FIT ICT Software Development

Lecture 8

Lecturer : Charles Tyner

This Session

- After discussing arrays, we introduce one of Java's predefined data structures from the Java API's collection classes.
  - Greater capabilities than traditional arrays.
  - Reusable, reliable, powerful and efficient and have been designed and tested to ensure quality and performance.
  - Focus on the `ArrayList` collection.
- `ArrayList`s are similar to arrays but provide additional functionality, such as dynamic resizing—they automatically increase their size at execution time to accommodate additional elements.

Primitive Types vs. Reference Types

- Java's types are divided into primitive types and reference types.
  - Primitive types: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- All non-primitive types are reference types.

Array Recap

- Data structures—collections of related data items.
- Arrays—data structures consisting of related data items of the same type.
  - Make it convenient to process related groups of values.
  - Remain the same length once they are created.
- An array is a group of variables (called elements or components) containing values that all have the same type.
- Arrays are objects, so they're considered reference types.
- Elements can be either primitive types or reference types.
- To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array.
  - The position number of the element is called the element's index or subscript.

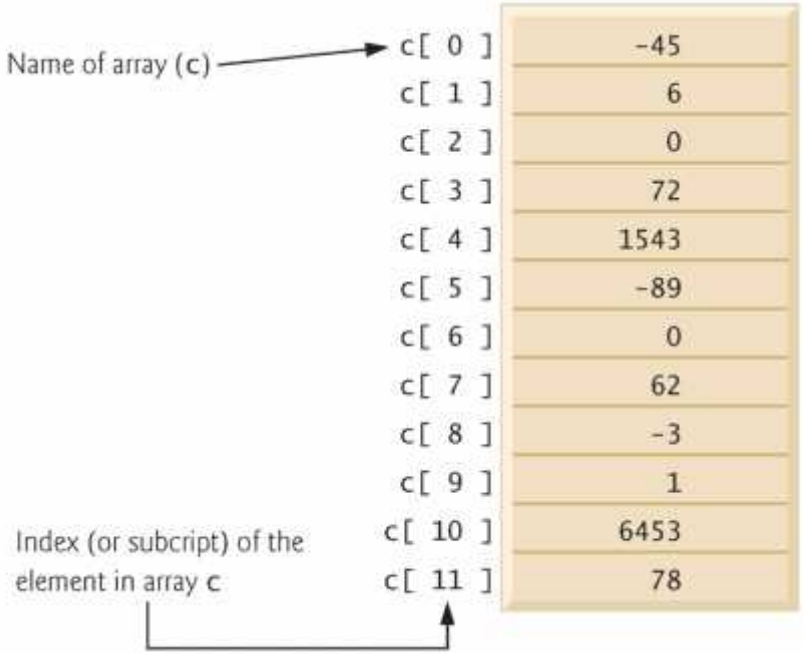


Fig. 6.1 | A 12-element array.

Array Recap

- A program refers to an array's elements with an array-access expression
  - The name of the array followed by the index of the particular element in square brackets ([]).
- The first element in every array has index zero and is sometimes called the zeroth element.
- The highest index in an array is one less than the number of elements.
- Array names follow the same conventions as other variable names.
- An index must be a nonnegative integer.
- A program can use an expression as an index.
- Array-access expressions can be used on the left side of an assignment to place a new value into an array element.

Common Programming Error 6.1

An index must be an `int` *value or a value of a type that can be promoted to* `int`*—namely,* `byte`, `short` *or* `char`, *but not* `long`*; otherwise, a compilation error occurs.*

Array Recap

- Every array object knows its own length and stores it in a `length` instance variable.
- Even though the `length` instance variable of an array is `public`, it cannot be changed because it's a `final` variable.

Declaring and Creating Arrays

- Like other objects, arrays are created with keyword `new`.
- To create an array object, you specify the **type** of the array elements and the **number** of elements as part of an array-creation expression that uses keyword `new`.
  - Returns a reference that can be stored in an array variable.
- The following declaration and array-creation expression create an array object containing 12 `int` elements and store the array's reference in array variable `c`:
  - `int[] c = new int[ 12 ];`
- The square brackets following the type indicate that the variable will refer to an array.
- When type of the array and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables.

NOTE:

- A program can declare arrays of any type.
- Every element of a primitive-type array contains a value of the array's declared element type.
- Similarly, in an array of a reference type, every element is a reference to an object of the array's declared element type.

```
1  // Fig. 6.2: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array; // declare array named array
9
10       array = new int[ 10 ]; // create the array object
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
13
14       // output each array element's value
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // end main
18 } // end class InitArray
```

Fig. 6.2 | Initializing the elements of an array to default values of zero. (Part 1 of 2.)

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Fig. 6.2 | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

Array Recap

- You can create an array and initialize its elements with an array initializer—a comma-separated list of expressions (called an initializer list) enclosed in braces.
- The array length is determined by the number of elements in the initializer list.
- The following statement creates a five-element array with index values 0–4
  - `int[] n = { 10, 20, 30, 40, 50 };`
- Element `n[0]` is initialized to `10`, `n[1]` is initialized to `20`, and so on.
- When the compiler encounters an array declaration that includes an initializer list, it counts the number of initializers in the list to determine the size of the array, then sets up the appropriate `new` operation "behind the scenes."

```
1  // Fig. 6.3: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // initializer list specifies the value for each element
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
12
13       // output each array element's value
14       for ( int counter = 0; counter < array.length; counter++ )
15          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16    } // end main
17 } // end class InitArray
```

Fig. 6.3 | Initializing the elements of an array with an array initializer. (Part 1 of 2.)

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. 6.3 | Initializing the elements of an array with an array initializer. (Part 2 of 2.)

