Chapter 7
Arrays and ArrayLists

Java™ How to Program, 8/e

7.2 Arrays

- Array
  - Group of variables (called elements) containing values of the same type.
  - Arrays are objects so they are reference types.
  - Elements can be either primitive or reference types.
- Refer to a particular element in an array
  - Use the element's index.
  - 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.
- The highest index in an array is one less than the number of elements in the array.
- Array names follow the same conventions as other variable names.

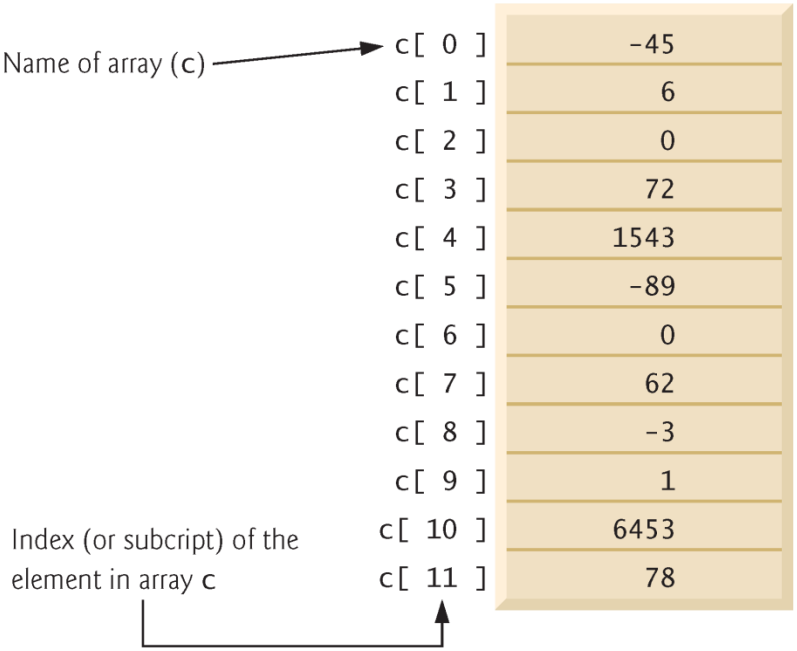


Fig. 7.1 | A 12-element array.

7.2 Arrays (Cont.)

- An index must be a nonnegative integer.
  - Can use an expression as an index.
- An indexed array name is an array-access expression.
  - Can be used on the left side of an assignment to place a new value into an array element.
- Every array object knows its own length and stores it in a `length` instance variable.
  - `length` cannot be changed because it's a `final` variable.

Common Programming Error 7.1

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.*

7.3 Declaring and Creating Arrays

- Array objects
  - Created with keyword `new`.
  - You specify the element type and the number of elements in an array-creation expression, which returns a reference that can be stored in an array variable.
- Declaration and array-creation expression for an array of 12 `int` elements

```
int[] c = new int[ 12 ];
```

- Can be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[ 12 ]; // creates the array
```

7.3 Declaring and Creating Arrays (Cont.)

- In a declaration, square brackets following a type indicate that a variable will refer to an array (i.e., store an array reference).
- When an array is created, each element of the array receives a default value
  - Zero for the numeric primitive-type elements, `false` for `boolean` elements and `null` for references.

Common Programming Error 7.2

In an array declaration, specifying the number of elements in the square brackets of the declaration (e.g., `int[12] c;`*) is a syntax error.*

7.3 Declaring and Creating Arrays (Cont.)

- When the element type and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables.
  - int[] a,b;
  - For readability, declare only one variable per declaration.

Common Programming Error 7.3

Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration `int[] a, b, c;`. *If* `a`, `b` *and* `c` *should be declared as array variables, then this declaration is correct—placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only* `a` *is intended to be an array variable, and* `b` *and* `c` *are intended to be individual* `int` *variables, then this declaration is incorrect—the declaration* `int a[], b, c;` *would achieve the desired result.*

7.3 Declaring and Creating Arrays (Cont.)

- Every element of a primitive-type array contains a value of the array's declared element type.
  - Every element of an `int` array is an `int` value.
- Every element of a reference-type array is a reference to an object of the array's declared element type.
  - Every element of a `String` array is a reference to a `String` object.

7.4 Examples Using Arrays

- Fig. 7.2 uses keyword `new` to create an array of 10 `int` elements, which are initially zero (the default for `int` variables).

