```
            // Calculate midpoint
            middle = (first + last) / 2;
            // If value is found at midpoint...
            if (array[middle] == value)
            {
               found = true;
               position = middle;
            }
            // else if value is in lower half...
            else if (array[middle] > value)
               last = middle - 1;
            // else if value is in upper half....
            else
               first = middle + 1;
         }
         // Return the position of the item, or -1
         // if it was not found.
         return position;
      }
```

The BinarySearchDemo program demonstrates this method. You can download this chapter's source code from the book's companion Web site at www.pearsonhighered.com/gaddis.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

7.18 What value in an array does the selection sort algorithm look for first? When the selection sort finds this value, what does it do with it?

7.19 How many times will the selection sort swap the smallest value in an array with another value?

7.20 Describe the difference between the sequential search and the binary search.

7.21 On average, with an array of 20,000 elements, how many comparisons will the sequential search perform? (Assume the items being searched for are consistently found in the array.)

7.22 If a sequential search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

7.12 Command-Line Arguments and Variable-Length Argument Lists

CONCEPT: When you invoke a Java program from the operating system command line, you can specify arguments that are passed into the main method of the program. In addition, you can write a method that takes a variable number of arguments. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.

Command-Line Arguments

Every program you have seen in this book and every program you have written uses a static `main` method with a header that looks like this:

```
public static void main(String[] args)
```

Inside the parentheses of the method header is the declaration of a parameter named `args`. This parameter is an array name. As its declaration indicates, it is used to reference an array of `Strings`. The array that is passed into the `args` parameter comes from the operating system command line. For example, look at Code Listing 7-21.

Code Listing 7-21 (`CommandLine.java`)

```
/**
   This program displays the arguments passed to
   it from the operating system command line.
*/

public class CommandLine
{
   public static void main(String[] args)
   {
      for (int index = 0; index < args.length; index++)
         System.out.println(args[index]);
   }
}
```

If this program is compiled and then executed with the following command:

```
java CommandLine How does this work?
```

its output will be as follows:

```
How
does
this
work?
```

Any items typed on the command line, separated by spaces, and after the name of the class are considered to be one or more arguments that are to be passed into the `main` method. In the previous example, four arguments are passed into `args`. The word "How" is passed into `args[0]`, "does" is passed into `args[1]`, "this" is passed into `args[2]`, and "work?" is passed into `args[3]`. The `for` loop in `main` simply displays each argument.

NOTE: It is not required that the name of `main`'s parameter array be `args`. You can name it anything you wish. It is a standard convention, however, for the name `args` to be used.

Variable-Length Argument Lists

Java provides a mechanism known as *variable-length argument lists*, which makes it possible to write a method that takes a variable number of arguments. In other words, you can write a method that accepts any number of arguments when it is called. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.

For example, suppose we need to write a method named sum that can accept any number of int values and then return the sum of those values. We might call the method as shown here:

```
result = sum(10, 20);
```

Here we pass two arguments to the method: 10 and 20. After this code executes, the value 30 would be stored in the result variable. But, the method does not have to accept two arguments each time it is called. We could call the method again with a different number of arguments, as shown here:

```
int firstVal = 1, secondVal = 2, thirdVal = 3, fourthVal = 4;
result = sum(firstVal, secondVal, thirdVal, fourthVal);
```

Here we pass four arguments to the method: firstVal (which is set to 1), secondVal (which is set to 2), thirdVal (which is set to 3), and fourthVal (which is set to 4). After this code executes, the value 10 would be stored in the result variable. Here's the code for the sum method:

```
public static int sum(int... numbers)
{
   int total = 0;  // Accumulator

   // Add all the values in the numbers array.
   for (int val : numbers)
      total += val;

   // Return the total.
   return total;
}
```

Notice the declaration of the numbers parameter in the method header. The ellipsis (three periods) that follows the data type indicates that numbers is a special type of parameter known as a *vararg parameter*. A vararg parameter can take a variable number of arguments.

