remainder) by only itself and 1. For example, 3, 13, and 31 are primes, but 4 and 21 are not.

To determine whether an integer `n` is prime, we try dividing `n` by each of the numbers in the set, {2, 3, 4, ..., `n`/2 }, to try to find a number that divides into `n` with remainder zero. (There is no need to try integers larger than `n`/2, of course.) If our search for a divisor fails, then `n` is a prime.

The method's algorithm is based on the searching pattern:

```
boolean item_found = false;
current = n / 2;   // start searching here for possible integer divisors
```

Figure 7.6: method for prime detection

```
/** isPrime examines an integer > 1 to see if it is a prime.
  * @param n - the integer > 1
  * @return  1, if the integer is a prime;
  *  return the largest divisor of the integer, if it is not a prime;
  * @throw RuntimeException, if the argument is invalid, that is, < 2.  */
public int isPrime(int n)
{ int answer;
  if ( n < 2 )
       { throw new RuntimeException("isPrime error: invalid argument " + n ); }
  else { boolean item_found = false;
         int current = n / 2;  // start search for possible integer divisor
         while ( !item_found  &&  current > 1 )
               // invariant:
               // (1) item_found == false  means  n  is not divisible
               //     by any of  n/2, (n/2)-1, ...down to... current+1
               // (2) item_found == true  means  current  divides  n
               { if ( n % current  ==  0 )
                      { item_found = true; }      // found a divisor
                 else { current = current - 1; }  // try again
               }
         if ( item_found )
              { answer = current; }  // current is the largest divisor of  n
         else { answer = 1; }        // n  is a prime
       }
  return answer;
}
```

```
                  // and count downwards towards 2
while ( !item_found  &&  current > 1 )
      { if ( current divides into n with remainder 0 )
             { item_found = true; }      // current  is a divisor of  n
        else { current = current - 1; }  // select another possible divisor
      }
```

The search through the set, {n/2, (n/2) - 1, ..., 3, 2}, terminates if we find a divisor that produces a remainder of 0 or if we search the entire set and fail. Figure 6 shows the completed method.

Exercises

1. Modify method `findChar` in Figure 3 so that it locates the rightmost occurrence of character `c` in string `s`. Remember to revise the loop's invariant.

2. Use the searching pattern to write the following method:

```
/** powerOfTwo determines whether its argument is a power of 2.
  * @param n - the argument, a nonnegative int
  * @return m, such that n == 2^m,  if  n  is a power of 2
  *  return -1, if  n  is not a power of 2 */
public int powerOfTwo(int n)
```

What is the "collection" that you are "searching" in this problem?

7.6 For-Statements

Definite-iteration loops pervade computer programming. When we use this pattern of definite iteration,

```
{ int i = INITIAL VALUE;
  while ( TEST ON i )
        { EXECUTE LOOP BODY;
          INCREMENT i;
        }
}
```

there is a special loop statement in Java, called a *for-statement*, that we can use to tersely code the above pattern; it looks like this:

```
for ( int i = INITIAL VALUE; TEST ON i; INCREMENT i; )
    { EXECUTE LOOP BODY; }
```

The semantics of the for-statement is exactly the semantics of the definite iteration pattern, but there is a small technical point: The scope of the declaration of `i` extends only as far as the for-statement's body—the statements following the for-statement cannot examine `i`'s value. If it is important that `i`'s value be available after the loop terminates, then an extra declaration must be prefixed to the loop:

```
int i;
for ( i = INITIAL VALUE; TEST ON i; INCREMENT i; )
    { EXECUTE LOOP BODY; }
// because  i  is declared before the loop starts, i's value can be read here
```

Here is the for-loop that corresponds to the while-loop we saw at the beginning of this Chapter, which prints the decimal values of the reciprocals, 1/2 to 1/20:

```
for ( int denominator = 2;  denominator <= 20;  denominator = denominator+1 )
    // invariant: 1/2, ...up to..., 1/(denominator-1) printed so far
    { System.out.println("1/" + denominator + " = " + (1.0 / denominator)); }
```

Compare this to the original while-loop:

```
int denominator = 2;
while ( denominator <= 20 )
      { System.out.println("1/" + denominator + " = " + (1.0 / denominator));
        denominator = denominator + 1;
      }
```

There are two advantages in using the for-statement for definite iteration:

- The for-statement is more concise than the while-statement, because it displays the initialization, test, and increment of the loop counter all on one line.
- The for-statement suggests to the reader that the loop is a definite iteration.

Begin Footnote: Java's for-statement can be used to compute an indefinite iteration (by omitting the `INCREMENT i` part), but we do not do this in this text. *End Footnote*

The for-statement is particularly useful for systematically computing upon all the items in a set; here is a loop that easily prints all the characters in a string `s`, one per line, in reverse order:

```
for ( int i = s.length() - 1;  i >= 0;  i = i - 1 )
    // invariant: printed characters at  s.length()-1 ...downto... (i+1)
    { System.out.println(s.charAt(i)); }
```

Exercises

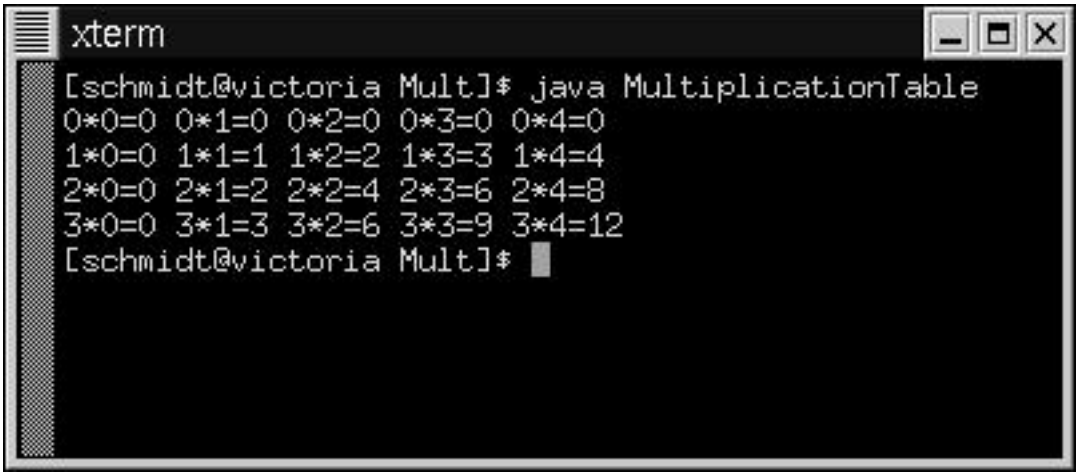
1. Rewrite the `computeAverage` method in Figure 1 so that it uses a for-statement.
2. Use a for-statement to write a function, `reverse`, that receives a string parameter, `s`, and returns as its result the string that looks like `s` reversed, e.g., `reverse("abcd")` returns `"dcba"`.
3. Rewrite the method in Figure 2 to use a for-statement.
4. Rewrite into for-loops the while-loops you wrote in the answers to the Exercises for the "Definite Iteration" Section

7.7 Nested Loops

A loop may be placed within the body of another loop; a *nested loop* is the result. Nested loops are the natural solution to problems that require a "table" of answers, so we begin with two examples that build tables.

Computing a Multiplication Table

We might wish to generate a 4-by-5 table of all the mutiplications of 0..3 by 0..4. The output might appear like this:



To do this, we must generate all possible combinations of `i * j`, when `i` varies in the range 0..3 and `i` varies in the range 0..4. The algorithm for printing the multiplications might go:

1. for `i` varying from 0 to 3, do the following: print `i * 0`, `i * 1`, ..., `i * 4`.

The "for" keyword in the algorithm suggests that a for-statement is the best way to write the required loop. Similarly, printing the multiplications, `i * 0`, `i * 1`, ... `i * 4`, can be done by varying the second operand over the range, 0..4:

1. for `i` varying from 0 to 3, do the following:
    - for `j` varying from 0 to 4, do the following: print `i * j`

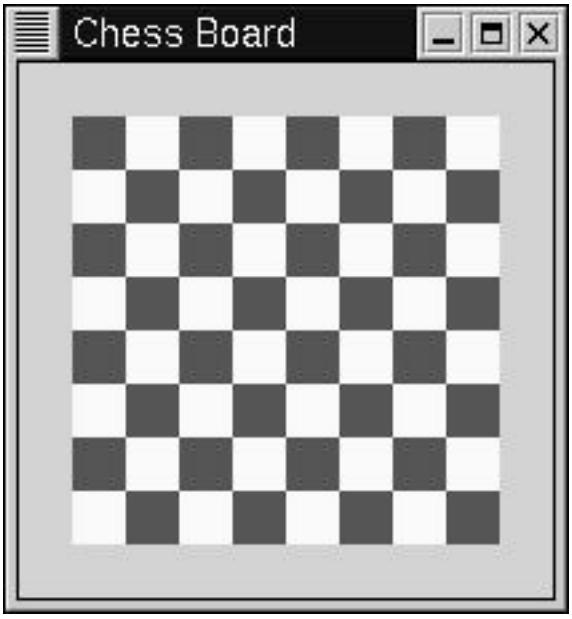
The completed algorithm uses its outer loop to count and print the rows of multiplications and uses its inner loop to print the individual multiplications on each row:

```
for ( int i = 0; i <= 3; i = i + 1 )
    { // invariant: printed  0*x  up to  (i-1)*x,  for all values  x  in 0..4
      for ( int j = 0; j <= 4; j = j + 1 )
          // invariant: printed  i*0  up to  i*(j-1)
          { System.out.print(i + "*" + j + "=" + (i * j) + " "); }
      System.out.println();
    }
```

Note the uses of `print` and `println` to format the rows of the table.

Drawing a Chessboard

How might we paint an `n`-by-`n` chessboard in a graphics window?



This task is also a table-printing problem, where the "values" in the table are red and white squares. To understand which squares should be red and which should be white, consider this numbering scheme for the squares, which was suggested by the multiplication table seen earlier:

```
0, 0     0, 1     0, 2    ...    0, n-1
1, 0     1, 1     1, 2    ...    1, n-1
 ...
n-1, 0   n-1, 1   n-1, 2  ...  n-1, n-1
```

Assuming that the board's upper-left corner (the one numbered by 0,0) is a red square, then it is easy to calculate the color of every square: If the square's number is `i,j`, then the square is red when `i + j` is even-valued; otherwise, it is white (when `i + j` is odd-valued).

We write the algorithm for painting the board by imitating the algorithm for printing the multiplication table:

1. for `i` varying from 0 to `n-1`, do the following:

    - for `j` varying from 0 to `n-1`, do the following: paint the square numbered `i, j`: If `i + j` is even-valued, paint a red square; otherwise, paint a white square.

Figure 7.7: method to paint a chessboard

```
/** paintBoard paints an n-by-n chessboard of red and white squares
  * @param start_x - position of the upper left corner of the board
  * @param start_y - position of the upper left corner of the board
  * @param total_rows - the number of rows of the board
  * @param square_size - the width of each square, in pixels
  * @param g - the graphics pen */
private void paintBoard(int start_x, int start_y,
                    int total_rows, int square_size, Graphics g)
{ for ( int x = 0;  x < total_rows;  x = x + 1 )
      // invariant: have painted  x  rows so far
      { // calculate position of row x:
        int x_position = start_x + (x * square_size);
        for ( int y = 0;  y < total_rows;  y = y + 1 )
            // invariant: have painted  y  squares of row  x
            { // calculate position of the  y-th  square:
              int y_position = start_y + (y * square_size);
              if ( ((x + y) % 2) == 0 )  // is square  x,y  a red one?
                   { g.setColor(Color.red); }
              else { g.setColor(Color.white); }
              g.fillRect(x_position, y_position, square_size, square_size);
            }
      }
}
```

We write the method so that a chessboard can be painted at whatever position, number of rows, and size a user desires. Figure 7 shows the method, which can be invoked by a panel's `paintComponent` method, as we saw in the bulls-eye-painting example earlier in this chapter in Figures 2 and 3.

Alphabetizing a String

When a set of items must be arranged or reordered, nested loops often give a solution. Given a string, `s`, say that we must construct a new string that has all of `s`'s characters but rearranged in alphabetical order. For example, if `s` is `butterfly`, then its alphabetized form is `beflrttuy`.

The algorithm for alphabetization will copy the characters, one by one, from string `s` into a new, alphabetized string, which we call `answer`. Here is an initial algorithm:

1. Set `answer = ""`
2. Working from left to right, for each character in `s`, copy the character into its proper position in `answer`.

The use of "for" in Step 2 suggests that a for-statement might be used to examine and extract the characters in `s`, say, as `s.charAt(0)`, `s.charAt(1)`, and so on:

```
answer = "";
for ( int i = 0;  i != s.length();  i = i + 1 )
    // invariant: characters at indices 0..i-1 have been inserted into  answer
    { insertAlphabetically(s.charAt(i), answer); }
```

The body of the loop contains a helper method, `insertAlphabetically`, which will place its first argument, the character, into the correct position within its second argument, `answer`, so that `answer` remains alphabetized.

What is the algorithm for `insertAlphabetically`? Let's consider the general probem of inserting a character, `c`, into its proper position in an alphabetized string, `alpha`. This is in fact a searching problem, where we search `alpha` from left to right until we find a character that appears later in the alphabet than does `c`. We adapt the searching pattern for loops seen earlier in this Chapter:

1. Set `index = 0` (this variable is used to look at individual characters within `alpha`), and set `searching_for_c_position = true`.
2. While (`searching_for_c_position` and there are still characters in `alpha` to examine), do the following
   - If `c` is less-or-equals to `alpha.charAt(index)`, then we have found the correct position for inserting `c`, so set `searching_for_c_position = false`;
   - Else, increment `index` (and keep searching).
3. Insert `c` into `alpha` in front of the character at position `index` within `alpha`.

Figure 8 shows the final versions of the two methods developed for generating the complete, alphabetized string. The public method, `alphabetize`, uses a for-statement to systematically copy each character in the orginal string into the alphabetized string.

Helper method `insertAlphabetically` uses a searching loop to find the correct position to insert each character into the alphabetized string. When the searching loop finishes, local variable `index` marks the position where the character should be inserted. The insertion is done by the last statement within `insertAlphabetically`, which makes clever use of the `substring` method:

```
return  alpha.substring(0, index) + c + alpha.substring(index, alpha.length());
```

That is, `alpha.substring(0, index)` is the front half of string `alpha`, from character 0 up to character `index`, and `alpha.substring(index, alpha.length())` is the back half of `alpha`, from character `index` to the end.

