7

Arrays

Introducing
JAVA SE 6 (Mustang) • Web Services •
Ajax-Enabled Web Applications •
JavaServer Faces

Now go, write it
before them in a table,
and note it in a book.
—Isaiah 30:8

To go beyond is as
wrong as to fall short.
—Confucius

Begin at the beginning, …
and go on till you come to
the end: then stop.
—Lewis Carroll

OBJECTIVES

In this chapter you will learn:

- What arrays are.
- To use arrays to store data in and retrieve data from lists and tables of values.
- To declare arrays, initialize arrays and refer to individual elements of arrays.
- To use the enhanced `for` statement to iterate through arrays.
- To pass arrays to methods.
- To declare and manipulate multidimensional arrays.
- To write methods that use variable-length argument lists.
- To read command-line arguments into a program.

Self-Review Exercises

7.1 Fill in the blank(s) in each of the following statements:

a) Lists and tables of values can be stored in __________.

ANS: arrays.

b) An array is a group of __________ (called elements or components) containing values that all have the same __________.

ANS: variables, type.

c) The __________ allows programmers to iterate through the elements in an array without using a counter.

ANS: enhanced `for` statement.

d) The number used to refer to a particular element of an array is called the element's __________.

ANS: index (or subscript or position number).

e) An array that uses two indices is referred to as a(n) __________ array.

ANS: two-dimensional.

f) Use the enhanced `for` statement __________ to walk through `double` array `numbers`.

ANS: `for ( double d : numbers )`.

g) Command-line arguments are stored in __________.

ANS: an array of `String`s, called `args` by convention.

h) Use the expression __________ to receive the total number of arguments in a command line. Assume that command-line arguments are stored in `String args[]`.

ANS: `args.length`.

i) Given the command `java MyClass test`, the first command-line argument is __________.

ANS: `test`.

j) An __________ in the parameter list of a method indicates that the method can receive a variable number of arguments.

ANS: ellipsis (`...`).

7.2 Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) An array can store many different types of values.

ANS: False. An array can store only values of the same type.

b) An array index should normally be of type `float`.

ANS: False. An array index must be an integer or an integer expression.

c) An individual array element that is passed to a method and modified in that method will contain the modified value when the called method completes execution.

ANS: For individual primitive-type elements of an array: False. A called method receives and manipulates a copy of the value of such an element, so modifications do not affect the original value. If the reference of an array is passed to a method, however, modifications to the array elements made in the called method are indeed reflected in the original. For individual elements of a nonprimitive type: True. A called method receives a copy of the reference of such an element, and changes to the referenced object will be reflected in the original array element.

d) Command-line arguments are separated by commas.

ANS: False. Command-line arguments are separated by white space.

7.3 Perform the following tasks for an array called `fractions`:

a) Declare a constant `ARRAY_SIZE` that is initialized to 10.

ANS:
```
final int ARRAY_SIZE = 10;
```

b) Declare an array with ARRAY_SIZE elements of type double, and initialize the elements to 0.

ANS: `double fractions[] = new double[ ARRAY_SIZE ];`

c) Refer to array element 4.

ANS: `fractions[ 4 ]`

d) Assign the value 1.667 to array element 9.

ANS: `fractions[ 9 ] = 1.667;`

e) Assign the value 3.333 to array element 6.

ANS: `fractions[ 6 ] = 3.333;`

f) Sum all the elements of the array, using a for statement. Declare the integer variable x as a control variable for the loop.

ANS:

```java
double total = 0.0;
for ( int x = 0; x < fractions.length; x++ )
   total += fractions[ x ];
```

```java
// Exercise 7.3 Solution: Sum.java
public class Sum
{
   public static void main( String args[] )
   {
      // a)
      final int ARRAY_SIZE = 10;

      // b)
      double fractions[] = new double[ ARRAY_SIZE ];

      // c) fractions[ 4 ]

      // d)
      fractions[ 9 ] = 1.667;

      // e)
      fractions[ 6 ] = 3.333;

      // f)
      double total = 0.0;
      for ( int x = 0; x < fractions.length; x++ )
         total += fractions[ x ];

      System.out.printf( "fractions[ 9 ] = %.3f\n", fractions[ 9 ] );
      System.out.printf( "fractions[ 6 ] = %.3f\n", fractions[ 6 ] );
      System.out.printf( "total = %.3f", total );
   } // end main
} // end class Sum
```

```
fractions[ 9 ] = 1.667
fractions[ 6 ] = 3.333
total = 5.000
```

7.4 Perform the following tasks for an array called table:

a) Declare and create the array as an integer array that has three rows and three columns. Assume that the constant ARRAY_SIZE has been declared to be 3.

ANS: `int table[][] = new int[ ARRAY_SIZE ][ ARRAY_SIZE ];`

b) How many elements does the array contain?

ANS: Nine.

c) Use a `for` statement to initialize each element of the array to the sum of its indices. Assume that the integer variables `x` and `y` are declared as control variables.

ANS:
```
for ( int x = 0; x < table.length; x++ )
   for ( int y = 0; y < table[ x ].length; y++ )
      table[ x ][ y ] = x + y;
```

```
// Exercise 7.4 Solution: Table.java
public class Table
{
   public static void main( String args[] )
   {
      final int ARRAY_SIZE = 3;

      // a)
      int table[][] = new int[ ARRAY_SIZE ][ ARRAY_SIZE ];

      // c)
      for ( int x = 0; x < table.length; x++ )
         for ( int y = 0; y < table[ x ].length; y++ )
            table[ x ][ y ] = x + y;
   } // end main
} // end class Table
```

7.5 Find and correct the error in each of the following program segments:

a)
```
final int ARRAY_SIZE = 5;
ARRAY_SIZE = 10;
```

ANS: Error: Assigning a value to a constant after it has been initialized.
Correction: Assign the correct value to the constant in a `final int ARRAY_SIZE` declaration or declare another variable.

```
// Exercise 7.5 Part A Solution: PartAError.java
public class PartAError
{
   public static void main( String args[] )
   {
      final int ARRAY_SIZE = 5;
      ARRAY_SIZE = 10;
   } // end main
} // end PartAError
```

```
PartAError.java:7: cannot assign a value to final variable ARRAY_SIZE
      ARRAY_SIZE = 10;
      ^
1 error
```

```
// Exercise 7.5 Part A Solution: PartACorrect.java
public class PartACorrect
```

```
{
   public static void main( String args[] )
   {
      final int ARRAY_SIZE = 10;
   } // end main
} // end PartACorrect
```

b) Assume `int b[] = new int[ 10 ];`

```
for ( int i = 0; i <= b.length; i++ )
   b[ i ] = 1;
```

ANS: Error: Referencing an array element outside the bounds of the array (`b[10]`).
Correction: Change the <= operator to <.

```
// Exercise 7.5 Part B Solution: PartBError.java
public class PartBError
{
   public static void main( String args[] )
   {
      int b[] = new int[ 10 ];

      for ( int i = 0; i <= b.length; i++ )
         b[ i ] = 1;
   } // end main
} // end PartBError
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at PartBError.main(PartBError.java:9)
```

```
// Exercise 7.5 Part B Solution: PartBCorrect.java
public class PartBCorrect
{
   public static void main( String args[] )
   {
      int b[] = new int[ 10 ];

      for ( int i = 0; i < b.length; i++ )
         b[ i ] = 1;
   } // end main
} // end PartBCorrect
```

c) Assume `int a[][] = { { 1, 2 }, { 3, 4 } };`

```
   a[ 1, 1 ] = 5;
```

ANS: Array indexing is performed incorrectly.
Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`.

```
// Exercise 7.5 Part C Solution: PartCError.java
public class PartCError
{
   public static void main( String args[] )
   {
      int a[][] = { { 1, 2 }, { 3, 4 } };

      a[ 1, 1 ] = 5;
   } // end main
} // end PartCError
```

```
PartC.java [9:1] ']' expected
      a[ 1, 1 ] = 5;
          ^
PartC.java [9:1] not a statement
      a[ 1, 1 ] = 5;
       ^
```

```
// Exercise 7.5 Part C Solution: PartCCorrect.java
public class PartCCorrect
{
   public static void main( String args[] )
   {
      int a[][] = { { 1, 2 }, { 3, 4 } };

      a[ 1 ][ 1] = 5;
   } // end main
} // end PartCCorrect
```

Exercises

7.6 Fill in the blanks in each of the following statements:

a) One-dimensional array `p` contains four elements. The array-access expressions for elements are ___________, ___________, ___________ and ___________.

ANS: `p[ 0 ]`, `p[ 1 ]`, `p[ 2 ]`, and `p[ 3 ]`

b) Naming an array, stating its type and specifying the number of dimensions in the array is called ___________ the array.

ANS: declaring

c) In a two-dimensional array, the first index identifies the ___________ of an element and the second index identifies the ___________ of an element.

ANS: row, column

d) An *m*-by-*n* array contains ___________ rows, ___________ columns and ___________ elements.

ANS: `m, n, m · n`

e) The name of the element in row 3 and column 5 of array `d` is ___________.

ANS: `d[ 3 ][ 5 ]`

7.7 Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.

ANS: False. The name of the array and the index are specified.

b) An array declaration reserves space for the array.

ANS: False. Arrays must be dynamically allocated with `new` in Java.

c) To indicate that 100 locations should be reserved for integer array `p`, the programmer writes the declaration

```
p[ 100 ];
```

ANS: False. The correct declaration is `int p[] = new int[ 100 ];`

d) An application that initializes the elements of a 15-element array to zero must contain at least one `for` statement.

ANS: False. Numeric arrays are automatically initialized to zero. Also, a member initializer list can be used.

e) An application that totals the elements of a two-dimensional array must contain nested `for` statements.

ANS: False. It is possible to total the elements of a two-dimensional array with nested `while` statements, nested `do…while` statements or even individual totaling statements.

7.8 Write Java statements to accomplish each of the following tasks:

a) Display the value of element 6 of array `f`.

ANS: `System.out.print( f[ 6 ] );`

b) Initialize each of the five elements of one-dimensional integer array `g` to `8`.

ANS: `int g[] = { 8, 8, 8, 8, 8 );`

c) Total the 100 elements of floating-point array `c`.

ANS:
```
for ( int k = 0; k < c.length; k++ )
   total += c[ k ];
```

```java
// Exercise 7.8c Solution: PartC.java
public class PartC
{
   public static void main( String args[] )
   {
      double c[] = new double[ 100 ];
      double total = 0;

      // c)
      for ( int k = 0; k < c.length; k++ )
         total += c[ k ];
   } // end main
} // end class PartC
```

d) Copy 11-element array `a` into the first portion of array `b`, which contains 34 elements.

ANS:
```
for ( int j = 0; j < a.length; j++ )
   b[ j ] = a[ j ];
```

```java
// Exercise 7.8d Solution: PartD.java
public class PartD
{
   public static void main( String args[] )
   {
      double a[] = new double[ 11 ];
      double b[] = new double[ 34 ];

      // d)
      for ( int j = 0; j < a.length; j++ )
         b[ j ] = a[ j ];
   } // end main
} // end class PartD
```

e) Determine and display the smallest and largest values contained in 99-element floating-point array w.

ANS:

```
// Exercise 7.8e Solution: PartE.java
public class PartE
{
   public static void main( String args[] )
   {
      double w[] = new double[ 99 ];
      double small = w[ 0 ];
      double large = w[ 0 ];

      // e)
      for ( int i = 0;  i < w.length; i++ )
         if ( w[ i ] < small )
            small = w[ i ];
         else if ( w[ i ] > large )
            large = w[ i ];

      System.out.printf( "%f   %f\n", small, large );
   } // end main
} // end class PartE
```

```
0.000000   0.000000
```

7.9 Consider a two-by-three integer array t.

a) Write a statement that declares and creates t.

ANS: `int t[][] = new int[ 2 ][ 3 ];`

b) How many rows does t have?

ANS: two.

c) How many columns does t have?

ANS: three.

d) How many elements does t have?

ANS: six.

e) Write the access expressions for all the elements in row 1 of t.

ANS: `t[ 1 ][ 0 ], t[ 1 ][ 1 ], t[ 1 ][ 2 ]`

f) Write the access expressions for all the elements in column 2 of t.

ANS: `t[ 0 ][ 2 ], t[ 1 ][ 2 ]`

g) Write a single statement that sets the element of t in row 0 and column 1 to zero.

ANS: `t[ 0 ][ 1 ] = 0;`

h) Write a series of statements that initializes each element of t to zero. Do not use a repetition statement.

ANS:

```
t[ 0 ][ 0 ] = 0;
t[ 0 ][ 1 ] = 0;
t[ 0 ][ 2 ] = 0;
t[ 1 ][ 0 ] = 0;
t[ 1 ][ 1 ] = 0;
t[ 1 ][ 2 ] = 0;
```

i) Write a nested for statement that initializes each element of t to zero.

ANS:
```
for ( int j = 0; j < t.length; j++ )
   for ( int k = 0; k < t[ j ].length; k++ )
      t[ j ][ k ] = 0;
```

j) Write a nested for statement that inputs the values for the elements of t from the user.

ANS:
```
for ( int j = 0; j < t.length; j++ )
   for ( int k = 0; k < t[ j ].length; k++ )
      t[ j ][ k ] = input.nextInt();
```

k) Write a series of statements that determines and displays the smallest value in t.

ANS:
```
int smallest = t[ 0 ][ 0 ];

for ( int j = 0; j < t.length; j++ )
   for ( int k = 0; k < t[ j ].length; k++ )
      if ( t[ x ][ y ] < smallest )
         smallest = t[ x ][ y ];

System.out.println( smallest );
```

l) Write a printf statement that displays the elements of the first row of t. Do not use repetition.

ANS:
```
System.out.printf( “%d %d %d\n", t[ 0 ][ 0 ], t[ 0 ][ 1 ], t[ 0 ][ 2 ] );
```

m) Write a statement that totals the elements of the third column of t. Do not use repetition.

ANS:
```
int total = t[ 0 ][ 2 ] + t[ 1 ][ 2 ];
```

n) Write a series of statements that displays the contents of t in tabular format. List the column indices as headings across the top, and list the row indices at the left of each row.

ANS:
```
System.out.println( "\t0\t1\t2\n" );

for ( int e = 0; e < t.length; e++ )
{
   System.out.print( e );
   for ( int r = 0; r < t[ e ].length; r++ )
      System.out.printf( "\t%d", t[ e ][ r ] );

   System.out.println();
} // end for
```

```
// Exercise 7.9 Solution: Array.java
import java.util.Scanner;

public class Array
{
   public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );

      // a)
      int t[][] = new int[ 2 ][ 3 ];

      // g)
      t[ 0 ][ 1 ] = 0;

      // h)
      t[ 0 ][ 0 ] = 0;
      t[ 0 ][ 1 ] = 0;
      t[ 0 ][ 2 ] = 0;
      t[ 1 ][ 0 ] = 0;
      t[ 1 ][ 1 ] = 0;
      t[ 1 ][ 2 ] = 0;

      // i)
      for ( int j = 0; j < t.length; j++ )
         for ( int k = 0; k < t[ j ].length; k++ )
            t[ j ][ k ] = 0;

      // j)
      for ( int j = 0; j < t.length; j++ )
         for ( int k = 0; k < t[ j ].length; k++ )
            t[ j ][ k ] = input.nextInt();

      // k)
      int small = t[ 0 ][ 0 ];

      for ( int j = 0; j < t.length; j++ )
         for ( int k = 0; k < t[ j ].length; k++ )
            if ( t[ j ][ k ] < small )
               small = t[ j ][ k ];

      System.out.println( small );

      // l)
      System.out.printf(
         "%d %d %d\n", t[ 0 ][ 0 ], t[ 0 ][ 1 ], t[ 0 ][ 2 ] );

      // m
      int total = t[ 0 ][ 2 ] + t[ 1 ][ 2 ];

      // n
      System.out.println( "\t0\t1\t2\n" );
      for ( int e = 0; e < t.length; e++ )
      {
         System.out.print( e );

         for ( int r = 0; r < t[ e ].length; r++ )
            System.out.printf( "\t%d", t[ e ][ r ] );

         System.out.println();
      } // end for
   } // end main
} // end class Array
```

```
1
2
3
4
5
6
1
1 2 3
    0   1   2

0   1   2   3
1   4   5   6
```

7.10 *(Sales Commissions)* Use a one-dimensional array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9% of $5000, or a total of $650. Write an application (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a) $200–299
b) $300–399
c) $400–499
d) $500–599
e) $600–699
f) $700–799
g) $800–899
h) $900–999
i) $1000 and over

Summarize the results in tabular format.

ANS:

```java
// Exercise 7.10 Solution: Sales.java
// Program calculates the amount of pay for a salesperson and counts the
// number of salespeople that earned salaries in given ranges.
import java.util.Scanner;

public class Sales
{
   // counts the number of people in given salary ranges
   public void countRanges()
   {
      Scanner input = new Scanner( System.in );

      int total[] = new int[ 9 ]; // totals for the various salaries

      // initialize the values in the array to zero
      for ( int counter = 0; counter < total.length; counter++ )
         total[ counter ] = 0;

      // read in values and assign them to the appropriate range
      System.out.print( "Enter sales amount (negative to end): " );
      double dollars = input.nextDouble();

      while ( dollars >= 0 )
      {
         double salary = dollars * 0.09 + 200;
         int range = ( int ) ( salary / 100 );

         if ( range > 10 )
            range = 10;

         ++total[ range - 2 ];

         System.out.print( "Enter sales amount (negative to end): " );
         dollars = input.nextDouble();
      } // end while

      // print chart
      System.out.println( "Range\t\tNumber" );

      for ( int range = 0; range < total.length - 1; range++ )
         System.out.printf( "$%d-$%d\t%d\n",
            (200 + 100 * range), (299 + 100 * range), total[ range ] );

      // special case for the last range
      System.out.printf( "$1000 and over\t%d\n",
         total[ total.length - 1 ] );
   } // end method countRanges
} // end class Sales
```

```
// Exercise 7.10 Solution: SalesTest.java
// Test application for class Sales
public class SalesTest
{
   public static void main( String args[] )
   {
      Sales application = new Sales();
      application.countRanges();
   } // end main
} // end class SalesTest
```

```
Enter sales amount (negative to end): 5000
Enter sales amount (negative to end): -1
Range           Number
$200-$299       0
$300-$399       0
$400-$499       0
$500-$599       0
$600-$699       1
$700-$799       0
$800-$899       0
$900-$999       0
$1000 and over  0
```

7.11 Write statements that perform the following one-dimensional-array operations:

a) Set the 10 elements of integer array `counts` to zero.