```
1  // Fig. 7.2: 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
```

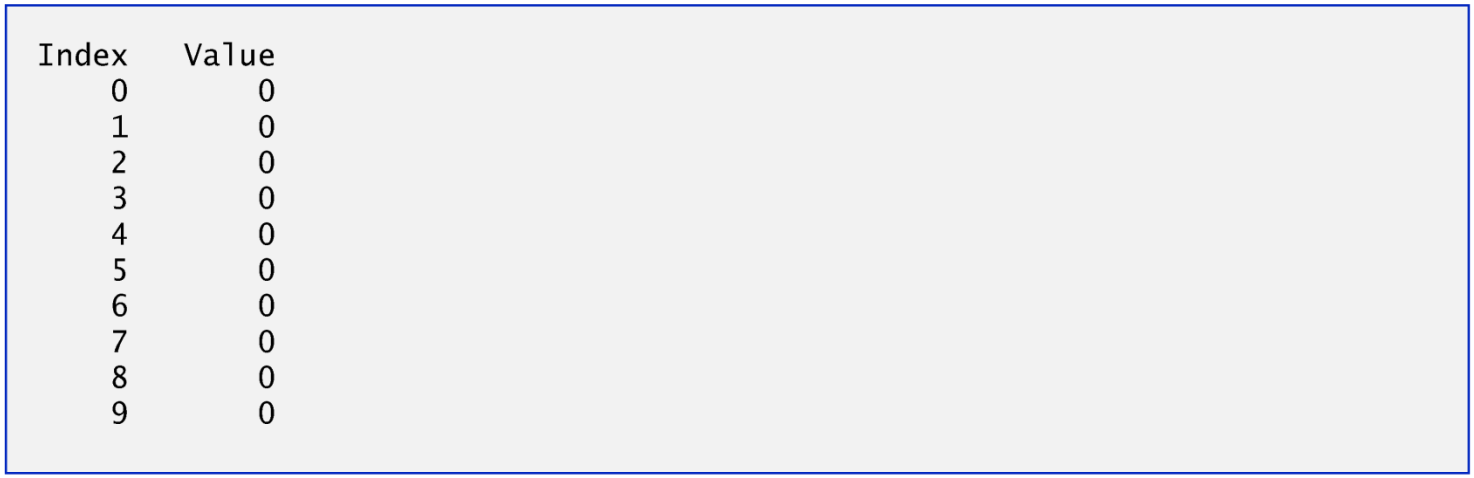
Variable `array` will refer to an array of `int` values.

Creates an array of `10` `int` elements, each with the value `0` by default

`for` statement iterates while `counter` is less than the array's `length`

Array-access expression gets the value at the index represented by `counter`

Fig. 7.2 | 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. 7.2 | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

7.4 Examples Using Arrays (Cont.)

- Array initializer
  - A comma-separated list of expressions (called an initializer list) enclosed in braces.
  - Used to create an array and initialize its elements.
  - Array length is determined by the number of elements in the initializer list.

```
int[] n = { 10, 20, 30, 40, 50 };
```

    - Creates a five-element array with index values 0–4.
- Compiler counts the number of initializers in the list to determine the size of the array
  - Sets up the appropriate `new` operation "behind the scenes."

```
1  // Fig. 7.3: 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
```

Array initializer list for a 10-element `int` array

Fig. 7.3 | 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. 7.3 | Initializing the elements of an array with an array initializer. (Part 2 of 2.)

7.4 Examples Using Arrays (Cont.)

- The application in Fig. 7.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20).

```
1  // Fig. 7.4: 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
```

Named constant representing the array length

Create an array with ARRAY_LENGTH elements

Calculates and stores a value for each element of the array

Fig. 7.4 | 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. 7.4 | Calculating the values to be placed into the elements of an array. (Part 2 of 2.)

7.4 Examples Using Arrays (Cont.)

- `final` variables must be initialized before they are used and cannot be modified thereafter.
- An attempt to modify a `final` variable after it's initialized causes a compilation error
  - `cannot assign a value to final variable` *variableName*
- An attempt to access the value of a `final` variable before it's initialized causes a compilation error
  - `variable` *variableName* `might not have been initialized`

Good Programming Practice 7.2

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 7.4

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

7.4 Examples Using Arrays (Cont.)

- Figure 7.5 sums the values contained in a 10-element integer array.
- Often, the elements of an array represent a series of values to be used in a calculation.