Note also within `insertAlphabetically` that the `<=` operation is used to compare two characters—for example, `'a' <= 'b'` computes to `true` but `'b' <= 'a'` is `false`.

Figure 7.8: methods for alphabetizing a string

```
/** alphabetize computes a string that has the same characters as
    *  its argument but arranged in alphabetical order
    * @param s - the input string
    * @return the alphabetized string  */
  public String alphabetize(String s)
  { String answer = "";
    for ( int i = 0;  i != s.length();  i = i + 1 )
    // update the  answer  by inserting  s.charAt(i)  into it:
    { answer = insertAlphabetically(s.charAt(i), answer); }
    return answer;
  }

  /** insertAlphabetically  inserts  c  into  alpha,  preserving
    *  alphabetical ordering.  */
  private String insertAlphabetically(char c, String alpha)
  { int index = 0;  // the position where we will possibly insert  c
    boolean searching_for_c_position = true;
    while ( searching_for_c_position  &&  index < alpha.length() )
          { if ( c <= alpha.charAt(index) )  // should  c  be placed at  index?
                 { searching_for_c_position = false; }
            else { index = index + 1; }
          }
    // ``break'' alpha  into two pieces and place  c  in the middle:
    return  alpha.substring(0, index)
              + c + alpha.substring(index, alpha.length());
  }
```

Alphabetization is a special case of *sorting*, where a collection of items are arranged ordered according to some standardized ordering. (For example, a dictionary is a collection of words, sorted alphabetically, and a library is a collection of books, sorted by the catalog numbers stamped on the books' spines.)

Exercises

1. Write nested loops that generate the addition table for 0+0 up to 5+5.

2. Write this method:

```
/** removeDuplicateLetters  constructs a string that contains the
  *  same letters as its argument except that all duplicate letters
  *  are removed, e.g.,  for argument, "butterflies", the result is
  *  "buterflis"
  * @param s - the argument string
```

```
  * @return a string that looks like  s  but with no duplicates  */
public String removeDuplicateLetters(String s)
```

3. Write nested loops that print this pattern:

```
0 0
1 0   1 1
2 0   2 1   2 2
3 0   3 1   3 2   3 3
```

4. Write nested loops that print this pattern:

```
0 3   0 2   0 1  0 0
1 3   1 2   0 1
2 3   2 2
3 3
```

7.8 Writing and Testing Loops

Loops are easily miswritten. For example, one must not inadvertently type an extra semicolon—this example,

```
int i = 2;
while ( i != 0 );
      { i = i - 1; }
```

fails to terminate because the semicolon after the test causes the Java compiler to read the code as `while ( i != 0 ) {}; { i = i - 1; }`—the loop has an empty body.

Problems also arise when a loop's test is carelessly formulated. For example, this attempt to compute the sum, `1 + 2 + ... + n`, fails,

```
int total = 0;
int i = 0;
while ( i <= n )
      { i = i + 1;
        total = total + i;
      }
```

because the loop iterates one time too many. *This form of error, where a loop iterates one time too many or one time too few, is commonly made, so be alert for it when testing the loops you write.*

A related problem is an improper starting value for the loop counter, e.g.,

```
int total = 0;
int i = 1;
while ( i <= n )
      { i = i + 1;
        total = total + i;
      }
```

This loop again attempts to compute the summation, `1 + 2 + ... + n`, but it forgets to add the starting value of `i` into its total.

Examples like these show that it is not always obvious when a loop iterates exactly the correct number of times. *When you test a loop with example data, be certain to know the correct answer so that you can compare it to the loop's output.*

Here is another technical problem with loop tests: Never code a loop's test expression as an equality of two doubles. For example, this loop

```
double d = 0.0;
while ( d != 1.0 )
      { d = d + (1.0 / 13);
        System.out.println(d);
      }
```

should terminate in 13 iterations but in reality does not due to computer arithmetic, where fractional numbers like 1/13 are imperfectly represented as fractions in decimal format.

Formulating test cases for loops is more difficult than formulating test cases for conditionals, because it is not enough to merely ensure that each statement in the loop body is executed at least once by some test case. A loop encodes a potentially infinite number of distinct execution paths, implying that an infinite number of test cases might be needed. Obviously, no testing strategy can be this exhaustive.

In practice, one narrows loop testing to these test cases:

- Which test cases should cause the loop to terminate with zero iterations?
- Which test cases should cause the loop to terminate with exactly one iteration?
- Which "typical" test cases should cause the loop to iterate multiple times and terminate?
- Which test cases might cause the loop to iterate forever or produce undesirable behavior?

Test cases of all four forms are applied to the loop code.

One more time, this attempt to compute the summation `1 + 2 + ... + n`,

```
int n = ... ; // the input value
int total = 0;
int i = 0;
while ( i != n )
      { total = total + i;
        i = i + 1;
      }
```

might be tested with `n` set to 0 (which we believe should cause immediate termination), set to 1 (should cause termination in one iteration), set to, say, 4 (a typical case that requires multiple iterations), and set to a negative number, say, -1, (which might cause unwanted behavior). These test cases quickly expose that the test expression forces termination one iteration too soon. Subtle errors arise when a loop's test expression stops the loop one iteration too soon or one iteration too late; testing should try to expose these "boundary cases."

A more powerful version of loop testing is *invariant monitoring*: If you wrote an invariant for the loop, you can monitor whether the invariant is holding true while the loop iterates. To do this, insert inside the loop one or more `println` statements that display the values of the variables used by the loop. (Or, use an IDE to halt the loop at each iteration and display the values of its variables.) Use the variables's values to validate that the loop's invariant is holding true. If the invariant is remaining true, this gives you great confidence that the loop is making proper progress towards the correct answer.

For example, consider Figure 1, which displays a loop that sums a sequence of exam scores. The loop's invariant,

```
// at each iteration: total_points == exam_1 + exam_2 + ... + exam_count
```

tells us what behavior to observe at the start (and the end) of each loop iteration. The `println` statement placed at the end of the loop in Figure 1 lets us verify that the invariant property that we believed to be true is in fact holding true during the execution.

If we monitor a loop's invariant, like the one in Figure 1, and we find that the invariant goes false at some interation, this is an indication that either the loop's code or the invariant is incorrectly written. In the former case, repair is clearly needed; in the latter case, we do not understand what our loop should do and we must study further.

Although loop testing can never be exhaustive, please remember that any testing is preferable to none—the confidence that one has in one's program increases by the number of test cases that the program has successfully processed.

7.9 Case Study: Bouncing Ball Animation

Because a loop lets us repeat an action over and over, we can write a loop that paints a graphical image over and over, changing the image slightly with each iteration—This gives the illusion of movement. An application that displays moving objects in a graphics window is an *animation*.

A simple but classic animation is a ball bouncing within a box:

(Although the printed page cannot show it, the red ball is travelling from side to side within the frame.)

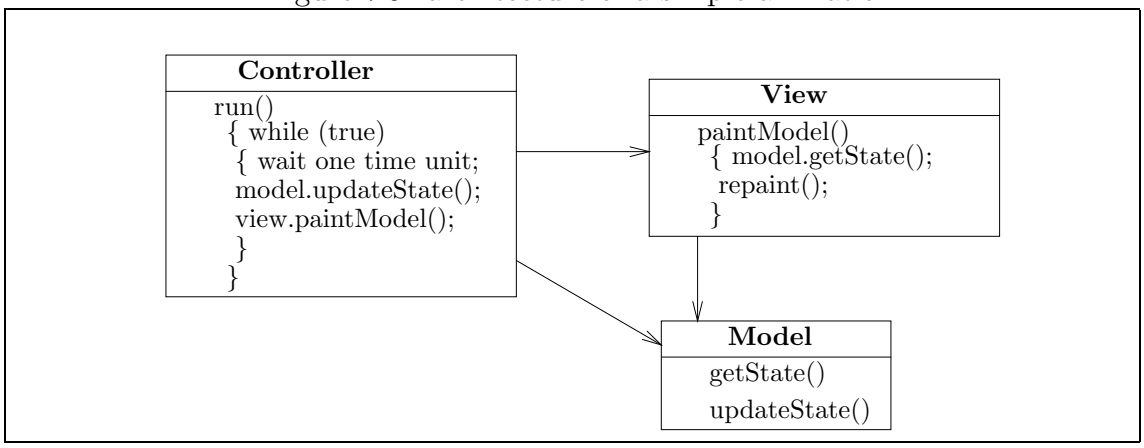
A program written in the Model-View-Controller architectural style does best: The animation is *modelled*, in this case, by objects that represent the ball and box. A *view* paints the model's current state on the display. Finally a *controller* monitors the passage of time and tells the model when to update the ball's position and repaint it on the display.

Figure 9 shows the architecture of the animation program. The controller contains a loop that executes an iteration for each unit of time that passes. At each unit of time, the loop's body tells the model to update its state, and it tells the view to repaint the model; the repeated paintings look like movement. The model and view have methods that respond to the controller's requests. Also, the view must ask the model for its current state each time the model must be painted.

Recall once more the steps we take to design and solve a programming problem:

1. *State the program's behavior, from the perspective of the program's user.*
2. *Select a software architecture that can implement the behavior.*
3. *For each of the architecture's components, specify classes with appropriate attributes and methods.*

Figure 7.9: architecture of a simple animation



4. *Write and test the individual classes.*

5. *Integrate the classes into the architecture and test the system.*

Let's move through these steps:

Behaviors and Architecture

Steps (1) and (2) are not a challenge for building the moving-ball animation: The behavior of the animation has already been indicated—there is nothing for the user to do but watch—and the architecture in Figure 9 is a good start towards implementing the desired behavior. The architecture shows that the application has distinct model, view, and controller subassemblies, so we develop each subassembly separately, beginning with the model.

Specifying the Model Subassembly

We begin with the model subassembly of the animation—What are the model's components? They are of course the ball and the box in which the ball moves. Our first attempt at specifying the model might place ball and box together in one class, like this:

`class BallAndBox`	maintains a ball that moves within a box
Attributes	
the box's size, the ball's size, the ball's position, the ball's velocity	
Methods	
`moveTheBall(int time_units)`	Updates the model's state by moving the ball a distance based on its current position, its velocity, and the amount of time, `time_units`, that have passed since the last time the state has been updated.
`getTheState()`	Returns the current position of the ball, its size, and the dimensions of the box.

This initial specification is a bit problematic: The attributes of the ball and the box are mixed together, and this will cause trouble if we later modify the animation, say by using two balls or using a different form of container. The specification of the `getTheState` also has some confusion about what exactly is the state of the model.

These concerns motivate us to design the model as two classes—one class for the ball and one for the box.

Let's consider the ball's class first—what are its attributes and what are its methods? Of course, a ball has a physical size and location (x- and y-coordinates). Because it is moving in two-dimensional space, it has a velocity in both x- and y-coordinates. The primary method the ball requires is the ability to move on request. This produces the specification of `class MovingBall` that appears in Table 10. In addition to its mutator method, `move`, the final version of `class MovingBall` will also possess accessor methods that return the ball's radius, position, and so on.

The description of the `move` method in Figure 10 makes clear that the moving ball must send messages to the box in which it is contained, to learn if it has hit a well and must change its direction. This motivates us to design `class Box`. A box's attributes are just the positions of its walls, and assuming that the box is shaped as a square, it suffices to remember just the box's width. A box does not actively participate in the moving-ball animation, but it must have methods that reply when asked if a given position is in contact with a wall.

Table 11 shows one way to specify `class Box`.

At this point, the design of the model is mostly complete. Refinements might be made as the programming problem is better understood, but the specifications are a good starting point for developing the remainder of the program.

Implementing and Testing the Model

Now, we can write the classes. As usual, `class Ball` has accessor methods that yield the values of its attributes. The most interesting method is `move`, which updates the

Figure 7.10: specification of moving ball

`class MovingBall`	models a ball that moves in two-dimensional space
Attributes	
`int x_pos, int y_pos`	the center coordinates of the ball's location
`radius`	the ball's radius (size)
`int x_velocity, int y_velocity`	the ball's horizontal and vertical velocities
Method	
`move(int time_units)`	moves to its new position based on its velocity the the elapsed amount of `time_units`, reversing direction if it comes into contact with a wall of the container (the `Box`) that holds it.

Figure 7.11: specification of box

`class Box`	models a square box
Attribute	
`int BOX_SIZE`	the width of the square box
Methods	
`inHorizontalContact(int x_position): boolean`	responds whether `x_position` (a horizontal coordinate) is in contact with one of the Box's (leftmost or rightmost) walls
`inVerticalContact(int y_position): boolean`	responds whether `y_position` (a vertical coordinate) is in contact with one of the Box's (upper or lower) walls

ball's state; its algorithm goes

1. Move the ball to its new position.

2. Ask the box that contains the ball whether the ball has come into contact with one of the box's horizontal or vertical walls; if it has, then change the ball's direction. (For example, if the ball makes contact with a horizontal wall, then its horizontal direction must reverse; this is done by negating the ball's horizontal velocity.)

Figure 12 shows the coding of `class Ball`'s `move` method.

The coding of `class Box` is simple, because the box's state is fixed when the box is constructed, so the box's methods merely return properties related to the positions of the box's walls. See Figure 13.