In fact, vararg parameters are actually arrays. In the sum method, the numbers parameter is an array of ints. All of the arguments that are passed to the sum method are stored in the elements of the numbers array. As you can see from the code, the method uses the enhanced for loop to step through the elements of the numbers array, adding up the values stored in its elements. (The *VarargsDemo1.java* program in this chapter's source code demonstrates the sum method.)

You can also write a method to accept a variable number of object references as arguments. For example, the program in Code Listing 7-22 shows a method that accepts a variable number of references to BankAccount objects. The method returns the total of the objects' balance fields.

Code Listing 7-22 (`VarargsDemo2.java`)

```java
/**
   This program demonstrates a method that accepts
   a variable number of arguments (varargs).
*/

public class VarargsDemo2
{
   public static void main(String[] args)
   {
      double total;  // To hold the total balances

      // Create BankAccount object with $100.
      BankAccount account1 = new BankAccount(100.0);

      // Create BankAccount object with $500.
      BankAccount account2 = new BankAccount(500.0);

      // Create BankAccount object with $1500.
      BankAccount account3 = new BankAccount(1500.0);

      // Call the method with one argument.
      total = totalBalance(account1);
      System.out.println("Total: $" + total);

      // Call the method with two arguments.
      total = totalBalance(account1, account2);
      System.out.println("Total: $" + total);

      // Call the method with three arguments.
      total = totalBalance(account1, account2, account3);
      System.out.println("Total: $" + total);
   }

   /**
      The totalBalance method takes a variable number
      of BankAccount objects and returns the total
      of their balances.
      @param accounts The target account or accounts.
      @return The sum of the account balances
   */

   public static double totalBalance(BankAccount... accounts)
   {
      double total = 0.0;  // Accumulator

      // Add all the values in the accounts array.
      for (BankAccount acctObject : accounts)
         total += acctObject.getBalance();

      // Return the total.
      return total;
   }
}
```

Program Output

```
Total: $100.0
Total: $600.0
Total: $2100.0
```

You can write a method to accept a mixture of fixed arguments and a variable-length argument list. For example, suppose we want to write a method named `courseAverage`, which accepts the name of a course as a `String`, and a variable-length list of test scores as `double`s. We could write the method header as follows:

```
public static void courseAverage(String course, double... scores)
```

This method has a regular `String` parameter named `course`, and a vararg parameter named `scores`. When we call this method, we always pass a `String` argument, then a list of `double` values. (This method is demonstrated in the program *VarargsDemo3.java*, which is in this chapter's source code folder.) Note that when a method accepts a mixture of fixed arguments and a variable-length argument list, the vararg parameter must be the last one declared.

You can also pass an array to a vararg parameter. This is demonstrated in the program *VarargsDemo4.java*. You can download this chapter's source code from the book's companion Web site at www.pearsonhighered.com/gaddis.

7.13 The `ArrayList` Class

CONCEPT: `ArrayList` is a class in the Java API that is similar to an array and allows you to store objects. Unlike an array, an `ArrayList` object's size is automatically adjusted to accommodate the number of items being stored in it.

The Java API provides a class named `ArrayList`, which can be used for storing and retrieving objects. Once you create an `ArrayList` object, you can think of it as a container for holding other objects. An `ArrayList` object is similar to an array of objects, but offers many advantages over an array. Here are a few:

- An `ArrayList` object automatically expands as items are added to it.
- In addition to adding items to an `ArrayList`, you can remove items as well.
- An `ArrayList` object automatically shrinks as items are removed from it.

The ArrayList class is in the java.util package, so the following import statement is required:

```
import java.util.ArrayList;
```

Creating and Using an ArrayList Object

Here is an example of how you create an ArrayList object:

```
ArrayList<String> nameList = new ArrayList<String>();
```

This statement creates a new ArrayList object and stores its address in the nameList variable. Notice that in this example the word String is written inside angled brackets <> immediately after the word ArrayList. This specifies that the ArrayList can hold String objects. If we try to store any other type of object in this ArrayList, an error will occur. (Later in this section, you will see an example that creates an ArrayList for holding other types of objects.)