ANS:
```
for ( int u = 0; u < counts.length; u++ )
   counts[ u ] = 0;
```

b) Add one to each of the 15 elements of integer array `bonus`.

ANS:
```
for ( int v = 0; v < bonus.length; v++ )
   bonus[ v ]++;
```

c) Display the five values of integer array `bestScores` in column format.

ANS:
```
for ( int w = 0; w < bestScores.length; w++ )
   System.out.println( bestScores[ w ] );
```

7.12 *(Duplicate Elimination)* Use a one-dimensional array to solve the following problem: Write an application that inputs five numbers, each between 10 and 100, inclusive. As each number is read, display it only if it is not a duplicate of a number already read. Provide for the "worst case," in which all five numbers are different. Use the smallest possible array to solve this problem. Display the complete set of unique values input after the user inputs each new value.

ANS:

```
// Exercise 7.12 Solution: Unique.java
// Reads in 5 unique numbers.
import java.util.Scanner;

public class Unique
{
   // gets 5 unique numbers from the user
   public void getNumbers()
   {
      Scanner input = new Scanner( System.in );

      int numbers[] = new int[ 5 ]; // list of unique numbers
      int count = 0; // number of uniques read

      while( count < numbers.length )
      {
         System.out.print( "Enter number: " );
         int number = input.nextInt();

         // validate the input
         if ( 10 <= number && number <= 100 )
         {
            // flags whether this number already exists
            boolean containsNumber = false;

            // compare input number to unique numbers in array
            for ( int i = 0; i < count; i++ )
               // if new number is duplicate, set the flag
               if ( number == numbers[ i ] )
                  containsNumber = true;

            // add only if the number is not there already
            if ( !containsNumber )
            {
               numbers[ count ] = number;
               count++;
```

```java
            } // end if
            else
               System.out.printf( "%d has already been entered\n",
                  number );
         } // end if
         else
            System.out.println( "number must be between 10 and 100" );

         // print the list
         for ( int i = 0; i < count; i++ )
            System.out.printf( "%d ", numbers[i] );
         System.out.println();
      } // end while
   } // end method getNumbers
} // end class Unique
```

```java
// Exercise 7.12 Solution: UniqueTest.java
// Test application for class Unique
public class UniqueTest
{
   public static void main( String args[] )
   {
      Unique application = new Unique();
      application.getNumbers();
   } // end main
} // end class UniqueTest
```

```
Enter number: 11
11
Enter number: 85
11 85
Enter number: 26
11 85 26
Enter number: 11
11 has already been entered
11 85 26
Enter number: 41
11 85 26 41
11 85 26
Enter number: 99
11 85 26 41 99
```

7.13 Label the elements of three-by-five two-dimensional array `sales` to indicate the order in which they are set to zero by the following program segment:

```java
for ( int row = 0; row < sales.length; row++ )
{
   for ( int col = 0; col < sales[ row ].length; col++ )
   {
      sales[ row ][ col ] = 0;
      }
   }
```

ANS: `sales[ 0 ][ 0 ], sales[ 0 ][ 1 ], sales[ 0 ][ 2 ], sales[ 0 ][ 3 ], sales[ 0 ][ 4 ], sales[ 1 ][ 0 ], sales[ 1 ][ 1 ], sales[ 1 ][ 2 ], sales[ 1 ][ 3 ], sales[ 1 ][ 4 ], sales[ 2 ][ 0 ], sales[ 2 ][ 1 ], sales[ 2 ][ 2 ], sales[ 2 ][ 3 ], sales[ 2 ][ 4 ]`

7.14 Write an application that calculates the product of a series of integers that are passed to method `product` using a variable-length argument list. Test your method with several calls, each with a different number of arguments.

ANS:

```java
// Exercise 7.14 Solution: VarargsTest.java
// Using variable-length argument lists.

public class VarargsTest
{
   // multiply numbers
   public static int product( int... numbers )
   {
      int product = 1;

      // process variable-length argument list
      for ( int number : numbers )
         product *= number;

      return product;
   } // end method product

   public static void main( String args[] )
   {
      // values to multiply
      int a = 1;
      int b = 2;
      int c = 3;
      int d = 4;
      int e = 5;

      // display integer values
      System.out.printf( "a = %d, b = %d, c = %d, d = %d, e = %d\n\n",
         a, b, c, d, e );

      // call product with different number of arguments in each call
      System.out.printf( "The product of a and b is: %d\n",
         product( a, b ) );
      System.out.printf( "The product of a, b and c is: %d\n",
         product( a, b, c ) );
      System.out.printf( "The product of a, b, c and d is: %d\n",
         product( a, b, c, d ) );
      System.out.printf( "The product of a, b, c, d and e is: %d\n",
         product( a, b, c, d, e ) );
   } // end main
} // end class VarargsTest
```

```
a = 1, b = 2, c = 3, d = 4, e = 5

The product of a and b is: 2
The product of a, b and c is: 6
The product of a, b, c and d is: 24
The product of a, b, c, d and e is: 120
```

7.15 Rewrite Fig. 7.2 so that the size of the array is specified by the first command-line argument. If no command-line argument is supplied, use 10 as the default size of the array.

ANS:

```java
// Exercise 7.15 Solution: InitArray.java
// Creating an array with size specified by the command-line argument.

public class InitArray
{
   public static void main( String args[] )
   {
      int[] array; // declare array
      int size = 10; // default size of the array

      // get size
      if ( args.length == 1 )
         size = Integer.parseInt( args[ 0 ] );

      array = new int[ size ]; // create array with specified size

      System.out.printf( "%s%8s\n", "Index", "Value" );

      // display array elements
      for ( int count = 0; count < array.length; count++ )
         System.out.printf( "%5d%8d\n", count, array[ count ] );
   } // end main
} // end class InitArray
```

```
java InitArray 5
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

7.16 Write an application that uses an enhanced `for` statement to sum the `double` values passed by the command-line arguments. [*Hint*: Use the `static` method `parseDouble` of class `Double` to convert a `String` to a `double` value.]

ANS:

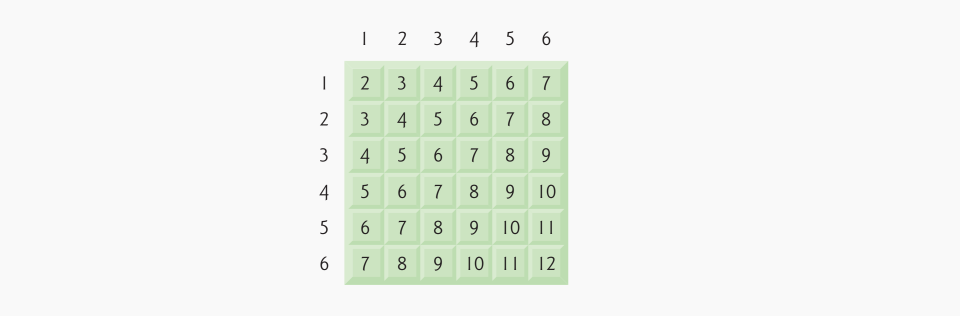
```java
// Exercise 7.16 Solution: CalculateTotal.java
// Calculates total of double values passed by the command-line arguments.
```

```java

public class CalculateTotal
{
   public static void main( String args[] )
   {
      double total = 0.0;

      // calculate total
      for ( String argument : args )
         total += Double.parseDouble( argument );

      System.out.printf( "total is: %.2f\n", total );
   } // end main
} // end class CalculateTotal
```

```
java CalculateTotal 1.1 2.2 3.3 4.4 5.5
total is: 16.50
```

```
java CalculateTotal
total is: 0.00
```

7.17 *(Dice Rolling)* Write an application to simulate the rolling of two dice. The application should use an object of class `Random` once to roll the first die and again to roll the second die. The sum of the two values should then be calculated. Each die can show an integer value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent, sum and 2 and 12 the least frequent. Figure 7.30 shows the 36 possible combinations of the two dice. Your application should roll the dice 36,000 times. Use a one-dimensional array to tally the number of times each possible sum appears. Display the results in tabular format. Determine whether the totals are reasonable (e.g., there are six ways to roll a 7, so approximately one-sixth of the rolls should be 7).



	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.30 | The 36 possible sums of two dice.

ANS:

```java
// Exercise 7.17 Solution: Roll36.java
// Program simulates rolling two six-sided dice 36,000 times.
```

```java
import java.util.Random;

public class Roll36
{
   // simulate rolling of dice 36000 times
   public void rollDice()
   {
      Random randomNumbers = new Random();

      int face1; // number on first die
      int face2; // number on second die
      int totals[] = new int[ 13 ]; // frequencies of the sums

      // initialize totals to zero
      for ( int index = 0; index < totals.length; index++ )
         totals[ index ] = 0;

      // roll the dice
      for ( int roll = 1; roll <= 36000; roll++ ) {
         face1 = 1 + randomNumbers.nextInt( 6 );
         face2 = 1 + randomNumbers.nextInt( 6 );
         totals[ face1 + face2 ]++;
      } // end for

      // print the table
      System.out.printf( "%3s%12s%12s\n",
         "Sum", "Frequency", "Percentage" );

      // ignore subscripts 0 and 1
      for ( int k = 2; k < totals.length; k++ )
      {
         int percent = totals[ k ] / ( 360 );
         System.out.printf( "%3d%12d%12d\n", k, totals[ k ], percent );
      } // end for
   } // end method rollDice
} // end class Roll36
```

```java
// Exercise 7.17 Solution: Roll36Test.java
// Test application for class Roll36
public class Roll36Test
{
   public static void main( String args[] )
   {
      Roll36 application = new Roll36();
      application.rollDice();
   } // end main
} // end class Roll36Test
```

```
Sum   Frequency  Percentage
  2        1007           2
  3        2012           5
  4        2959           8
  5        3946          10
  6        5020          13
  7        6055          16
  8        5014          13
  9        4022          11
 10        2993           8
 11        1997           5
 12         975           2
```

7.18 *(Game of Craps)* Write an application that runs 1000 games of craps (Fig. 6.9) and answers the following questions:

a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?

b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?

c) What are the chances of winning at craps? [*Note*: You should discover that craps is one of the fairest casino games. What do you suppose this means?]

d) What is the average length of a game of craps?

e) Do the chances of winning improve with the length of the game?

ANS:

```java
// Exercise 7.18 Solution: Craps.java
// Program plays 1000 games of craps and displays winning
// and losing statistics.
import java.util.Random;

public class Craps
{
   // create random number generator for use in method rollDice
   private Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST };

   int wins[]; // number of wins, by rolls
   int losses[]; // number of losses, by rolls
   int winSum = 0; // total number of wins
   int loseSum = 0; // total number of losses

   // plays one game of craps
   public void play()
   {
      int sumOfDice = 0; // sum of the dice
      int myPoint = 0; // point if no win or loss on first roll

      Status gameStatus; // can contain CONTINUE, WON or LOST

      int roll; // number of rolls for the current game

      wins = new int[ 22 ]; // frequency of wins
      losses = new int[ 22 ]; // frequency of losses

      for ( int i = 1; i <= 1000; i++ )
      {
         sumOfDice = rollDice(); // first roll of the dice
         roll = 1;

         // determine game status and point based on sumOfDice
         switch ( sumOfDice )
         {
            case 7:  // win with 7 on first roll
            case 11: // win with 11 on first roll
               gameStatus = Status.WON;
               break;
            case 2:  // lose with 2 on first roll
            case 3:  // lose with 3 on first roll
            case 12: // lose with 12 on first roll
               gameStatus = Status.LOST;
               break;
            default: // did not win or lose, so remember point
               gameStatus = Status.CONTINUE; // game is not over
               myPoint = sumOfDice; // store the point
               break; // optional for default case at end of switch
         } // end switch

         // while game is not complete ...
         while ( gameStatus == Status.CONTINUE )
         {
            sumOfDice = rollDice(); // roll dice again
            roll++;

            // determine game status
            if ( sumOfDice == myPoint ) // win by making point
               gameStatus = Status.WON;
            else if ( sumOfDice == 7 ) // lose by rolling 7
               gameStatus = Status.LOST;
         } // end while

         // all roll results after 20th roll placed in last element
         if ( roll > 21 )
            roll = 21;

         // increment number of wins in that roll
         if ( gameStatus == Status.WON )
         {
            ++wins[ roll ];
            ++winSum;
         } // end if
         else // increment number of losses in that roll
         {
            ++losses[ roll ];
            ++loseSum;
         } // end else
      } // end for

      printStats();
   } // end method play

   // print win/loss statistics
   public void printStats()
   {
      int totalGames = winSum + loseSum; // total number of games
      int length = 0; // total length of the games

      // display number of wins and losses on all rolls
      for ( int i = 1; i <= 21; i++ )
      {
         if ( i == 21 )
            System.out.printf( "%d %s %d %s\n",
               wins[ i ], "games won and", losses[ i ],
               "games lost on rolls after the 20th roll" );
         else
            System.out.printf( "%d %s %d %s%d\n",
               wins[ i ], "games won and", losses[ i ],
               "games lost on roll #", i );

         // for calculating length of game
         // number of wins/losses on that roll multiplied
         // by the roll number, then add them to length
         length += wins[ i ] * i + losses[ i ] * i;
      } // end for

      // calculate chances of winning
      System.out.printf( "\n%s %d / %d = %.2f%%\n",
         "The chances of winning are", winSum, totalGames,
         ( 100.0 * winSum / totalGames ) );


      System.out.printf( "The average game length is %.2f rolls.\n",
         ( ( double ) length / totalGames ) );
   } // end method printStats

   // roll dice, calculate sum and display results
   public int rollDice()
   {
      // pick random die values
      int die1 = 1 + randomNumbers.nextInt( 6 );
      int die2 = 1 + randomNumbers.nextInt( 6 );
      int sum = die1 + die2; // sum die values

      return sum; // return sum of dice
   } // end method rollDice
} // end class Craps
```

```
// Exercise 7.18 Solution: CrapsTest.java
// Test application for class Craps
public class CrapsTest
{
   public static void main( String args[] )
   {
      Craps game = new Craps();
      game.play();
   } // end main
} // end class CrapsTest
```

```
224 games won and 99 games lost on roll #1
74 games won and 119 games lost on roll #2
50 games won and 96 games lost on roll #3
33 games won and 54 games lost on roll #4
23 games won and 47 games lost on roll #5
22 games won and 37 games lost on roll #6
18 games won and 13 games lost on roll #7
8 games won and 18 games lost on roll #8
7 games won and 14 games lost on roll #9
5 games won and 6 games lost on roll #10
5 games won and 6 games lost on roll #11
4 games won and 3 games lost on roll #12
1 games won and 3 games lost on roll #13
1 games won and 0 games lost on roll #14
0 games won and 4 games lost on roll #15
1 games won and 0 games lost on roll #16
0 games won and 0 games lost on roll #17
0 games won and 1 games lost on roll #18
0 games won and 0 games lost on roll #19
0 games won and 0 games lost on roll #20
3 games won and 1 games lost on rolls after the 20th roll

The chances of winning are 479 / 1000 = 47.90%
The average game length is 3.37 rolls.
```

7.19 (*Airline Reservations System*) A small airline has just purchased a computer for its new automated reservations system. You have been asked to develop the new system. You are to write an application to assign seats on each flight of the airline's only plane (capacity: 10 seats).

Your application should display the following alternatives: `Please type 1 for First Class` and `Please type 2 for Economy`. If the user types `1`, your application should assign a seat in the first-class section (seats 1–5). If the user types `2`, your application should assign a seat in the economy section (seats 6–10). Your application should then display a boarding pass indicating the person's seat number and whether it is in the first-class or economy section of the plane.

Use a one-dimensional array of primitive type `boolean` to represent the seating chart of the plane. Initialize all the elements of the array to `false` to indicate that all the seats are empty. As each seat is assigned, set the corresponding elements of the array to `true` to indicate that the seat is no longer available.

Your application should never assign a seat that has already been assigned. When the economy section is full, your application should ask the person if it is acceptable to be placed in the first-class section (and vice versa). If yes, make the appropriate seat assignment. If no, display the message `"Next flight leaves in 3 hours."`

ANS:

```java
// Exercise 7.19 Solution: Plane.java
// Program reserves airline seats.
import java.util.Scanner;

public class Plane
{
   // checks customers in and assigns them a boarding pass
   public void checkIn()
   {
      Scanner input = new Scanner( System.in );

      boolean seats[] = new boolean[ 10 ]; // array of seats
      int firstClass = 0; // next available first class seat
      int economy = 5; // next available economy seat

      while ( ( firstClass < 5 ) || ( economy < 10 ) )
      {
         System.out.println( "Please type 1 for First Class" );
         System.out.println( "Please type 2 for Economy" );
         System.out.print( "choice: " );
         int section = input.nextInt();

         if ( section == 1 ) // user chose first class
         {
            if ( firstClass < 5 )
            {
               firstClass++;
               System.out.printf( "First Class. Seat #%d\n", firstClass );
            } // end if
            else if ( economy < 10 ) // first class is full
            {
               System.out.println(
                  "First Class is full, Economy Class?" );
               System.out.print( "1. Yes, 2. No. Your choice: " );
               int choice = input.nextInt();

               if ( choice == 1 )
               {
                  economy++;
                  System.out.printf( "Economy Class. Seat #%d\n",
                     economy );
               }
               else
                  System.out.println( "Next flight leaves in 3 hours." );
            } // end else if
         } // end if
         else if ( section == 2 ) // user chose economy
         {
            if ( economy < 10 )
            {
               economy++;
               System.out.printf( "Economy Class. Seat #%d\n", economy );
            } // end if
            else if ( firstClass < 5 ) // economy class is full
            {
               System.out.println(
                  "Economy Class is full, First Class?" );
               System.out.print( "1. Yes, 2. No. Your choice: " );
               int choice = input.nextInt();

               if ( choice == 1 )
               {
                  firstClass++;
                  System.out.printf( "First Class. Seat #%d\n",
                     firstClass );
               } // end if
               else
                  System.out.println( "Next flight leaves in 3 hours." );
            } // end else if
         } // end else if

         System.out.println();
      } // end while

      System.out.println( "The plane is now full." );
   } // end method checkIn
} // end class Plane
```

```java
// Exercise 7.19 Solution: PlaneTest.java
// Test application for class Plane
public class PlaneTest
{
   public static void main( String args[] )
   {
      Plane application = new Plane();
      application.checkIn();
   } // end main
} // end class PlaneTest
```

```
Please type 1 for First Class
Please type 2 for Economy
choice: 1
First Class. Seat #1

Please type 1 for First Class
Please type 2 for Economy
choice: 2
Economy Class. Seat #6

.
.
.