Figure 7.12: model of moving ball

```
/** MovingBall models a moving ball */
public class MovingBall
{ private int x_pos;  // ball's center x-position
  private int y_pos;  // ball's center y-position
  private int radius; // ball's radius

  private int x_velocity = +5; // horizonal speed; positive is to the right
  private int y_velocity = +2; // vertical speed; positive is downwards

  private Box container;  // the container in which the ball travels

  /** Constructor MovingBall constructs the ball.
    * @param x_initial - the center of the ball's starting horizontal position
    * @param y_initial - the center of the ball's starting vertical position
    * @param r - the ball's radius
    * @param box - the container in which the ball travels */
  public MovingBall(int x_initial, int y_initial, int r, Box box)
  { x_pos = x_initial;
    y_pos = y_initial;
    radius = r;
    container = box;
  }

  /** xPosition returns the ball's current horizontal position */
  public int xPosition()
  { return x_pos; }

  /** yPosition returns the ball's current vertical position */
  public int yPosition()
  { return y_pos; }

  /** radiusOf returns the ball's radius */
  public int radiusOf()
  { return radius; }

  /** move moves the ball
    * @param time_units - the amount of time since the ball last moved */
  public void move(int time_units)
  { x_pos = x_pos + (x_velocity * time_units);
    if ( container.inHorizontalContact(x_pos) )
       { x_velocity = -x_velocity; }   // reverse horizontal direction
    y_pos = y_pos + (y_velocity * time_units);
    if ( container.inVerticalContact(y_pos) )
       { y_velocity = -y_velocity; }   // reverse vertical direction
  }
}
```

Figure 7.13: model of a box

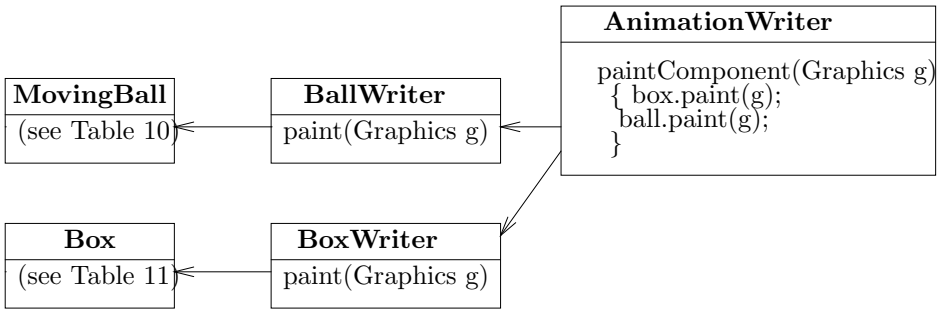
```
/** Box models a box in which moving objects live */
public class Box
{ private int box_size;  // the size of the box

  /** Constructor Box builds the box
    * @param size - the size of the box */
  public Box(int size)
  { box_size = size; }

  /** inHorizontalContact replies whether a position has contacted a
    *  horizontal wall.
    * @param x_position - the position examined
    * @return true, if  x_position  equals or exceeds the positions of the
    *   horizontal walls; return false, otherwise  */
  public boolean inHorizontalContact(int x_position)
  { return (x_position <= 0 ) || (x_position >= box_size); }

  /** inVerticalContact replies whether a position has contacted a
    *  vertical wall.
    * @param y_position - the position examined
    * @return true, if  y_position  equals or exceeds the positions of the
    *   vertical walls; return false, otherwise  */
  public boolean inVerticalContact(int y_position)
  { return (y_position <= 0 ) || (y_position >= box_size); }

  /** sizeOf returns the size of the box */
  public int sizeOf()
  { return box_size; }
}
```

Because `class MovingBall` depends on `class Box`, we test the classes together, say, by writing a testing program that constructs a box and places the ball in the middle of the box. Then, we tell the ball to move and we print its new position. Of course, we should move the ball until it comes into contact with a wall so that we can see how the ball's direction changes as a result. Here is some testing code:

```
Box box = new Box(50);   // 50 pixels by 50 pixels in size
MovingBall ball = new MovingBall(25, 25, 10, box);  // radius is 10 pixels
while ( true )
      { ball.move(1);    // 1 unit of elapsed time
        System.out.println("x = " + ball.xPosition()
                       + "; y = " + ball.yPosition());
      }
```

Specifying and Implementing the View Subassembly

The ball and box become more interesting once we write a view to paint them. Since the model is designed in two components, we propose a view class that paints the box and a view class that paints the ball. This will make it easier to extend the animation later to contain multiple moving balls, and it suggests the pleasant idea that each model component should have its own view component. Here is a diagram of how the view classes might be specified:



When the animation must be painted, an `AnimationWriter` is asked to paint the entire picture on a panel. This class asks the `BoxWriter` to paint the box and it asks the `BallWriter` to paint a ball. The latter two classes query the accessor methods of their respective model components for the state of the box and ball.

There is little more to specify about the view classes, so we study their codings. Figure 14 presents `class AnimationWriter`; notice how this class gives its graphics pen to `class BoxWriter` and then to `class BallWriter`, presented in Figure 15, so that the box and ball are painted on the window with which the graphics pen is associated.

We can test the view classes with the model we have already built and tested.

Implementing the Controller

Finally, the program's controller uses a loop to control the progress of the simulation; its algorithm is

1. loop forever, doing the following steps:

    - Tell the ball to move one time unit.
    - Tell the writer to repaint the state of the animation on the display

Figure 16 gives the coding of this loop and also the coding of the start-up class that contructs all the animation's objects.

The while-loop in the `run` method of `class BounceController` is intended not to terminate, hence its test is merely `true`. To present an illusion of movement, the same

Figure 7.14: view class for moving-ball simulation

```
import java.awt.*;
import javax.swing.*;
/** AnimationWriter displays a box with a ball in it.  */
public class AnimationWriter extends JPanel
{ private BoxWriter box_writer;    // the output-view of the box
  private BallWriter ball_writer;  // the output-view of the ball in the box

  /** Constructor AnimationWriter constructs the view of box and ball
    * @param b - the box's writer
    * @param l - the ball's writer
    * @param size - the frame's size */
  public AnimationWriter(BoxWriter b, BallWriter l, int size)
  { box_writer = b;
    ball_writer = l;
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("Bounce");
    my_frame.setSize(size, size);
    my_frame.setVisible(true);
  }

  /** paintComponent paints the box and ball
    * @param g - the graphics pen */
  public void paintComponent(Graphics g)
  { box_writer.paint(g);
    ball_writer.paint(g);
  }
}
```

Figure 7.15: view classes for box and ball

```
import java.awt.*;
/** BoxWriter displays a box  */
public class BoxWriter
{ private Box box;   // the (address of the) box object that is displayed

  /** Constructor BoxWriter displays the box
    * @param b - the box that is displayed */
  public BoxWriter(Box b)
  { box = b; }

  /** paint paints the box
    * @param g - the graphics pen used to paint the box */
  public void paint(Graphics g)
  { int size = box.sizeOf();
    g.setColor(Color.white);
    g.fillRect(0, 0, size, size);
    g.setColor(Color.black);
    g.drawRect(0, 0, size, size);
  }
}

import java.awt.*;
/** BallWriter displays a moving ball */
public class BallWriter
{ private MovingBall ball;   // the (address of the) ball object displayed
  private Color balls_color; // the ball's color

  /** Constructor BallWriter
    * @param x - the ball to be displayed
    * @param c - its color */
  public BallWriter(MovingBall x, Color c)
  { ball = x;
    balls_color = c;
  }

  /** paint paints the ball on the view
    * @param g - the graphics pen used to paint the ball */
  public void paint(Graphics g)
  { g.setColor(balls_color);
    int radius = ball.radiusOf();
    g.fillOval(ball.xPosition() - radius,
               ball.yPosition() - radius, radius * 2, radius * 2);
  }
}
```

technique from motion pictures is used: Method `delay` pauses the program slightly (here, 20 milliseconds, but this should be adjusted to look realistic for the processor's speed) before advancing the ball one more step. The method uses a built-in method, `Thread.sleep`, which must be enclosed by an exception handler. These details are unimportant, and the `delay` method may be copied and used whereever needed.

The moving-ball animation is a good start towards building more complex animations, such as a billiard table or a pinball machine. Indeed, here is a practical tip: If you are building a complex animation, it helps to build a simplistic first version, like the one here, that implements only some of the program's basic behaviors. Once the first version performs correctly, then add the remaining features one by one.

For example, if our goal is indeed to build an animated pinball machine, we should design the complete pinball machine but then design and code an initial version that is nothing more than an empty machine (a box) with a moving pinball inside it. Once the prototype operates correctly, then add to the machine one or more of its internal "bumpers" (obstacles that the ball hits to score points). Once the pinball correctly hits and deflects from the bumpers, then add the scoring mechanism and other parts.

Each feature you add to the animation causes you to implement more and more of the attributes and methods of the components. By adding features one by one, you will not be overwhelmed by the overall complexity of the animation. The Exercises that follow give a small example of this approach.

Exercises

1. Execute the moving ball animation. Occasionally, you will observe that the ball appears to bounce off a wall "too late," that is, the ball changes direction after it intersects the wall. Find the source of this problem and suggest ways to improve the animation.

2. Revise the moving-ball animation so that there are two balls in the box. (Do not worry about collisions of the two balls.)

3. If you completed the previous exercise, rebuild the animation so that a collision of the two balls causes both balls to reverse their horizontal and vertical directions.

4. Place a small barrier in the center of the box so that the ball(s) must bounce off the barrier.

7.10 Recursion

No doubt, you have seen a "recursive picture," that is, a picture that contains a smaller copy of itself that contains a smaller copy of itself that contains.... (You can

Figure 7.16: controller and start-up class for moving-ball animation

```
/** BounceController controls a moving ball within a box.  */
public class BounceController
{ private MovingBall ball;  // model object
  private AnimationWriter writer; // output-view object

  /** Constructor BounceController initializes the controller
    * @param b - the model object
    * @param w - the output-view object  */
  public BounceController(MovingBall b, AnimationWriter w)
  { ball = b;
    writer = w;
  }

  /** runAnimation  runs the animation by means of an internal clock */
  public void runAnimation()
  { int time_unit = 1;    // time unit for each step of the animation
    int painting_delay = 20;  // how long to delay between repaintings
    while ( true )
          { delay(painting_delay);
            ball.move(time_unit);
            System.out.println(ball.xPosition() + ", " + ball.yPosition());
            writer.repaint();  // redisplay box and ball
          }
  }

  /** delay pauses execution for  how_long  milliseconds */
  private void delay(int how_long)
  { try { Thread.sleep(how_long); }
    catch (InterruptedException e) { }
  }
}
```

Figure 7.16: start-up class for moving-ball animation (concl.)

```
import java.awt.*;
/** BounceTheBall constructs and starts the objects in the animation.  */
public class BounceTheBall
{ public static void main(String[] args)
  { // construct the model objects:
    int box_size = 200;
    int balls_radius = 6;
    Box box = new Box(box_size);
    // place the ball not quite in the box's center; about 3/5 position:
    MovingBall ball = new MovingBall((int)(box_size / 3.0),
                                     (int)(box_size / 5.0),
                                     balls_radius, box);
    BallWriter ball_writer = new BallWriter(ball, Color.red);
    BoxWriter box_writer  = new BoxWriter(box);
    AnimationWriter writer
             = new AnimationWriter(box_writer, ball_writer, box_size);
    // construct the controller and start it:
    new BounceController(ball, writer).runAnimation();
  }
}
```

simulate this by hanging two large mirrors on opposite walls, standing in the middle, and looking at one of the mirrors.) It is self-reference that characterizes recursion, and when a method contains a statement that invokes (restarts) itself, we say that the method is *recursively defined*.

In the bank-accounting example in Chapter 6. we used recursion to restart a method, but now we study a more sophisticated use of recursion, where a complex problem is solved by solving simpler instances of the same problem.

Here is an informal example of recursive problem solving: How do you make a three-layer cake? A curt answer is, "make a two-layer cake and top it with one more layer!" In one sense, this answer begs the question, because it requires that we know already how to make a cake in order to make a cake, but in another sense, the answer says we should learn how to make first a simpler version of the cake and use the result to solve the more complex problem.

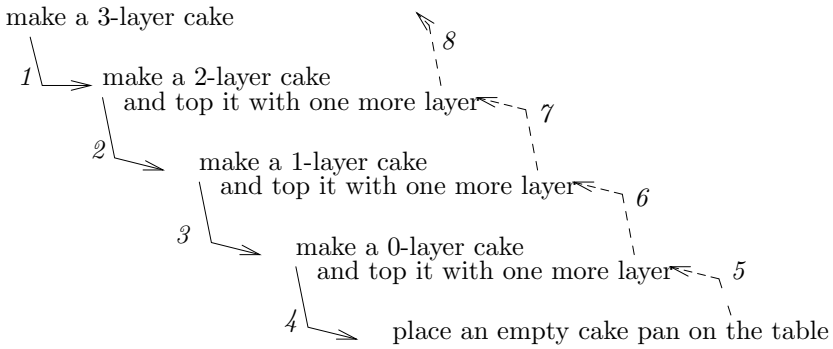
If we take seriously the latter philosophy, we might write this recipe for making a cake with some nonzero number of layers, say, `N+1`:

To make an `(N+1)`*-layer cake, make an* `N`*-layer cake and top it with one more layer.*

The reciple simplifies `(N+1)`-layer cake making into the problem of `N`-layer cake making. But something is missing—where do we *begin* cake making? The answer is startlingly simple:

To make a `zero`*-layer cake, place an empty cake pan on the table.*

The solution to the original problem becomes clearer—to make a three-layer cake, we must first make a two layer cake (and then top it with one more layer); but to make a two-layer cake, we must first make a one-layer cake (and then top it with one more layer); but to make a one-layer cake, we must make a zero-layer cake (and then top it with one more layer); and we make a "zero-layer cake" from just an empty pan. Perhaps this explanation is a bit wordy, but all the steps are listed. A diagram displays the steps more pleasantly:



By following the arrows, we understand how the problem of making a multi-layer case is decomposed into simpler problems (see the downwards arrows) and the result is assembled from from the solutions—the pan and layers (see the upwards arrows).

The strategy of solving a task by first solving a simpler version of the task and building on the simpler solution is called *problem solving by recursion* or just *recursion*, for short. When we solve a problem by recursion, we can approach it as a task of writing simultaneous algebraic equations, like these, for cake making:

```
bakeACake(0) = ...
bakeACake(N + 1) = ... bakeACake(N) ...,  where N is nonnegative
```

The first equation states how a 0-layer cake is made, and the second explains how an `N+1`-layer cake is made in terms of making an `N`-layer one. Based on the narrative for cake making, we would finish the equations like this:

```
bakeACake(0) = place an empty pan on the table
bakeACake(N + 1) = bakeACake(N) and top it with one more layer
```

The algorithm can be used to generate the steps for a 3-layer cake:

```
bakeACake(3)
=> bakeACake(2)
     and top it with one more layer
=> (bakeACake(1)
      and top it with one more layer)
```

Figure 7.17: class Cake

```
/** class Cake is a simple model of a multi-layer cake  */
public class Cake
{ int how_many_layers;

  /** Constructor Cake constructs a 0-layer cake---a pan */
  public Cake()
  { how_many_layers = 0; }

  /** addALayer makes the Cake one layer taller */
  public void addALayer()
  { how_many_layers = how_many_layers + 1; }

  /** displayTheCake prints a representation of the cake */
  public void displayTheCake()
  { ... left as a project for you to do ... }
}
```

```
     and top it with one more layer
=> ((bakeACake(0)
       and top it with one more layer))
      and top it with one more layer)
     and top it with one more layer
=> ((place an empty pan on the table
       and top it with one more layer))
      and top it with one more layer)
     and top it with one more layer
```

A recursive algorithm can be reformatted into a Java method that uses recursion. Perhaps there is a Java data type, named `Cake`, such that `new Cake()` constructs a 0-layer cake. Perhaps `Cake` objects have a method, named `addALayer`, which places one more layer on a cake. For fun, a version of `class Cake` is displayed in Figure 17.