To add items to the ArrayList object, you use the add method. For example, the following statements add a series of String objects to nameList:

```
nameList.add("James");
nameList.add("Catherine");
nameList.add("Bill");
```

After these statements execute, nameList will hold three references to String objects. The first will reference "James", the second will reference "Catherine", and the third will reference "Bill".

The items that are stored in an ArrayList have a corresponding index. The index specifies the item's location in the ArrayList, so it is much like an array subscript. The first item that is added to an ArrayList is stored at index 0. The next item that is added to the ArrayList is stored at index 1, and so forth. After the previously shown statements execute, "James" will be stored at index 0, "Catherine" will be stored at index 1, and "Bill" will be stored at index 2.

The ArrayList class has a size method that reports the number of items stored in an ArrayList. It returns the number of items as an int. For example, the following statement uses the method to display the number of items stored in nameList:

```
System.out.println("The ArrayList has " +
                   nameList.size() +
                   " objects stored in it.");
```

Assuming that nameList holds the Strings "James", "Catherine", and "Bill", the following statement will display:

```
The ArrayList has 3 objects stored in it.
```

The ArrayList class's get method returns the item stored at a specific index. You pass the index as an argument to the method. For example, the following statement will display the item stored at index 1 of nameList:

```
System.out.println(nameList.get(1));
```

The program in Code Listing 7-23 demonstrates the topics discussed so far.

Code Listing 7-23 **(ArrayListDemo1.java)**

```
 1  import java.util.ArrayList; // Needed for ArrayList class
 2
 3  /**
 4     This program demonstrates an ArrayList.
 5  */
 6
 7  public class ArrayListDemo1
 8  {
 9     public static void main(String[] args)
10     {
11        // Create an ArrayList to hold some names.
12        ArrayList<String> nameList = new ArrayList<String>();
13
14        // Add some names to the ArrayList.
15        nameList.add("James");
16        nameList.add("Catherine");
17        nameList.add("Bill");
18
19        // Display the size of the ArrayList.
20        System.out.println("The ArrayList has " +
21                           nameList.size() +
22                           " objects stored in it.");
23
24        // Now display the items in nameList.
25        for (int index = 0; index < nameList.size(); index++)
26           System.out.println(nameList.get(index));
27     }
28  }
```

Program Output

```
The ArrayList has 3 objects stored in it.
James
Catherine
Bill
```

Notice in line 25 that the `for` loop uses the value returned from `nameList`'s `size` method to control the number of times the loop iterates. This is to prevent a bounds checking error from occurring. The last item stored in an `ArrayList` will have an index that is 1 less than the `size` of the `ArrayList`. If you pass a value larger than this to the `get` method, an error will occur.

Using the Enhanced `for` Loop with an `ArrayList`

Earlier in this chapter, you saw how the enhanced `for` loop can be used to iterate over each `element` in an array. You can also use the enhanced `for` loop to iterate over each item in an

ArrayList. Code Listing 7-24 demonstrates. The enhanced for loop is used in lines 26 and 27 to display all of the items stored in the ArrayList.

Code Listing 7-24 (ArrayListDemo2.java)

```
 1 import java.util.ArrayList; // Needed for ArrayList class
 2
 3 /**
 4    This program demonstrates how the enhanced for loop
 5    can be used with an ArrayList.
 6 */
 7
 8 public class ArrayListDemo2
 9 {
10    public static void main(String[] args)
11    {
12       // Create an ArrayList to hold some names.
13       ArrayList<String> nameList = new ArrayList<String>();
14
15       // Add some names to the ArrayList.
16       nameList.add("James");
17       nameList.add("Catherine");
18       nameList.add("Bill");
19
20       // Display the size of the ArrayList.
21       System.out.println("The ArrayList has " +
22                          nameList.size() +
23                          " objects stored in it.");
24
25       // Now display the items in nameList.
26       for (String name : nameList)
27          System.out.println(name);
28    }
29 }
```