```
1  // Fig. 7.5: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int counter = 0; counter < array.length; counter++ )
13          total += array[ counter ];
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // end main
17 } // end class SumArray
```

Adds each value in `array` to `total`, which is displayed when the loop terminates

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

Fig. 7.5 | Computing the sum of the elements of an array.

7.4 Examples Using Arrays (Cont.)

- Sometimes, programs use counter variables to summarize data, such as the results of a survey.
- Fig. 6.8 used separate counters in a die-rolling program to track the number of occurrences of each side of a six-sided die as the program rolled the die 6000 times.
- Fig. 7.7 shows an array version of this application.
  - Line 14 of this program replaces lines 23–46 of Fig. 6.8.
- Array `frequency` must be large enough to store six counters.
  - We use a seven-element array in which we ignore `frequency[0]`
  - More logical to have the face value 1 increment `frequency[1]` than `frequency[0]`.

```
1  // Fig. 7.7: RollDie.java
2  // Die-rolling program using arrays instead of switch.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // random number generator
10       int[] frequency = new int[ 7 ]; // array of frequency counters
11
12       // roll die 6000 times; use die value as frequency index
13       for ( int roll = 1; roll <= 6000; roll++ )
14          ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];
15
16       System.out.printf( "%s%10s\n", "Face", "Frequency" );
17
18       // output each array element's value
19       for ( int face = 1; face < frequency.length; face++ )
20          System.out.printf( "%4d%10d\n", face, frequency[ face ] );
21    } // end main
22 } // end class RollDie
```

Random number from 1 to 6 is used as index into `frequency` array to determine which element to increment

Fig. 7.7 | Die-rolling program using arrays instead of `switch`. (Part 1 of 2.)

```
Face Frequency
   1        988
   2        963
   3       1018
   4       1041
   5        978
   6       1012
```

Fig. 7.7 | Die-rolling program using arrays instead of `switch`. (Part 2 of 2.)

Error-Prevention Tip 7.1

An exception indicates that an error has occurred in a program. 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.*

Error-Prevention Tip 7.2

When writing code to loop through an array, ensure that the array index is always greater than or equal to 0 and less than the length of the array. The loop-continuation condition should prevent accessing elements outside this range.

7.5 Case Study: Card Shuffling and Dealing Simulation

- Examples thus far used arrays containing elements of primitive types.
- Elements of an array can be either primitive types or reference types.
- Next example uses an array of reference-type elements—objects representing playing cards—to develop a class that simulates card shuffling and dealing.

7.5 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Class Card contains two String instance variables—face and suit—that are used to store references to the face and suit names for a specific Card.
- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.
- every class extends object class by default.
- Override toString() in Object class to print the fields in an appropriate manner.
- Method toString creates a String consisting of the face of the card, " of " and the suit of the card.
  - Can invoke explicitly to obtain a string representation of a Card.
  - Called implicitly when the object is used where a String is expected.

```
1  // Fig. 7.9: Card.java
2  // Card class represents a playing card.
3
4  public class Card
5  {
6     private String face; // face of card ("Ace", "Deuce", ...)
7     private String suit; // suit of card ("Hearts", "Diamonds", ...)
8
9     // two-argument constructor initializes card's face and suit
10    public Card( String cardFace, String cardSuit )
11    {
12       face = cardFace; // initialize face of card
13       suit = cardSuit; // initialize suit of card
14    } // end two-argument Card constructor
15
16    // return String representation of Card
17    public String toString()
18    {
19       return face + " of " + suit;
20    } // end method toString
21 } // end class Card
```

Must be declared with this first line it is to be called implicitly to convert `Card` objects to `String` representations

Fig. 7.9 | `Card` class represents a playing card.

7.5 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Class `DeckOfCards` (Fig. 7.10) declares as an instance variable a `Card` array named `deck`.
- `Deck`'s elements are `null` by default
  - Constructor fills the `deck` array with `Card` objects.
- Method `shuffle` shuffles the `Card`s in the deck.
  - Loops through all 52 `Card`s (array indices 0 to 51).
  - Each `Card` swapped with a randomly chosen other card in the deck.
- Method `dealCard` deals one `Card` in the array.
  - `currentCard` indicates the index of the next `Card` to be dealt
  - Returns `null` if there are no more cards to deal