Now, we can use `class Cake` to reformat the two algebraic equations for cake making into the Java method displayed in Figure 18.

Now, the assignment,

```
Cake c = bakeACake(0);
```

constructs a 0-layer cake. More interesting is

```
Cake c = bakeACake(3);
```

because this triggers `bakeACake(2)`, which invokes `bakeACake(1)`, which starts `bakeACake(0)`. The last invocation constructs a new cake object, which is then topped by three layers

Figure 7.18: a recursive method for cake making

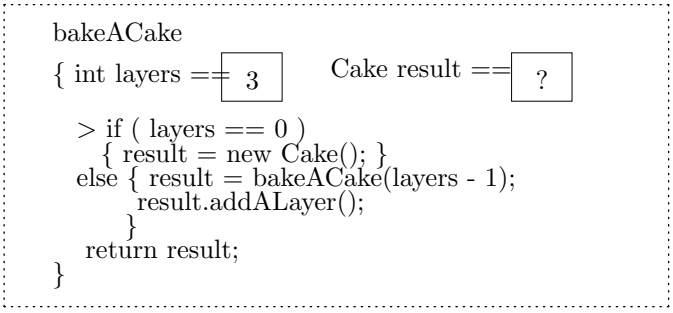
```
/** bakeACake makes a multi-layer cake
  * @param layers - the number of the cake's layers
  * @return the cake  */
public Cake bakeACake(int layers)
{ Cake result;
  if ( layers == 0 )
       // then the first equation applies:
       //  bakeACake(0) = place an empty pan on the table
       { result = new Cake(); }
  else // the second equation applies:
       //  bakeACake(N + 1) = bakeACake(N) and top it with one more layer
       { result = bakeACake(layers - 1);
         result.addALayer();
       }
  return result;
}
```

and eventually assigned to variable `c`. To understand the process, we should study an execution trace.

7.10.1 An Execution Trace of Recursion

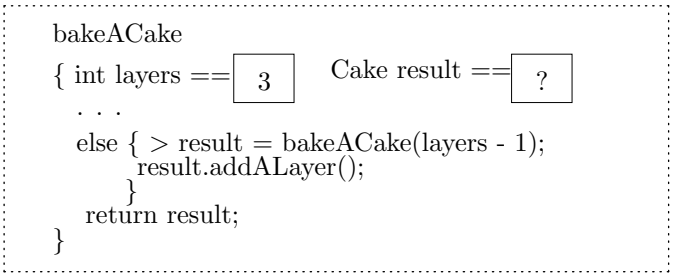
We now study how the method in Figure 12 contructs its answer by recursion in an example invocation of `bakeACake(3)`.

The invocation causes 3 to bind to the the formal parameter, `layers`; it and the local variable, `result`, are created:

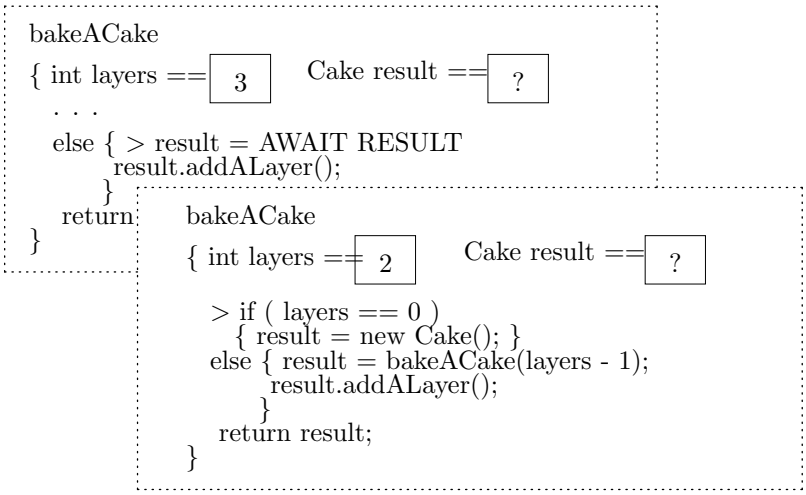


The test marked by `>1???` computes to `false`, meaning that the next step is a

recursive invocation:



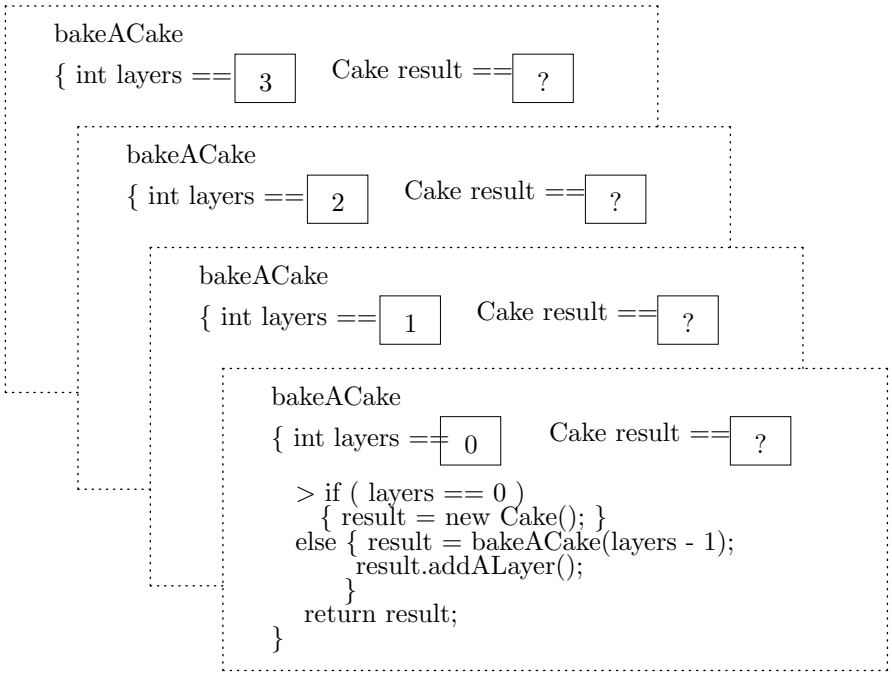
The invocation causes a fresh, *distinct* activation record of `bakeACake` to be executed:



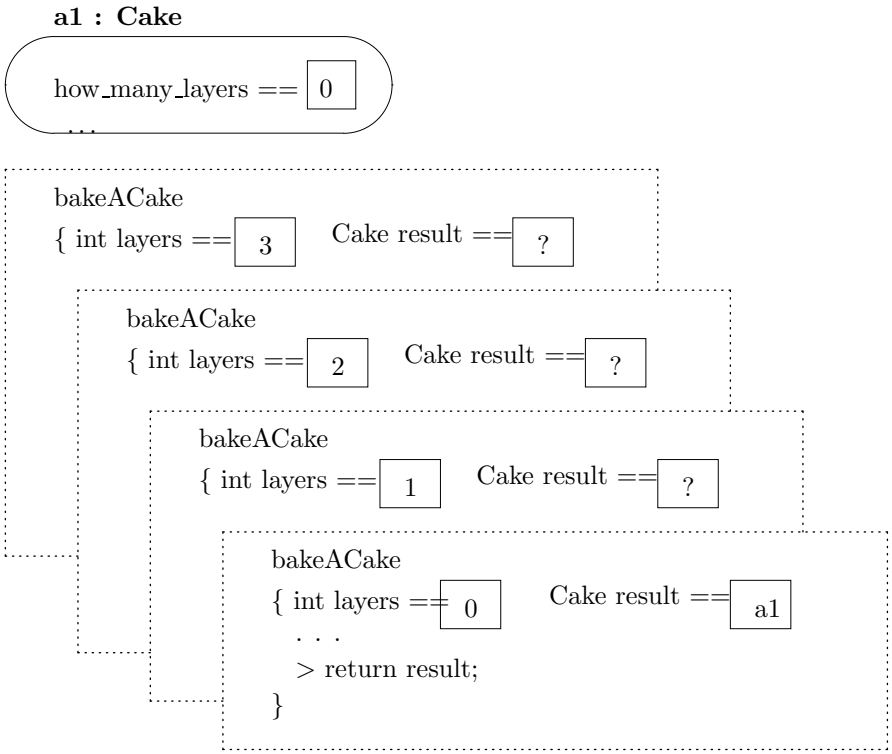
This shows that a *recursive method invocation works like any other invocation:* actual parameters are bound to formal parameters, and a fresh copy of the method's body executes.

Further execution causes two more copies of `bakeACake` to be invoked, producing

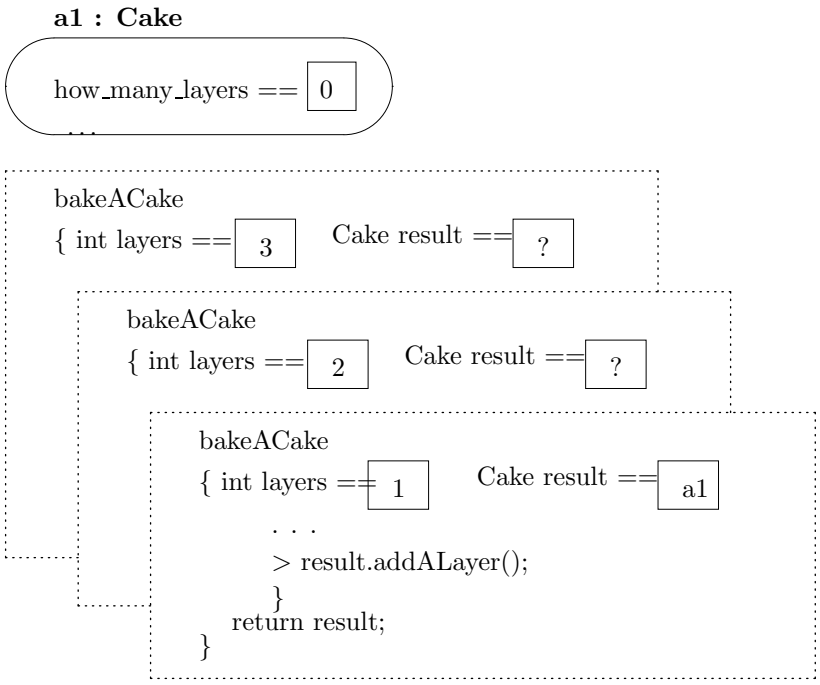
this configuration:



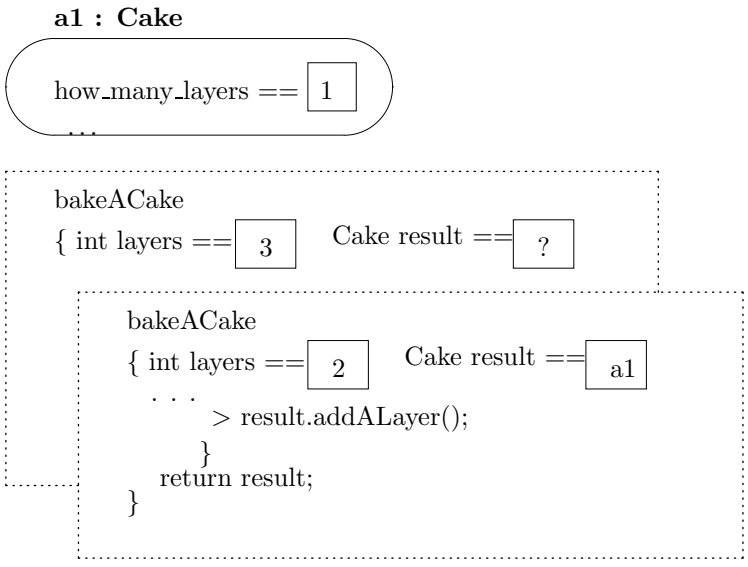
Because the conditional's test evaluates to true, the conditional's then-arm constructs a new `Cake` object and places its address, `a1`, in the local variable, `result`.



The 0-layer cake is returned as the result, and the activation record for the completed invocation disappears.



Next, `result.addALayer()` adds a layer to the cake, and the one-layer cake is returned as the result:



This process repeats until the `result` variable at the initial invocation receives the cake object at address `a1` and places the third layer on it.

The example shows how each recursive invocation decreases the actual parameter by 1, halting when the parameter has 0. When an invoked method returns its result, the result is used by the caller to build *its* result. This matches our intuition that recursion is best used to solve a problem in terms of solving a simpler or smaller one.

Admittedly, the cake-making example is contrived, and perhaps you have already noticed that, given `class Cake`, we can "make" an `N`-layer cake with this for-loop:

```
Cake c = new Cake();
for ( int i = 1;  i != N;  i = i + 1 )
    { c.addALayer(); }
```

But the style of thinking used when writing the for-loop is different than the style used when writing the recursively defined method.

The loop in the previous paragraph should remind you that an incorrectly written repetitive computation might not terminate—this holds true for loops *and it holds true for recursively defined methods as well.* It is not an accident that the cake-making example used recursive invocations with an argument that counted downward from an nonnegative integer, `N`, to 0. As a rule:

> *A recursively defined method should use a parameter whose value decreases at each recursive invocation until the parameter obtains a value that stops the recursive invocations.*

The "counting-down" pattern suggested above will protect us from making the foolish mistake of writing a recursively defined method that restarts itself forever.

7.11 Counting with Recursion

Many computing problems that require counting can be solved by recursion. One classic example is permutation counting: Say that we want to calculate the total number of *permutations* (orderings) of `n` distinct items. For example, the permutations of the three letters `a`, `b`, and `c` are `abc`, `bac`, `bca`, `acb`, `cab`, and `cba`—there. are six permutations.

Permutation generation has tremendous importance to planning and scheduling problems, for example:

- You must visit three cities, `a`, `b`, and `c`, next week. What are the possible orders in which you can visit the cities, and which of these orders produces the shortest or least-expensive trip?

- You must complete three courses to finish your college degree; what are the possible orders in which the courses might be taken, and which ordering allows you to apply material learned in an earlier course to the courses that follow?

- Three courses that are taught at the same hour must be scheduled in three classrooms. What are the possible assignments of courses to classrooms, and which assignments will accommodate all students who wish to sit the courses?

- A printer must print three files; what are the orders in which the files can be printed, and which orderings ensure that the shorter files print before the longer files?