Program Output

```
The ArrayList has 3 objects stored in it.
James
Catherine
Bill
```

The ArrayList Class's toString method

The ArrayList class has a toString method that returns a string representing all of the items stored in an ArrayList object. For example, suppose we have set up the nameList object as previously shown, with the Strings "James", "Catherine", and "Bill". We could use the following statement to display all of the names:

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

The contents of the ArrayList will be displayed in the following manner:

```
[James, Catherine, Bill]
```

This is demonstrated in the program *ArrayListToString.java*, which is in this chapter's source code folder, available at www.pearsonhighered.com/gaddis.

Removing an Item from an ArrayList

The ArrayList class has a remove method that removes an item at a specific index. You pass the index as an argument to the method. The program in Code Listing 7-25 demonstrates.

Code Listing 7-25 (ArrayListDemo3.java)

```
import java.util.ArrayList; // Needed for ArrayList class

/**
   This program demonstrates an ArrayList.
*/

public class ArrayListDemo3
{
   public static void main(String[] args)
   {
      // Create an ArrayList to hold some names.
      ArrayList<String> nameList = new ArrayList<String>();

      // Add some names to the ArrayList.
      nameList.add("James");
      nameList.add("Catherine");
      nameList.add("Bill");

      // Display the items in nameList and their indices.
      for (int index = 0; index < nameList.size(); index++)
      {
         System.out.println("Index: " + index + " Name: " +
                            nameList.get(index));
      }

      // Now remove the item at index 1.
      nameList.remove(1);

      System.out.println("The item at index 1 is removed. " +
                         "Here are the items now.");

      // Display the items in nameList and their indices.
      for (int index = 0; index < nameList.size(); index++)
      {
         System.out.println("Index: " + index + " Name: " +
                            nameList.get(index));
      }
   }
}
```

Program Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
The item at index 1 is removed. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Bill
```

Note that when the item at index 1 was removed (in line 27), the item that was previously stored at index 2 was shifted in position to index 1. When an item is removed from an `ArrayList`, the items that come after it are shifted downward in position to fill the empty space. This means that the index of each item after the removed item will be decreased by one.

Note that an error will occur if you call the `remove` method with an invalid index.

Inserting an Item

The `add` method, as previously shown, adds an item at the last position in an `ArrayList` object. The `ArrayList` class has an overloaded version of the `add` method that allows you to add an item at a specific index. This causes the item to be inserted into the `ArrayList` object at a specific position. The program in Code Listing 7-26 demonstrates.

Code Listing 7-26 **(ArrayListDemo4.java)**

```
import java.util.ArrayList; // Needed for ArrayList class

/**
   This program demonstrates inserting an item.
*/

public class ArrayListDemo4
{
   public static void main(String[] args)
   {
      // Create an ArrayList to hold some names.
      ArrayList<String> nameList = new ArrayList<String>();

      // Add some names to the ArrayList.
      nameList.add("James");
      nameList.add("Catherine");
17          nameList.add("Bill");
18
19          // Display the items in nameList and their indices.
20          for (int index = 0; index < nameList.size(); index++)
21          {
22             System.out.println("Index: " + index + " Name: " +
23                                nameList.get(index));
24          }
25
26          // Now insert an item at index 1.
27          nameList.add(1, "Mary");
28
29          System.out.println("Mary was added at index 1. " +
30                             "Here are the items now.");
31
32          // Display the items in nameList and their indices.
33          for (int index = 0; index < nameList.size(); index++)
34          {
35             System.out.println("Index: " + index + " Name: " +
36                                nameList.get(index));
37          }
38       }
39    }
```

Program Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
Mary was added at index 1. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Mary
Index: 2 Name: Catherine
Index: 3 Name: Bill
```

Note that when a new item was added at index 1 (in line 27), the item that was previously stored at index 1 was shifted in position to index 2. When an item is added at a specific index, the items that come after it are shifted upward in position to accommodate the new item. This means that the index of each item after the new item will be increased by one.