Named Constants

- The application in Fig. 6.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20).
- Modifier `final` indicates that a variable is a constant.
- Constant variables must be initialized before they're used and cannot be modified thereafter.
- If you attempt to modify a `final` variable after it's initialized in its declaration, the compiler issues an error message like
  - `cannot assign a value to final variable` *variableName*
- If an attempt is made to access the value of a `final` variable before it's initialized, the compiler issues an error message like
  - `variable` *variableName* `might not have been initialized`

```
1  // Fig. 6.4: InitArray.java
2  // Calculating values to be placed into elements of an array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        final int ARRAY_LENGTH = 10; // declare constant
9        int[] array = new int[ ARRAY_LENGTH ]; // create array
10
11       // calculate value for each array element
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
16
17       // output each array element's value
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20    } // end main
21 } // end class InitArray
```

Fig. 6.4 | Calculating the values to be placed into the elements of an array. (Part 1 of 2.)

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Fig. 6.4 | Calculating the values to be placed into the elements of an array. (Part 2 of 2.)

Good Programming Practice 6.2

Constant variables also are called ***named constants****. They often make programs more readable than programs that use literal values (e.g.,* 10*)—a named constant such as* ARRAY_LENGTH *clearly indicates its purpose, whereas a literal value could have different meanings based on its context.*

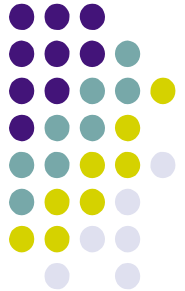
Common Programming Error 6.4

Assigning a value to a constant variable after it has been initialized is a compilation error.

Common Programming Error 6.5

Attempting to use a constant before it's initialized is a compilation error.

Summing Array Values



- Often, the elements of an array represent a series of values to be used in a calculation.
- Figure 6.5 sums the values contained in a 10- element integer array.

```java
// Fig. 6.5: SumArray.java
// Computing the sum of the elements of an array.

public class SumArray
{
   public static void main( String[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int counter = 0; counter < array.length; counter++ )
         total += array[ counter ];

      System.out.printf( "Total of array elements: %d\n", total );
   } // end main
} // end class SumArray
```

```
Total of array elements: 849
```

Fig. 6.5 | Computing the sum of the elements of an array.

Array to Summarise Results

- Figure 6.8 uses arrays to summarize the results of data collected in a survey:
  - Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (where 1 means awful and 10 means excellent). Place the 40 responses in an integer array, and summarize the results of the poll.
- Typical array-processing application.
- Summarize the number of responses of each type (i.e., 1 through 10).
- The array `responses` is a 40-element `int` array of the survey responses.
- 11-element array `frequency` stores the number of occurrences of each response.
- Each element is initialized to zero by default.
- As in Fig. 6.7, we ignore `frequency[0]`.

```
1  // Fig. 6.8: StudentPoll.java
2  // Poll analysis program.
3
4  public class StudentPoll
5  {
6     public static void main( String[] args )
7     {
8        // array of survey responses
9        int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
10          10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
11          4, 8, 6, 8, 10 };
12       int[] frequency = new int[ 11 ]; // array of frequency counters
13
14       // for each answer, select responses element and use that value
15       // as frequency index to determine element to increment
16       for ( int answer = 0; answer < responses.length; answer++ )
17          ++frequency[ responses[ answer ] ];
18
19       System.out.printf( "%s%10s", "Rating", "Frequency" );
20
21       // output each array element's value
22       for ( int rating = 1; rating < frequency.length; rating++ )
23          System.out.printf( "%d%10d", rating, frequency[ rating ] );
24    } // end main
25 } // end class StudentPoll
```

Fig. 6.8 | Poll analysis program. (Part 1 of 2.)

```
Rating Frequency
     1         2
     2         2
     3         2
     4         2
     5         5
     6        11
     7         5
     8         7
     9         1
    10         3
```

Fig. 6.8 | Poll analysis program. (Part 2 of 2.)

Array to Summarise Results

- The `for` loop at lines 16–17 takes the responses one at a time from array `responses` and increments one of the 10 counters in the `frequency` array (`frequency[1]` to `frequency[10]`).
- The key statement in the loop is line 17, which increments the appropriate `frequency` counter, depending on the value of `responses[answer]`.

Array to Summarise Results

- If the data in the `responses` array had contained invalid values, such as 13, the program would have attempted to add `1` to `frequency[13]`, which is outside the bounds of the array.
- Java doesn't allow this.
- The JVM checks array indices to ensure that they're greater than or equal to 0 and less than the length of the array
  - Called bounds checking.
- If a program uses an invalid index, Java generates a so-called exception to indicate that an error occurred in the program at execution time.

Error-Prevention Tip 6.1

An exception indicates that an error has occurred. You often can write code to recover from an exception and continue program execution, rather than abnormally terminating the program. When a program attempts to access an element outside the array bounds, an `ArrayIndexOutOfBoundsException` *occurs. Exception handling is discussed in Chapter 11.*

Enhanced `for` Statement

- Iterates through the elements of an array without using a counter, thus avoiding the possibility of "stepping outside" the array.
- Syntax:
  - `for ( parameter : arrayName )`
    statement
  - where *parameter* has a type and an identifier, and *arrayName* is the array through which to iterate.
  - Parameter type must be consistent with the type of the elements in the array.
- Figure 6.9 uses the enhanced `for` statement to sum the integers in an
- array of student grades.
- Can be used only to obtain array elements—it cannot be used to modify
- elements.
- Can be used in place of the counter-controlled `for` statement whenever
- code looping through an array does not require access to the counter indicating the index of the current array element.

```java
// Fig. 6.9: EnhancedForTest.java
// Using enhanced for statement to total integers in an array.

public class EnhancedForTest
{
   public static void main( String[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int number : array )
         total += number;

      System.out.printf( "Total of array elements: %d\n", total );
   } // end main
} // end class EnhancedForTest
```

```
Total of array elements: 849
```

Fig. 6.9 | Using the enhanced `for` statement to total integers in an array.