It is valuable for us to know how to generate and count permutations. To study this problem, say that we have `n` distinct letters, and we want to count all the permutations of the letters. The following observation holds the secret to the solution:

> By magic, say that we already know the quantity of permutations of `n-1` distinct letters—say there are `m` of them, that is, there are distinct words, `word_ 1`, `word_ 2`, ..., `word_ m`, where each word is `n-1` letters in length.
>
> Next, given the very last, `n`th letter, we ask, "how many permutations can we make from the `m` words using one more letter?" The answer is, we can insert the new letter it into all possible positions in each of the `m` words: We take `word_ 1`, *which has length* `n-1`, and we insert the new letter in all `n` possible positions in `word_ 1`. *This generates* `n` *distinct permutations of length* `n`. We do the same for `word_ 2`, giving us `n` more permutations. If we do this for all `m` words, we generate `m` sets of `n` permutations each, that is, a total of `n * m` permutations of length `n`. This is the answer.

Finally, we note that when we start with an alphabet of one single letter, there is exactly *one* permutation.

The above insights tell us how to count the quantity of permutations:

For an alphabet of just one letter, there is just one permutation:

```
number_of_permutations_of(1) = 1
```

If the alphabet has `n+1` letters, we count the permutations made from `n` letters, and we use the technique described above to count the permutations generated by the addition of one more letter:

```
number_of_permutations_of(n+1) = (n + 1) * number_of_permutations_of(n)
```

These two equations define an algorithm for computing the quantity of permutations. We can use this algorithm to quickly count the permutations of 4 distinct letters:

```
number_of_permutations_of(4)
= 4 * number_of_permutations_of(3)
= 4 * ( 3 * number_of_permutations_of(2) )
= 4 * ( 3 * ( 2 * number_of_permutations_of(1) ))
= 4 * 3 * 2 * 1  =  24
```

Calculations with this algorithm quickly convince us that even small alphabets generate huge quantities of permutations.

Not surprisingly, permutation counting occurs often in probability and statistics theory, where it is known as the *factorial function*. It is traditional to write the factorial of a nonnegative integer, `n`, as `n!`, and calculate it with these equations:

```
 0!  = 1

 (n+1)!  = (n+1) * n!, when n is nonnegative
```

(Notice that the starting number is lowered to zero, and that `1!` equals 1, just like we determined earlier.)

It is easy to translate these two equations into a recursively defined method that computes factorials of integers:

```
public int fac(int n)
{ int answer;
  if ( n == 0 )
       { answer = 1; }                // 0! = 1
  else { answer = n * fac(n - 1); }  // (n+1)! = (n+1) * n!
  return answer;
}
```

If you test the function, `fac`, on the computer, you will see that, when the argument to `fac` is greater than 13, the answer is so huge that it *overflows* the internal computer representation of an integer!

BeginFootnote: Unfortunately, a Java `int` can hold a value only between the range of about negative 2 billion and positive 2 billion. *EndFootnote*.

For this reason, we revise method `fac` so that it uses Java's `long` version of integers. We also make check that the argument to the function is not so large that the computed answer will overflow the internal representation of a `long` integer. See Figure 19 for the implementation of the factorial function.

When you test the `factorial` function, say by

```
System.out.println("4! = " + factorial(4));
```

you will find that the invocation of `factorial(4)` generates the invocation to `factorial(3)`, and so on, down to `factorial(0)`. Then, once the invoked functions starting returning their answers, a series of four multiplications are performed, which compute the result, 24. This pattern of invocations and multiplications are displayed when we compute with factorial's algorithm:

```
4! =>  4 * 3!
   =>  4 * (3 * 2!)
   =>  4 * (3 * (2 * 1!))
   =>  4 * (3 * (2 * (1 * 0!)))
   =>  4 * (3 * (2 * (1 * 1)))
   =>  4 * (3 * (2 * 1))  =>  4 * (3 * 2)  =>  4 * 6  =>  24
```

Figure 7.19: recursively defined method that computes factorial

```
/** factorial computes  n!  for  n  in the range 0..20.
  * (note: answers for  n > 20  are too large for Java to compute,
  *  and answers for  n < 0   make no sense.)
  * @param n - the input; should be in the range 0..20
  * @return  n!,  if  n in 0..20
  * @throw RuntimeException, if  n < 0  or  n > 20  */
public long factorial(int n)
{ long answer;
  if ( n < 0  ||  n > 20 )
       { throw new RuntimeException("factorial error: illegal input"); }
  else { if ( n == 0 )
              { answer = 1; }                    // 0! = 1
         else { answer = n * factorial(n - 1); } // (n+1)! = (n+1) * n!
       }
  return answer;
}
```

7.11.1 Loops and Recursions

Now that we understand the reason why the `factorial` function gives the answers it does, we can study its Java coding and ask if there is an alternative way to program the function. Indeed, because the coding of `factorial` invokes itself once, we can rearrange its statements into a for-loop. Here is the loop version of the function:

```
public long factorial(int n)
{ long answer;
  if ( n < 0  ||  n > 20 )
       { throw new RuntimeException("factorial error: illegal input"); }
  else { answer = 1;
         int count = 0;
         while ( count != n )
               // invariant:  answer == count!
               { count = count + 1;
                 answer = count * answer;
               }
       }
  return answer;
}
```

The loop mimicks the sequence of multiplications that compute the answer. Since the statements in the body of the loop bear no resemblance to the original recursive algorithm, we rely on the loop's invariant to understand why the loop computes the correct result.

In practice, many counting problems are best solved with recursive techniques. If the recursive solution has a simple form, like those seen in the cake-making and factorial examples, then it may be possible to rewrite the recursive implementation into a loop. But the next section shows counting problems where solutions with multiple recursions are needed; such solutions do not always convert to loop code.

7.11.2 Counting with Multiple Recursions

Some problems simply cannot be understood and resolved without recursive techniques, and here is a counting problem that makes this point.

Perhaps we are obsessed by insects and want to know how reproductive house flies are. Say that, in its lifetime, one house fly lays exactly two eggs. Each egg produces a fly that itself lays exactly two eggs. If this process occurs for `n` generations of egg-laying, how many flies result? (Assume that the flies never die.)

Because each fly gives birth to two more flies, which themselves give birth to even more, the pattern of counting cannot be done by a simple iteration—we must solve the problem by thinking about it as a counting problem that "decomposes" into two more counting problems.

An answer to the problem is that the total number of flies produced by the original fly is the *sum* of the flies produced by the first egg, plus the flies produced by the second egg, plus the original fly. If we say that the original fly is `n`-generations old, then its child flies are one generation younger—each are `n-1` generations old. We have this recursive equation:

```
flies_at_generation(n) = flies_at_generation(n-1) + flies_at_generation(n-1) + 1
```

A newly hatched fly is 0-generations old:

```
flies_at_generation(0) = 1
```

that is, the number of flies produced by a newly hatched fly is just the fly itself.

These two equations generate a Java method that contains *two* recursions:

```
public int fliesAtGeneration(int n)
{ int answer;
  if ( n < 0 )
       { throw new RuntimeException("error: negative argument"); }
  else { if ( n == 0 )  // is it a newly hatched fly?
               { answer = 1; }
         else { int first_egg_produces = fliesAtGeneration(n - 1);
                int second_egg_produces = fliesAtGeneration(n - 1);
                answer = first_egg_produces + second_egg_produces + 1;
               }
  return answer;
}
```

When executing the innermost else-clause, the computer computes the first recursion completely to its integer result before it starts the second recursive invocation. Because each recursive invocation uses an argument that is smaller than the one given to the caller method, all the recursions will terminate.

Now that we have used recursive problem solving, we can readily see that the two recursions in the method's innermost else-clause can be replaced by one:

```
int one_egg_produces = fliesAtGeneration(n - 1);
answer = (2 * one_egg_produces) + 1;
```

This simplifies the method and even allows us to rewrite the method's body into a loop, if we so desire.

Although the fly-counting example is a bit contrived, there is a related counting problem that is not: the *Fibonacci function*. The Fibonacci number of a nonnegative integer is defined in the following way:

```
Fib(0) = 1

Fib(1) = 1

Fib(n) = Fib(n-1) + Fib(n-2), when n >= 2
```

This recursively defined algorithm was proposed in the 13th century for counting the number of pairs of rabbits produced by one initial pair, assuming that a pair of rabbits takes one month to mature and from then on the pair produces one more pair of rabbits every month thereafter! Since that time, Fibonacci numbers have arisen in surprising places in problems in biology, botany, and mathematics.

The algorithm for the Fibonacci function has an easy coding as a method that uses two recursions in its body. It is a fascinating project to rewrite the method so that it uses only one recursion.

Exercises

1. Write an application that prints the values of `3!`, `6!`, `9!`, ..., `18!`.

2. Remove from the `factorial` method the first conditional, the one that tests `n < 0 || n > 20`. Then, try to compute `factorial(20)`, `factorial(21)`, `factorial(99)`, and `factorial(-1)`.

3. With a bit of thought, you can use a while-loop to program the recursive equations for factorial. Do this.

4. Implement the *Fibonacci function* with a method that contains two recursions, and try your function with the actual parameters 20; 30; 35; 40. Why does the computation go so slowly for the test cases?

(Incidentally, the Fibonacci function was proposed in the 13th century for counting the number of pairs of rabbits produced by one initial pair, assuming that a pair of rabbits takes one month to mature and from then on the pair produces one more pair of rabbits every month thereafter!)

5. Here is a recursive algorithm for computing the product of a sequence of nonnegative integers:

```
product(a, b) = 1, when b < a

product(a, b) = product(a, b-1) * b, when b >= a
```

   Write a recursively defined method that computes `product`.

6. Even an activity as simple as the addition of two nonnegative integers can be solved recursively in terms of just "-1" and "+1":

```
add(0, b) = b

add(a, b) = add(a-1, b) + 1, when a > 0
```

   (a) Write a recursively defined method that computes this definition.

   (b) In a similar style, define multiplication, `mult`, of two nonnegative integers in terms of "-1" and `add`.

   (c) Define exponentiation, `exp`, of two nonnegative integers in terms of "-1" and `mult`.

7. Ackermann's function is yet another famous recursive definition:

```
A(0, n) = n + 1

A(m, 0) = A(m-1, 1), when m > 0

A(m, n) = A(m-1, A(m, n-1)), when m > 0 and n > 0
```

   Write a recursively defined method based on this definition. The function is famous because its recursions cause the values of the the second parameter to zoom upwards while the first parameter slowly decreases, yet the function always terminates.

8. Say that a word is *sorted* if all its letters are distinct and are listed in alphabetical order, e.g., `"bdez"` is sorted but `"ba"` and `"bbd"` are not. Write a recursive algorithm that calculates the number of sorted words of length `n` or less that one can create from `n` distinct letters.

9. Solve this modified version of the fly-generation problem: Say that an "ordinary" fly lays exactly two eggs. One of the eggs produces another "ordinary" fly, but the other egg produces a "queen" fly, which can lay, for every generation thereafter, two eggs that hatch into ordinary flies.

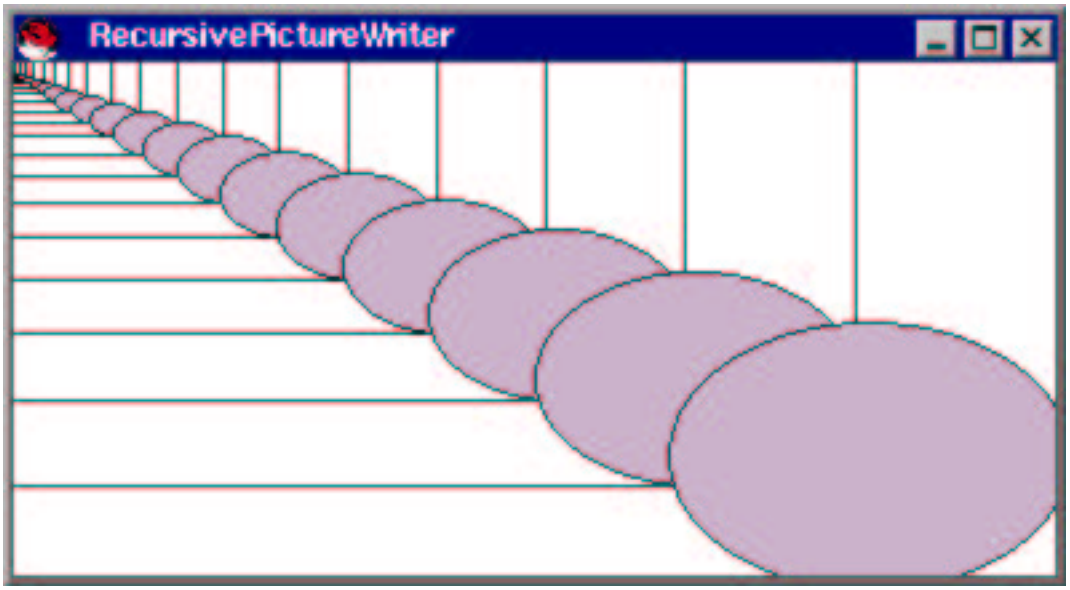
   Starting from one ordinary fly, how many flies are produced from it in `n` generations? Starting from one queen fly, how many flies are produced from it in `n` generations?

   (Hint: You will need one recursive algorithm that counts the flies produced from one ordinary fly and one recursive algorithm that counts the number of children flies produced from one queen fly.)

7.12 Drawing Recursive Pictures

Recursively defined methods are ideal for drawing beautiful recursive pictures.

Perhaps we wish to paint a picture of an egg resting in front of a picture of another egg resting in front of a picture of another egg, ..., until the eggs recede into nothingness:



You might give a painter a bucket of paint, a brush, and these recursive instructions: Paint a slightly smaller egg-picture in the background and next paint a big egg in the front!

More precisely stated, the algorithm goes as follows:

1. First, paint a completed an egg-picture, sized slightly smaller than the desired size of the final picture. (If this makes the background eggs have zero size, then don't bother with it.)