Please type 1 for First Class
Please type 2 for Economy
choice: 1
First Class is full, Economy Class?
1. Yes, 2. No. Your choice: 1
Economy Class. Seat #7
```

7.20 *(Total Sales)* Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each type of product sold. Each slip contains the following:

a) The salesperson number
b) The product number
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write an application that will read all this information for last month's sales and summarize the total sales by salesperson and by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, display the results in tabular format, with each column representing a particular salesperson and each row representing a particular product. Cross-total each row to get the total sales of each product for last month. Cross-total each column to get the total sales by salesperson for last month. Your tabular output should include these cross-totals to the right of the totaled rows and to the bottom of the totaled columns.

ANS:

```java
// Exercise 7.20 Solution: Sales2.java
// Program totals sales for salespeople and products.
import java.util.Scanner;

public class Sales2
{
   public void calculateSales()
   {
      Scanner input = new Scanner( System.in );
      // sales array holds data on number of each product sold
      // by each salesperson
      double sales[][] = new double[ 5 ][ 4 ];

      System.out.print( "Enter salesperson number (-1 to end): " );
      int person = input.nextInt();

      while ( person != -1 )
      {
         System.out.print( "Enter product number: " );
         int product = input.nextInt();
         System.out.print( "Enter sales amount: " );
         double amount = input.nextDouble();

         // error-check the input
         if ( person >= 1 && person < 5 &&
               product >= 1 && product < 6 && amount >= 0 )
            sales[ product - 1 ][ person - 1 ] += amount;
         else
            System.out.println( "Invalid input!" );

         System.out.print( "Enter salesperson number (-1 to end): " );
         person = input.nextInt();
      } // end while

      // total for each salesperson
      double salesPersonTotal[] = new double[ 4 ];

      // display the table
      for ( int column = 0; column < 4; column++ )
         salesPersonTotal[ column ] = 0;

      System.out.printf( "%8s%14s%14s%14s%14s%10s\n",
            "Product", "Salesperson 1", "Salesperson 2",
            "Salesperson 3", "Salesperson 4", "Total" );

      // for each column of each row, print the appropriate
      // value representing a person's sales of a product
      for ( int row = 0; row < 5; row++ )
      {
         double productTotal = 0.0;
         System.out.printf( "%8d", ( row + 1 ) );

         for ( int column = 0; column < 4; column++ ) {
            System.out.printf( "%14.2f", sales[ row ][ column ] );
            productTotal += sales[ row ][ column ];
            salesPersonTotal[ column ] += sales[ row ][ column ];
         } // end for

         System.out.printf( "%10.2f\n", productTotal );
      } // end for

      System.out.printf( "%8s", "Total" );

      for ( int column = 0; column < 4; column++ )
         System.out.printf( "%14.2f", salesPersonTotal[ column ] );

      System.out.println();
```

```java
   } // end method calculateSales
} // end class Sales2
```

```java
// Exercise 7.20 Solution: Sales2Test.java
// Test application for class Sales2
public class Sales2Test
{
   public static void main( String args[] )
   {
      Sales2 application = new Sales2();
      application.calculateSales();
   } // end main
} // end class Sales2Test
```

```
Enter salesperson number (-1 to end): 1
Enter product number: 4
Enter sales amount: 1082
Enter salesperson number (-1 to end): 2
Enter product number: 3
Enter sales amount: 998
Enter salesperson number (-1 to end): 3
Enter product number: 1
Enter sales amount: 678
Enter salesperson number (-1 to end): 4
Enter product number: 1
Enter sales amount: 1554
Enter salesperson number (-1 to end): -1
 Product Salesperson 1 Salesperson 2 Salesperson 3 Salesperson 4     Total
       1          0.00          0.00        678.00       1554.00   2232.00
       2          0.00          0.00          0.00          0.00      0.00
       3          0.00        998.00          0.00          0.00    998.00
       4       1082.00          0.00          0.00          0.00   1082.00
       5          0.00          0.00          0.00          0.00      0.00
   Total       1082.00        998.00        678.00       1554.00
```

7.21 (*Turtle Graphics*) The Logo language made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a Java application. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves, and while the pen is up, the turtle moves about freely without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad.

Use a 20-by-20 array `floor` that is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor with its pen up. The set of turtle commands your application must process are shown in Fig. 7.31.

Command	Meaning
1	Pen up

Fig. 7.31 | Turtle graphics commands. (Part 1 of 2.)

Command	Meaning
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (replace 10 for a different number of spaces)
6	Display the 20-by-20 array
9	End of data (sentinel)

Fig. 7.31 | Turtle graphics commands. (Part 2 of 2.)

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and display a 12-by-12 square, leaving the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to `1`s. When the `6` command (display the array) is given, wherever there is a `1` in the array, display an asterisk or any character you choose. Wherever there is a `0`, display a blank.

Write an application to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

ANS:

```java
// Exercise 7.21: TurtleGraphics.java
// Drawing turtle graphics based on turtle commands.
import java.util.Scanner;

public class TurtleGraphics
{
   final int MAXCOMMANDS = 100; // maximum size of command array
   final int SIZE = 20; // size of the drawing area

   int floor[][]; // array representing the floor
   int commandArray[][]; // list of commands

   int count; // the current number of commands
   int xPos; // the x position of the turtle
   int yPos; // the y position of the turtle

   // enters the commands for the turtle graphics
   public void enterCommands()
   {
      Scanner input = new Scanner( System.in );

      count = 0;
      commandArray = new int[ MAXCOMMANDS ][ 2 ];
      floor = new int[ SIZE ][ SIZE ];

      System.out.print( "Enter command (9 to end input): " );
      int inputCommand = input.nextInt();

      while ( inputCommand != 9 && count < MAXCOMMANDS )
      {
         commandArray[ count ][ 0 ] = inputCommand;

         // prompt for forward spaces
         if ( inputCommand == 5 )
         {
            System.out.print( "Enter forward spaces: " );
            commandArray[ count ][ 1 ] = input.nextInt();
         } // end if

         count++;

         System.out.print( "Enter command (9 to end input): " );
         inputCommand = input.nextInt();
      } // end while

      executeCommands();
   } // end method enterCommands

   // executes the commands in the command array
   public void executeCommands()
   {
      int commandNumber = 0; // the current position in the array
      int direction = 0; // the direction the turtle is facing
      int distance = 0; // the distance the turtle will travel
      int command; // the current command
      boolean penDown = false; // whether the pen is up or down
      xPos = 0;
      yPos = 0;

      command = commandArray[ commandNumber ][ 0 ];

      // continue executing commands until either reach the end
      // or reach the max commands
      while ( commandNumber < count )
      {
         //System.out.println("Executing...");
         // determine what command was entered
         // and perform desired action
         switch ( command )
         {
            case 1: // pen down
               penDown = false;
               break;

            case 2: // pen up
               penDown = true;
               break;

            case 3: // turn right
               direction = turnRight( direction );
               break;

            case 4: // turn left
               direction = turnLeft( direction );
               break;

            case 5: // move
               distance = commandArray[ commandNumber ][ 1 ];
               movePen( penDown, floor, direction, distance );
               break;

            case 6: // display the drawing
               System.out.println( "\nThe drawing is:\n" );
               printArray( floor );
               break;
         }  // end switch

         command = commandArray[ ++commandNumber ][ 0 ];
      }  // end while
   } // end method executeCommands

   // method to turn turtle to the right
   public int turnRight( int d )
   {
      return ++d > 3 ? 0 : d;
   } // end method turnRight

   // method to turn turtle to the left
   public int turnLeft( int d )
   {
      return --d < 0 ? 3 : d;
   } // end method turnLeft

   // method to move the pen
   public void movePen( boolean down, int a[][], int dir, int dist )
   {
      int j; // looping variable

      // determine which way to move pen
      switch ( dir )
      {
         case 0: // move to right
            for ( j = 1; j <= dist && yPos + j < SIZE; ++j )
               if ( down )
                  a[ xPos ][ yPos + j ] = 1;

            yPos += j - 1;
            break;

         case 1: // move down
            for ( j = 1; j <= dist && xPos + j < SIZE; ++j )
               if ( down )
                  a[ xPos + j ][ yPos ] = 1;

            xPos += j - 1;
            break;

         case 2: // move to left
            for ( j = 1; j <= dist && yPos - j >= 0; ++j )
               if ( down )
                  a[ xPos ][ yPos - j ] = 1;

            yPos -= j - 1;
            break;

         case 3: // move up
            for ( j = 1; j <= dist && xPos - j >= 0; ++j )
               if ( down )
                  a[ xPos - j ][ yPos ] = 1;

            xPos -= j - 1;
            break;
      } // end switch
   } // end method movePen

   // method to print array drawing
   public void printArray( int a[][] )
   {
      // display array
      for ( int i = 0; i < SIZE; ++i )
      {
         for ( int j = 0; j < SIZE; ++j )
            System.out.print( ( a[ i ][ j ] == 1 ? "*" : " " ) );

         System.out.println();
      } // end for
   } // end method printArray
} // end class TurtleGraphics
```

```
// Exercise 7.21 Solution: TurtleGraphicsTest.java
// Test application for class TurtleGraphics
public class TurtleGraphicsTest
{
   public static void main( String args[] )
```

```
    {
       TurtleGraphics drawing = new TurtleGraphics();
       drawing.enterCommands();
    } // end main
} // end class TurtleGraphicsTest
```

```
Enter command (9 to end input): 2
Enter command (9 to end input): 5
Enter forward spaces: 12
Enter command (9 to end input): 3
Enter command (9 to end input): 5
Enter forward spaces: 12
Enter command (9 to end input): 3
Enter command (9 to end input): 5
Enter forward spaces: 12
Enter command (9 to end input): 3
Enter command (9 to end input): 5
Enter forward spaces: 12
Enter command (9 to end input): 1
Enter command (9 to end input): 6
Enter command (9 to end input): 9

The drawing is:

*************
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*           *
*************
```

7.22 (*Knight's Tour*) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem, originally proposed by the mathematician Euler. Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). Thus, as shown in Fig. 7.32, from a square near the middle of an empty chessboard, the knight (labeled K) can make eight different moves (numbered 0 through 7).

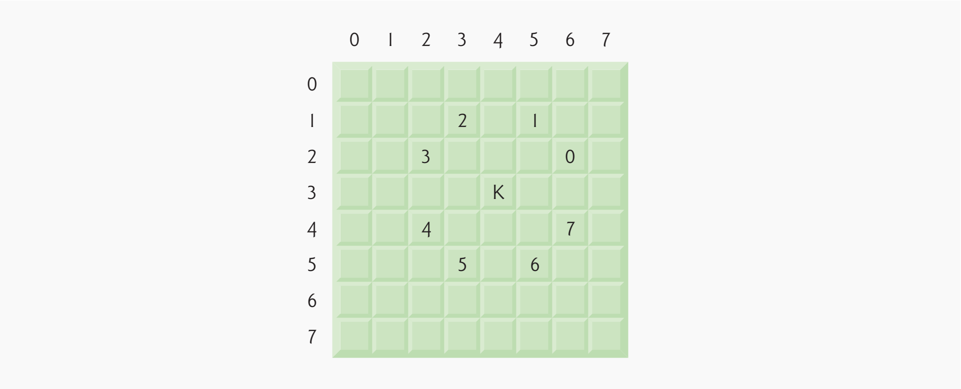


Fig. 7.32 | The eight possible moves of the knight.

a) Draw an eight-by-eight chessboard on a sheet of paper, and attempt a Knight's Tour by hand. Put a `1` in the starting square, a `2` in the second square, a `3` in the third, and so on. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

b) Now let us develop an application that will move the knight around a chessboard. The board is represented by an eight-by-eight two-dimensional array `board`. Each square is initialized to zero. We describe each of the eight possible moves in terms of their horizontal and vertical components. For example, a move of type 0, as shown in Fig. 7.32, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

```
horizontal[ 0 ] = 2        vertical[ 0 ] = -1
horizontal[ 1 ] = 1        vertical[ 1 ] = -2
horizontal[ 2 ] = -1       vertical[ 2 ] = -2
horizontal[ 3 ] = -2       vertical[ 3 ] = -1
horizontal[ 4 ] = -2       vertical[ 4 ] = 1
horizontal[ 5 ] = -1       vertical[ 5 ] = 2
horizontal[ 6 ] = 1        vertical[ 6 ] = 2
horizontal[ 7 ] = 2        vertical[ 7 ] = 1
```

Let the variables `currentRow` and `currentColumn` indicate the row and column, respectively, of the knight's current position. To make a move of type `moveNumber`, where `moveNumber` is between 0 and 7, your application should use the statements

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Write an application to move the knight around the chessboard. Keep a counter that varies from `1` to `64`. Record the latest count in each square the knight moves to.

Test each potential move to see if the knight has already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Run the application. How many moves did the knight make?

ANS:

```java
// Exercise 7.22 Part B Solution: Knight1.java
// Knight's Tour
import java.util.Random;

public class Knight1
{
   Random randomNumbers = new Random();

   int board[][]; // gameboard

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   // runs a tour
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard
      int moveNumber = 0; // the current move number

      board = new int[ 8 ][ 8 ]; // gameboard

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move

      // randomize initial board position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      board[ currentRow ][ currentColumn ] = ++moveNumber;
      boolean done = false;

      // continue until knight can no longer move
      while ( !done )
      {
         boolean goodMove = false;

         // check all possible moves until we find one that's legal
         for ( int moveType = 0; moveType < 8 && !goodMove;
            moveType++ )
         {
            testRow = currentRow + vertical[ moveType ];
            testColumn = currentColumn + horizontal[ moveType ];
            goodMove = validMove( testRow, testColumn );

            // test if new move is valid
            if ( goodMove )
            {
               currentRow = testRow;
               currentColumn = testColumn;
               board[ currentRow ][ currentColumn ] = ++moveNumber;
            } // end if
         } // end for

         // if no valid moves, knight can no longer move
         if ( !goodMove )
            done = true;
         // if 64 moves have been made, a full tour is complete
         else if ( moveNumber == 64 )
            done = true;
      } // end while

      System.out.printf( "The tour ended with %d moves.\n", moveNumber );

      if ( moveNumber == 64 )
         System.out.println( "This was a full tour!" );
      else
         System.out.println( "This was not a full tour." );

      printTour();
   }  // end method start

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display Knight's tour path
   public void printTour()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%d", board[ row ][ column ] );

         System.out.println();
      } // end for
   } // end method printTour
} // end class Knight1
```

```java
// Exercise 7.22 Part B Solution: Knight1Test.java
// Test application for class Knight1
public class Knight1Test
{
   public static void main( String args[] )
   {
      Knight1 application = new Knight1();
      application.tour();
   } // end main
} // end class Knight1Test
```

```
The tour ended with 43 moves.
This was not a full tour.
        0       1       2       3       4       5       6       7

0       8       43      28      13      6       11      18      23
1       29      14      7       10      17      22      5       20
2       42      9       16      27      12      19      24      37
3       15      30      0       40      25      36      21      4
4       0       41      26      0       0       3       38      35
5       31      0       0       2       39      34      0       0
6       0       1       0       33      0       0       0       0
7       0       32      0       0       0       0       0       0
```

c) After attempting to write and run a Knight's Tour application, you have probably developed some valuable insights. We will use these insights to develop a *heuristic* (or "rule of thumb") for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We could develop an "accessibility heuristic" by classifying each of the squares according to how accessible it is and always moving the knight (using the knight's L-shaped moves) to the most inaccessible square. We label a two-dimensional array `accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each of the 16 squares nearest the center is rated as `8`, each corner square is rated as `2`, and the other squares have accessibility numbers of `3`, `4` or `6` as follows:

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Write a new version of the Knight's Tour, using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [*Note*: As the knight moves around the chessboard, your application should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your application. Did you get a full tour? Modify the application to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

ANS:

```java
// Exercise 7.22 Part C Solution: Knight2.java
// Knight's Tour - heuristic version
import java.util.Random;

public class Knight2
{
   Random randomNumbers = new Random();

   int access[][] = { { 2, 3, 4, 4, 4, 4, 3, 2 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 2, 3, 4, 4, 4, 4, 3, 2 } };

   int board[][]; // gameboard
   int accessNumber; // the current access number

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   // initialize applet
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard
      int moveNumber = 0; // the current move number

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move
      int minRow = -1; // row position of move with minimum access
      int minColumn = -1; // row position of move with minimum access

      board = new int[ 8 ][ 8 ];

      // randomize initial board position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      board[ currentRow ][ currentColumn ] = ++moveNumber;
      boolean done = false;

      // continue touring until finished traversing
      while ( !done )
      {
         accessNumber = 99;

         // try all possible moves
         for ( int moveType = 0; moveType < board.length; moveType++ )
         {
            // new position of hypothetical moves
            testRow = currentRow + vertical[ moveType ];
            testColumn = currentColumn + horizontal[ moveType ];

            if ( validMove( testRow, testColumn ) )
            {
               // obtain access number
               if ( access[ testRow ][ testColumn ] < accessNumber )
               {
                  // if this is the lowest access number thus far,
                  // then set this move to be our next move
                  accessNumber = access[ testRow ][ testColumn ];

                  minRow = testRow;
                  minColumn = testColumn;
               } // end if

               // position access number tried
               --access[ testRow ][ testColumn ];
            } // end if
         } // end for

         // traversing done
         if ( accessNumber == 99 ) // no valid moves
            done = true;
         else
         { // make move
            currentRow = minRow;
            currentColumn = minColumn;
            board[ currentRow ][ currentColumn ] = ++moveNumber;
         } // end else
      } // end while

      System.out.printf( "The tour ended with %d moves.\n", moveNumber );

      if ( moveNumber == 64 )
         System.out.println( " This was a full tour!" );
      else
         System.out.println( " This was not a full tour." );

      printTour();
   } // end method tour

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display Knight's tour path
   public void printTour()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%d", board[ row ][ column ] );

         System.out.println();
      } // end for
   } // end method printTour
} // end class Knight2
```

```
// Exercise 7.22 Part C Solution: Knight2Test.java
// Test application for class Knight2
public class Knight2Test
{
   public static void main( String args[] )
   {
      Knight2 application = new Knight2();
      application.tour();
   } // end main
} // end class Knight2Test
```

```
The tour ended with 64 moves.
 This was a full tour!
        0       1       2       3       4       5       6       7

0       5       24      7       38      3       22      17      36
1       8       39      4       23      18      37      2       21
2       25      6       41      44      1       20      35      16
3       40      9       50      19      52      43      54      61
4       49      26      45      42      55      60      15      34
5       10      29      56      51      46      53      62      59
6       27      48      31      12      57      64      33      14
7       30      11      28      47      32      13      58      63
```

d) Write a version of the Knight's Tour application that, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your application should move to the tied square for which the next move would arrive at a square with the lowest accessibility number.

ANS:

```java
// Exercise 7.22 Part D Solution: Knight3.java
// Knight's Tour - heuristic version
import java.util.Random;

public class Knight3
{
   Random randomNumbers = new Random();

   int access[][] = { { 2, 3, 4, 4, 4, 4, 3, 2 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 2, 3, 4, 4, 4, 4, 3, 2 } };

   int board[][]; // gameboard
   int accessNumber; // the current access number

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   // initialize applet
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard
      int moveNumber = 0; // the current move number

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move
      int minRow = -1; // row position of move with minimum access
      int minColumn = -1; // row position of move with minimum access

      board = new int[ 8 ][ 8 ];

      // randomize initial board position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      board[ currentRow ][ currentColumn ] = ++moveNumber;
      boolean done = false;

      // continue touring until finished traversing
      while ( !done )
      {
         accessNumber = 99;

         // try all possible moves
         for ( int moveType = 0; moveType < board.length; moveType++ )
         {
            // new position of hypothetical moves
            testRow = currentRow + vertical[ moveType ];
            testColumn = currentColumn + horizontal[ moveType ];

            if ( validMove( testRow, testColumn ) )
            {
               // obtain access number
               if ( access[ testRow ][ testColumn ] < accessNumber )
               {
                  // if this is the lowest access number thus far,
                  // then set this move to be our next move
                  accessNumber = access[ testRow ][ testColumn ];

                  minRow = testRow;
                  minColumn = testColumn;
               } // end if
               else if
                  ( access[ testRow ][ testColumn ] == accessNumber )
               {
                  // if the lowest access numbers are the same,
                  // look ahead to the next move to see which has the
                  // lower access number
                  int lowestTest = nextMove( testRow, testColumn );
                  int lowestMin = nextMove( minRow, minColumn );

                  if ( lowestTest <= lowestMin )
                  {
                     accessNumber = access[ testRow ][ testColumn ];

                     minRow = testRow;
                     minColumn = testColumn;
                  } // end if
               } // end else if

```

```java
               // position access number tried
               --access[ testRow ][ testColumn ];
            } // end if
         } // end for

         // traversing done
         if ( accessNumber == 99 )
            done = true;
         else // make move
         {
            currentRow = minRow;
            currentColumn = minColumn;
            board[ currentRow ][ currentColumn ] = ++moveNumber;
         } // end else
      } // end while

      System.out.printf( "The tour ended with %d moves.\n", moveNumber );

      if ( moveNumber == 64 )
         System.out.println( " This was a full tour!" );
      else
         System.out.println( " This was not a full tour." );

      printTour();
   } // end method tour

   // checks for next move
   public int nextMove( int row, int column )
   {
      int tempRow, tempColumn, tempMinRow, tempMinColumn;
      int tempAccessNumber = accessNumber;
      int tempAccess[][] = new int[ 8 ][ 8 ];

      for ( int i = 0; i < access.length; i++ )
         for ( int j = 0; j < access[ i ].length; j++ )
            tempAccess[ i ][ j ] = access[ i ][ j ];

      // try all possible moves
      for ( int moveType = 0; moveType < board.length; moveType++ )
      {
         // new position of hypothetical moves
         tempRow = row + vertical[ moveType ];
         tempColumn = column + horizontal[ moveType ];

         if ( validMove( tempRow, tempColumn ) )
         {
            // obtain access number
            if ( access[ tempRow ][ tempColumn ] < tempAccessNumber )
               tempAccessNumber = tempAccess[ tempRow ][ tempColumn ];

            // position access number tried
            --tempAccess[ tempRow ][ tempColumn ];
         } // end if
      } // end for

      return tempAccessNumber;
   }  // end method nextMove

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display Knight's tour path
   public void printTour()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%d", board[ row ][ column ] );

         System.out.println();
      } // end for
   } // end method printTour
} // end class Knight3
```

```
// Exercise 7.22 Part D Solution: Knight3Test.java
// Test application for class Knight3
public class Knight3Test
{
   public static void main( String args[] )
   {
      Knight3 application = new Knight3();
      application.tour();
   } // end main
} // end class Knight3Test
```

```
The tour ended with 64 moves.
 This was a full tour!
        0       1       2       3       4       5       6       7

0       48      13      36      33      54      15      38      19
1       35      32      49      14      37      18      55      16
2       12      47      34      63      58      53      20      39
3       31      62      59      50      41      56      17      52
4       60      11      46      57      64      51      40      21
5       27      30      61      42      45      24      3       6
6       10      43      28      25      8       5       22      1
7       29      26      9       44      23      2       7       4
```

7.23 (*Knight's Tour: Brute-Force Approaches*) In part (c) of Exercise 7.22, we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue to increase in power, we will be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let us call this approach "brute-force" problem solving.

a) Use random-number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves) at random. Your application should run one tour and display the final chessboard. How far did the knight get?

ANS:

```java
// Exercise 7.23 Part A Solution: Knight4.java
// Knights tour - Brute Force Approach. Uses random number
// generation to move around the board.
import java.util.Random;

public class Knight4
{
   Random randomNumbers = new Random();

   int board[][]; // gameboard

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   // runs a tour
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard
      int moveNumber = 0; // the current move number

      board = new int[ 8 ][ 8 ]; // gameboard

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move

      // randomize initial board position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      board[ currentRow ][ currentColumn ] = ++moveNumber;
      boolean done = false;

      // continue until knight can no longer move
      while ( !done )
      {
         boolean goodMove = false;

         // start with a random move
         int moveType = randomNumbers.nextInt( 8 );

         // check all possible moves until we find one that's legal
         for ( int count = 0; count < 8 && !goodMove;
            count++ )
         {
            testRow = currentRow + vertical[ moveType ];
            testColumn = currentColumn + horizontal[ moveType ];
            goodMove = validMove( testRow, testColumn );

            // test if new move is valid
            if ( goodMove )
            {
               currentRow = testRow;
               currentColumn = testColumn;
               board[ currentRow ][ currentColumn ] = ++moveNumber;
            } // end if

            moveType = ( moveType + 1 ) % 8;
         } // end for

         // if no valid moves, knight can no longer move
         if ( !goodMove )
            done = true;
         // if 64 moves have been made, a full tour is complete
         else if ( moveNumber == 64 )
            done = true;
      } // end while

      System.out.printf( "The tour ended with %d moves.\n", moveNumber );

      if ( moveNumber == 64 )
         System.out.println( "This was a full tour!" );
      else
         System.out.println( "This was not a full tour." );

      printTour();
   } // end method start

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display Knight's tour path
   public void printTour()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%d", board[ row ][ column ] );

         System.out.println();
      } // end for
   } // end method printTour
} // end class Knight4
```

```
// Exercise 7.23 Part A Solution: Knight4Test.java
// Test application for class Knight4
public class Knight4Test
{
   public static void main( String args[] )
   {
      Knight4 application = new Knight4();
      application.tour();
   } // end main
} // end class Knight4Test
```

```
The tour ended with 31 moves.
This was not a full tour.
        0       1       2       3       4       5       6       7

0       6       0       0       15      4       0       0       0
1       13      30      5       0       21      0       0       0
2       0       7       14      0       16      3       20      0
3       31      12      29      22      0       0       0       0
4       8       0       10      17      2       0       0       19
5       11      28      1       26      23      18      0       0
6       0       9       24      0       0       0       0       0
7       0       0       27      0       25      0       0       0
```

b) Most likely, the application in part (a) produced a relatively short tour. Now modify your application to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your application finishes attempting the 1000 tours, it should display this information in neat tabular format. What was the best result?

ANS:

```java
// Exercise 7.23 Part B Solution: Knight5.java
// Knights tour program - Brute Force Approach. Use random
// number generation to traverse the board. ( 1000 tours )
import java.util.Random;

public class Knight5
{
   Random randomNumbers = new Random();

   int board[][] = new int[ 8 ][ 8 ]; // gameboard

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   int moveTotals[] = new int[ 65 ]; // total number of tours per move

   // runs a tour
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move

      for ( int k = 0; k < 1000; k++ )
      {
         clearBoard();
         int moveNumber = 0; // the current move number

         // randomize initial board position
         currentRow = randomNumbers.nextInt( 8 );
         currentColumn = randomNumbers.nextInt( 8 );

         board[ currentRow ][ currentColumn ] = ++moveNumber;
         boolean done = false;

         // continue until knight can no longer move
         while ( !done )
         {
            boolean goodMove = false;

            int moveType = randomNumbers.nextInt( 8 );

            // check all possible moves until we find one that's legal
            for ( int count = 0; count < 8 && !goodMove; count++ )
            {
               testRow = currentRow + vertical[ moveType ];
               testColumn = currentColumn + horizontal[ moveType ];
               goodMove = validMove( testRow, testColumn );

               // test if new move is valid
               if ( goodMove )
               {
                  currentRow = testRow;
                  currentColumn = testColumn;
                  board[ currentRow ][ currentColumn ] = ++moveNumber;
               } // end if

               moveType = ( moveType + 1 ) % 8;
            } // end for

            // if no valid moves, knight can no longer move
            if ( !goodMove )
               done = true;
            // if 64 moves have been made, a full tour is complete
            else if ( moveNumber == 64 )
               done = true;
         } // end while

         ++moveTotals[ moveNumber ]; // update the statistics
      } // end for

      printResults();
   } // end method start

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display results on applet window
   public void printResults()
   {
      System.out.print( "# tours having # moves  " );
      System.out.print( "# tours having # moves\n\n" );

      // display results in tabulated columns
      for ( int row = 1; row < 33; row++ )
      {
         System.out.printf( "%-15d%-9d%-15d%d\n", moveTotals[ row ], row,
            moveTotals[ row + 32 ], ( row + 32 ) );
      } // end for
   } // end method printResults
```

```java

   // resets board
   public void clearBoard()
   {
      for ( int j = 0; j < board.length; j++ )
         for ( int k = 0; k < board[ j ].length; k++ )
            board[ j ][ k ] = 0;
   } // end method clearBoard
} // end class Knight5
```

```java
// Exercise 7.23 Part B Solution: Knight5Test.java
// Test application for class Knight5
public class Knight5Test
{
   public static void main( String args[] )
   {
      Knight5 application = new Knight5();
      application.tour();
   } // end main
} // end class Knight5Test
```

```
# tours having # moves  # tours having # moves

0              1        20             33
0              2        37             34
0              3        29             35
1              4        36             36
0              5        41             37
0              6        30             38
1              7        42             39
3              8        27             40
1              9        26             41
4              10       38             42
1              11       38             43
6              12       27             44
7              13       22             45
7              14       43             46
6              15       35             47
12             16       36             48
5              17       24             49
14             18       23             50
4              19       20             51
12             20       27             52
10             21       12             53
16             22       17             54
14             23       7              55
17             24       13             56
12             25       2              57
23             26       4              58
21             27       1              59
29             28       0              60
24             29       2              61
21             30       1              62
17             31       0              63
32             32       0              64
```

c) Most likely, the application in part (b) gave you some “respectable” tours, but no full tours. Now let your application run until it produces a full tour. (*Caution*: This version of the application could run for hours on a powerful computer.) Once again, keep a table of the number of tours of each length, and display this table when the first full tour is found. How many tours did your application attempt before producing a full tour? How much time did it take?

ANS:

```
// Exercise 7.23 Part C Solution: Knight6.java
// Knights tour program - Brute Force Approach. Use random
// number generation to traverse the board until a full tour is found
import java.util.Random;

public class Knight6
{
   Random randomNumbers = new Random();

   int board[][] = new int[ 8 ][ 8 ]; // gameboard

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   int moveTotals[] = new int[ 65 ]; // total number of tours per move

   // runs a tour
   public void tour()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard

      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move

      boolean fullTour = false;

      while ( !fullTour )
      {
         clearBoard();
         int moveNumber = 0; // the current move number

         // randomize initial board position
         currentRow = randomNumbers.nextInt( 8 );
         currentColumn = randomNumbers.nextInt( 8 );

         board[ currentRow ][ currentColumn ] = ++moveNumber;
         boolean done = false;

         // continue until knight can no longer move
         while ( !done )
         {
            boolean goodMove = false;

            int moveType = randomNumbers.nextInt( 8 );

            // check all possible moves until we find one that's legal
            for ( int count = 0; count < 8 && !goodMove; ++count )
            {
               testRow = currentRow + vertical[ moveType ];
               testColumn = currentColumn + horizontal[ moveType ];
               goodMove = validMove( testRow, testColumn );

               // test if new move is valid
               if ( goodMove )
               {
                  currentRow = testRow;
                  currentColumn = testColumn;
                  board[ currentRow ][ currentColumn ] = ++moveNumber;
               } // end if

               moveType = ( moveType + 1 ) % 8;
            } // end for

            // if no valid moves, knight can no longer move
            if ( !goodMove )
               done = true;
            // if 64 moves have been made, a full tour is complete
            else if ( moveNumber == 64 )
            {
               done = true;
               fullTour = true;
            } // end else if
         } // end while

         ++moveTotals[ moveNumber ]; // update the statistics

      } // end for

      printResults();
   } // end method start

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display results on applet window
   public void printResults()
   {
      int totalTours = 0; // total number of moves

      System.out.print( "# tours having # moves  " );
      System.out.print"# tours having # moves\n\n" );

      // display results in tabulated columns
      for ( int row = 1; row < 33; row++ )
      {
         System.out.printf( "%-15d%-9d%-15d%d\n", moveTotals[ row ], row,
            moveTotals[ row + 32 ], ( row + 32 ) );

         totalTours += moveTotals[ row ] + moveTotals[ row + 32 ];
      } // end for

      System.out.printf( "\nIt took %d tries to get a full tour\n",
         totalTours );
   } // end method printResults

   // resets board
   public void clearBoard()
   {
      for ( int j = 0; j < board.length; j++ )
         for ( int k = 0; k < board[ j ].length; k++ )
            board[ j ][ k ] = 0;
   } // end method clearBoard
} // end class Knight6
```

```
// Exercise 7.23 Part C Solution: Knight6Test.java
// Test application for class Knight6
public class Knight6Test
{
   public static void main( String args[] )
   {
      Knight6 application = new Knight6();
      application.tour();
   } // end main
} // end class Knight6Test
```

```
# tours having # moves  # tours having # moves

0              1        636            33
0              2        783            34
0              3        676            35
15             4        802            36
16             5        771            37
48             6        941            38
36             7        802            39
77             8        970            40
64             9        805            41
83             10       922            42
99             11       784            43
135            12       907            44
115            13       776            45
162            14       834            46
154            15       687            47
202            16       794            48
184            17       637            49
253            18       646            50
222            19       493            51
318            20       475            52
311            21       362            53
394            22       324            54
345            23       241            55
416            24       192            56
384            25       144            57
518            26       113            58
463            27       54             59
553            28       53             60
538            29       22             61
601            30       8              62
566            31       0              63
734            32       1              64

It took 24661 tries to get a full tour
```

d) Compare the brute-force version of the Knight's Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem solving in general.

7.24 (*Eight Queens*) Another puzzler for chess buffs is the Eight Queens problem, which asks the following: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other (i.e., no two queens are in the same row, in the same column or along the same diagonal)? Use the thinking developed in Exercise 7.22 to formulate a heuristic for solving the Eight Queens problem. Run your application. (*Hint*: It is possible to assign a value to each square of the chessboard to indicate how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as demonstrated by Fig. 7.33. Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be as follows: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

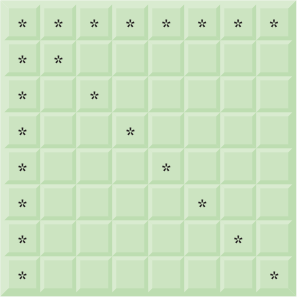


Fig. 7.33 | The 22 squares eliminated by placing a queen in the upper left corner.

ANS:

```java
// Exercise 7.24 Solution: EightQueens.java
// EightQueens - heuristic version
import java.util.Random;

public class EightQueens
{
   Random randomNumbers = new Random();

   boolean board[][]; // gameboard

   // accessibility values for each board position
   int access[][] = { { 22, 22, 22, 22, 22, 22, 22, 22 },
                      { 22, 24, 24, 24, 24, 24, 24, 22 },
                      { 22, 24, 26, 26, 26, 26, 24, 22 },
                      { 22, 24, 26, 28, 28, 26, 24, 22 },
                      { 22, 24, 26, 28, 28, 26, 24, 22 },
                      { 22, 24, 26, 26, 26, 26, 24, 22 },
                      { 22, 24, 24, 24, 24, 24, 24, 22 },
                      { 22, 22, 22, 22, 22, 22, 22, 22 } };
   int maxAccess = 99; // dummy value to indicate a queen has been placed

   int queens; // number of queens placed on the board

   // attempts to place eight queens on a chessboard
   public void placeQueens()
   {
      int currentRow; // the row position on the chessboard
      int currentColumn; // the column position on the chessboard

      board = new boolean[ 8 ][ 8 ];

      // initialize board to false
      for ( int i = 0; i < board.length; i++ )
      {
         for ( int j = 0; j < board[ i ].length; j++ )
            board[ i ][ j ] = false;
      } // end for

      // randomize initial first queen position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      board[ currentRow ][ currentColumn ] = true;
      ++queens;

      updateAccess( currentRow, currentColumn ); // update access

      boolean done = false;

      // continue until finished traversing
      while ( !done )
      {
         // the current lowest access number
         int accessNumber = maxAccess;

         // find square with the smallest elimination number
         for ( int row = 0; row < board.length; row++ )
         {
            for ( int col = 0; col < board.length; col++ )
            {
               // obtain access number
               if ( access[ row ][ col ] < accessNumber )
               {
                  accessNumber = access[ row ][ col ];
                  currentRow = row;
                  currentColumn = col;
               } // end if
            } // end innner for
         } // end outer for

         // traversing done
         if ( accessNumber == maxAccess )
            done = true;
         // mark the current location
         else
         {
            board[ currentRow ][ currentColumn ] = true;
            updateAccess( currentRow, currentColumn );
            queens++;
         } // end else
      } // end while

      printBoard();
   } // end method placeQueens

   // update access array
   public void updateAccess( int row, int column )
   {
      for ( int i = 0; i < 8; i++ )
      {
         // set elimination numbers to 99
         // in the row occupied by the queen
         access[ row ][ i ] = maxAccess;

         // set elimination numbers to 99
         // in the column occupied by the queen
         access[ i ][ column ] = maxAccess;
      } // end for

      // set elimination numbers to 99 in diagonals occupied by the queen
      updateDiagonals( row, column );
   } // end method updateAccess

   // place 99 in diagonals of position in all 4 directions
   public void updateDiagonals( int rowValue, int colValue )
   {
      int row = rowValue; // row postion to be updated
      int column = colValue; // column position to tbe updated

      // upper left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, --column ); diagonal++ )
         access[ row ][ column ] = maxAccess;

      row = rowValue;
      column = colValue;

      // upper right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, ++column ); diagonal++ )
         access[ row ][ column ] = maxAccess;

      row = rowValue;
      column = colValue;

      // lower left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, --column ); diagonal++ )
         access[ row ][ column ] = maxAccess;

      row = rowValue;
```

```java
      column = colValue;

      // lower right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, ++column ); diagonal++ )
         access[ row ][ column ] = maxAccess;
   } // end method updateDiagonals

   // check for valid move
   public boolean validMove( int row, int column )
   {
      return ( row >= 0 && row < 8 && column >= 0 && column < 8 );
   } // end method validMove

   // display the board
   public void printBoard()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
         {
            System.out.print( "\t" );

            if ( board[ row ][ column ] )
               System.out.print( "Q" );
            else
               System.out.print( "." );
         } // end for

         System.out.println();
      } // end for

      System.out.printf ( "\n%d queens placed on the board.\n", queens );
   } // end method printBoard
} // end class EightQueens
```

```java
// Exercise 7.24 Solution: EightQueensTest.java
// Test application for class EightQueens
public class EightQueensTest
{
   public static void main( String args[] )
   {
      EightQueens application = new EightQueens();
      application.placeQueens();
   } // end main
} // end class EightQueensTest
```

```
        0       1       2       3       4       5       6       7

0       .       .       .       .       .       .       Q       .
1       Q       .       .       .       .       .       .       .
2       .       .       .       .       .       .       .       Q
3       .       .       .       .       .       .       .       .
4       .       .       .       .       .       .       .       .
5       .       .       Q       .       .       .       .       .
6       .       .       .       .       Q       .       .       .
7       .       Q       .       .       .       .       .       .