Passing Arrays to Methods
(References and reference parameters)

- Two ways to pass arguments in method calls in many programming languages are call-by-value (access to copy) and call-by-reference (access to original).
- When an argument is passed by value, a copy of the argument's *value* is passed to the called method.
  - The called method works exclusively with the copy.
  - Changes to the called method's copy do not affect the original variable's value in the caller.
- When an argument is passed by reference, the called method can access the argument's value in the caller directly and modify that data, if necessary.
  - Improves performance by eliminating the need to copy possibly large amounts of data.

References and Reference types

- Java does not allow programmers to choose call-by-value or call-by-reference

- Primitive data types are passed call-by-value
- Reference data types are passed call-by reference.

- Objects themselves cannot be passed to methods
- We pass a copy of the reference to the object as an argument to the method.

Call-by-value

When an argument is passed call-by-value a copy of the argument's value is made and passed to the called method

Call-by-reference

With call-by-reference, the caller gives the called method the ability to **directly access the caller's data** and to modify that data if the called method chooses.

Call-by-reference can improve performance because it can eliminate the overhead of copying large amounts of data.

Call-by-reference can weaken security because the called data method can access the caller's data

To pass an object call-by-reference, specify in the method call a reference to the object by name.

Mentioning the object by its parameter name in the body of the called method actually refers to the original object in memory, and the original object can be accessed directly by the called method.

Because Java treats **arrays as objects**,
… arrays are passed to methods call-by-reference.

A called method can acess the element values in the caller's original arrays.

The name of an array is a reference to the object that contains the array elements.

The **length** instance variable indicates the number of elements in the array.

Passing arrays by reference makes sense for **performance** reasons.

If arrays were passed by value, a copy of each element would be passed.

For large arrays, this would waste time and storage.

Passing Arrays to methods

To pass an array to a method, specify the name of the array without any brackets.

i.e.

```
int hourlyTemperature[ ] = new int[ 24 ];

    o
    o
    o

modifyArray( hourlyTemperature );
```

- This passes the array **hourlyTemperature** to method **modifyArray**

Passing Arrays to methods

To receive an array through a method, the **method's parameters list** must specify that an array will be received.

```
modifyArray( int [ ] b)
{
//instructions;
}
```

This indicates that **modifyArray** expects to receive an integer array in parameter **b**.

Because the arrays are passed by reference, when the called method uses the array name **b**, it refers to the actual array in the caller.

This is **hourlyTemperature** in the preceding call.

In Java, every array knows it's own size via the **length** instance variable!!

Therefore, when we pass an array object to a method we do not need to specify the size of the array as an argument.

Although: **entire arrays** are passed call-by-reference, …..

- **individual** array elements of primitive data types are passed call-by-value … exactly as simple variables.

These simple single pieces of data are called **scalars** or **scalar quantities**.

To pass an array element to a method,

- **use the subscripted name** of the array element as an argument in the method call.

```
modifyElement( hourlyTemperature [ 3 ] );
```

For a method to receive an array element through a method,

- the **method's parameters list** must specify that an array element will be received.

```
void modifyElement( int element)
```

This indicates that **modifyArray** expects to receive a primitive type array element in parameter **b**.

Passing Arrays to Methods

- Figure 6.10 demonstrates the difference between passing an entire array and passing a primitive-type array element to a method.
- Method `modifyArray` receives a **copy** of `array`'s reference and uses the reference to multiply each of `array`'s elements by 2.
- When a **copy** of an individual primitive-type array element is passed to a method, modifying the copy in the called method does not affect the original value of that element in the calling method's array.

Passing Arrays to Methods (recap)

- To pass an array argument to a method, specify the name of the array without any brackets.
- When we pass an array object's reference into a method, we need not pass the array length as an additional argument because every array knows its own length.
- For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter.
- When a method argument is an entire array or an individual array element of a reference type, the called method receives a copy of the reference to the array or individual element.
- When a method argument is an individual primitive-type array element, the called method receives a copy of the element's value.
- Such primitive values are called scalars or scalar quantities.
- To pass an individual array element to a method, use the indexed name of the array element.

```
1  // Fig. 6.10: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray
5  {
6     // main creates array and calls modifyArray and modifyElement
7     public static void main( String[] args )
8     {
9        int[] array = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Effects of passing reference to entire array:\n" +
13          "The values of the original array are:" );
14
15       // output original array elements
16       for ( int value : array )
17          System.out.printf( "   %d", value );
18
19       modifyArray( array ); // pass array reference
20       System.out.println( "\n\nThe values of the modified array are:" );
21
22       // output modified array elements
23       for ( int value : array )
24          System.out.printf( "   %d", value );
```

Fig. 6.10 | Passing arrays and individual array elements to methods. (Part 1 of 3.)

```
25
26          System.out.printf(
27             "\n\nEffects of passing array element value:\n" +
28             "array[3] before modifyElement: %d\n", array[ 3 ] );
29
30          modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
31          System.out.printf(
32             "array[3] after modifyElement: %d\n", array[ 3 ] );
33       } // end main
34
35       // multiply each element of an array by 2
36       public static void modifyArray( int[] array2 )
37       {
38          for ( int counter = 0; counter < array2.length; counter++ )
39             array2[ counter ] *= 2;
40       } // end method modifyArray
41
42       // multiply argument by 2
43       public static void modifyElement( int element )
44       {
45          element *= 2;
46          System.out.printf(
47             "Value of element in modifyElement: %d\n", element );
48       } // end method modifyElement
49    } // end class PassArray
```

Fig. 6.10 | Passing arrays and individual array elements to methods. (Part 2 of 3.)