```
1  // Fig. 7.10: DeckOfCards.java
2  // DeckOfCards class represents a deck of playing cards.
3  import java.util.Random;
4
5  public class DeckOfCards
6  {
7     private Card[] deck; // array of Card objects
8     private int currentCard; // index of next Card to be dealt
9     private static final int NUMBER_OF_CARDS = 52; // constant # of Cards
10    // random number generator
11    private static final Random randomNumbers = new Random();
12
13    // constructor fills deck of Cards
14    public DeckOfCards()
15    {
16       String[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
17          "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
18       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
19
20       deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
21       currentCard = 0; // set currentCard so first Card dealt is deck[ 0 ]
22
```

Declares the array of `Cards`; `deck` is `null` until the array is created

Creates the array of `Card` variables

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards. (Part 1 of 3.)

```
23          // populate deck with Card objects
24          for ( int count = 0; count < deck.length; count++ )
25             deck[ count ] =
26                new Card( faces[ count % 13 ], suits[ count / 13 ] );
27       } // end DeckOfCards constructor
28
29       // shuffle deck of Cards with one-pass algorithm
30       public void shuffle()
31       {
32          // after shuffling, dealing should start at deck[ 0 ] again
33          currentCard = 0; // reinitialize currentCard
34
35          // for each Card, pick another random Card and swap them
36          for ( int first = 0; first < deck.length; first++ )
37          {
38             // select a random number between 0 and 51
39             int second =  randomNumbers.nextInt( NUMBER_OF_CARDS );
40
41             // swap current Card with randomly selected Card
42             Card temp = deck[ first ];
43             deck[ first ] = deck[ second ];
44             deck[ second ] = temp;
45          } // end for
46       } // end method shuffle
```

Creates a `Card` for the current array element

Swaps the current element with the randomly selected element

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards. (Part 2 of 3.)

```
47
48      // deal one Card
49      public Card dealCard()
50      {
51         // determine whether Cards remain to be dealt
52         if ( currentCard < deck.length )
53            return deck[ currentCard++ ]; // return current Card in array
54         else
55            return null; // return null to indicate that all Cards were dealt
56      } // end method dealCard
57   } // end class DeckOfCards
```

Ensures that `currentCard` is less than the `length` of the array; if so, a `Card` is returned; otherwise, `null` is returned

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards. (Part 3 of 3.)

7.5 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Figure 7.11 demonstrates class `DeckOfCards` (Fig. 7.10).
- When a `Card` is output as a `String`, the `Card`'s `toString` method is implicitly invoked.

```
1  // Fig. 7.11: DeckOfCardsTest.java
2  // Card shuffling and dealing.
3
4  public class DeckOfCardsTest
5  {
6     // execute application
7     public static void main( String[] args )
8     {
9        DeckOfCards myDeckOfCards = new DeckOfCards();
10       myDeckOfCards.shuffle(); // place Cards in random order
11
12       // print all 52 Cards in the order in which they are dealt
13       for ( int i = 1; i <= 52; i++ )
14       {
15          // deal and display a Card
16          System.out.printf( "%-19s", myDeckOfCards.dealCard() );
17
18          if ( i % 4 == 0 ) // output newline every 4 cards
19             System.out.println();
20       } // end for
21    } // end main
22 } // end class DeckOfCardsTest
```

Deals a `Card`; the `Card`'s `toString` method is called implicitly to obtain the `String` representation that is output

Fig. 7.11 | Card shuffling and dealing. (Part 1 of 2.)