6 queens placed on the board.
```

7.25 (*Eight Queens: Brute-Force Approaches*) In this exercise, you will develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 7.24.

a) Use the random brute-force technique developed in Exercise 7.23 to solve the Eight Queens problem.

ANS:

```
// Exercise 7.25 PartA Solution: EightQueens1.java
// Uses a random brute force approach to solve the eight queens problem.
import java.util.Random;

public class EightQueens1
{
   Random randomNumbers = new Random();

   char board[][]; // chess board
   int queens; // number of queens placed

   // place queens on board
   public void placeQueens()
   {
      // repeat until solved
      while (queens < 8)
      {
         int rowMove; // column move
         int colMove; // row move
         boolean done = false; // indicates if all squares filled

         // reset the board
         board = new char[ 8 ][ 8 ];
         queens = 0;

         // continue placing queens until no more squares
         // or not all queens placed
         while ( !done )
         {
            // randomize move
            rowMove = randomNumbers.nextInt( 8 );
            colMove = randomNumbers.nextInt( 8 );

            // if valid move, place queen and mark off conflict squares
            if ( queenCheck( rowMove, colMove ) )
            {
               board[ rowMove ][ colMove ] = 'Q';
               xConflictSquares( rowMove, colMove );
               ++queens;
            } // end if

            // done when no more squares left
            if ( !availableSquare() )
               done = true;
         } // end inner while loop
      } // end outer while loop

      printBoard();
   } // end method placeQueens

   // check for valid move
   public boolean validMove( int row, int column )
   {
      return ( row >= 0 && row < 8 && column >= 0 && column < 8 );
   } // end method validMove

   // check if any squares left
   public boolean availableSquare()
   {
      for ( int row = 0; row < board.length; row++ )
         for ( int col = 0; col < board[ row ].length; col++ )
            if ( board[ row ][ col ] == '\0' )
               return true; // at least one available square

      return false; // no available squares
   } // end method availableSquare

   // check if a queen can be placed without being attacked
   public boolean queenCheck( int rowValue, int colValue )
   {
      int row = rowValue, column = colValue;

      // check row and column for a queen
      for ( int position = 0; position < 8; position++ )
         if ( board[ row ][ position ] == 'Q' ||
            board[ position ][ column ] == 'Q' )
            return false;

      // check upper left diagonal for a queen
      for ( int square = 0; square < 8 &&
         validMove( --row, --column ); square++ )
         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check upper right diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, ++column ); diagonal++ )
         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check lower left diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, --column ); diagonal++ )
         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check lower right diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, ++column ); diagonal++ )
         if ( board[ row ][ column ] == 'Q' )
            return false;

      return true; // no queen in conflict
   } // end method queenCheck

   // conflicting square marked with *
   public void xConflictSquares( int row, int col )
   {
      for ( int i = 0; i < 8; i++ ) {

         // place a '*' in the row occupied by the queen
         if ( board[ row ][ i ] == '\0' )
            board[ row ][ i ] = '*';

         // place a '*' in the col occupied by the queen
         if ( board[ i ][ col ] == '\0' )
            board[ i ][ col ] = '*';
      } // end for

      // place a '*' in the diagonals occupied by the queen
      xDiagonals( row, col );
   } // end method xConflictSquares

   // place * in diagonals of position in all 4 directions
   public void xDiagonals( int rowValue, int colValue )
   {
      int row = rowValue, column = colValue;

      // upper left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, --column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // upper right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, ++column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // lower left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, --column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // lower right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, ++column ); diagonal++ )
         board[ row ][ column ] = '*';
   } // end method xDiagonals

   // prints the chessboard
   public void printBoard()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%c", board[ row ][ column ] );

         System.out.println();
      } // end for

      System.out.printf ( "\n%d queens placed on the board.\n", queens );
   } // end method printBoard
} // end class EightQueens1
```

```java
// Exercise 7.25 Part A Solution: EightQueens1Test.java
// Test application for class EightQueens1
public class EightQueens1Test
{
   public static void main( String args[] )
   {
      EightQueens1 application = new EightQueens1();
      application.placeQueens();
   } // end main
} // end class EightQueens1Test
```

```
        0       1       2       3       4       5       6       7

0       *       *       *       Q       *       *       *       *
1       *       *       *       *       *       *       Q       *
2       *       *       Q       *       *       *       *       *
3       *       *       *       *       *       *       *       Q
4       *       Q       *       *       *       *       *       *
5       *       *       *       *       Q       *       *       *
6       Q       *       *       *       *       *       *       *
7       *       *       *       *       *       Q       *       *

8 queens placed on the board.
```

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard) to solve the Eight Queens problem.

ANS:

```java
// Exercise 7.25 Part B Solution: EightQueens2.java
// Uses an exhaustive technique to solve the eight queens problem
import java.util.Random;

public class EightQueens2
{
   char board[][] = new char[ 8 ][ 8 ]; // chess board
   int queens; // number of queens placed

   // place queens on board
   public void placeQueens()
   {
      for ( int firstQueenRow = 0;
         firstQueenRow < board[ 0 ].length && queens < 8;
         firstQueenRow++ )
      {
         for ( int firstQueenCol = 0;
            firstQueenCol < board[ 0 ].length && queens < 8;
            firstQueenCol++ )
         {
            // reset the board
            board = new char[ 8 ][ 8 ];
            queens = 0;

            // place first queen at starting position
            board[ firstQueenRow ][ firstQueenCol ] = 'Q';
            xConflictSquares( firstQueenRow, firstQueenCol );
            ++queens;

            // remaining queens will be placed in board

            boolean done = false;   // indicates if all squares filled

            // try all possible locations on board
            for ( int rowMove = 0;
               rowMove < board[ 0 ].length && !done; rowMove++ )
            {
               for ( int colMove = 0;
                  colMove < board[0].length && !done; colMove++ )
               {
                  // if valid move, place queen
                  // and mark off conflict squares
                  if ( queenCheck( rowMove, colMove ) )
                  {
                     board[ rowMove ][ colMove ] = 'Q';
                     xConflictSquares( rowMove, colMove );
                     ++queens;
                  } // end if

                  // done when no more squares left
                  if ( !availableSquare() )
                     done = true;
               } // end for colMove
            } // end for rowMove
         } // end for firstQueenCol
      } // end for firstQueenRow

      printBoard();
   }  // end method placeQueens

   // check for valid move
   public boolean validMove( int row, int column )
   {
      return ( row >= 0 && row < 8 && column >= 0 && column < 8 );
   } // end method validMove

   // check if any squares left
   public boolean availableSquare()
   {
      for ( int row = 0; row < board.length; row++ )
         for ( int col = 0; col < board[ row ].length; col++ )
            if ( board[ row ][ col ] == '\0' )
               return true; // at least one available square

      return false; // no available squares

   } // end method availableSquare

   // conflicting square marked with *
   public void xConflictSquares( int row, int col )
   {
      for ( int i = 0; i < 8; i++ ) {

         // place a '*' in the row occupied by the queen
         if ( board[ row ][ i ] == '\0' )
            board[ row ][ i ] = '*';

         // place a '*' in the col occupied by the queen
         if ( board[ i ][ col ] == '\0' )
            board[ i ][ col ] = '*';
      } // end for

      // place a '*' in the diagonals occupied by the queen
      xDiagonals( row, col );
   } // end method xConflictSquares

   // check if queens can "attack" each other
   public boolean queenCheck( int rowValue, int colValue )
   {
      int row = rowValue, column = colValue;

      // check row and column for a queen
      for ( int position = 0; position < 8; position++ )
         if ( board[ row ][ position ] == 'Q' ||
            board[ position ][ column ] == 'Q' )

            return false;

      // check upper left diagonal for a queen
      for ( int square = 0; square < 8 &&
         validMove( --row, --column ); square++ )

         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check upper right diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, ++column ); diagonal++ )

         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check lower left diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, --column ); diagonal++ )

         if ( board[ row ][ column ] == 'Q' )
            return false;

      row = rowValue;
      column = colValue;

      // check lower right diagonal for a queen
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, ++column ); diagonal++ )

         if ( board[ row ][ column ] == 'Q' )
            return false;

      return true;   // no queen in conflict
   }  // end method queenCheck

   // place * in diagonals of position in all 4 directions
   public void xDiagonals( int rowValue, int colValue )
   {
      int row = rowValue, column = colValue;

      // upper left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, --column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // upper right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( --row, ++column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // lower left diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, --column ); diagonal++ )
         board[ row ][ column ] = '*';

      row = rowValue;
      column = colValue;

      // lower right diagonal
      for ( int diagonal = 0; diagonal < 8 &&
         validMove( ++row, ++column ); diagonal++ )
         board[ row ][ column ] = '*';
   } // end method xDiagonals

   // prints the chessboard
   public void printBoard()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%c", board[ row ][ column ] );

         System.out.println();
      } // end for

      System.out.printf ( "\n%d queens placed on the board.\n", queens );
   } // end method printBoard
} // end class EightQueens2
```

```
// Exercise 7.25 Part B Solution: EightQueens2Test.java
// Test application for class EightQueens2
public class EightQueens2Test
{
   public static void main( String args[] )
   {
      EightQueens2 application = new EightQueens2();
      application.placeQueens();
   } // end main
} // end class EightQueens2Test
```

```
        0       1       2       3       4       5       6       7

0       *       Q       *       *       *       *       *       *
1       *       *       *       Q       *       *       *       *
2       *       *       *       *       *       Q       *       *
3       *       *       *       *       *       *       *       Q
4       *       *       Q       *       *       *       *       *
5       Q       *       *       *       *       *       *       *
6       *       *       *       *       *       *       Q       *
7       *       *       *       *       Q       *       *       *

8 queens placed on the board.
```

c) Why might the exhaustive brute-force approach not be appropriate for solving the Knight's Tour problem?

d) Compare and contrast the random brute-force and exhaustive brute-force approaches.

7.26 (*Knight's Tour: Closed-Tour Test*) In the Knight's Tour (Exercise 7.22), a full tour occurs when the knight makes 64 moves, touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the application you wrote in Exercise 7.22 to test for a closed tour if a full tour has occurred.

ANS:

```java
// Exercise 7.26 Solution: Knight7.java
// Knight's Tour - heuristic version, Closed Tour
import java.util.Random;

public class Knight7
{
   Random randomNumbers = new Random();

   int access[][] = { { 2, 3, 4, 4, 4, 4, 3, 2 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 4, 6, 8, 8, 8, 8, 6, 4 },
                      { 3, 4, 6, 6, 6, 6, 4, 3 },
                      { 2, 3, 4, 4, 4, 4, 3, 2 } };

   int board[][]; // gameboard
   int currentRow; // the row position on the chessboard
   int currentColumn; // the column position on the chessboard
   int firstRow; // the initial row position
   int firstColumn; // the initial column position
   int moveNumber = 0; // the current move number
   int accessNumber; // the current access number

   // moves
   int horizontal[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int vertical[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

   // initialize applet
   public void tour()
   {
      int testRow; // row position of next possible move
      int testColumn; // column position of next possible move
      int minRow = -1; // row position of move with minimum access
      int minColumn = -1; // row position of move with minimum access

      board = new int[ 8 ][ 8 ];

      // randomize initial board position
      currentRow = randomNumbers.nextInt( 8 );
      currentColumn = randomNumbers.nextInt( 8 );

      firstRow = currentRow;
      firstColumn = currentColumn;

      board[ currentRow ][ currentColumn ] = ++moveNumber;
      boolean done = false;

      // continue touring until finished traversing
      while ( !done )
      {
         accessNumber = 99;

         // try all possible moves
         for ( int moveType = 0; moveType < board.length; moveType++ )
         {
            // new position of hypothetical moves
            testRow = currentRow + vertical[ moveType ];
            testColumn = currentColumn + horizontal[ moveType ];

            if ( validMove( testRow, testColumn ) )
            {
               // obtain access number
               if ( access[ testRow ][ testColumn ] < accessNumber )
               {
                  // if this is the lowest access number thus far,
                  // then set this move to be our next move
                  accessNumber = access[ testRow ][ testColumn ];

                  minRow = testRow;
                  minColumn = testColumn;
               } // end if

               // position access number tried
               --access[ testRow ][ testColumn ];
            } // end if
         } // end for

         // traversing done
         if ( accessNumber == 99 ) // no valid moves
            done = true;
         else // make move
         {
            currentRow = minRow;
            currentColumn = minColumn;
            board[ currentRow ][ currentColumn ] = ++moveNumber;
         } // end else
      } // end while

      System.out.printf( "The tour ended with %d moves.\n", moveNumber );

      if ( moveNumber == 64 )
      {
         if ( closedTour() )
            System.out.println( " This was a CLOSED tour!" );
         else
            System.out.println(
               " This was a full tour, but it wasn't closed." );
      } // end if
      else
         System.out.println( " This was not a full tour." );

      printTour();
   } // end method tour

   // check for a closed tour if the last move can reach the initial
   // starting position
   public boolean closedTour()
   {
      // test all 8 possible moves to check if move
      // would position knight on first move
      for ( int moveType = 0; moveType < 8; moveType++ )
      {
         int testRow = currentRow + vertical[ moveType ];
         int testColumn = currentColumn + horizontal[ moveType ];

         // if one move away from initial move
         if ( testRow == firstRow &&
            testColumn == firstColumn )
         {
            return true;
         } // end if
      } // end for

      return false;
   } // end method closedTour

   // checks for valid move
   public boolean validMove( int row, int column )
   {
      // returns false if the move is off the chessboard, or if
      // the knight has already visited that position
      // NOTE: This test stops as soon as it becomes false
      return ( row >= 0 && row < 8 && column >= 0 && column < 8
         && board[ row ][ column ] == 0 );
   } // end method validMove

   // display Knight's tour path
   public void printTour()
   {
      // display numbers for column
      for ( int k = 0; k < 8; k++ )
         System.out.printf( "\t%d", k );

      System.out.print( "\n\n" );

      for ( int row = 0; row < board.length; row++ )
      {
         System.out.print ( row );

         for ( int column = 0; column < board[ row ].length; column++ )
            System.out.printf( "\t%d", board[ row ][ column ] );

         System.out.println();
      } // end for
   } // end method printTour
} // end class Knight7
```

```java
// Exercise 7.26 Solution: KnightTest7.java
// Test application for class Knight7
public class KnightTest7
{
   public static void main( String args[] )
   {
      Knight7 application = new Knight7();
      application.tour();
   } // end main
} // end class KnightTest7
```

```
The tour ended with 64 moves.
 This was a CLOSED tour!
        0       1       2       3       4       5       6       7

0       14      11      16      47      30      9       32      41
1       17      48      13      10      43      40      29      8
2       12      15      52      39      46      31      42      33
3       53      18      49      44      51      56      7       28
4       22      1       54      61      38      45      34      57
5       19      62      21      50      55      60      27      6
6       2       23      64      37      4       25      58      35
7       63      20      3       24      59      36      5       26
```

7.27 (*Sieve of Eratosthenes*) A prime number is any integer greater than 1 that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create a primitive type `boolean` array with all elements initialized to `true`. Array elements with prime indices will remain `true`. All other array elements will eventually be set to `false`.

b) Starting with array index 2, determine whether a given element is `true`. If so, loop through the remainder of the array and set to `false` every element whose index is a multiple of the index for the element with value `true`. Then continue the process with the next element with value `true`. For array index 2, all elements beyond element 2 in the array that have indices which are multiples of 2 (indices 4, 6, 8, 10, etc.) will be set to `false`; for array index 3, all elements beyond element 3 in the array that have indices which are multiples of 3 (indices 6, 9, 12, 15, etc.) will be set to `false`; and so on.

When this process completes, the array elements that are still `true` indicate that the index is a prime number. These indices can be displayed. Write an application that uses an array of 1000 elements to determine and display the prime numbers between 2 and 999. Ignore array elements 0 and 1.

ANS:

```java
// Exercise 7.27 Solution: Sieve.java
// Sieve of Eratosthenes
public class Sieve
{
   public static void main( String args[] )
   {
      int count = 0; // the number of primes found

      boolean primes[] = new boolean[ 1000 ]; // array of primes

      // initialize all array values to true
      for ( int index = 0; index < primes.length; index++ )
         primes[ index ] = true;

      // starting at the third value, cycle through the array and put 0
      // as the value of any greater number that is a multiple
      for ( int i = 2; i < primes.length; i++ )
         if ( primes[ i ] )
         {
            for ( int j = i + i; j < primes.length; j += i )
               primes[ j ] = false;
         } // end if

      // cycle through the array one last time to print all primes
      for ( int index = 2; index < primes.length; index++ )
         if ( primes[ index ] )
         {
            System.out.printf( "%d is prime.\n", index );
            ++count;
         } // end if

      System.out.printf( "\n%d primes found.\n", count );
   } // end main
} // end class Sieve
```

```
2 is prime.
3 is prime.
5 is prime.
7 is prime.