Note that an error will occur if you call the `add` method with an invalid index.

Replacing an Item

The `ArrayList` class's `set` method can be used to replace an item at a specific index with another item. For example, the following statement will replace the item currently at index 1 with the String "Becky":

```
nameList.set(1, "Becky");
```

This is demonstrated in the program *ArrayListDemo5.java*, which is in this chapter's source code folder, available at www.pearsonhighered.com/gaddis. Note that an error will occur if you specify an invalid index.

Capacity

Previously you learned that an ArrayList object's size is the number of items stored in the ArrayList object. When you add an item to the ArrayList object, its size increases by one, and when you remove an item from the ArrayList object, its size decreases by one.

An ArrayList object also has a *capacity*, which is the number of items it can store without having to increase its size. When an ArrayList object is first created, using the no-arg constructor, it has an initial capacity of 10 items. This means that it can hold up to 10 items without having to increase its size. When the eleventh item is added, the ArrayList object must increase its size to accommodate the new item. You can specify a different starting capacity, if you desire, by passing an int argument to the ArrayList constructor. For example, the following statement creates an ArrayList object with an initial capacity of 100 items:

```
ArrayList<String> list = new ArrayList<String>(100);
```

All of the examples we have looked at so far use ArrayList objects to hold Strings. You can create an ArrayList to hold any type of object. For example, the following statement creates an ArrayList that can hold BankAccount objects:

```
ArrayList<BankAccount> accountList = new ArrayList<BankAccount>();
```

By specifying BankAccount inside the angled brackets, we are declaring that the ArrayList can hold only BankAccount objects. Code Listing 7-27 demonstrates such an ArrayList.

Code Listing 7-27 (ArrayListDemo6.java)

```
import java.util.ArrayList; // Needed for ArrayList class

/**
   This program demonstrates how to store BankAccount
   objects in an ArrayList.
*/

public class ArrayListDemo6
{
   public static void main(String[] args)
   {
      // Create an ArrayList to hold BankAccount objects.
      ArrayList<BankAccount> list = new ArrayList<BankAccount>();

      // Add three BankAccount objects to the ArrayList.
      list.add(new BankAccount(100.0));
      list.add(new BankAccount(500.0));
      list.add(new BankAccount(1500.0));

      // Display each item.
      for (int index = 0; index < list.size(); index++)
      {
         BankAccount account = list.get(index);
         System.out.println("Account at index " + index +
                            "\nBalance: " + account.getBalance());
      }
   }
}
```

Program Output

```
Account at index 0
Balance: 100.0
Account at index 1
Balance: 500.0
Account at index 2
Balance: 1500.0
```

Using the Diamond Operator for Type Inference (Java 7)

Beginning with Java 7, you can simplify the instantiation of an ArrayList by using the *diamond operator* (<>). For example, in this chapter you have seen several programs that create an ArrayList object with a statement such as this:

```
ArrayList<String> list = new ArrayList<String>();
```

Notice that the data type (in this case, String) appears between the angled brackets in two locations: first in the part that declares the reference variable, and then again in the part that calls the ArrayList constructor. Beginning with Java 7, you are no longer required to write the data type in the part of the statement that calls the ArrayList constructor. Instead, you can simply write a set of empty angled brackets, as shown here:

```
ArrayList<String> list = new ArrayList<>();
```

This set of empty angled brackets (<>) is called the diamond operator. It causes the compiler to infer the required data type from the reference variable declaration. Here is another example:

```
ArrayList<InventoryItem> list = new ArrayList<>();
```

This creates an ArrayList that can hold InventoryItem objects. Keep in mind that type inference was introduced in Java 7. If you are using an earlier version of the Java language, you will have to use the more lengthy form of the declaration statement to create an ArrayList.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

7.23 What import statement must you include in your code in order to use the ArrayList class?

7.24 Write a statement that creates an ArrayList object and assigns its address to a variable named frogs.

7.25 Write a statement that creates an `ArrayList` object and assigns its address to a variable named `lizards`. The `ArrayList` should be able to store `String` objects only.