```
Six of Spades      Eight of Spades    Six of Clubs       Nine of Hearts
Queen of Hearts    Seven of Clubs     Nine of Spades     King of Hearts
Three of Diamonds  Deuce of Clubs     Ace of Hearts      Ten of Spades
Four of Spades     Ace of Clubs       Seven of Diamonds  Four of Hearts
Three of Clubs     Deuce of Hearts    Five of Spades     Jack of Diamonds
King of Clubs      Ten of Hearts      Three of Hearts    Six of Diamonds
Queen of Clubs     Eight of Diamonds  Deuce of Diamonds  Ten of Diamonds
Three of Spades    King of Diamonds   Nine of Clubs      Six of Hearts
Ace of Spades      Four of Diamonds   Seven of Hearts    Eight of Clubs
Deuce of Spades    Eight of Hearts    Five of Hearts     Queen of Spades
Jack of Hearts     Seven of Spades    Four of Clubs      Nine of Diamonds
Ace of Diamonds    Queen of Diamonds  Five of Clubs      King of Spades
Five of Diamonds   Ten of Clubs       Jack of Spades     Jack of Clubs
```

Fig. 7.11 | Card shuffling and dealing. (Part 2 of 2.)

7.6 Enhanced `for` Statement

- Enhanced `for` statement
  - Iterates through the elements of an array without using a counter.
  - Avoids 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 array's element type.
- The enhanced `for` statement simplifies the code for iterating through an array.

```
1  // Fig. 7.12: EnhancedForTest.java
2  // Using enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int number : array )
13          total += number;
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // end main
17 } // end class EnhancedForTest
```

For each element in `array`, assign the element's value to `number`, then add `number` to `total`

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

Fig. 7.12 | Using the enhanced `for` statement to total integers in an array.

The enhanced `for` statement simplifies the code for iterating through an array.

7.6 Enhanced `for` Statement (Cont.)

- The enhanced `for` statement can be used only to obtain array elements
  - Since the indexed element is put in a variable, it is not the actual element in the array!!!
  - It cannot be used to modify elements.
  - To modify elements, use the traditional counter-controlled `for` statement.
- Can be used in place of the counter-controlled `for` statement if you don't need to access the index of the element.

7.7 Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets.
  - Since every array object "knows" its own length, we need not pass the array length as an additional argument.
- To receive an array, the method's parameter list must specify an array parameter.
- When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference.
- When an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's value.

```
1  // Fig. 7.13: 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 );
```

Passes the reference to `array` into method `modifyArray`

Fig. 7.13 | 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
```

Passes a copy of `array[3]`'s `int` value into `modifyElement`

Method receives copy of an array's reference, which gives the method direct access to the original array in memory

Method receives copy of an `int` value; the method cannot modify the original `int` value in `main`

Fig. 7.13 | 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. 7.13 | Passing arrays and individual array elements to methods. (Part 3 of 3.)

7.7 Passing Arrays to Methods (Cont.)

- Pass-by-value (also called call-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.
- Pass-by-reference (also called call-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.

7.7 Passing Arrays to Methods (Cont.)

- All arguments in Java are passed by value.
- A method call can pass two types of values to a method
  - Copies of primitive values
  - Copies of references to objects
- Objects cannot be passed to methods.
- If a method modifies a reference-type parameter so that it refers to another object, only the parameter refers to the new object
  - The reference stored in the caller's variable still refers to the original object.
- Although an object's reference is passed by value, a method can still interact with the referenced object by calling its methods using the copy of the object's reference.

7.8 Case Study: Class GradeBook Using an Array to Store Grades

- WILL BE COVERED IN LAB.

7.9 Multidimensional Arrays

- Two-dimensional arrays are often used to represent tables of values consisting of information arranged in rows and columns.
- Identify a particular table element with two indices.
  - By convention, the first identifies the element's row and the second its column.
- Multidimensional arrays can have more than two dimensions.
- Java does not support multidimensional arrays directly
  - Allows you to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect.
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array.

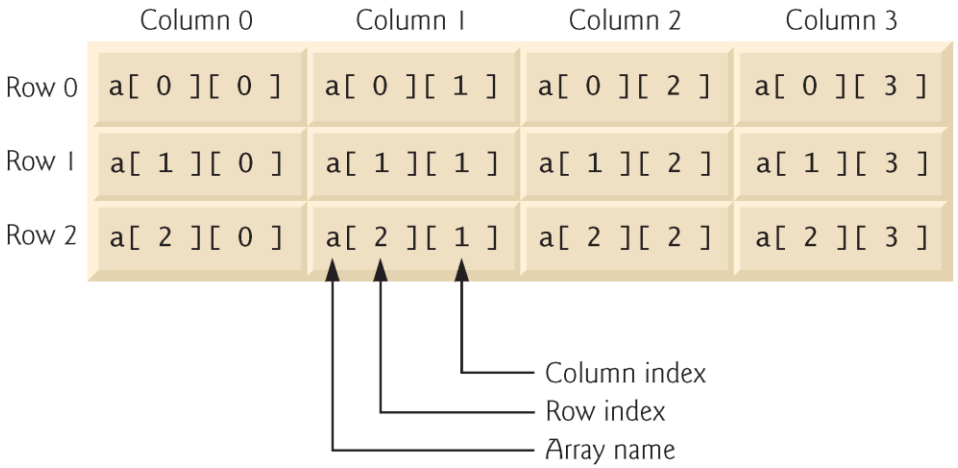


Fig. 7.16 | Two-dimensional array with three rows and four columns.