.
.
.

977 is prime.
983 is prime.
991 is prime.
997 is prime.

168 primes found.
```

7.28 (*Simulation: The Tortoise and the Hare*) In this problem, you will re-create the classic race of the tortoise and the hare. You will use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in Fig. 7.34. Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the “starting gate”). If an animal slips left before square 1, move it back to square 1.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 7.34 | Rules for adjusting the positions of the tortoise and the hare.

Generate the percentages in Fig. 7.34 by producing a random integer i in the range $1 \le i \le 10$. For the tortoise, perform a “fast plod” when $1 \le i \le 5$, a “slip” when $6 \le i \le 7$ or a “slow plod” when $8 \le i \le 10$. Use a similar technique to move the hare.

Begin the race by displaying

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

Then, for each tick of the clock (i.e., each repetition of a loop), display a 70-position line showing the letter `T` in the position of the tortoise and the letter `H` in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display `OUCH!!!` beginning at that position. All output positions other than the `T`, the `H` or the `OUCH!!!` (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display `TORTOISE WINS!!! YAY!!!` If the hare wins, display `Hare wins. Yuch.` If both animals win on the same tick of the clock, you may want to favor the tortoise (the “underdog”), or you may want to display `It's a tie`. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you are ready to run your application, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

Later in the book, we introduce a number of Java capabilities, such as graphics, images, animation, sound and multithreading. As you study those features, you might enjoy enhancing your tortoise-and-hare contest simulation.

ANS:

```java
// Exercise 7.28 Solution: Race.java
// Program simulates the race between the tortoise and the hare
import java.util.Random;

```

```java
public class Race
{
   static final int RACE_END = 70;  // final position

   Random randomNumbers = new Random();

   int tortoise; // toroise's position
   int hare; // hare's position
   int timer; // clock ticks elapsed

   // run the race
   public void startRace()
   {
      tortoise = 1;
      hare = 1;
      timer = 0;

      System.out.println( "ON YOUR MARK, GET SET" );
      System.out.println( "BANG !!!!!" );
      System.out.println( "AND THEY'RE OFF !!!!!" );

      while ( tortoise < RACE_END && hare < RACE_END )
      {
         moveHare();
         moveTortoise();
         printCurrentPositions();

         // slow down race
         for ( int temp = 0; temp < 100000000; temp++ );

         ++timer;
      } // end while

      // tortoise beats hare or a tie
      if ( tortoise >= hare )
         System.out.println( "\nTORTOISE WINS!!! YAY!!!" );
      // hare beat tortoise
      else
         System.out.println( "\nHare wins. Yuch!" );

      System.out.printf( "TIME ELAPSED = %d seconds\n", timer );
   }

   // move tortoise's position
   public void moveTortoise()
   {
      // randomize move to choose
      int percent = 1 + randomNumbers.nextInt( 10 );

      // determine moves by percent in range in Fig 7.32
      // fast plod
      if ( percent >= 1 && percent <= 5 )
         tortoise += 3;
      // slip
      else if ( percent == 6 || percent == 7 )
         tortoise -= 6;
      // slow plod
      else
         ++tortoise;

      // ensure tortoise doesn't slip beyond start position
      if ( tortoise < 1 )
         tortoise = 1;

      // ensure tortoise doesn't pass the finish
      else if ( tortoise > RACE_END )
         tortoise = RACE_END;
   } // end method move Tortoise

   // move hare's position
   public void moveHare()
   {
      // randomize move to choose
      int percent = 1 + randomNumbers.nextInt( 10 );

      // determine moves by percent in range in Fig 7.32
      // big hop
      if ( percent == 3 || percent == 4 )
         hare += 9;
      // big slip
      else if ( percent == 5 )
         hare -= 12;
      // small hop
      else if ( percent >= 6 && percent <= 8 )
         ++hare;
      // small slip
      else if ( percent > 8 )
         hare -= 2;

      // ensure that hare doesn't slip beyond start position
      if ( hare < 1 )
         hare = 1;
      // ensure hare doesn't pass the finish
      else if ( hare > RACE_END )
         hare = RACE_END;
   } // end method moveHare

   // display positions of tortoise and hare
   public void printCurrentPositions()
   {
      // goes through all 70 squares, printing H
      // if hare on position and T for tortoise
      for ( int count = 1; count <= RACE_END; count++ )
         // tortoise and hare positions collide
         if ( count == tortoise && count == hare )
            System.out.print( "OUCH!!!" );
         else if ( count == hare )
            System.out.print( "H" );
         else if ( count == tortoise )
            System.out.print( "T" );
         else
            System.out.print( " " );

      System.out.println();
   } // end printCurrentPositions
} // end class Race
```

```
// Exercise 7.28 Solution: RaceTest.java
// Test application for class Race
public class RaceTest
{
   public static void main( String args[] )
   {
      Race application = new Race();
      application.startRace();
   } // end main
} // end class RaceTest
```

```
ON YOUR MARK, GET SET
BANG !!!!!
AND THEY'RE OFF !!!!!
H  T
H   T
H      T
 H        T
  H          T
  H             T
  H                T
           H T
            H   T
H                  T
H                     T
H                      T
 H                        T
H                          T
 H                            T
H                                T
 H                                T
H                                    T
H                                       T
         H                                 T
         H                                    T
          H                                    T
                   H                     T
       H                                    T
     H                                         T
      H                                           T
    H                                                T
             H                                        T
              H                                          T
            H                                               T
            H                                                  T
H                                                                 T
 H                                                                   T

TORTOISE WINS!!! YAY!!!
TIME ELAPSED = 33 seconds
```

7.29 *(Fibonacci Series)* The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

a) Write a method `fibonacci( n )` that calculates the *n*th Fibonacci number. Incorporate this method into an application that enables the user to enter the value of `n`.

ANS:

```java
// Exercise 7.29 Part A Solution: Series1.java
// Program calculates the Fibonacci series iteratively
import java.util.Scanner;

```

```java
public class Series1
{
   // finds elements in the Fibonacci series
   public void findElements()
   {
      Scanner input = new Scanner( System.in );

      System.out.print( "Enter n: (n < 0 to exit): " );
      int element = input.nextInt();

      while ( element >= 0 )
      {
         int value = fibonacci( element );
         System.out.printf( "Fibonacci number is " );
         System.out.println( value );
         System.out.print( "Enter n: (n < 0 to exit): " );
         element = input.nextInt();
      } // end while
   } // end method findElements

   // returns fibonacci number of nth element
   public int fibonacci( int nElement )
   {
      int temp = 1; // number to be added
      int fibNumber = 0; // fibonacci number

      if ( nElement == 1 )
         return 0;

      // find nth element
      for ( int n = 2; n <= nElement; n++ )
      {
         int last = fibNumber;
         fibNumber += temp;

         temp = last;
      } // end for

      return fibNumber;
   } // end method fibonacci
} // end class Series1
```

```java
// Exercise 7.29 Part A Solution: Series1Test.java
// Test application for class Series1
public class Series1Test
{
   public static void main( String args[] )
   {
      Series1 application = new Series1();
      application.findElements();
   } // end main
} // end class Series1Test
```

```
Enter n: (n < 0 to exit): 5
Fibonacci number is 3
Enter n: (n < 0 to exit): 77
Fibonacci number is 1412467027
Enter n: (n < 0 to exit): -1
```

b) Determine the largest Fibonacci number that can be displayed on your system.

c) Modify the application you wrote in part (a) to use `double` instead of `int` to calculate and return Fibonacci numbers, and use this modified application to repeat part (b).

ANS:

```java
// Exercise 7.29 Part C Solution: Series2.java
// Program calculates the Fibonacci series iteratively
import java.util.Scanner;

public class Series2
{
   // finds elements in the Fibonacci series
   public void findElements()
   {
      Scanner input = new Scanner( System.in );

      System.out.print( "Enter n: (n < 0 to exit): " );
      int element = input.nextInt();

      while ( element >= 0 )
      {
         double value = fibonacci( element );
         System.out.printf( "Fibonacci number is " );
         System.out.println( value );
         System.out.print( "Enter n: (n < 0 to exit): " );
         element = input.nextInt();
      } // end while
   } // end method findElements

   // returns fibonacci number of nth element
   public double fibonacci( int nElement )
   {
      double temp = 1; // number to be added
      double fibNumber = 0; // fibonacci number

      if ( nElement == 1 )
         return 0;

      // find nth element
      for ( int n = 2; n <= nElement; n++ )
      {
         double last = fibNumber;
         fibNumber += temp;

         temp = last;
      } // end for

      return fibNumber;
   } // end method fibonacci
} // end class Series2
```

```
// Exercise 7.29 Part C Solution: Series2Test.java
// Test application for class Series2
public class Series2Test
{
   public static void main( String args[] )
   {
      Series2 application = new Series2();
      application.findElements();
   } // end main
} // end class Series2Test
```

```
Enter n: (n < 0 to exit): 5
Fibonacci number is 3.0
Enter n: (n < 0 to exit): 1477
Fibonacci number is 1.3069892237633987E308
Enter n: (n < 0 to exit): -1
```

Exercises 7.30—7.33 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.

7.30 *(Card Shuffling and Dealing)* Modify the application of Fig. 7.11 to deal a five-card poker hand. Then modify class DeckOfCards of Fig. 7.10 to include methods that determine whether a hand contains

a) a pair
b) two pairs
c) three of a kind (e.g., three jacks)
d) four of a kind (e.g., four aces)
e) a flush (i.e., all five cards of the same suit)
f) a straight (i.e., five cards of consecutive face values)
g) a full house (i.e., two cards of one face value and three cards of another face value)

[*Hint:* Add methods getFace and getSuit to class Card of Fig. 7.9.]

ANS:

```
// Exercise 7.30 Solution: Card.java
// Card class represents a playing card.

public class Card
{
   private String face; // face of card
   private String suit; // suit of card

   // two-argument constructor initializes card's face and suit
   public Card( String cardFace, String cardSuit )
```

```java
   {
      face = cardFace; // initialize face of card
      suit = cardSuit; // initialize suit of card
   } // end two-argument Card constructor

   // return card face
   public String getFace()
   {
      return face;
   } // end method getFace

   // return card suit
   public String getSuit()
   {
      return suit;
   } // end method getSuit

   // return String representation of Card
   public String toString()
   {
      return face + " of " + suit;
   } // end method toString
} // end class Card
```

```java
// Exercise 7.30 Solution: DeckOfCards.java
// DeckOfCards class represents a deck of playing cards.
import java.util.Random;

public class DeckOfCards
{
   String faces[] = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
      "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
   String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
   private Card deck[]; // array of Card objects
   private int currentCard; // the index of the next Card to be dealt
   private final int NUMBER_OF_CARDS = 52; // constant number of cards
   private Random randomNumbers; // random number generator

   // constructor fills deck of cards
   public DeckOfCards()
   {
      deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
      currentCard = 0; // initialize currentCard
      randomNumbers = new Random(); // create random number generator

      // populate deck with Card objects
      for ( int count = 0; count < deck.length; count++ )
         deck[ count ] =
            new Card( faces[ count % 13 ], suits[ count / 13 ] );
   } // end DeckOfCards constructor

   // shuffle deck of cards with one-pass algorithm
   public void shuffle()
   {
      currentCard = 0; // reinitialize currentCard

      // for each card, pick another random card and swap them
      for ( int first = 0; first < deck.length; first++ )
      {
         int second =  randomNumbers.nextInt( NUMBER_OF_CARDS );
         Card temp = deck[ first ];
         deck[ first ] = deck[ second ];
         deck[ second ] = temp;
      } // end for
   } // end method shuffle

   // deal one card
   public Card dealCard()
   {
      // determine whether cards remain to be dealt
      if ( currentCard < deck.length )
         return deck[ currentCard++ ]; // return current Card in array
      else
         return null; // return null to indicate that all cards were dealt
   } // end method dealCard

   // tally the number of each face card in hand
   private int[] totalHand( Card hand[] )
   {
      int numbers[] = new int[ faces.length ]; // store number of face

      // initialize all elements of numbers[] to zero
      for ( int i = 0; i < 13; i++ )
         numbers[ i ] = 0;

      // compare each card in the hand to each element in the faces array
      for ( int h = 0; h < hand.length; h++ )
      {
         for ( int f = 0; f < 13; f++ )
         {
            if ( hand[ h ].getFace() == faces[ f ] )
               ++numbers[ f ];
         } // end for
      } // end for

      return numbers;
   } // end method totalHand

   // determine if hand contains pairs
   public int pairs( Card hand[] )
   {
      int couples = 0;
      int numbers[] = totalHand( hand );

      // count pairs
      for ( int k = 0; k < numbers.length; k++ )
      {
         if ( numbers[ k ] == 2 )
         {
            System.out.printf( "Pair of %ss\n", faces[ k ] );
            ++couples;
         } // end if
   } // end for

      return couples;
   } // end method pairs

   // determine if hand contains a three of a kind
   public int threeOfAKind( Card hand[] )
   {
      int triples = 0;
      int numbers[] = totalHand( hand );

      // count three of a kind
      for ( int k = 0; k < numbers.length; k++ )
      {
         if ( numbers[ k ] == 3 )
         {
            System.out.printf( "Three %ss\n", faces[ k ] );
            ++triples;
            break;
         } // end if
      } // end for

      return triples;
   } // end method threeOfAKind

   // determine if hand contains a four of a kind
   public void fourOfAKind( Card hand[] )
   {
      int numbers[] = totalHand( hand );

      for ( int k = 0; k < faces.length; k++ )
      {
         if ( numbers[ k ] == 4 )
            System.out.printf ( "Four %ss\n", faces[ k ] );
      } // end for
   } // end fourOfAKind

   // determine if hand contains a flush
   public void flush( Card hand[] )
   {
      String theSuit = hand[ 0 ].getSuit();

      for ( int s = 1; s < hand.length; s++ )
      {
         if ( hand[ s ].getSuit() != theSuit )
            return;   // not a flush
      } // end for

      System.out.printf( "Flush in %s\n", theSuit );
   } // end method flush

   // determine if hand contains a straight
   public void straight( Card hand[] )
   {
      int locations[] = new int[ 5 ];
      int z = 0;
      int numbers[] = totalHand( hand );

      for ( int y = 0; y < numbers.length; y++ )
      {
         if ( numbers[ y ] == 1 )
            locations[ z++ ] = y;
      } // end for

      int faceValue = locations[ 0 ];

      if ( faceValue == 0 ) // special case, faceValue is Ace
      {
         faceValue = 13;

         for ( int m = locations.length - 1; m >= 1; m-- )
         {
            if ( faceValue != locations[ m ] + 1 )
               return; // not a straight
            else
               faceValue = locations[ m ];
         } // end if
      } // end if
      else
      {
         for ( int m = 1; m < locations.length; m++ )
         {
            if ( faceValue != locations[ m ] - 1 )
               return; // not a straight
            else
               faceValue = locations[ m ];
         } // end if
      } // end else

      System.out.println( "Straight" );
   } // end method straight

   // determine if hand contains a full house
   public void fullHouse( int couples, int triples )
   {
      if ( couples == 1 && triples == 1 )
         System.out.println( "\nFull House!" );
   } // end method fullHouse

   // determine if hand contains two pairs
   public void twoPairs( int couples )
   {
      if ( couples == 2 )
```

```java
         System.out.println( "\nTwo Pair!" );
   } // end method twoPair
} // end class DeckOfCards
```

```java
// Exercise 7.30 Solution: DeckOfCardsTest.java
// Card shuffling and dealing application.

public class DeckOfCardsTest
{
   // execute application
   public static void main( String args[] )
   {
      DeckOfCards myDeckOfCards = new DeckOfCards();
      myDeckOfCards.shuffle(); // place Cards in random order

      Card[] hand = new Card[ 5 ]; // store five cards

      // get first five cards
      for ( int i = 0; i < 5; i++ )
      {
         hand[ i ] = myDeckOfCards.dealCard(); // get next card
         System.out.println( hand[ i ] );
      } // end for

      // display result
      System.out.println( "\nHand contains:" );

      int couples = myDeckOfCards.pairs( hand ); // a pair
      myDeckOfCards.twoPairs( couples ); // two pairs
      int triples = myDeckOfCards.threeOfAKind( hand ); // three of a kind
      myDeckOfCards.fourOfAKind( hand ); // four of a kind
      myDeckOfCards.flush( hand ); // a flush
      myDeckOfCards.straight( hand ); // a straight
      myDeckOfCards.fullHouse( couples, triples ); // a full house
   } // end main
} // end class DeckOfCardsTest
```

```
Queen of Hearts
Nine of Hearts
Nine of Spades
Queen of Diamonds
Three of Hearts

Hand contains:
Pair of Nines
Pair of Queens

Two Pair!
```

7.31 *(Card Shuffling and Dealing)* Use the methods developed in Exercise 7.30 to write an application that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

ANS:

```java
// Exercise 7.31 Solution: DeckOfCards.java
// DeckOfCards class represents a deck of playing cards.
import java.util.Random;

public class DeckOfCards
{
   String faces[] = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
      "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
   String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
   private Card deck[]; // array of Card objects
   private int currentCard; // the index of the next Card to be dealt
   private final int NUMBER_OF_CARDS = 52; // constant number of cards
   private Random randomNumbers; // random number generator
   private boolean straightHand1, straightHand2, pair1, pair2;
   private int hand1Value, hand2Value;
   private final int ONEPAIR = 2;
   private final int TWOPAIR = 4;
   private final int THREEKIND = 6;
   private final int STRAIGHT = 8;
   private final int FULLHOUSE = 10;
   private final int FLUSH = 12;
   private final int FOURKIND = 14;
   private final int STRAIGHTFLUSH = 16;

   // constructor fills deck of cards
   public DeckOfCards()
   {
      deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
      currentCard = 0; // initialize currentCard
      randomNumbers = new Random(); // create random number generator

      // populate deck with Card objects
      for ( int count = 0; count < deck.length; count++ )
         deck[ count ] =
            new Card( faces[ count % 13 ], suits[ count / 13 ] );
   } // end DeckOfCards constructor

   // shuffle deck of cards with one-pass algorithm
   public void shuffle()
   {
      currentCard = 0; // reinitialize currentCard