2. Next, paint the largest egg, of the desired size, in the foreground.

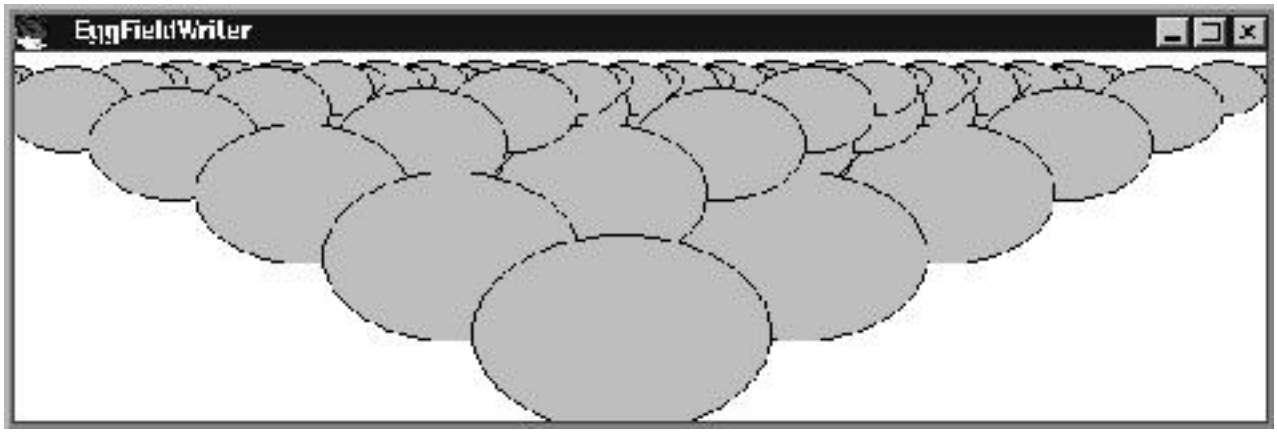
3. Finally, draw a border around the largest egg and its background.

Of course, Step 1 is a recursive reference, and an argument, `size`, will be used to determine how large to paint the picture. A recursive invocation will decrease `size` to draw the background picture, and the drawings of the background pictures terminate when `size` reaches 0.

Figure 20 shows the method, named `paintPicture`, that uses this algorithm.

When `paintPicture` draws a picture, it first sends a message to itself to paint a background picture at 0.8 size. The regress of drawing successively smaller background pictures halts when `paintPicture` notices that a background picture would have a 0-sized egg. In this way, the recursions "count down" to 0, and the quantity of recursive invocations *count how many background pictures must be drawn to fill the entire frame.* When the background pictures are drawn, from smallest to largest, the eggs are properly positioned, smallest to largest.

Figure 21 shows an ever more clever example, where a "field of eggs" is drawn by a method that invokes itself two times:



The work is done by method `paintEggField`, which paints two triangular fields of eggs, one to the left rear and one to the right rear, of a lead egg. The recursively defined method, `paintEggField`, uses parameter `layer` to count downwards by one each time the rear layers of eggs are painted. Because the left rear egg field is painted after the right rear field, the left field's eggs rest on the top of the right field's.

Exercises

1. Revise `class RecursivePictureWriter` by removing the border around each picture; next, move the eggs so that they rest in the left corner of the picture rather than the right corner.

Figure 7.20: class that paints a recursively defined picture

```
import java.awt.*;  import javax.swing.*;
/** RecursivePictureWriter displays a recursively defined picture of an
  * egg in front of a picture (of eggs). */
public class RecursivePictureWriter extends JPanel
{ private double scale = 0.8;  // the size of the background picture relative
                               //   to the foreground
  private int width = 400;     // frame width and depth
  private int depth = 200;
  private int first_egg = 150; // size of the egg in the foreground

  /** Constructor RecursivePictureWriter creates the window */
  public RecursivePictureWriter()
  { JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("RecursivePictureWriter");
    my_frame.setSize(width, depth);
    my_frame.setBackground(Color.white);
    my_frame.setVisible(true);
  }

  /** paintComponent paints the recursively defined picture
    * @param g - the ``graphics pen'' */
  public void paintComponent(Graphics g)
  { g.setColor(Color.white);
    g.fillRect(0, 0, width, depth);  // paint the background
    paintPicture(width, depth, first_egg, g);
  }

  /** paintPicture paints a picture of an egg in front of a picture of eggs.
    * @param right_border - the right border of the picture painted
    * @param bottom - the bottom border of the picture painted
    * @param size - the size of the egg to be painted in the foreground
    * @param g - the graphics pen  */
 private void paintPicture(int right_border, int bottom,
                           int size, Graphics g)
  { // paint the background picture first:
    int background_size = (int)(size * scale);
    if ( background_size > 0 )  // is the background worth painting?
       { paintPicture((int)(right_border * scale), (int)(bottom * scale),
                      background_size, g);
       }
    // paint an egg in the foreground:
    paintAnEgg(right_border - size, bottom, size, g);
    g.setColor(Color.black);
    g.drawRect(0, 0, right_border, bottom);  // draw the border
  }
 ...
```

Figure 7.20: class that paints a recursively defined picture (concl.)

```
  /** paintAnEgg  paints an egg in 2-by-3 proportion
    * @param left_edge - the position of the egg's left edge
    * @param bottom - the position of the egg's bottom
    * @param width - the egg's width
    * @param g - the graphics pen  */
  private void paintAnEgg(int left_edge, int bottom, int width, Graphics g)
  { int height = (2 * width) / 3;
    int top = bottom - height;
    g.setColor(Color.pink);
    g.fillOval(left_edge, top, width, height);
    g.setColor(Color.black);
    g.drawOval(left_edge, top, width, height);
  }

  public static void main(String[] args)
  { new RecursivePictureWriter(); }
}
```

2. Write a class that generates a circle of diameter 200 pixels, in which there is another circle of 0.8 size of the first, in which there is another circle of 0.8 size of the second, ..., until the circles shrink to size 0.

3. To better understand `class EggFieldWriter`, do the following. First, replace method `paintAnEgg` with this version, which draws a transparent egg:

   ```
   private void paintAnEgg(int left_edge, int bottom, int width, Graphics g)
   { int height = (2 * width) / 3;
     int top = bottom - height;
     g.setColor(Color.black);
     g.drawOval(left_edge, top, width, height);
   }
   ```

   Retry `EggFieldWriter`.

4. Write a recursive definition, like that for the factorial and Fibonacci functions, that defines precisely how many eggs are drawn in an egg field of `n` layers. How many eggs are drawn by the output view in Figure 6?

7.13 Summary

This chapter presented concepts about repetition:

Figure 7.21: a field of eggs

```
import java.awt.*;  import javax.swing.*;
/** EggFieldWriter displays a binary tree depicted as a ‘‘field of eggs’’ */
public class EggFieldWriter extends JPanel
{ private int size = 20;  // size for eggs
  private int total_layers_of_eggs = 7; // how many layers to paint
  private int frame_width = 600;
  private int frame_height = 200;

  /** Constructor EggFieldWriter creates the window and makes it visible */
  public EggFieldWriter()
  { JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("EggFieldWriter");
    my_frame.setSize(frame_width, frame_height);
    my_frame.setBackground(Color.white);
    my_frame.setVisible(true);
  }

  /** paintComponent fills the window with the field of eggs
    * @param g - the ‘‘graphics pen’’ */
  public void paintComponent(Graphics g)
  { paintEggField(220, 200, total_layers_of_eggs, g); }

  /** paintEggField paints two fields of eggs behind a lead egg
    * @param left_border - the left edge of the lead egg in the field
    * @param baseline - the bottom of the lead egg
    * @param layer - how many layers of eggs to draw
    * @param g - the graphics pen  */
  private void paintEggField(int left_border, int baseline, int layer,
                             Graphics g)
  { if ( layer > 0 )
       { int egg_size = size * layer; // invent a size for the lead egg
         // paint right sub-egg field:
         paintEggField(left_border + ((2 * egg_size) / 3),
                       baseline - (egg_size / 3), layer - 1, g);
         // paint left sub-egg field:
         paintEggField(left_border - (egg_size / 2),
                       baseline - (egg_size / 3), layer - 1, g);
         // paint leading egg:
         paintAnEgg(left_border, baseline, egg_size, g);
       }
  }
  // see Figure 20 for method  paintAnEgg
}
```

New Constructions

- *while-statement* (from Figure 1):

```
int total = 0;
int count = 0;
while ( count != n )
      { System.out.println("count = " + count + "; total = " + total);
        count = count + 1;
        total = total + count;
      }
```

- *for-statement*:

```
int total = 0;
for ( int count = 1;  count <= n;  count = count + 1 )
    { total = total + count;
      System.out.println("count = " + count + "; total = " + total);
    }
```

New Terminology

- *loop*: a statement (or sequence of statements) that repeats computation, such as a while-statement
- *iteration*: repetitive computation performed by a loop; also refers to one execution of a loop's body
- *definite iteration*: iteration where the number of the loop's iterations is known the moment the loop is started
- *indefinite iteration*: iteration that is not definite
- *nontermination*: iteration that never terminates; also called "infinite looping"
- *invariant property*: a logical (true-false) fact that holds true at the start and at the end of each iteration of a loop; used to explain the workings of a loop and argue the loop's correctness
- *loop counter*: a variable whose value remembers the number of times a loop has iterated; typically used in the loop's test to terminate iteration. Also called a *loop control variable*.
- *recursive definition*: a method that invokes itself, directly or indirectly; used to solve a problem by recursive invocations to first solve simpler versions of the problem and next combine the results to compute overall result.

Points to Remember

- While-loops are used in two forms:
  - *definite iteration*, where the number of times the loop repeats is known when the loop is started. A standard pattern for definite iteration reads,

```
int count = INITIAL VALUE;
while ( TEST ON count )
      { EXECUTE LOOP BODY;
        INCREMENT count;
      }
```

    where `count` is the loop control variable. The *for-statement* is used to tersely code the above pattern:

```
for ( int count = INITIAL VALUE; TEST ON count; INCREMENT count; )
    { EXECUTE LOOP BODY (DO NOT ALTER count); }
```

  - *indefinite iteration*, where the number of repetitions is not known. The pattern for indefinite iteration of input processing reads

```
boolean processing = true;  // this variable announces when it's time to stop
while ( processing )
      { READ AN INPUT TRANSACTION;
        if ( THE INPUT INDICATES THAT THE LOOP SHOULD STOP )
             { processing = false; }  // reset  processing  to stop loop
        else { PROCESS THE TRANSACTION; }
      }
```

    and the pattern for indefinite iteration for searching a "set" of items reads

```
boolean item_found = false;
DETERMINE THE FIRST ``ITEM'' TO EXAMINE FROM THE SET;
while ( !item_found  &&  ITEMS REMAIN IN THE SET TO SEARCH )
      { EXAMINE ITEM FROM THE SET;
        if ( THE ITEM IS THE DESIRED ONE )
             { item_found = true; }
        else { DETERMINE THE NEXT ITEM TO EXAMINE FROM THE SET; }
      }
```

- Inside the design of every loop is an *invariant property*, which states what the loop is accomplishing as it progresses towards completion. When you design a loop, you should also write what you believe is the loop's invariant property; this clarifies your thinking and provides excellent documentation of the loop.
- Use a recursively defined method when a problem is solved by using the solution to a slightly simpler/smaller-valued version of the same problem. Write the

recursively defined method so that it uses a parameter whose value becomes simpler/smaller each time the method invokes itself.

7.14 Programming Projects

1. Write these methods for comparing strings and insert them into an application that lets a user ask questions about strings.

   (a)
   ```
   /** palindrome decides whether a string is a palindrome, that is,
     * the same sequence of letters spelled forwards or backwards.
     * (e.g., "abcba" is a palindrome, but "abccda" is not.)
     * @param s - the string to be analyzed
     * @return whether s is a palindrome */
   private static boolean palindrome(String s)
   ```

   (b)
   ```
   /** permutation decides whether one string is a permutation of another,
     * that is, the two strings contain exactly the same letters in the
     * same quantities, but in possibly different orders.
     * (e.g., "abcd" and "abdc" are permutations, but "abcd" and "aabdc" are not)
     * @param s1 - the first string
     * @param s2 - the second string
     * @return whether s1 and s2 are permutations of each other */
   private static boolean permutation(String s1, String s2)
   ```

   (Note: your efforts will be eased if you consult the Java API for `java.lang.String` and learn about the `indexOf` method for strings.)

2. Write an application that reads a series of nonnegative integers and prints the list of all the nonzero divisors of the input integer. (Recall that `b` is a *divisor* of `a` if `a` divided by `b` yields a remainder of zero.) The program terminates when a negative integer is typed; it should behave like this:

   ```
   Type a positive integer: 18
   The divisors are: 2 3 6 9
   Type a positive integer: 23
   The divisors are: none---it's a prime
   Type a positive integer: -1
   Bye!
   ```

3. Write an application that calculates the odds of a person winning the lottery by guessing correctly all `m` numbers drawn from a range of `1..n`. The formula that calculates the probability is

   ```
   probability = m!  / product((n-m)+1, n)
   ```

where `m!` is the factorial of `m` as seen earlier, and `product(a,b)` is the iterated product from `a` up to `b`, which is defined as follows:

```
product(a, b) = a * (a+1) * (a+2) * ...  * b
```

(*Footnote:* Here is the rationale for the probability formula. Consider the task of matching 3 chosen numbers in the range 1..10; the chances of matching on the first number drawn at the lottery is of course 3 in 10, or 3/10, because we have 3 picks and there are 10 possible values for the first number. Assuming success with the first draw, then our chances for matching the second number drawn are 2/9, because we have 2 numbers left and the second number is drawn from the 9 remaining. The chance of matching the third number drawn is 1/8. In total, our chances of matching all three numbers are (3/10) * (2/9) * (1/8), which we reformat as (1*2*3) / (8*9*10). A pattern appears: The odds of picking m numbers out of the range *1..n* are (1*2*...*m) / ((n-m)+1 * (n-m)+2 *...*n) which is succinctly coded as the formula presented above. *End Footnote.*)

4. Write an application that can translate between Arabic numbers (ordinary non-negative integers) and Roman numerals. (Recall that Roman numeral `I` denotes 1, `V` denotes 5, `X` denotes 10, `L` denotes 50, and `C` denotes 100. The symbols of a numeral should be in nonincreasing order, e.g. `LXXVIII` is 78, but an out-of-order symbol can be used to *subtract* from the symbol that follows it, e.g., `XCIV` is 94.)

   The application must be able to translate any Arabic numeral in the range 1 to 399 into a Roman numeral and vice versa. (Recall that no letter in a roman number can appear more than three times consecutively, e.g., you must write `IX` rather than `VIIII` for 9.)

5. Write an application that implements *pattern matching* on strings: The program's inputs are `s`, a string to be searched, and `p`, a pattern. The program searches for the leftmost position in `s` where `p` appears as a substring. If such a position is found, the program displays the index in `s` where `p` begins; if `p` cannot be found in `s`, the answer displayed is -1.