7.9 Multidimensional Arrays (Cont.)

- 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 number of nested array initializers (represented by sets of braces within the outer braces) determines the number of rows.
  - The number of initializer values in the nested array initializer for a row determines the number of columns in that row.
  - Rows can have different lengths.

7.9 Multidimensional Arrays (Cont.)

- The lengths of the rows in a two-dimensional array are not required to be the same:

```
int[][] b = { { 1, 2 }, { 3, 4, 5 } };
```

  - Each element of `b` is a reference to a one-dimensional array of `int` variables.
  - The `int` array for row `0` is a one-dimensional array with two elements (`1` and `2`).
  - The `int` array for row `1` is a one-dimensional array with three elements (`3`, `4` and `5`).

7.9 Multidimensional Arrays (Cont.)

- A multidimensional array with the same number of columns in every row can be created with an array-creation expression.

```
int[][] b = new int[ 3 ][ 4 ];
```

  - 3 rows and 4 columns.
- 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
```

  - Creates a two-dimensional array with two rows.
  - Row 0 has five columns, and row 1 has three columns.

7.9 Multidimensional Arrays (Cont.)

- Figure 7.17 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to traverse the arrays.

```
1  // Fig. 7.17: 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
```

array1 has two rows, each with three columns

array2 has three rows with two, one and three columns, respectively

Fig. 7.17 | 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
```

Method can received any two-dimensional array of `int` values

Outer loop determines number of rows in the array

Inner loop determines number of columns in the *current row*

```
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. 7.17 | Initializing two-dimensional arrays. (Part 2 of 2.)

7.10 Case Study: Class GradeBook Using a Two-Dimensional Array

- WILL BE COVERED IN LAB.

7.11 Variable-Length Argument Lists

- Variable-length argument lists
  - Can be used to create methods that receive an unspecified number of arguments.
  - Parameter type followed by an ellipsis (. . .) indicates that the method receives a variable num-ber of arguments of that particular type.
  - The ellipsis can occur only once at the end of a parameter list.

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average( double... numbers )
8     {
9        double total = 0.0; // initialize total
10
11       // calculate total using the enhanced for statement
12       for ( double d : numbers )
13          total += d;
14
15       return total / numbers.length;
16    } // end method average
17
18    public static void main( String[] args )
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
```

Variable number of `double` values can be passed to this method

Variable arguments are automatically placed in an array referenced by the parameter

Fig. 7.20 | Using variable-length argument lists. (Part 1 of 2.)

```
25         System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26            d1, d2, d3, d4 );
27
28         System.out.printf( "Average of d1 and d2 is %.1f\n",
29            average( d1, d2 ) );
30         System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
31            average( d1, d2, d3 ) );
32         System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33            average( d1, d2, d3, d4 ) );
34      } // end main
35   } // 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. 7.20 | Using variable-length argument lists. (Part 2 of 2.)

Common Programming Error 7.6

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.

7.12 Using Command-Line Arguments

- Command-line arguments
  - Can pass arguments from the command line to an application.
  - Arguments that appear after the class name in the `java` command are received by `main` in the `String` array `args`.
  - The number of command-line arguments is obtained by accessing the array's `length` attribute.
  - Command-line arguments are separated by white space, not commas.
  - Example from Eclipse→Chapter7

```
1  // Fig. 7.21: InitArray.java
2  // Initializing an array using command-line arguments.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // check number of command-line arguments
9        if ( args.length != 3 )
10          System.out.println(
11             "Error: Please re-enter the entire command, including\n" +
12             "an array size, initial value and increment." );
13       else
14       {
15          // get array size from first command-line argument
16          int arrayLength = Integer.parseInt( args[ 0 ] );
17          int[] array = new int[ arrayLength ]; // create array
18
19          // get initial value and increment from command-line arguments
20          int initialValue = Integer.parseInt( args[ 1 ] );
21          int increment = Integer.parseInt( args[ 2 ] );
22
```

Check whether there are 3 command-line arguments

Use first command-line argument as length of array to create

Use second and third command-line arguments as a starting value and increment for the values that will be generated in lines 24–25

Fig. 7.21 | Initializing an array using command-line arguments. (Part 1 of 3.)