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Fig. 6.10 | Passing arrays and individual array elements to methods. (Part 3 of 3.)

Performance Tip 6.1

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and consume considerable storage for the copies of the arrays.

Multidimensional Arrays

- Multidimensional arrays with two dimensions often represent tables of values consisting of information arranged in rows and columns.
  - To identify a particular table element, we must specify two indices—the first identifies the element's row and the second its column.
- Arrays that require two indices to identify an element are called two-dimensional arrays.
- Multidimensional arrays can have more than two dimensions.
  - Java does not support multidimensional arrays directly, but it does allow you to specify one-dimensional arrays whose elements are one-dimensional arrays, thus achieving the same effect.

Multidimensional Arrays

- Figure 6.11 shows a two-dimensional array named `a` that contains three rows and four columns (i.e., a three-by-four array).
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array.

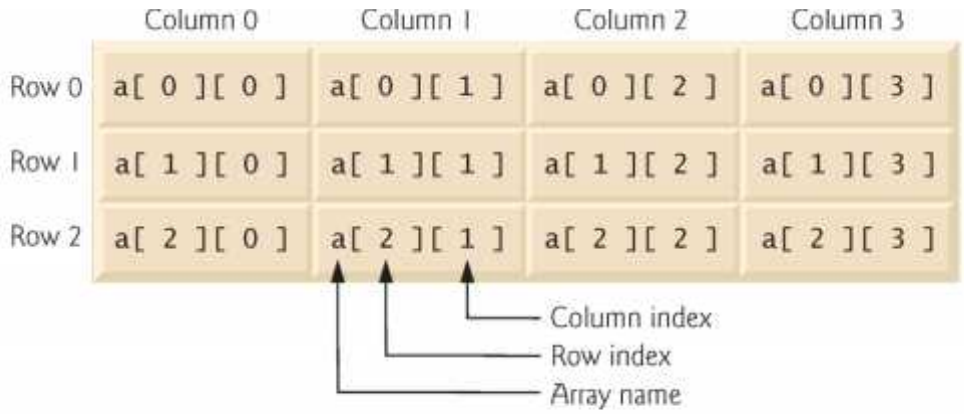


Fig. 6.11 | Two-dimensional array with three rows and four columns.

Multidimensional Arrays

- Multidimensional arrays can be initialized with array initializers in declarations.
- A two-dimensional array b with two rows and two columns could be declared and initialized with nested array initializers as follows:
  - `int[][] b = { { 1, 2 }, { 3, 4 } };`
- The initial values are grouped by row in braces.
- The compiler counts the number of nested array initializers to determine the number of rows in array b.
- The compiler counts the initializer values in the nested array initializer for a row to determine the number of columns in that row.
- Rows can have different lengths.
- Multidimensional arrays are maintained as arrays of one-dimensional arrays.

Multidimensional Arrays

- A multidimensional array with the same number of columns in every row can be created with an array-creation expression, as in:
  - `int[][] b = new int[ 3 ][ 4 ];`
- Programs can also use variables to specify array dimensions, because `new` creates arrays at execution time—not at compile time.
- As with one-dimensional arrays, the elements of a multidimensional array are initialized when the array object is created.
- A multidimensional array in which each row has a different number of columns can be created as follows:

```
int[][] b = new int[ 2 ][ ]; // create 2 rows
b[ 0 ] = new int[ 5 ]; // create 5 columns for row 0

b[ 1 ] = new int[ 3 ]; // create 3 columns for row 1
```

Multidimensional Arrays

- Figure 6.12 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to traverse the arrays (i.e., manipulate every element of each array).

```
1  // Fig. 6.12: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray
5  {
6     // create and output two-dimensional arrays
7     public static void main( String[] args )
8     {
9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // displays array1 by row
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // displays array2 by row
17    } // end main
18
```

Fig. 6.12 | Initializing two-dimensional arrays. (Part 1 of 2.)

```
19      // output rows and columns of a two-dimensional array
20      public static void outputArray( int[][] array )
21      {
22         // loop through array's rows
23         for ( int row = 0; row < array.length; row++ )
24         {
25            // loop through columns of current row
26            for ( int column = 0; column < array[ row ].length; column++ )
27               System.out.printf( "%d  ", array[ row ][ column ] );
28
29            System.out.println(); // start new line of output
30         } // end outer for
31      } // end method outputArray
32   } // end class InitArray
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. 6.12 | Initializing two-dimensional arrays. (Part 2 of 2.)

Case Study: Summarizing Student Exam Grades Using a Two-Dimensional Array

- Figure 6.13 uses a two-dimensional array `gradesArray` (lines 8–18) to store the grades of several students on multiple exams.
- Each row of the array represents a single student's grades for the entire course, and each column represents the grades of all the students who took a particular exam.

```
1  // Fig. 6.13: SummarizeGrades.java
2  // SummarizeGrades using a two-dimensional array to store grades.
3  public class SummarizeGrades
4  {
5     public static void main( String[] args )
6     {
7        // two-dimensional array of student grades
8        int[][] gradesArray =
9           { { 87, 96, 70 },
10            { 68, 87, 90 },
11            { 94, 100, 90 },
12            { 100, 81, 82 },
13            { 83, 65, 85 },
14            { 78, 87, 65 },
15            { 85, 75, 83 },
16            { 91, 94, 100 },
17            { 76, 72, 84 },
18            { 87, 93, 73 } };
19
20       // output grades array
21       outputGrades( gradesArray );
22
```

Fig. 6.13 | SummarizeGrades uses a two-dimensional array to store grades. (Part 1 of 10.)