      // for each card, pick another random card and swap them
      for ( int first = 0; first < deck.length; first++ )
      {
         int second =  randomNumbers.nextInt( NUMBER_OF_CARDS );
         Card temp = deck[ first ];
         deck[ first ] = deck[ second ];
         deck[ second ] = temp;
      } // end for
   } // end method shuffle

   // deal one card
   public Card dealCard()
   {
      // determine whether cards remain to be dealt
      if ( currentCard < deck.length )
         return deck[ currentCard++ ]; // return current Card in array
      else
         return null; // return null to indicate that all cards were dealt
   } // end method dealCard

   // tally the number of each face card in hand
   private int[] totalHand( Card hand[] )
   {
      int numbers[] = new int[ faces.length ]; // store number of face

      // initialize all elements of numbers[] to zero
      for ( int i = 0; i < 13; i++ )
         numbers[ i ] = 0;

      // compare each card in the hand to each element in the faces array
      for ( int h = 0; h < hand.length; h++ )
      {
         for ( int f = 0; f < 13; f++ )
         {
            if ( hand[ h ].getFace() == faces[ f ] )
               ++numbers[ f ];
         } // end for
      } // end for

      return numbers;
   } // end method totalHand

   // determine if hand contains pairs
   public void pairs( Card hand1[], Card hand2[] )
   {
      int numberPairs1 = 0; // number of pairs in hand1
      int numberPairs2 = 0; // number of pairs in hand2
      int highest1 = 0; // highest number of pair in hand1
      int highest2 = 0; // highest number of pair in hand2
      int numbers1[] = totalHand( hand1 ); // tally the number of each
      int numbers2[] = totalHand( hand2 ); // face in hand1 and hand2

      // count number of pairs in hands
      for ( int k = 0; k < faces.length; k++ )
      {
         // pair found in hand 1
         if ( numbers1[ k ] == 2 )
         {
            pair1 = true;

            // store highest pair
            if ( k == 0 )
               highest1 = 13; // special value for ace

            if ( k > highest1 )
               highest1 = k;

            ++numberPairs1;
         } // end if

         // pair found in hand 2
         if ( numbers2[ k ] == 2 )
         {
            pair2 = true;

            // store highest pair
            if ( k == 0 )
               highest2 = 13; // special value for ace

            if ( k > highest2 )
               highest2 = k;

            ++numberPairs2;
         } // end if
      } // end for

      // evaluate number of pairs in each hand
      if ( numberPairs1 == 1 )
         hand1Value = ONEPAIR;
      else if ( numberPairs1 == 2 )
         hand1Value = TWOPAIR;

      if ( numberPairs2 == 1 )
         hand2Value = ONEPAIR;
      else if ( numberPairs2 == 2 )
         hand2Value = TWOPAIR;

      if ( highest1 > highest2 )
         ++hand1Value;
      else if ( highest2 > highest1 )
         ++hand2Value;
   } // end method pairs

   // determine if hand contains a three of a kind
   public void threeOfAKind( Card hand1[], Card hand2[] )
   {
      int tripletValue1 = 0; // highest triplet value in hand1
      int tripletValue2 = 0; // highest triplet value in hand2
      boolean flag1 = false;
      boolean flag2 = false;
      int numbers1[] = totalHand( hand1 ); // tally the number of each
      int numbers2[] = totalHand( hand2 ); // face in hand1 and hand2

      // check for three of a kind
      for ( int k = 0; k < faces.length; k++ )
      {
         // three of a kind found in hand 1
         if ( numbers1[ k ] == 3 )
         {
            hand1Value = THREEKIND;
            flag1 = true;

            // store value of triplet
            if ( k == 0 )
               tripletValue1 = 13; // special value for ace

            if ( k > tripletValue1 )
               tripletValue1 = k;

            if ( pair1 == true )
               hand1Value = FULLHOUSE;
         } // end if

         // three of a kind found in hand 2
         if ( numbers2[ k ] == 3 )
         {
            hand2Value = THREEKIND;
            flag2 = true;

            // store value of triplet
            if ( k == 0 )
               tripletValue2 = 13;   // special value for ace

            if ( k > tripletValue2 )
               tripletValue2 = k;

            if ( pair2 == true )
               hand2Value = FULLHOUSE;
         } // end if
      } // end for

      // both hands have three of a kind,
      // determine which triplet is higher in value
      if ( flag1 == true && flag2 == true )
      {
         if ( tripletValue1 > tripletValue2 )
            ++hand1Value;

         else if ( tripletValue1 < tripletValue2 )
            ++hand2Value;
      } // end if
   } // end method threeOfAKind

   // determine if hand contains a four of a kind
   public void fourOfAKind( Card hand1[], Card hand2[] )
   {
      int highest1 = 0;
      int highest2 = 0;
      boolean flag1 = false;
      boolean flag2 = false;
      int numbers1[] = totalHand( hand1 ); // tally the number of each
      int numbers2[] = totalHand( hand2 ); // face in hand1 and hand2

      // check for four of a kind
      for ( int k = 0; k < faces.length; k++ )
      {
         // hand 1
         if ( numbers1[ k ] == 4 )
         {
            hand1Value = FOURKIND;
            flag1 = true;

            if ( k == 0 )
               highest1 = 13; // special value for ace

            if ( k > highest1 )
               highest1 = k;
         } // end if

         // hand 2
         if ( numbers2[ k ] == 4 )
         {
            hand2Value = FOURKIND;
            flag2 = true;

            if ( k == 0 )
               highest2 = 13;    // special value for ace

            if ( k > highest2 )
               highest2 = k;
         } // end if
      } // end for

      // if both hands contain four of a kind, determine which is higher
      if ( flag1 == true && flag2 == true )
      {
         if ( highest1 > highest2 )
            ++hand1Value;
         else if ( highest1 < highest2 )
            ++hand2Value;
      } // end if
   } // end fourOfAKind

   // determine if hand contains a flush
   public void flush( Card hand1[], Card hand2[] )
   {
      String hand1Suit = hand1[ 0 ].getSuit();
      String hand2Suit = hand2[ 0 ].getSuit();
      boolean flag1 = true, flag2 = true;

      // check hand1
      for ( int s = 1; s < hand1.length && flag1 == true; s++ )
      {
         if ( hand1[ s ].getSuit() != hand1Suit )
            flag1 = false;   // not a flush
      } // end for

      // check hand2
      for ( int s = 1; s < hand2.length && flag2 == true; s++ )
      {
         if ( hand2[ s ].getSuit() != hand2Suit )
            flag2 = false;   // not a flush
      } // end for

      // hand 1 is a flush
      if ( flag1 == true )
      {
         hand1Value = FLUSH;

         // straight flush
         if ( straightHand1 == true )
            hand1Value = STRAIGHTFLUSH;
      } // end if

      // hand 2 is a flush
      if ( flag2 == true )
      {
         hand2Value = FLUSH;

         // straight flush
         if ( straightHand2 == true )
            hand2Value = STRAIGHTFLUSH;
      } // end if
   } // end method flush

   // determine if hand contains a straight
   public void straight( Card hand1[], Card hand2[] )
   {
      int locations1[] = new int[ 5 ];
      int locations2[] = new int[ 5 ];
      int value;
      int numbers1[] = totalHand( hand1 ); // tally the number of each
      int numbers2[] = totalHand( hand2 ); // face in hand1 and hand2

      // check each card in both hands
      for ( int y = 0, z = 0; y < numbers1.length; y++ )
      {
         if ( numbers1[ y ] == 1 )
            locations1[ z++ ] = y;
      } // end for

      for ( int y = 0, z = 0; y < numbers2.length; y++ )
      {
         if ( numbers1[ y ] == 1 )
            locations1[ z++ ] = y;
      } // end for

      int faceValue = locations1[ 0 ];
      boolean flag1 = true, flag2 = true;

      if ( faceValue == 0 ) // special case, faceValue is Ace
      {
         faceValue = 13;

         for ( int m = locations1.length - 1; m >= 1; m-- )
         {
            if ( faceValue != locations1[ m ] + 1 )
               return; // not a straight
            else
               faceValue = locations1[ m ];
         } // end if
      } // end if
      else
      {
         for ( int m = 1; m < locations1.length; m++ )
         {
            if ( faceValue != locations1[ m ] - 1 )
               return; // not a straight
            else
               faceValue = locations1[ m ];
         } // end if
      } // end else

      faceValue = locations2[ 0 ];

      if ( faceValue == 0 ) // special case, faceValue is Ace
      {
         faceValue = 13;

         for ( int m = locations2.length - 1; m >= 1; m-- )
         {
            if ( faceValue != locations2[ m ] + 1 )
               return; // not a straight
            else
               faceValue = locations2[ m ];
         } // end if
      } // end if
      else
      {
         for ( int m = 1; m < locations2.length; m++ )
         {
            if ( faceValue != locations2[ m ] - 1 )
               return; // not a straight
            else
               faceValue = locations2[ m ];
         } // end if
      } // end else

      int highest1 = 0;
      int highest2 = 0;

      // hand 1 is a straight
      if ( flag1 == true )
      {
         straightHand1 = true;
         hand1Value = STRAIGHT;

         if ( locations1[ 0 ] != 0 )
            highest1 = locations1[ 4 ];
         else
            highest1 = 13;
      } // end if

      // hand 2 is a straight
      if ( flag2 == true )
      {
         straightHand2 = true;
         hand2Value = STRAIGHT;

         if ( locations2[ 0 ] != 0 )
            highest2 = locations2[ 4 ];
         else
            highest2 = 13;
      } // end if

      // if both hands contain straights,
      // determine which is higher
      if ( straightHand1 == true && straightHand2 == true )
      {
         if ( highest1 > highest2 )
            ++hand1Value;
         else if ( highest2 > highest1 )
            ++hand2Value;
      } // end if
   } // end method straight

   // compare two hands
   public void compareTwoHands( Card hand1[], Card hand2[] )
   {
      // calculate contents of the two hand
      pairs( hand1, hand2 );
      threeOfAKind( hand1, hand2 );
      fourOfAKind( hand1, hand2 );
      straight( hand1, hand2 );
      flush( hand1, hand2 );
      displayHandValues(); // display hand values

      int numbers1[] = totalHand( hand1 ); // tally the number of each
      int numbers2[] = totalHand( hand2 ); // face in hand1 and hand2
      int highestValue1 = 0;
      int highestValue2 = 0;

      // calculate highest value in hand1
      if ( numbers1[ 0 ] > 0 ) // hand1 contains Ace
         highestValue1 = 13;
      else
      {
         for ( int i = 1; i < numbers1.length; i++ )
         {
            if ( numbers1[ i ] > 0 )
            {
               highestValue1 = i;
            } // end if
         } // end for
      } // end else

      // calculate highest value in hand2
      if ( numbers2[ 0 ] > 0 ) // hand2 contains Ace
         highestValue2 = 13;
      else
      {
        for ( int i = 1; i < numbers2.length; i++ )
        {
            if ( numbers2[ i ] > 0 )
            {
               highestValue2 = i;
            } // end if
         } // end for
      } // end else

      // compare and display result
      if ( hand1Value > hand2Value )
         System.out.println( "\nResult: left hand is better" );
      else if ( hand1Value < hand2Value )
         System.out.println( "\nResult: right hand is better" );
      else
      {
         // test for the highest card
         if ( highestValue1 > highestValue2 )
            System.out.println( "\nResult: left hand is better" );
         else if ( highestValue1 < highestValue2 )
            System.out.println( "\nResult: right hand is better" );
         else
            System.out.println( "\nResult: they are equal" );
      } // end else
   } // end method compareTwoHands

   // display hand values
   public void displayHandValues()
   {
      String handValue[] = { "none", "none" };
      int value = hand1Value;

      for ( int i = 0; i < 2; i++ )
      {
         if ( i == 1 )
            value = hand2Value;

         switch ( value ) {
            case 2: case 3:
               handValue[ i ] = "One Pair";
               break;
            case 4: case 5:
               handValue[ i ] = "Two Pair";
               break;
            case 6: case 7:
               handValue[ i ] = "Three of a Kind";
               break;
            case 8: case 9:
               handValue[ i ] = "Straight";
               break;
            case 10: case 11:
               handValue[ i ] = "Full House";
               break;
            case 12: case 13:
               handValue[ i ] = "Flush";
               break;
            case 14: case 15:
               handValue[ i ] = "Four of a Kind";
               break;
            case 16:
               handValue[ i ] = "Straight Flush";
               break;
         } // end switch
      } // end for

      System.out.printf( "%-20s", handValue[ 0 ] );
      System.out.printf( "%-20s\n", handValue[ 1 ] );
   } // end method displayHandValues
} // end class DeckOfCards
```

```
// Exercise 7.31 Solution: DeckOfCardsTest.java
// Card shuffling and dealing application.

public class DeckOfCardsTest
{
   // execute application
   public static void main( String args[] )
   {
      DeckOfCards myDeckOfCards = new DeckOfCards();
      myDeckOfCards.shuffle(); // place Cards in random order

      Card[] hand1 = new Card[ 5 ]; // store first hand
      Card[] hand2 = new Card[ 5 ]; // store second hand

      // get first five cards
      for ( int i = 0; i < 5; i++ )
      {
         hand1[ i ] = myDeckOfCards.dealCard(); // get next card
         hand2[ i ] = myDeckOfCards.dealCard(); // get next card
      } // end for

      // print hand1 and hand2
      System.out.printf( "%-20s%-20s\n", "Left hand:", "Right hand:" );

      for ( int i = 0; i < hand1.length; i++ )
         System.out.printf( "%-20s%-20s\n", hand1[ i ], hand2[ i ] );


      // display result
      System.out.println( "\nHand Values:" );
      myDeckOfCards.compareTwoHands( hand1, hand2 ); // compare two hands
   } // end main
} // end class DeckOfCardsTest
```

```
Left hand:          Right hand:
Ace of Spades       Deuce of Spades
Jack of Hearts      Four of Spades
Jack of Diamonds    Ten of Diamonds
Nine of Clubs       Nine of Hearts
Jack of Clubs       Deuce of Diamonds

Hand Values:
Three of a Kind     One Pair

Result: left hand is better
```

7.32 *(Card Shuffling and Dealing)* Modify the application developed in Exercise 7.31 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down," so the player cannot see it. The application should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The application should then reevaluate the dealer's hand. [*Caution*: This is a difficult problem!]

7.33 *(Card Shuffling and Dealing)* Modify the application developed in Exercise 7.32 so that it can handle the dealer's hand automatically, but the player is allowed to decide which cards of the player's hand to replace. The application should then evaluate both hands and determine who wins. Now use this new application to play 20 games against the computer. Who wins more games, you or the computer? Have a friend play 20 games against the computer. Who wins more games? Based on the results of these games, refine your poker-playing application. (This, too, is a difficult problem.) Play 20 more games. Does your modified application play a better game?

Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion from the world of high-level language programming; to "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (through the technique of software-based *simulation*) on which you can execute your machine-language programs.

7.34 (*Machine-Language Programming*) Let us create a computer called the Simpletron. As its name implies, it is a simple, but powerful, machine. The Simpletron runs programs written in the only language it directly understands: Simpletron Machine Language (SML).

The Simpletron contains an *accumulator*—a special register in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All the information in the Simpletron is handled in terms of *words*. A word is a signed four-digit decimal number, such as `+3364`, `-1293`, `+0007` and `-0001`. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers `00`, `01`, …, `99`.

Before running an SML program, we must *load*, or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location `00`. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory (and hence instructions are signed four-digit decimal numbers). We shall assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the *operation code* specifying the operation to be performed. SML operation codes are summarized in Fig. 7.35.

Operation code	Meaning
Input/output operations:	
`final int READ = 10;`	Read a word from the keyboard into a specific location in memory.
`final int WRITE = 11;`	Write a word from a specific location in memory to the screen.
Load/store operations:	
`final int LOAD = 20;`	Load a word from a specific location in memory into the accumulator.
`final int STORE = 21;`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`final int ADD = 30;`	Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
`final int SUBTRACT = 31;`	Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).
`final int DIVIDE = 32;`	Divide a word from a specific location in memory into the word in the accumulator (leave result in the accumulator).
`final int MULTIPLY = 33;`	Multiply a word from a specific location in memory by the word in the accumulator (leave the result in the accumulator).

Fig. 7.35 | Simpletron Machine Language (SML) operation codes. (Part 1 of 2.)

Operation code	Meaning
Transfer-of-control operations:	
`final int BRANCH = 40;`	Branch to a specific location in memory.
`final int BRANCHNEG = 41;`	Branch to a specific location in memory if the accumulator is negative.
`final int BRANCHZERO = 42;`	Branch to a specific location in memory if the accumulator is zero.
`final int HALT = 43;`	Halt. The program has completed its task.

Fig. 7.35 | Simpletron Machine Language (SML) operation codes. (Part 2 of 2.)

The last two digits of an SML instruction are the *operand*—the address of the memory location containing the word to which the operation applies. Let's consider several simple SML programs.

The first SML program (Fig. 7.36) reads two numbers from the keyboard and computes and displays their sum. The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to 0). Then instruction `+1008` reads the next number into location `08`. The *load* instruction, `+2007`, puts the first number into the accumulator, and the *add* instruction, `+3008`, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, `+2109`, places the result back into memory location `09`, from which the *write* instruction, `+1109`, takes the number and displays it (as a signed four-digit decimal number). The *halt* instruction, `+4300`, terminates execution.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 7.36 | SML program that reads two integers and computes their sum.

The second SML program (Fig. 7.37) reads two numbers from the keyboard and determines and displays the larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as Java's `if` statement.

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 7.37 | SML program that reads two integers and determines which is larger.

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read 10 positive numbers. Compute and display their sum.

ANS: Note: This program terminates when a negative number is input. The problem statement should state that only positive numbers should be input.