```
23            // calculate value for each array element
24            for ( int counter = 0; counter < array.length; counter++ )
25               array[ counter ] = initialValue + increment * counter;
26
27            System.out.printf( "%s%8s\n", "Index", "Value" );
28
29            // display array index and value
30            for ( int counter = 0; counter < array.length; counter++ )
31               System.out.printf( "%5d%8d\n", counter, array[ counter ] );
32         } // end else
33      } // end main
34   } // end class InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 2 of 3.)

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

Three command-line arguments passed to `InitArray`

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Three command-line arguments passed to `InitArray`

Fig. 7.21 | Initializing an array using command-line arguments. (Part 3 of 3.)

7.13 Class Arrays

- `Arrays` class
  - Provides `static` methods for common array manipulations.
- Methods include
  - `sort` for sorting an array (ascending order by default)
  - `binarySearch` for searching a sorted array
  - `equals` for comparing arrays
  - `fill` for placing values into an array.
- Methods are overloaded for primitive-type arrays and for arrays of objects.
- `System` class `static` `arraycopy` method
  - Copies contents of one array into another.
  - public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
- Try the following code in Eclipse and play with it.
- In the next lab, try 3 more functions of the Array class not mentioned in the slides, take the printout of your code and output, hand them to assistants.

```
1  // Fig. 7.22: 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
```

Sorts an array's contents into their default sort order

Fills an array's elements with the value specified as the second argument

Fig. 7.22 | `Arrays` class methods. (Part 1 of 4.)

```
22         // copy array intArray into array intArrayCopy
23         int[] intArray = { 1, 2, 3, 4, 5, 6 };
24         int[] intArrayCopy = new int[ intArray.length ];
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 );
```

Copies elements of the array in the first argument, into the array specified as the third argument

Compares contents of two arrays for equality

Compares contents of two arrays for equality

Fig. 7.22 | `Arrays` class methods. (Part 2 of 4.)

```
45          else
46             System.out.println( "5 not found in intArray" );
47
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
```

Searches for second argument in the array specified as the first argument

Fig. 7.22 | `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. 7.22 | `Arrays` class methods. (Part 4 of 4.)

Common Programming Error 7.7

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

7.14 Introduction to Collections and Class `ArrayList`

- Java API provides several predefined data structures, called collections, used to store groups of related objects.
  - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
  - Reduce application-development time.
- Arrays do not automatically change their size at execution time to accommodate additional elements.
- `ArrayList<T>` (package `java.util`) can dynamically change its size to accommodate more elements.
  - `T` is a placeholder for the type of element stored in the collection.
  - 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.

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

7.14 Introduction to Collections and Class `ArrayList` (Cont.)

- Figure 7.24 demonstrates some common `ArrayList` capabilities.
- An `ArrayList`'s capacity indicates how many items it can hold without growing.
- When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
  - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
  - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

7.14 Introduction to Collections and Class `ArrayList` (Cont.)

- Method `add` adds elements to the `ArrayList`.
  - One-argument version appends its argument to the end of the `ArrayList`.
  - Two-argument version inserts a new element at the specified position.
  - Collection indices start at zero.
- Method `size` returns the number of elements in the `ArrayList`.
- Method `get` obtains the element at a specified index.
- Method `remove` deletes an element with a specific value.
  - An overloaded version of the method removes the element at the specified index.
- Method `contains` determines if an item is in the `ArrayList`.

```
1  // Fig. 7.24: 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
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
```

Creates an `ArrayList` that stores `String` elements

Add elements the the `ArrayList`

Method `size` returns the number of elements in the collection; method `get` returns the element at the specified index

Method `remove` deletes the first occurrence of the specified value

This version of `remove` deletes the element at the specified index

Method `contains` determines whether the specified value is in the collection

```
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
```

Can use the enhanced `for` statement with collections

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