   Here are some hints: If you did pattern matching with pencil and paper, you would align the pattern underneath the string and check for a match of the alignment. For example, for string `"abcdefg"` and pattern `"cde"`, you begin:

   ```
   s ==  a b c d e f g
   p ==  c d e
   current position == 0
   ```

   The "current position" remembers the index where the pattern is aligned. Since the match fails, the search moves one character:

```
s ==  a b c d e f g
p ==    c d e
current position == 1
```

The process proceeds until a match succeeds or the pattern cannot be shifted further.

6. Write a method that computes the value of PI by using this equation:

```
PI = 4 * ( 1 - 1/3 + 1/5 - 1/7 + ... )
```

The parameter to the method is the maximum value of the denominator, `n`, used in the fractions, `1/n`, in the series. Insert the method in a test program and try it on various parameters and compare your results with the value of the Java constant `Math.PI`.

7. (a) Design a calendar program, which takes two inputs: a month (an integer in range 1..12) and a year (for simplicity, we require an integer of 1900 or larger) and presents as its output the calendar for the month and year specified. We might use the program to see the calendar for December, 2010:

```
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

The application's model will need methods to calculate how many days are in a given month (within a given year), on what day of the week a given month, year begins, whether a year is a leap year, etc.

(b) Next, revise the application so that, on demand, the calendar prints a proper French calendar, that is, the days are presented Monday-Tuesday-...-Sunday (lundi-mardi-mercredi-jeudi-vendredi-dimanche) with the labels, `lu ma me je ve sa di`

8. Write a program that implements the word game, "hangman." Two people play: The first inserts a "secret word" into the game, and the second tries to guess the secret word, one letter at a time, before six wrong guesses are made. After the first player has typed the secret word (e.g., `apple`), the second sees the following on the console:

```
Pick a letter:
```

The second player guesses a letter, e.g., `e`. The program replies whether or not the guess was correct:

```
Pick a letter:e
Correct!
   --
  |  |
  |
  |
  |
_ _ _ _ e
```

The program displays the partially guessed word and a "hangman's pole" upon which the wrong guesses, drawn as a stick figure, are compiled. The game continues with the second player making guesses, one by one, until either the word is guessed correctly, or the six-piece stick figure is displayed:

```
Pick a letter:x
   --
  |  |
  |  0
  | /|\
  |  /\
 _ p p _ e
You lose--the word was "apple"!
```

(Note: this program can be written with judicious use of the `charAt` and `substring` methods for strings; see Table 9, Chapter 3, for details about these methods.)

9. Euclid's algorithm for calculating the greatest common divisor of two nonnegative integers is based on these laws:

   - `GCD(x,y) = x, if x equals y`
   - `GCD(x,y) = GCD(x-y,y), if x > y`
   - `GCD(x,y) = GCD(x,y-x), if y > x`

   Implement these equations with a method that accepts x and y as parameters and returns as its result the greatest common divisor. Write the method with a while-loop first. Next, write the method with recursive message sending.

10. Newton's approximation method for square roots goes as follows: The square root of a double, `n`, can be approximated as the limit of the sequence of values, n_i, where

- $n_0 = n$
- $n_{i+1} = ((n/n_i) + n_i)/2$

Write a method that takes as its parameters two doubles: the number `n` and a precision, `epsilon`. The method prints the values of n_i until the condition $|n_{k+1} - n_k| < epsilon$ holds for some $k > 0$. The result of the method is n_{k+1}. (Hint: the Java operation `Math.abs(E)` computes `|E|`, the absolute value of `E`.) Insert this method into a test program and try it with various values of `epsilon`.

11. Recall yet again this formula for the monthly payment on a loan of principal, p (double), at annual interest rate, i (double), for a loan of duration of y (int) years:

$$annual\_payment = \frac{(1+i)^y * p * i}{(1+i)^y - 1}$$

$$monthly\_payment = annual\_payment/12.0$$

The monthly payment history on a loan of principal, `p`, at annual interest rate, `i`, is computed as follows:

- $p_0 = p$
- $p_{j+1} = ((1 + (i/12))p_j) - monthly\_payment$

for $j >= 0$.

Use these formulas to write an application that that calculates the monthly payment for a loan and prints the corresponding payment history. Test your application on various inputs, and explain why the principal is completely repaid a bit sooner than exactly `y` full years.

12. Write an output-view class that contains these two methods for formatting numbers. The first is

```
/** format  formats an integer for display.
  * @param formatcode - the format in which the integer should print.
  *   The variants are:  "n"  - format the integer to have length  n  characters
  *                      "n.m" - format the integer with length  n  characters
  *   followed by a decimal point, followed by  m  zeroes.
  * (Pad with leading blanks if the integer needs less than  n  characters.)
  * Note: if the integer is too large to fit within the formatcode, the
  * formatcode is ignored and the entire integer is used.
  * @param i - the integer to be formatted
  * @return a string containing  i,  formatted according to the formatcode */
public String format(String formatcode, int i)
```

The second method is also called `format` and uses the following interface:

```
/** format  formats a double for display.
  * @param formatcode - the format in which the double should print.
  *   The format must have form "n1.n2" -- format the double using  n1
  *   characters for the whole number part, followed by a decimal point,
  *   followed by  n2  characters for the fraction part.
  * (Pad with leading blanks if the whole part of the double needs less than
  *  n1  characters.) Note: if the whole number part of the double is too large
  * to fit within formatcode, the entire whole number part is included.
  * @param d - the double to be formatted
  * @return a string containing  d, formatted according to the  formatcode */
public void printf(String formatcode, double d)
```

Insert the two methods in a class called `Formatter` and use the class to print nicely formatted output, e.g.,

```
Formatter f = new Formatter();
int i = ... read an input integer ...;
System.out.println("For input, " +  f.format("3", i));
System.out.println("The square root is " +  f.format("2.3", Math.sqrt(i)));
```

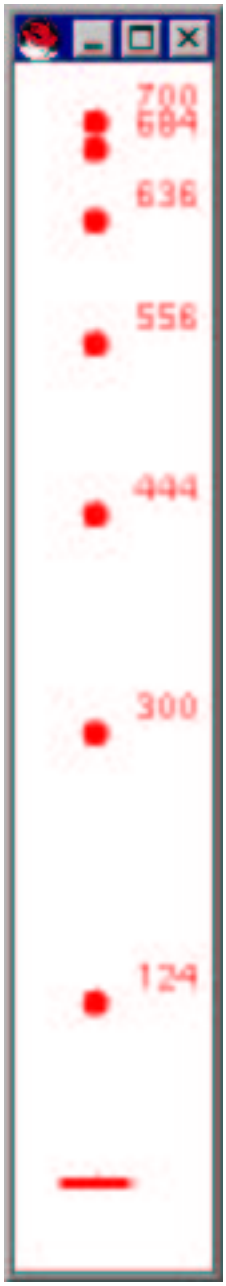
Don't forget to print a leading negative sign for negative numbers.

13. Write a program that lets two players play a game of tic-tac-toe (noughts and crosses). (Hint: the model simulates the the game board, whose internal state can be represented by three strings, representing the three rows of the board. Write methods that can insert a move into the board and check for three-in-a-row. (Note: use the `charAt` and `substring` methods for strings; see Table 9, Chapter 3 for details about `substring`.)

    It is simplest to build the game for only one player at first; then add a second player. Next, revise the game so that the computer can compete against a human player.

14. Program an animation of a falling egg. The animation will show, at correct

velocity, an egg falling from a great height and hitting the ground.



The egg's fall is calculated by this famous formula from physics:

$$height = I0 - (0.5 * g * t^2)$$

where $I0$ is the egg's initial height (in meters), t is the time (in seconds) that the egg has been falling, and g is the acceleration due to gravity, which is $9.81 meters/second^2$.

The animation should let its user specify the initial height from which the egg is dropped. When the egg reaches height 0, it must "smash" into the ground.

15. Program an animation of a cannon ball shot into the air: The position, `x,y`, of the cannon ball in the air after `t` seconds have elapsed is defined by these formulas:

$$x = initial\_velocity * cosine(radians\_angle) * t$$

$$y = (initial\_velocity * sine(radians\_angle) * t) - ((gravity * t^2)/2)$$

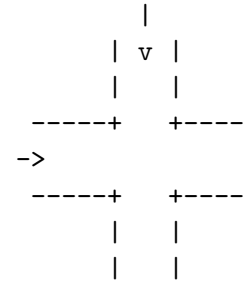
where *initial_velocity* is the velocity of the ball when it first leaves the cannon, *gravity* is the pull of gravity, and *radians_angle* is computed as $radians\_angle = (degrees * PI)/180$ *degrees* is the angle that the cannon was pointed when it shot the cannon ball.

Your application should let the user experiment with different initial velocities, degrees angles, and gravitational constants.

16. Make the cannon-ball animation into a game by drawing a bucket (pail) at a random spot near the end of the graphics window and letting the user adjust the velocity and angle of the cannon to try to make her cannon ball fall into the bucket.

17. Program an animation of a clock—every second, the clock redisplays the time. (To start, program a digital clock. Next, program a clock with a face and three moving hands; see Chapter 4 for advice at drawing clock hands.)

18. Program a simulation of traffic flow at a traffic intersection:



For simplicity, assume that traffic on one street travels only from north to south and traffic on the other street travels only from west to east. The simulation maintains at most one car at a time travelling from north to south and at most one car at a time travelling from east to west. When one car disappears from view, another car, travelling in the same direction, may appear at the other end of the street. All cars travel at the same velocity of 10 meters (pixels) per second.

Assume that both roadway are 90 meters in length and that the intersection has size 10-by-10 meters.

Build the simulation in stages:

(a) First, generate north-south cars only: A new north-south car appears within a random period of 0-2 seconds after the existing north-south car disappears from view. Next, add a counter object that counts how many cars complete their journey across the intersection in one minute.

(b) Next, generate east-west cars: A new east-west car appears within a random period of 0-5 seconds after the existing one disappears. Count these cars as well. (Do not worry about collisions at the intersection.)

(c) Now, program the cars so that a car must stop 20 meters (pixels) before the intersection if a car travelling in another direction is 20 meters or closer to entering the intersection. (If both cars are equi-distant from the intersection, the east-west car—the car on the "right"—has right-of-way.) Count the total flow of cars in one minute.

(d) Experiment with stop signs at the intersection: A stop sign causes a car to stop for 1 second, and if another car is within 20 meters of the intersection at the end of the 1 second, the stopped car must wait until the moving car crosses the intersection. Place a stop sign on just the east-west street and count the total traffic flow for one minute. Do the same for a stop sign just on the north-south street.

(e) Finish the simulation with two stop signs; count cars.

Which arrangement of stop signs, if any, caused optimal traffic flow?

7.15 Beyond the Basics

7.15.1 Loop termination with `break`

7.15.2 The `do-while` *Loop*

7.15.3 Loop Invariants

7.15.4 Loop Termination

7.15.5 More Applets

These optional sections expand upon the materials in the chapter. In particular, correctness properties of loops and recursively defined methods are studied in depth.

7.15.1 Loop Termination with `break`

A loop that does indefinite iteration must terminate at some point, and often the decision to terminate is made in the middle of the loop's body. Java provides a statement named `break` that makes a loop quit at the point where the termination decision is made.

For example, Figure 5 showed how the `findChar` method used indefinite iteration to search for the leftmost occurrence of a character in a string. Once the character is located, a boolean variable, `found`, is set to `true`, causing the loop to quit at the beginning of the next iteration. We can terminate the loop immediately by using a `break` statement instead of the assignment, `found = true`:

```
public int findChar(char c, String s)
{ int index = 0;  // where to look within  s  for  c
  while ( index < s.length() )
        // invariant: at this program point,
        //   c  is not any of chars 0..(index-1) in  s
        { if ( s.charAt(index) == c )
               { break; }  // exit loop immediately
          else { index = index + 1; }
        }
  // If the break executed, this means  index < s.length()  is still true
  //  at this point and  that  s.charAt(index) == c.
  if ( !(index < s.length()) )  // did the loop terminate normally?
     { index = -1; }            // then  c  is not found in  s
  return index;
}
```

The `break` statement causes the flow of control to move immediately from the middle of the loop to the first statement that follows the loop. For this reason, the `found` variable no longer controls loop termination. This simplifies the loop's test but forces us to write a more complex conditional that follows the loop.

When a `break` statement is inserted into a loop, it means that the loop has more than one way of exiting—it can exit by failure of its test and by execution of the `break`. For this reason, *a* `break` *statement should be used sparingly and with caution:*

insert a `break` *only in a position where the reason for loop exit is perfectly clear*

In the above example, the `break` occurs exactly when `s.charAt(index) == c`, which is a crucial property for describing what the loop does. This property is crucial to the the conditional statement that follows the loop. *If you are not completely certain about inserting a* `break` *statement into a loop, then don't do it.*

Finally, note that a `break` exits only *one* loop's body. For example, the following nested loop appears to make the values of variables `i` and `j` vary from 0 to 3. But the `break` statement in the body of the inner loop causes that loop to terminate prematurely:

```
int i = 0;
while ( i != 4 )
      { int j = 0;
        while ( j != 4 )
              { if ( i + j == 3 )
                   { break; }
                j = j + 1;
              }
        System.out.println("i = " + i + ", j = " + j);
        i = i + 1;
      }
```

The loops print

```
i = 0, j = 3
i = 1, j = 2
i = 2, j = 1
i = 3, j = 0
```

The above example should make clear that a `break` statement can make a program more difficult to understand.

7.15.2 The `do-while` Loop

The Java language provides a variant of while-loop that performs its test at the end of the loop's body, rather than the beginning; the new loop is a `do-while`-statement:

```
do { S } while ( E );
```

and its semantics states

1. The body, `S`, executes.
2. The test, `E`, computes to a boolean answer. If `false`, the loop is finished; if `true`, Steps 1 and 2 repeat.

The do-while-statement can be convenient for programming repetitive tasks where there is at least one repetition. Input-transition processing is one such example, and its pattern can be written with a do-while-statement:

```
boolean processing = true;
do { READ AN INPUT TRANSACTION;
     if ( THE INPUT INDICATES THAT THE LOOP SHOULD STOP )
          { processing = false; }
     else { PROCESS THE TRANSACTION; }
   }
while ( processing );
```

7.15.3 Loop Invariants

The two crucial questions of loop design are:

1. What is the purpose of the loop's body?
2. How does the loop's test ensure termination?

We study the first question in this section and the second in the next.

A loop must achieve its goal in multiple steps, and surprisingly, one understands a loop better by asking about what the loop accomplishes in its steps towards its goal, rather than asking about the ultimate goal itself. Precisely, one must answer the following question, ideally in one sentence:

Say that the loop has been iterating for some time; what has it accomplished so far?

The answer to this question is called a loop's *invariant*.

The term "invariant" is used to describe the answer, because the answer must be a property that holds true whether the loop has executed for 2 repetitions, or 20 repetitions, or 200 repetitions, or even 0 repetitions—the answer must be invariant of the number of repetitions of the loop.

A loop invariant also reveals what the loop's ultimate goal will be: Once the loop terminates, it will be the case that *(i)* the invariant is still true, and *(ii)* the loop's test has evaluated to false. These two facts constitute the loop's total achievement.

Let's consider several examples. Here is a loop:

```
int n =  ...; // read integer from user
int i = 0;
while ( i != (n + 1) )
      { System.out.println(i * i);
        i = i + 1;
      }
```

Answer the question: *Say that the loop has been iterating for some time; what has it accomplished so far?* An informal but accurate answer is, "the loop has been printing squares." Indeed, this is an invariant, and it is useful to document the loop exactly this way:

```
while ( i != (n + 1) )
      // the loop has been printing squares
      { System.out.println(i * i);
        i = i + 1;
      }
```

Someone who is more mathematically inclined might formulate a more precise answer: The loop has been printing the squares `0*0, 1*1, 2*2`, etc. Indeed, the value of its counter variable, `i`, states precisely how many squares have been printed:

```
while ( i != (n + 1) )
      // the loop has printed the squares  0*0, 1*1, ...up to... (i-1)*(i-1)
      { System.out.println(i * i);
        i = i + 1;
      }
```

This is also an invariant, because whether the loop has iterated 2 times, 20 times, etc., the invariant states exactly what has happened. For example, after 20 iterations, `i` has value 21, and indeed the loop has printed from 0*0 up to 20*20. (If the loop has iterated zero times, `i` has value 0 and the invariant says the loop has printed from 0*0 up to -1*-1, that is, it has printed nothing so far.)

When the loop terminates, its test, `i != (n + 1)`, evaluates to false. Hence, `i` has the value, `n+1`. But we know that the invariant is still true, and by substituting `n+1` for `i`, we discover that the loop has printed exactly the squares from `0*0` up to `n*n`. This exposes the ultimate goal of the loop.

Next, reconsider the summation example and its proposed invariant:

```
int n =  ... ;  // read input value
int total = 0;  int i = 0;
while ( i != n )
      // proposed invariant: total == 0 + 1 + ...up to... + i
      { i = i + 1;
        total = total + i;
      }
```

How do we verify that the invariant is stated correctly? To prove an invariant, we have two tasks:

1. *basis step*: we show the invariant is true the very first time the loop is encountered.

2. *inductive step*: We assume the invariant is already holding true at the start of an arbitrary iteration of the loop, and we show that when the iteration completes the invariant is still true.

Such a proof style is known as *proof by induction* on the number of iterations of the loop.

The proof of the invariant for the summation loop is an induction:

1. When the loop is first encountered, both `total` and `i` have value 0, therefore it is true that `total == 0 + ...up to...  + 0`.

2. Next, we assume the invariant is true at the start of an arbitary iteration:

   ```
   total_start == 0 + 1 + ...up to... + i_start
   ```

and we show that the statements in the loop's body update the invariant so that it holds true again at the end of the iteration:

```
total_next == 0 + 1 + ...up to... + i_next
```

We use `V_start` to represent the value of variable `V` at the start of the iteration and `V_next` to represent the new value `V` receives during the iteration.

Here is the proof:

- Because of `i = i + 1`, we know that `i_next == i_start + 1`, and because of `total = total + i`, we know that `total_next = total_start + i_next`
- Use algebra to add `i_next` to both sides of the starting invariant: `total_start + i_next == (0+1+...up to...+ i_start) + i_next`.
- Using the definitions of `total_next` and `i_next`, we substitute and conclude that `total_next == 0+1+...up to...+ i_next`

We conclude, no matter how long the loop iterates, the invariant remains true.

The previous examples were of definite iteration loops, where the pattern of invariant takes the form

```
int i = INITIAL VALUE;
while ( TEST ON i )
      // all elements in the range INITIAL VALUE ...up to... (i-1)
      //  have been combined into a partial answer
      { EXECUTE LOOP BODY;
        i = i + 1;
      }
```

In the case of input processing, the natural invariant for the transaction-processing loop is simply,

```
boolean processing = true;
while ( processing )
      // every transaction read so far has been processed correctly
      { READ AN INPUT TRANSACTION;
        if ( THE INPUT INDICATES THAT THE LOOP SHOULD STOP )
             { processing = false; }
        else { PROCESS THE TRANSACTION; }
      }
```

Searching loops often have invariants in two parts, because the loops can exit in two different ways; the general format is

```
boolean item_found = false;
DETERMINE THE FIRST ‘‘ITEM’’ TO EXAMINE FROM THE COLLECTION;
while ( !item_found  &&  ITEMS REMAIN IN THE COLLECTION TO SEARCH )
      // Clause 1:  item_found == false  implies the DESIRED ITEM is not found
      //  in that part of the COLLECTION examined so far
      // Clause 2:  item_found == true  implies the DESIRED ITEM is ITEM
      { EXAMINE ITEM FROM THE COLLECTION;
        if ( ITEM IS THE DESIRED ONE )
             { item_found = true; }
        else { DETERMINE THE NEXT ITEM TO EXAMINE FROM THE COLLECTION; }
      }
```

Clauses 1 and 2 are necessary because the value of `item_found` can possibly change from `false` to `true` within an iteration.

If you write a loop and you "know" what the loop does, but you find it difficult to state its invariant, then you should generate execution traces that exhibit values of the loop's variables and examine the relationships between the variable's values at the various iterations. These relationships often suggest an invariant. For example, say that we believe this loop computes multiplication of two nonnegative integers `a` and `b` via repeated additions:

```
int i = 0;
int answer = 0;
while ( i != a )
      { answer = answer + b;
        i = i + 1;
      }
```

The question is: *Say that the loop has been iterating for some time; what has it accomplished so far?* We can use the debugging tool of an IDE to print an execution trace, or we might make the loop generate its own trace with a `println` statement:

```
int i = 0;
int answer = 0;
while ( i != a )
      { System.out.println("i=" + i + "  a=" + a + " b=" + b
                           + " c=" + c + " answer=" + answer);
        answer = answer + b;
        i = i + 1;
      }
```

When `a` is initialized to 3 and `b` is initialized to 2, we get this trace:

```
i=0 a=3 b=2 answer=0
i=1 a=3 b=2 answer=2
i=2 a=3 b=2 answer=4
i=3 a=3 b=2 answer=6
```

We see that `i * b = answer` holds at the start of each iteration. This is indeed the crucial property of the loop, and when the loop terminates, we conclude that `i = a` and therefore `a * b = answer`.

These examples hint that one can use loop invariants and elementary symbolic logic to construct formal proofs of correctness of computer programs. This is indeed the case and unfortunately this topic goes well beyond the scope of this text, but a practical consequence of invariants does not: *A programmer who truly understands her program will be capable of stating invariants of the loops it contains.* As a matter of policy, the loops displayed in this text will be accompanied by invariants. Sometimes the invariant will be stated in mathematical symbols, and often it will be an English description, but in either case it will state the property that remains true as the iterations of the loop work towards the loop's goal.

Exercises

State invariants for these loops and argue as best you can that the invariants remain true each time the loop does an additional iteration:

1. Multiplication:

```
int a = ...;  int b =  ...;
int i = b;  int answer = 0;
while ( !(i == 0 ) )
      { answer = answer + a;
        i = i - 1;
      }
System.out.println(answer);
```

2. Division:

```
int n = ...;  int d = ...;
if ( n >= 0  &&  d > 0 )
   { int q = 0;
     int rem = n;
     while ( rem >= d )
           { q = q + 1;
             rem = rem - d;
           }
   }
```

3. Exponentiation:

```
int n = ...;  int p = ...;
if ( p >= 0 )
     { int i = 0;
       int total = 1;
       while ( i != p )
             { total = total * n;
               i = i+1;
             }
     }
else { ... };
```

4. Search for character, 'A':

```
String s = ... ;
boolean item_found = false;
int index = 0;
while ( !item_found  && (index < s.length()) )
      { if ( s.charAt(index) == 'A' )
             { item_found = true; }
        else { index = index + 1; }
      }
```

5. String reverse:

```
String s = ...;
String t = "";
int i = 0;
while ( i != s.length() )
      { t = s.charAt(i) + t;
        i = i + 1;
      }
```

7.15.4 Loop Termination

How do we write the test of a loop so that it ensures that the loop *must* terminate? Imagine that every loop comes with an "alarm clock" that ticks downwards towards 0—when the clock reaches 0, the loop *must terminate*. (The loop might terminate before the alarm reaches 0, but it cannot execute further once the alarm reaches 0.) Our job is to show that a loop indeed has such an "alarm clock."

Usually, the alarm clock is stated as an arithmetic expression, called a *termination expression*. Each time the loop executes one more iteration, the value of the termination expression must decrease, meaning that eventually it must reach zero. Consider the loop that multiplies nonnegative integers `a` and `b`:

```
int i = 0;  int answer = 0;
while ( i != a )
      { answer = answer + b;
        i = i + 1;
      }
```

The termination expression is `a - i`, because each time the loop completes an iteration, the value of `a - i` decreases. Since `a` is a nonnegative number, at some point `i`'s value will equal `a`'s, and the termination expression's value reaches 0, and at this very moment, the loop does indeed terminate. (Again, we require that `a` and `b` are *nonnegative* integers.)

Definite iteration loops use termination expressions that compare the value of the loop counter to a stopping value, because the increment of the loop counter within the loop's body makes the counter move closer to the stopping value.

Searching loops ensure termination by limiting their search to a finite collection. In the case of the example in Figure 4, where the loop searches for divisors, the termination expression is `current - 1`, because variable `current` is searching for possible divisors by counting from `n/2` down to 1. When the termination expression reaches 0, it forces the loop's test to go false.

Unfortunately, not all loops have such termination expressions; a loop that implements a divergent infinite series clearly lacks such an "alarm clock." An even simpler example is a loop that reads a sequence of transactions submitted by a user at the keyboard—there is no termination expression because there is no guarantee that the user will tire and terminate the sequence of inputs.

Exercises

For each of the loops in the previous exercise set, state the conditions under which each is guaranteed to terminate and explain why.

7.15.5 More Applets

Part of the fun in writing animations and drawing recursive pictures is displaying them. As noted at the end of Chapter 4, we can use an applet to display an output-view object within a web page.

Because they are designed for use only with graphical-user-interface (GUI) controllers,

Begin Footnote: We study these in Chapter 10. *End Footnote*

Java applets do *not* execute the controller objects we have written so far. So, if we wish to display this Chapter's moving-ball animation as an applet, we must rewrite the controller in Figure 16.

First, we review the key modifications to make an output-view class into a Java applet:

Figure 7.22: output-view class for animation applet

```
import java.awt.*;
import javax.swing.*;
/** AnimationApplet contains the start-up class, controller,
    and output-view of the moving ball animation.  */
public class AnimationApplet extends JApplet
{ private BoxWriter box_writer;    // the output-view of the box
  private BallWriter ball_writer;  // the output-view of the ball in the box

  private MovingBall ball;  // the (address of the) moving ball object

  // this was formerly the application's  main  method:
  /** init  initializes (constructs) the applet  */
  public void init()
  { // construct the model objects:
    int box_size = 200;
    int balls_radius = 6;
    Box box = new Box(box_size);
    ball = new MovingBall((int)(box_size / 3.0), (int)(box_size / 5.0),
                          balls_radius, box);
    // construct the output-view objects:
    ball_writer = new BallWriter(ball, Color.red);
    box_writer  = new BoxWriter(box);
    // constructor method for  class AnimationWriter  is unneeded:
    // AnimationWriter writer
    //       = new AnimationWriter(box_writer, ball_writer, box_size);
    // the controller's  runAnimation  method is renamed  paint,  so
    //  following statement is unneeded:
    // new BounceController(ball, writer).runAnimation();
  }

  /** paint  paints the applet---this is the  runAnimation  method.
    * @param g - the graphics pen */
  public void paint(Graphics g)
  { while ( true )
          { delay(20);
            ball.move();
            // System.out.println(ball.xPosition() + ", " + ball.yPosition());
            // the following statement replaces  writer.repaint():
            paintAnimation(g);
          }
  }
 ...
```

Figure 7.22: output-view class for animation applet (concl.)

```
  /** delay pauses execution for  how_long  milliseconds */
  private void delay(int how_long)
  { try { Thread.sleep(how_long); }
    catch (InterruptedException e) { }
  }

  /** paintAnimation  is formerly  AnimationWriter's  paintComponent  method */
  public void paintAnimation(Graphics g)
  { box_writer.paint(g);
    ball_writer.paint(g);
  }
}
```

- Change the class from `extends JPanel` to `extends JApplet`.
- Remove the header line from the constructor method and replace it by `public void init()`.
- Remove the enclosing frame and all invocations of `setSize`, `setTitle`, and `setVisible`.

  If the constructor method uses parameters, then you are out of luck. (But Chapter 10 presents an awkward way of binding strings to variables within `init`.)
- Rename `paintComponent` to `paint`.

For example, the class in Figure 20 is revised this way:

```
import java.awt.*;
import javax.swing.*;
/** RecursivePictureApplet displays a recursively defined picture of an
  * egg in front of a picture (of eggs). */
public class RecursivePictureApplet extends JApplet
{ private double scale = 0.8;  // the size of the background picture relative
                               //   to the foreground
  private int width = 400;     // frame width and depth
  private int depth = 200;
  private int first_egg = 150; // size of the egg in the foreground

  /** the constructor method is renamed  init:  */
  public void init()
  { setBackground(Color.white); }
```

```
  /** paintComponent is renamed paint: */
  public void paint(Graphics g)
  { ... }

  ... the other methods stay the same ...
}
```

As noted in Chapter 4, the applet is executed by the `applet` command in an HTML-coded web page. Here is a sample web-page:

```
<body>
Here is my applet:
 <p>
  <applet code = "RecursivePictureApplet.class" width=400 height=200>
  Comments about the applet go here.<applet>
</body>
```

We must work harder to make the animation example into an applet—the start-up and controller classes must be squeezed into the output-view class, `AnimationWriter`. The unfortunate result appears in Figure 22. The controller must be moved into the applet's `paint` method, because a newly created applet first executes its `init` method, followed by `paint`. The "real" painting method, that of the output view, is renamed `paintAnimation` and is invoked from within `paint`.

In Chapter 10, we learn how to avoid such surgeries.