```
23          // call methods getMinimum and getMaximum
24          System.out.printf( "\n%s %d\n%s %d\n\n",
25             "Lowest grade is", getMinimum( gradesArray ),
26             "Highest grade is", getMaximum( gradesArray ) );
27
28          // output grade distribution chart of all grades on all tests
29          outputBarChart( gradesArray );
30       } // end main
31
```

Fig. 6.13 | SummarizeGrades uses a two-dimensional array to store grades. (Part 2 of 10.)

```
32      // find minimum grade
33      public static int getMinimum( int grades[][] )
34      {
35         // assume first element of grades array is smallest
36         int lowGrade = grades[ 0 ][ 0 ];
37
38         // loop through rows of grades array
39         for ( int[] studentGrades : grades )
40         {
41            // loop through columns of current row
42            for ( int grade : studentGrades )
43            {
44               // if grade less than lowGrade, assign it to lowGrade
45               if ( grade < lowGrade )
46                  lowGrade = grade;
47            } // end inner for
48         } // end outer for
49
50         return lowGrade; // return lowest grade
51      } // end method getMinimum
52
```

Fig. 6.13 | SummarizeGrades uses a two-dimensional array to store grades. (Part 3 of 10.)

```
53      // find maximum grade
54      public static int getMaximum( int grades[][] )
55      {
56         // assume first element of grades array is largest
57         int highGrade = grades[ 0 ][ 0 ];
58
59         // loop through rows of grades array
60         for ( int[] studentGrades : grades )
61         {
62            // loop through columns of current row
63            for ( int grade : studentGrades )
64            {
65               // if grade greater than highGrade, assign it to highGrade
66               if ( grade > highGrade )
67                  highGrade = grade;
68            } // end inner for
69         } // end outer for
70
71         return highGrade; // return highest grade
72      } // end method getMaximum
73
```

Fig. 6.13 | `SummarizeGrades` uses a two-dimensional array to store grades. (Part 4 of 10.)

```
74      // determine average grade for particular set of grades
75      public static double getAverage( int[] setOfGrades )
76      {
77         int total = 0; // initialize total
78
79         // sum grades for one student
80         for ( int grade : setOfGrades )
81            total += grade;
82
83         // return average of grades
84         return (double) total / setOfGrades.length;
85      } // end method getAverage
86
```

Fig. 6.13 | `SummarizeGrades` uses a two-dimensional array to store grades. (Part 5 of 10.)

```
87      // output bar chart displaying overall grade distribution
88      public static void outputBarChart( int grades[][] )
89      {
90         System.out.println( "Overall grade distribution:" );
91
92         // stores frequency of grades in each range of 10 grades
93         int[] frequency = new int[ 11 ];
94
95         // for each grade in GradeBook, increment the appropriate frequency
96         for ( int[] studentGrades : grades )
97         {
98            for ( int grade : studentGrades )
99               ++frequency[ grade / 10 ];
100        } // end outer for
101
102        // for each grade frequency, print bar in chart
103        for ( int count = 0; count < frequency.length; count++ )
104        {
105           // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
106           if ( count == 10 )
107              System.out.printf( "%5d: ", 100 );
```

Fig. 6.13 | SummarizeGrades uses a two-dimensional array to store grades. (Part 6 of 10.)

```
108          else
109             System.out.printf( "%02d-%02d: ",
110                count * 10, count * 10 + 9  );
111
112          // print bar of asterisks
113          for ( int stars = 0; stars < frequency[ count ]; stars++ )
114             System.out.print( "*" );
115
116          System.out.println(); // start a new line of output
117       } // end outer for
118    } // end method outputBarChart
119
120    // output the contents of the grades array
121    public static void outputGrades( int grades[][] )
122    {
123       System.out.println( "The grades are:\n" );
124       System.out.print( "            " ); // align column heads
125
126       // create a column heading for each of the tests
127       for ( int test = 0; test < grades[ 0 ].length; test++ )
128          System.out.printf( "Test %d  ", test + 1 );
129
```

Fig. 6.13 | SummarizeGrades uses a two-dimensional array to store grades. (Part 7 of 10.)

```
130         System.out.println( "Average" ); // student average column heading
131
132         // create rows/columns of text representing array grades
133         for ( int student = 0; student < grades.length; student++ )
134         {
135            System.out.printf( "Student %2d", student + 1 );
136
137            for ( int test : grades[ student ] ) // output student's grades
138               System.out.printf( "%8d", test );
139
140            // call method getAverage to calculate student's average grade;
141            // pass row of grades as the argument to getAverage
142            double average = getAverage( grades[ student ] );
143            System.out.printf( "%9.2f\n", average );
144         } // end outer for
145      } // end method outputGrades
146   } // end class SummarizeGrades
```

Fig. 6.13 | `SummarizeGrades` uses a two-dimensional array to store grades. (Part 8 of 10.)

```
The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
```

Fig. 6.13 | `SummarizeGrades` uses a two-dimensional array to store grades. (Part 9 of 10.)

```
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade is 65
Highest grade is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 6.13 | `SummarizeGrades` uses a two-dimensional array to store grades. (Part 10 of 10.)

6.9 Case Study: Summarizing Student Exam Grades Using a Two-Dimensional Array (cont.)

- Methods `getMinimum`, `getMaximum`, `outputBarChart` and `outputGrades` each loop through array `grades` by using nested `for` statements.
- The outer enhanced `for` statement iterates through the two-dimensional array `grades`, assigning successive rows to parameter `studentGrades` on successive iterations.
- The square brackets following the parameter name indicate that `studentGrades` refersto a one-dimensional `int` array—namely, a row in array`grades` containing one student's grades.
- The inner enhanced `for` statement then loops through `studentGrades`.