7.26 How do you add items to an `ArrayList` object?

7.27 How do you remove an item from an `ArrayList` object?

7.28 How do you retrieve a specific item from an `ArrayList` object?

7.29 How do you insert an item at a specific location in an `ArrayList` object?

7.30 How do you determine an `ArrayList` object's size?

7.31 What is the difference between an `ArrayList` object's size and its capacity?

7.14 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Using an invalid subscript.** Java does not allow you to use a subscript value that is outside the range of valid subscripts for an array.
- **Confusing the contents of an integer array element with the element's subscript.** An element's subscript and the value stored in the element are not the same thing. The subscript identifies an element, which holds a value.
- **Causing an off-by-one error.** When processing arrays, the subscripts start at zero and end at one less than the number of elements in the array. Off-by-one errors are commonly caused when a loop uses an initial subscript of one and/or uses a maximum subscript that is equal to the number of elements in the array.
- **Using the = operator to copy an array.** Assigning one array reference variable to another with the = operator merely copies the address in one variable to the other. To copy an array, you should copy the individual elements of one array to another.
- **Using the == operator to compare two arrays.** You cannot use the == operator to compare two array reference variables and determine whether the arrays are equal. When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables.
- **Reversing the row and column subscripts when processing a two-dimensional array.** When thinking of a two-dimensional array as having rows and columns, the first subscript accesses a row and the second subscript accesses a column. If you reverse these subscripts, you will access the wrong element.

Review Questions and Exercises

Multiple Choice and True/False

1. In an array declaration, this indicates the number of elements that the array will have.
   a. subscript
   b. size declarator
   c. element sum
   d. reference variable

2. Each element of an array is accessed by a number known as a(n) __________.
   a. subscript
   b. size declarator
   c. address
   d. specifier
3. The first subscript in an array is always __________.
   a. 1
   b. 0
   c. −1
   d. 1 less than the number of elements
4. The last subscript in an array is always __________.
   a. 100
   b. 0
   c. −1
   d. 1 less than the number of elements
5. Array bounds checking happens __________.
   a. when the program is compiled
   b. when the program is saved
   c. when the program runs
   d. when the program is loaded into memory
6. This array field holds the number of elements that the array has.
   a. `size`
   b. `elements`
   c. `length`
   d. `width`
7. This search algorithm steps through an array, comparing each item with the search value.
   a. binary search
   b. sequential search
   c. selection search
   d. iterative search
8. This search algorithm repeatedly divides the portion of an array being searched in half.
   a. binary search
   b. sequential search
   c. selection search
   d. iterative search
9. This is the typical number of comparisons performed by the sequential search on an array of N elements (assuming the search values are consistently found).
   a. 2N
   b. N
   c. N^2
   d. N/2

10. When initializing a two-dimensional array, you enclose each row's initialization list in ___________.
    a. braces
    b. parentheses
    c. brackets
    d. quotation marks
11. To insert an item at a specific location in an `ArrayList` object, you use this method.
    a. `store`
    b. `insert`
    c. `add`
    d. `get`
12. To delete an item from an `ArrayList` object, you use this method.
    a. `remove`
    b. `delete`
    c. `erase`
    d. `get`
13. To determine the number of items stored in an `ArrayList` object, you use this method.
    a. `size`
    b. `capacity`
    c. `items`
    d. `length`
14. **True or False:** Java does not allow a statement to use a subscript that is outside the range of valid subscripts for an array.
15. **True or False:** An array's sitze declarator can be a negative integer expression.
16. **True or False:** Both of the following declarations are legal and equivalent:
    ```
    int[] numbers;
    int numbers[];
    ```
17. **True or False:** The subscript of the last element in a single-dimensional array is one less than the total number of elements in the array.
18. **True or False:** The values in an initialization list are stored in the array in the order that they appear in the list.
19. **True or False:** The Java compiler does not display an error message when it processes a statement that uses an invalid subscript.
20. **True or False:** When an array is passed to a method, the method has access to the original array.
21. **True or False:** The first size declarator in the declaration of a two-dimensional array represents the number of columns. The second size declarator represents the number of rows.
22. **True or False:** A two-dimensional array has multiple `length` fields.
23. **True or False:** An `ArrayList` automatically expands in size to accommodate the items stored in it.