```
00  +1009       (Read Value)
01  +2009       (Load Value)
02  +4106       (Branch negative to 06)
03  +3008       (Add Sum)
04  +2108       (Store Sum)
05  +4000       (Branch 00)
06  +1108       (Write Sum)
07  +4300       (Halt)
08  +0000       (Storage for Sum)
09  +0000       (Storage for Value)
```

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

ANS:

```
00  +2018       (Load Counter)
01  +3121       (Subtract Termination)
02  +4211       (Branch zero to 11)
03  +2018       (Load Counter)
```

```
04  +3019       (Add Increment)
05  +2118       (Store Counter)
06  +1017       (Read Value)
07  +2016       (Load Sum)
08  +3017       (Add Value)
09  +2116       (Store Sum)
10  +4000       (Branch 00)
11  +2016       (Load Sum)
12  +3218       (Divide Counter)
13  +2120       (Store Result)
14  +1120       (Write Result)
15  +4300       (Halt)
16  +0000       (Variable Sum)
17  +0000       (Variable Value)
18  +0000       (Variable Counter)
19  +0001       (Variable Increment)
20  +0000       (Variable Result)
21  +0007       (Variable Termination)
```

c) Read a series of numbers, and determine and print the largest number. The first number read indicates how many numbers should be processed.

ANS:

```
00  +1017       (Read Endvalue)
01  +2018       (Load Counter)
02  +3117       (Subtract Endvalue)
03  +4215       (Branch zero to 15)
04  +2018       (Load Counter)
05  +3021       (Add Increment)
06  +2118       (Store Counter)
07  +1019       (Read Value)
08  +2020       (Load Largest)
09  +3119       (Subtract Value)
10  +4112       (Branch negative to 12)
11  +4001       (Branch 01)
12  +2019       (Load Value)
13  +2120       (Store Largest)
14  +4001       (Branch 01)
15  +1120       (Write Largest)
16  +4300       (Halt)
17  +0000       (Variable EndValue)
18  +0000       (Variable Counter)
19  +0000       (Variable Value)
20  +0000       (Variable Largest)
21  +0001       (Variable Increment)
```

7.35 (*Computer Simulator*) In this problem, you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of *software-based simulation* to create an object-oriented *software model* of the Simpletron of Exercise 7.34. Your

Simpletron simulator will turn the computer you are using into a Simpletron, and you will actually be able to run, test and debug the SML programs you wrote in Exercise 7.34.

When you run your Simpletron simulator, it should begin by displaying:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction  ***
*** (or data word) at a time into the input    ***
*** text field. I will display the location    ***
*** number and a question mark (?). You then   ***
*** type the word for that location. Press the ***
*** Done button to stop entering your program. ***
```

Your application should simulate the memory of the Simpletron with a one-dimensional array `memory` that has 100 elements. Now assume that the simulator is running, and let us examine the dialog as we enter the program of Fig. 7.37 (Exercise 7.34):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Your program should display the memory location followed by a question mark. Each value to the right of a question mark is input by the user. When the sentinel value `-99999` is input, the program should display the following:

```
*** Program loading completed ***
*** Program execution begins  ***
```

The SML program has now been placed (or loaded) in array `memory`. Now the Simpletron executes the SML program. Execution begins with the instruction in location `00` and, as in Java, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use the variable `operationCode` to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use the variable `operand` to indicate the memory location on which the current instruction operates. Thus, `operand` is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in `operationCode`, and "pick off" the right two digits and place them in `operand`. When the Simpletron begins execution, the special registers are all initialized to zero.

Now, let us "walk through" execution of the first SML instruction, `+1009` in memory location `00`. This procedure is called an *instruction execution cycle.*

The `instructionCounter` tells us the location of the next instruction to be performed. We *fetch* the contents of that location from `memory` by using the Java statement

```
instructionRegister = memory[ instructionCounter ];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, and so on). A `switch` differentiates among the 12 operations of SML. In the `switch` statement, the behavior of various SML instructions is simulated as shown in Fig. 7.38. We discuss branch instructions shortly and leave the others to you.

Instruction	Description
read:	Display the prompt `"Enter an integer"`, then input the integer and store it in location `memory[ operand ]`.
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
halt:	This instruction displays the message `*** Simpletron execution terminated ***`

Fig. 7.38 | Behavior of several SML instructions in the Simpletron.

When the SML program completes execution, the name and contents of each register as well as the complete contents of memory should be displayed. Such a printout is often called a computer dump (no, a computer dump is not a place where old computers go). To help you program your dump method, a sample dump format is shown in Fig. 7.39. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

```
REGISTERS:
accumulator          +0000
instructionCounter      00
instructionRegister  +0000
operationCode           00
operand                 00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 7.39 | A sample dump.

Let us proceed with the execution of our program's first instruction—namely, the +1009 in location 00. As we have indicated, the switch statement simulates this task by prompting the user to enter a value, reading the value and storing it in memory location memory[operand]. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Since the instruction just performed was not a transfer of control, we need merely increment the instruction-counter register as follows:

```
++instructionCounter;
```

This action completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to execute.

Now let us consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the switch as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 )
   instructionCounter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 7.34. If you desire, you may embellish SML with additional features and provide for these features in your simulator.

Your simulator should check for various types of errors. During the program-loading phase, for example, each number the user types into the Simpletron's memory must be in the range -9999 to +9999. Your simulator should test that each number entered is in this range and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, and accumulator overflows (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999). Such serious errors are called *fatal errors.* When a fatal error is detected, your simulator should display an error message such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should display a full computer dump in the format we discussed previously. This treatment will help the user locate the error in the program.

ANS:

```java
// Exercise 7.35 Solution: Simulator.java
// A computer simulator
import java.util.Scanner;

public class Simulator
{
   // list of SML instructions
   static final int READ = 10;
   static final int WRITE = 11;
   static final int LOAD = 20;
   static final int STORE = 21;
   static final int ADD = 30;
   static final int SUBTRACT = 31;
   static final int MULTIPLY = 32;
   static final int DIVIDE = 33;
   static final int BRANCH = 40;
   static final int BRANCH_NEG = 41;
   static final int BRANCH_ZERO = 42;
   static final int HALT = 43;

   Scanner input = new Scanner( System.in );

   int accumulator; // accumulator register
   int instructionCounter; // instruction counter, a memory address
   int operand; // argument for the operator
   int operationCode; // determines the operation
   int instructionRegister; // register holding the SML instruction

   int memory[]; // simpletron memory
   int index = 0; // number of instructions entered in memory

   // runs the simpletron simulator, reads instructions and executes
   public void runSimulator()
   {
      // initialize the registers
      initializeRegisters();

      // prompt the user to enter instructions
      printInstructions();
      loadInstructions();

      // execute the program and print the memory dump when finished
      execute();
      dump();
   } // end method runSimulator

   // set all registers to the correct start value
   public void initializeRegisters()
   {
      memory = new int[ 100 ];
      accumulator = 0;
      instructionCounter = 0;
      instructionRegister = 0;
      operand = 0;
      operationCode = 0;

      for ( int k = 0; k < memory.length; k++ )
         memory[ k ] = 0;
   } // end method initializeRegisters

   // print out user instructions
   public void printInstructions()
   {
```

```java
      System.out.printf( "%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
         "*** Welcome to Simpletron! ***",
         "*** Please enter your program one instruction ***",
         "*** ( or data word ) at a time into the input ***",
         "*** text field. I will display the location ***",
         "*** number and a question mark (?). You then ***",
         "*** type the word for that location. Enter ***",
         "*** -99999 to stop entering your program ***" );
   } // end method instructions

   // read in user input, test it, perform operations
   public void loadInstructions()
   {
      System.out.printf( "%02d ? ", index );
      int instruction = input.nextInt();

      while ( instruction != -99999 && index < 100 )
      {
         if ( validate( instruction ) )
            memory[ index++ ] = instruction;
         else
            System.out.println( "Input invalid." );

         System.out.printf( "%02d ? ", index );
         instruction = input.nextInt();
      } // end while

      System.out.println( "*** Program loading completed ***" );
   } // end method inputInstructions

   // ensure value is within range
   // returns true if the value is within range, otherwise returns false
   public boolean validate( int value )
   {
      return ( -9999 <= value ) && ( value <= 9999 );
   } // end method validate

   // ensure that accumulator has not overflowed
   public boolean testOverflow()
   {
      if ( !validate( accumulator ) ) {
         System.out.println(
            "*** Fatal error. Accumulator overflow. ***" );
         return true;
      } // end if

      return false;
   } // end method testOverflow

   // perform all simulator functions
   public void execute()
   {
      System.out.println( "*** Program execution begins  ***" );

      // continue executing until we reach the end of the program
      // it is possible that the program can terminate beforehand though
      while ( instructionCounter < index )
      {
         // read the instruction into the registers
         instructionRegister = memory[ instructionCounter ];
         operationCode = instructionRegister / 100;
         operand = instructionRegister % 100;

         // go to next instruction, this will only be overridden
         // by the branch commands
         ++instructionCounter;

         switch( operationCode )
         {
            case READ:
               // read an integer
               System.out.print( "Enter an integer: " );
               memory[ operand ] = input.nextInt();
               break;

            case WRITE:
               // outputs the contents of a memory address
               System.out.printf( "Contents of %02d is %d\n",
                  operand, memory[ operand ] );
               break;

            case LOAD:
               // load a memory address into the accumulator
               accumulator = memory[ operand ];
               break;

            case STORE:
               // store the contents of the accumulator to an address
               memory[ operand ] = accumulator;
               break;

            case ADD:
               // adds the contents of an address to the accumulator
               accumulator += memory[ operand ];

               if ( testOverflow() )
                  return;

               break;

            case SUBTRACT:
               // subtracts the contents of an address from the
               // accumulator
               accumulator -= memory[ operand ];

               if ( testOverflow() )
                  return;

               break;

            case MULTIPLY:
               // multiplies the accumulator with the contents of an
               // address
               accumulator *= memory[ operand ];

               if ( testOverflow() )
                  return;

               break;

            case DIVIDE:
               // divides the accumulator by the contents of an address
               if ( memory[ operand ] == 0 )
               {
                  System.out.println(
                     "*** Fatal error. Attempt to divide by zero. ***" );
                  return;
               } // end if

               accumulator /= memory[ operand ];
               break;

            case BRANCH:
               // jumps to an address
               instructionCounter = operand;
               break;

            case BRANCH_NEG:
               // jumps to an address if the accumulator is negative
               if ( accumulator < 0 )
                  instructionCounter = operand;

               break;

            case BRANCH_ZERO:
               // jumps to an address if the accumulator is zero
               if ( accumulator == 0 )
                  instructionCounter = operand;

               break;

            case HALT:
               // terminates execution
               System.out.println(
                  "*** Simpletron execution terminated ***" );
               return;

            default:
               // all other cases are not valid opcodes
               System.out.println(
                  "*** Fatal error. Invalid operation code. ***" );
               return;
```

```
         } // end switch
      } // end while
   } // end method execute

   // prints the values of the registers
   public void displayRegisters()
   {
      System.out.println( "REGISTERS:" );
      System.out.printf( "%-24s%+05d\n", "Accumulator:", accumulator );
      System.out.printf( "%-27s%02d\n", "InstructionCounter:",
         instructionCounter );
      System.out.printf( "%-24s%+05d\n", "InstructionRegister:",
         instructionRegister );
      System.out.printf( "%-27s%02d\n", "OperationCode:",
         operationCode );
      System.out.printf( "%-27s%02d\n", "Operand:", operand );
   } // end method displayRegisters

   // output memory information
   public void dump()
   {
      displayRegisters();

      System.out.println( "\nMEMORY:" );

      // print column headings
      System.out.print( "  " );

      for ( int k = 0; k < 10; k++)
         System.out.printf( "%7d", k );

      System.out.println();

      // print the memory dump
      for ( int k = 0; k < 10; k++ )
      {
         // print the row label
         System.out.printf( "%2d", k * 10 );

         // print the contents of each memory location
         for ( int i = 0; i < 10; i++ )
            System.out.printf( "  %+05d", memory[ k * 10 + i ] );

         System.out.println();
      } // end for
   } // end method dump
} // end class Simulator
```

```
// Exercise 7.35 Solution: SimulatorTest.java
// Test application for class Simulator
public class SimulatorTest
{
   public void main( String args[] )
```

```java
   {
      Simulator simpletron = new Simulator();
      simpletron.runSimulator();
   } // end main
} // end class SimulatorTest
```

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** ( or data word ) at a time into the input ***
*** text field. I will display the location ***
*** number and a question mark (?). You then ***
*** type the word for that location. Enter ***
*** -99999 to stop entering your program ***
00 ? 1007
01 ? 1008
02 ? 2007
03 ? 3008
04 ? 2109
05 ? 1109
06 ? 4300
07 ? 0000
08 ? 0000
09 ? 0000
10 ? -99999
*** Program loading completed ***
*** Program execution begins  ***
Enter an integer: 5
Enter an integer: 10
Contents of 09 is 15
*** Simpletron execution terminated ***
REGISTERS:
Accumulator:            +0015
InstructionCounter:        07
InstructionRegister:    +4300
OperationCode:             43
Operand:                   00

MEMORY:
        0     1     2     3     4     5     6     7     8     9
 0  +1007 +1008 +2007 +3008 +2109 +1109 +4300 +0005 +0010 +0015
10  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

7.36 (*Simpletron Simulator Modifications*) In Exercise 7.35, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron simulator. In Exercise 17.26 and Exercise 17.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of Basic) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:

a) Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.
b) Allow the simulator to perform remainder calculations. This modification requires an additional SML instruction.
c) Allow the simulator to perform exponentiation calculations. This modification requires an additional SML instruction.
d) Modify the simulator to use hexadecimal values rather than integer values to represent SML instructions.
e) Modify the simulator to allow output of a newline. This modification requires an additional SML instruction.
f) Modify the simulator to process floating-point values in addition to integer values.
g) Modify the simulator to handle string input. [*Hint*: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII (see Appendix B) decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string, beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half-word.]
h) Modify the simulator to handle output of strings stored in the format of part (g). [*Hint*: Add a machine-language instruction that will display a string, beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and displays the string by translating each two-digit number into its equivalent character.]

(Optional) GUI and Graphics Case Study

7.1 (*Drawing Spirals*) In this exercise, you will draw spirals with methods `drawLine` and `drawArc`.

a) Draw a square-shaped spiral (as in the left screen capture of Fig. 7.40), centered on the panel, using method `drawLine`. One technique is to use a loop that increases the line length after drawing every second line. The direction in which to draw the next line should follow a distinct pattern, such as down, left, up, right.

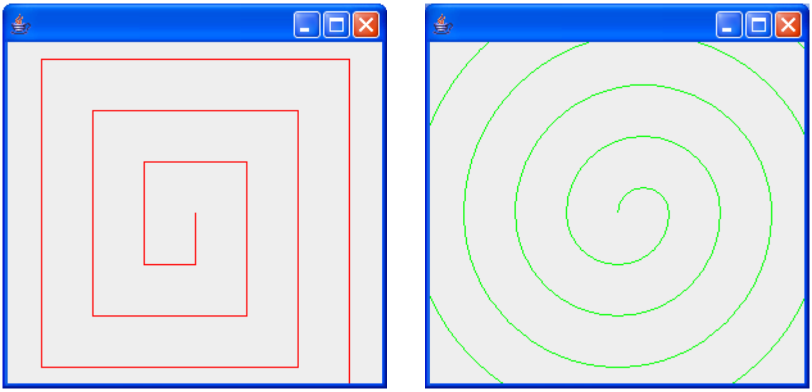
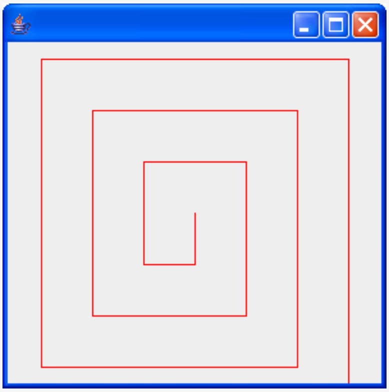


Fig. 7.40 | Drawing a spiral using `drawLine` (left) and `drawArc` (right).

ANS:

```java
// GCS Exercise 7.1 Part A Solution: DrawSpiral1.java
// Draws a square shaped spiral.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawSpiral1 extends JPanel
{
   // draws a square shape that continually spirals outward
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      g.setColor( Color.RED ); // draw a red spiral

      int oldX = getWidth() / 2; // starting X
      int oldY = getHeight() / 2; // starting Y

      int distance = 0; // distance to move

      // draws individual lines in to form a spiral
      for ( int i = 0; i < 20; i++ )
      {
         int newX = oldX; // new X position
         int newY = oldY; // new Y position

         if ( i % 2 == 0 ) // increment the distance every other leg
            distance += 40; // sets the distance between lines

         // set the endpoint depending on the desired direction
         switch ( i % 4 )
         {
            case 0:
               newY += distance;
               break;
            case 1:
               newX -= distance;
               break;
            case 2:
               newY -= distance;
               break;
            case 3:
               newX += distance;
               break;
         } // end switch

         g.drawLine( oldX, oldY, newX, newY );
         oldX = newX; // replace the old position
         oldY = newY; // with the new position
      } // end for
   } // end method paintComponent
} // end class DrawSpiral1
```

```
// GCS Exercise 7.1 Part A Solution: DrawSpiralTest1.java
// Test application to display class DrawSpiral1.
import javax.swing.JFrame;

public class DrawSpiralTest1
{
   public static void main( String args[] )
   {
      DrawSpiral1 panel = new DrawSpiral1();
      JFrame application = new JFrame();

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.add( panel );
      application.setSize( 300, 300 );
      application.setVisible( true );
   } // end main
} // end class DrawSpiralTest1
```



b) Draw a circular spiral (as in the right screen capture of Fig. 7.40), using method `drawArc` to draw one semicircle at a time. Each successive semicircle should have a larger radius (as specified by the bounding rectangle's width) and should continue drawing where the previous semicircle finished.

ANS:

```java
// GCS Exercise 7.1 Part B Solution: DrawSpiral2.java
// Draws a circular spiral.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawSpiral2 extends JPanel
{
   // draws a square shape that continually spirals outward
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      g.setColor( Color.GREEN ); // draw a green spiral

      int x = getWidth() / 2; // x coordinate of upperleft corner
      int y = getHeight() / 2; // y coordinate of upperleft corner

      int radiusStep = 20; // distance the radius changes
      int diameter = 0; // diameter of the arc

      int arc = 180; // amount and direction of arc to sweep

      // draws individual lines in to form a spiral
      for ( int i = 0; i < 20; i++ )
      {
         if ( i % 2 == 1 ) // move the x position every other repetition
            x -= 2 * radiusStep;

         y -= radiusStep; // move the y position

         diameter += 2 * radiusStep; // increase the diameter

         g.drawArc( x, y, diameter, diameter, 0, arc ); // draw the arc

         arc = -arc; // reverse the direction of the arc
      } // end for
   } // end method paintComponent
} // end class DrawSpiral2
```

```java
// GCS Exercise 7.1 Part B Solution: DrawSpiralTest2.java
// Test application to display class DrawSpiral2.
import javax.swing.JFrame;

public class DrawSpiralTest2
{
   public static void main( String args[] )
   {
      DrawSpiral2 panel = new DrawSpiral2();
      JFrame application = new JFrame();

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.add( panel );
      application.setSize( 300, 300 );
      application.setVisible( true );
   } // end main
} // end class DrawSpiralTest2
```