Variable-Length Argument Lists

- With variable-length argument lists, you can create methods that receive an unspecified number of arguments.
- A type followed by an ellipsis (...) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type.
  - Can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list.
- Figure 6.14 demonstrates method `average` which receives a variable-length sequence of `double`s.
- Java treats the variable-length argument list as an array of the specified type.

Common Programming Error 6.6

Placing an ellipsis indicating a variable-length argument list in the middle of a parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

```
1  // Fig. 6.14: VarargsTest.java
2  // Using variable-l
3  ength argument lists.
4
5  public class VarargsTest
6  {
7     // calculate average
8     public static double average( double... numbers )
9     {
10       double total = 0.0; // initialize total
11
12       // calculate total using the enhanced for statement
13       for ( double d : numbers )
14          total += d;
15
16       return total / numbers.length;
17    } // end method average
18
19    public static void main( String[] args )
20    {
21       double d1 = 10.0;
22       double d2 = 20.0;
23       double d3 = 30.0;
24       double d4 = 40.0;
```

Fig. 6.14 | Using variable-length argument lists. (Part 1 of 2.)

```
25
26          System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
27             d1, d2, d3, d4 );
28
29          System.out.printf( "Average of d1 and d2 is %.1f\n",
30             average( d1, d2 ) );
31          System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
32             average( d1, d2, d3 ) );
33          System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
34             average( d1, d2, d3, d4 ) );
35       } // end main
36    } // end class VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 6.14 | Using variable-length argument lists. (Part 2 of 2.)

Sorting Arrays

Sorting Arrays

Sorting data means placing the data in to **some particular order.**

i.e

Ascending
Descending

- Sorting is one of the most important computing applications.
  - Banks sort all cheques by account number.
  - Phone accounts are sorted by last name and then by first name and middle initial.

Bubble Sort

The following program sorts data into ascending order

The sorting technique used is called a bubble sort.

Smaller elements in the array gradually "bubble" their way to the top of the array

The algorithm needs several passes through the array data.

On each pass, successive pairs of elements are compared.

- If a pair is in **increasing order** or identical then the pair remain unchanged
- If the pair of elements are in **decreasing order** their values are swapped in the array.

Bubble Sort

- The program compares

  a[0] and a[1], then
  a[1] and a[2], then
  a[2] and a[3], etc ……. until
  a[8] and a[9] are compared

- Only nine comparisons are performed on the array of size 10

- A small value may move up only one position.

Bubble Sort

If a **swap** is needed, it is done in three moves

```
hold = a[ j ];
a[ j ] = a[ j + 1 ];
a[ j + 1 ] = hold;
```

where **hold** temporarily stores one of the two values being swapped.

Java Bubble Sort Example

```
/*
    Java Bubble Sort Example
    This Java bubble sort example shows how to sort an array of int using bubble
    sort algorithm. Bubble sort is the simplest sorting algorithm.
*/

public class BubbleSort {

    public static void main(String[] args) {

    //create an int array we want to sort using bubble sort algorithm
        int intArray[] = new int[]{5,90,35,45,150,3};

        //print array before sorting using bubble sort algorithm
        System.out.println("Array Before Bubble Sort");
        for(int i=0; i < intArray.length; i++){
            System.out.print(intArray[i] + " ");
        }

        //sort an array using bubble sort algorithm
        bubbleSort(intArray);

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

```
        //print array after sorting using bubble sort algorithm
        System.out.println("Array After Bubble Sort");
        for(int i=0; i < intArray.length; i++){
            System.out.print(intArray[i] + " ");
        }

    }
/*
    * In bubble sort, we traverse the array from first
    * to array_length - 1 position and compare the element with the next one.
    * Elements are swapped with the next element if the next element is greater.
    *
    * Bubble sort steps are as follows.
    *
    * 1. Compare array[0] & array[1]
    * 2. If array[0] > array [1] swap it.
    * 3. Compare array[1] & array[2]
    * 4. If array[1] > array[2] swap it.
    * ...
    * 5. Compare array[n-1] & array[n]
    * 6. if [n-1] > array[n] then swap it.
    *
    * After this step we will have largest element at the last index.
    *
    * Repeat the same steps for array[1] to array[n-1]
    *
    */
```

```
    private static void bubbleSort(int[] intArray)
        int n = intArray.length;
        int hold = 0;

        for(int i=0; i < n; i++){//allows us to pass or loop around array
                for(int j=1; j < (n-i); j++){//allows on pass or comparison

                        if(intArray[j] > intArray[j+1]){
                                //swap the elements!
                                hold = intArray[j];
                                intArray[j] = intArray[j + 1];
                                intArray[j + 1] = hold;
                        }

                }
        }

    }
}
```

Output of the Bubble Sort Example would be:

```
/*
Output of the Bubble Sort Example would be

Array Before Bubble Sort
5 90 35 45 150 3
Array After Bubble Sort
3 5 35 45 90 150

*/
```

Binary Search

Binary Search

Let's say you want to read page 99 in a book, or find the name Tyner in the telephone directory ….

You wouldn't start at page 1 and leaf through each item on each page until you got to what you wanted!!!

When there is some order in the data being searched we can use that to speed things up.

This is what a *binary search* does ...

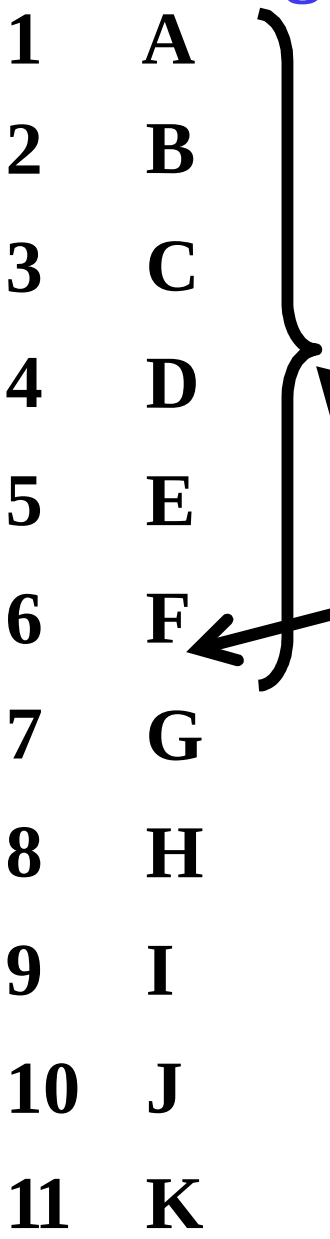
Searching ...

In overview a binary search repeatedly splits the array of data in half, until quite quickly it will have homed in on the required item ….

However in order for the binary search to work -

the data MUST BE SORTED

Searching ...



Say we're searching for B (it works the same for numbers) in the list opposite.

Go to midway point in set, which is at F.

Compare B (what we're looking for) with F - Since B is lower than F we now know to only search A to F, and we start again ...

Searching ...

1	**A**
2	**B**
3	**C**
4	**D**
5	**E**
6	**F**

Go to midway point again, which in this case will be 3 (or 4 if we round up)

Compare B with C - Since B is lower than C we now know to only search A to B, and we start again ...

This is binary searching in its simplest form.

Another example ...

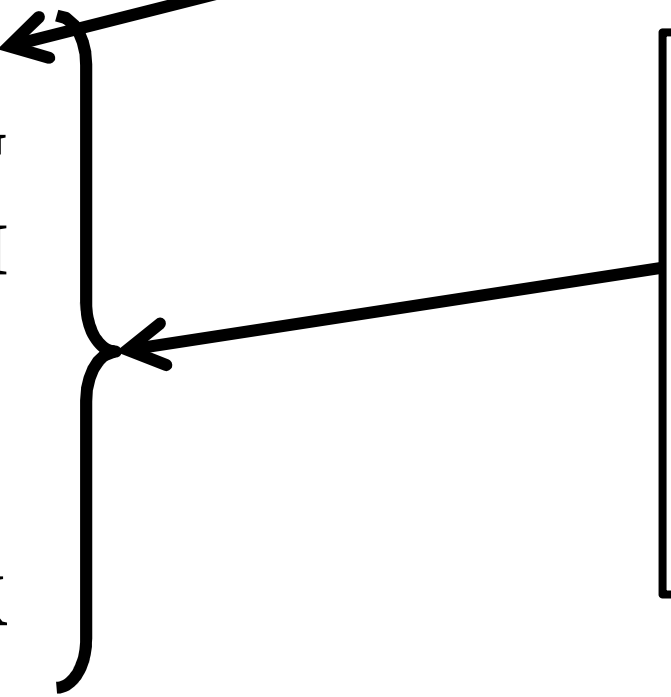
Searching ...

1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I
10	J
11	K

Say we're searching for J …

Go to midway point in set, which is at F.

Compare J (what we're looking for) with F - Since J is higher than F we now know to only search F to K, and we start again ...



Algorithm for Binary Search

```
bottom = first element
top = last element
While (top >=bottom) and (not found)
        mid = (top+bottom)/2
        if (list(mid) = item to find) then
                found = true
        elseif item to find > list(mid) then
                bottom = mid + 1
        else
                top = mid - 1
        end if
end loop

If found = true
        Display “Item found” list(mid)
Else
        Display “Item not found”
End If
```

Binary Search

This is a huge increase in performance over the **linear search** that requires comparing the search key with an average of half of the elements in the array.

For a one billion element array:

- Average of 5000 million comparisons - **linear** search of unsorted array
- Maximum of 30 comparisons – **binary** search of sorted array

Program Example - Binary Search

```
import java.util.Scanner;

class BinarySearch
{
  public static void main(String args[])
  {
    int c, first, last, middle, n, search, array[];

    Scanner in = new Scanner(System.in);
    System.out.println("Enter number of elements");
    n = in.nextInt();
    array = new int[n];

    System.out.println("Enter " + n + " integers");
```

Program Example (Contd.) - Binary Search

```
for (c = 0; c < n; c++)
      array[c] = in.nextInt();

    System.out.println("Enter value to find");
    search = in.nextInt();

    first  = 0;
    last   = n - 1;
    middle = (first + last)/2;

    while( first <= last )
    {
      if ( array[middle] < search )
        first = middle + 1;
      else if ( array[middle] == search )
```

Program Example (Contd.) - Binary Search

```
 {
        System.out.println(search + " found at location " +
(middle + 1) + ".");
        break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
    }
    if ( first > last )
      System.out.println(search + " is not present in the list.
\n");
   }
}
```

Arrays class binary search

- binarySearch method returns the location if a match occurs
- Otherwise -(x+1) where x is the number of elements in the array
- For example in the second case in our example on the next slide: when p is not present in characters array the returned value will be -6.

Arrays class binary search

```
import java.util.Arrays;

class BS
{
  public static void main(String args[])
  {
    char characters[] = { ’a’, ’b’, ’c’, ’d’, ’e’ };

    System.out.println(Arrays.binarySearch(characters, ’a’));
    System.out.println(Arrays.binarySearch(characters, ’p’));
  }
}
```

Class `Arrays`

- Class Arrays helps you avoid reinventing the wheel by providing `static` methods for common array manipulations.
- These methods include sort for sorting an array (i.e., arranging elements into increasing order), binarySearch for searching an array (i.e., determining whether an array contains a specific value and, if so, where the value is located), equals for comparing arrays and fill for placing values into an array.
- These methods are overloaded for primitive-type arrays and for arrays of objects.
- You can copy arrays with class `System`'s `static` arraycopy method.

```
1  // Fig. 6.16: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations
6  {
7     public static void main( String[] args )
8     {
9        // sort doubleArray into ascending order
10       double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort( doubleArray );
12       System.out.printf( "\ndoubleArray: " );
13
14       for ( double value : doubleArray )
15          System.out.printf( "%.1f ", value );
16
17       // fill 10-element array with 7s
18       int[] filledIntArray = new int[ 10 ];
19       Arrays.fill( filledIntArray, 7 );
20       displayArray( filledIntArray, "filledIntArray" );
21
22       // copy array intArray into array intArrayCopy
23       int[] intArray = { 1, 2, 3, 4, 5, 6 };
24       int[] intArrayCopy = new int[ intArray.length ];
```

Fig. 6.16 | Arrays class methods. (Part 1 of 4.)

```
25          System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
26          displayArray( intArray, "intArray" );
27          displayArray( intArrayCopy, "intArrayCopy" );
28
29          // compare intArray and intArrayCopy for equality
30          boolean b = Arrays.equals( intArray, intArrayCopy );
31          System.out.printf( "\n\nintArray %s intArrayCopy\n",
32             ( b ? "==" : "!=" ) );
33
34          // compare intArray and filledIntArray for equality
35          b = Arrays.equals( intArray, filledIntArray );
36          System.out.printf( "intArray %s filledIntArray\n",
37             ( b ? "==" : "!=" ) );
38
39          // search intArray for the value 5
40          int location = Arrays.binarySearch( intArray, 5 );
41
42          if ( location >= 0 )
43             System.out.printf(
44                "Found 5 at element %d in intArray\n", location );
45          else
46             System.out.println( "5 not found in intArray" );
47
```

Fig. 6.16 | `Arrays` class methods. (Part 2 of 4.)

```
48          // search intArray for the value 8763
49          location = Arrays.binarySearch( intArray, 8763 );
50
51          if ( location >= 0 )
52             System.out.printf(
53                "Found 8763 at element %d in intArray\n", location );
54          else
55             System.out.println( "8763 not found in intArray" );
56       } // end main
57
58       // output values in each array
59       public static void displayArray( int[] array, String description )
60       {
61          System.out.printf( "\n%s: ", description );
62
63          for ( int value : array )
64             System.out.printf( "%d ", value );
65       } // end method displayArray
66    } // end class ArrayManipulations
```

Fig. 6.16 | `Arrays` class methods. (Part 3 of 4.)

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Fig. 6.16 | `Arrays` class methods. (Part 4 of 4.)

Common Programming Error 6.7

Passing an unsorted array to `binarySearch` *is a logic error—the value returned is undefined.*

Introduction to Collections and Class `ArrayList`

- Collections provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- The collection class ArrayList<T> (from package `java.util`) can dynamically change its size to accommodate more elements.
- The `T` is a placeholder—when declaring a new `ArrayList`, replace it with the type of elements that you want the `ArrayList` to hold.
- This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes.
- Classes with this kind of placeholder that can be used with any type are called generic classes.
- Figure 6.17 shows some common methods of class `ArrayList<T>`.
- Lines 20–21 display the items in the `ArrayList`.

Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Removes the first occurrence of the specified value.
`remove`	Removes the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 6.17 | Some methods and properties of class `ArrayList<T>`.

```
1  // Fig. 6.18: ArrayListCollection.java
2  // Generic ArrayList collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main( String[] args )
8     {
9        // create a new ArrayList of Strings
10       ArrayList< String > items = new ArrayList< String >();
11
12       items.add( "red" ); // append an item to the list
13       items.add( 0, "yellow" ); // insert the value at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:" );
18
19       // display the colors in the list
20       for ( int i = 0; i < items.size(); i++ )
21          System.out.printf( " %s", items.get( i ) );
22
```

Fig. 6.18 | Generic `ArrayList<T>` collection demonstration. (Part 1 of 3.)

```
23          // display colors using foreach in the display method
24          display( items,
25             "\nDisplay list contents with enhanced for statement:" );
26
27          items.add( "green" ); // add "green" to the end of the list
28          items.add( "yellow" ); // add "yellow" to the end of the list
29          display( items, "List with two new elements:" );
30
31          items.remove( "yellow" ); // remove the first "yellow"
32          display( items, "Remove first instance of yellow:" );
33
34          items.remove( 1 ); // remove item at index 1
35          display( items, "Remove second list element (green):" );
36
37          // check if a value is in the List
38          System.out.printf( "\"red\" is %sin the list\n",
39             items.contains( "red" ) ? "": "not " );
40
41          // display number of elements in the List
42          System.out.printf( "Size: %s\n", items.size() );
43       } // end main
44
```

Fig. 6.18 | Generic `ArrayList<T>` collection demonstration. (Part 2 of 3.)

```
45      // display the ArrayList's elements on the console
46      public static void display( ArrayList< String > items, String header )
47      {
48         System.out.print( header ); // display header
49
50         // display each element in items
51         for ( String item : items )
52            System.out.printf( " %s", item );
53
54         System.out.println(); // display end of line
55      } // end method display
56   } // end class ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Fig. 6.18 | Generic `ArrayList<T>` collection demonstration. (Part 3 of 3.)