Find the Error

1.
   ```
   int[] collection = new int[-20];
   ```

2.
   ```
   int[] hours = 8, 12, 16;
   ```

3.
   ```
   int[] table = new int[10];
   for (int x = 1; x <= 10; x++)
   {
      table[x] = 99;
   }
   ```

4.
   ```
   String[] names = { "George", "Susan" };
   int totalLength = 0;
   for (int i = 0; i < names.length(); i++)
      totalLength += names[i].length;
   ```

5.
   ```
   String[] words = { "Hello", "Goodbye" };
   System.out.println(words.toUpperCase());
   ```

Algorithm Workbench

1. The variable `names` references an integer array with 20 elements. Write a `for` loop that prints each element of the array.
2. The variables `numberArray1` and `numberArray2` reference arrays that each have 100 elements. Write code that copies the values in `numberArray1` to `numberArray2`.
3. a. Write a statement that declares a `String` array initialized with the following strings:
   "Einstein", "Newton", "Copernicus", and "Kepler".
   b. Write a loop that displays the contents of each element in the array that you declared in Question 3(a).
   c. Write code that displays the total length of all the strings in the array that you declared in Question 3(a).
4. In a program you need to store the populations of 12 countries.
   a. Define two arrays that may be used in parallel to store the names of the countries and their populations.
   b. Write a loop that uses these arrays to print each country's name and its population.
5. In a program you need to store the identification numbers of ten employees (as `int` values) and their weekly gross pay (as `double` values).
   a. Define two arrays that may be used in parallel to store the 10 employee identification numbers and gross pay amounts.
   b. Write a loop that uses these arrays to print each of the employees' identification number and weekly gross pay.
6. Declare a two-dimensional `int` array named `grades`. It should have 30 rows and 10 columns.
7. Write code that calculates the average of all the elements in the `grades` array that you declared in Question 6.

8. Look at the following array declaration:

```
int[][] numberArray = new int[9][11];
```

   a. Write a statement that assigns 145 to the first column of the first row of this array.
   b. Write a statement that assigns 18 to the last column of the last row of this array.

9. The `values` variable references a two-dimensional double array with 10 rows and 20 columns. Write code that sums all the elements in the array and stores the sum in the variable `total`.

10. An application uses a two-dimensional array declared as follows:

```
int[][] days = new int[29][5];
```

   a. Write code that sums each row in the array and displays the results.
   b. Write code that sums each column in the array and displays the results.

11. Write code that creates an `ArrayList` that can hold `String` objects. Add the names of three cars to the `ArrayList`, and then display the contents of the `ArrayList`.

Short Answer

1. What is the difference between a size declarator and a subscript?

2. Look at the following array definition:

```
int[] values = new int[10];
```

   a. How many elements does the array have?
   b. What is the subscript of the first element in the array?
   c. What is the subscript of the last element in the array?

3. Look at the following array definition:

```
int[] values = { 4, 7, 6, 8, 2 };
```

What does each of the following code segments display?

```
System.out.println(values[4]);          a. ______________________

x = values[2] + values[3];
System.out.println(x);                  b. ______________________

x = ++values[1];
System.out.println(x);                  c. ______________________
```

4. How do you define an array without providing a size declarator?

5. Assuming that `array1` and `array2` are both array reference variables, why is it not possible to assign the contents of the array referenced by `array2` to the array referenced by `array1` with the following statement?

```
array1 = array2;
```

6. How do you establish an array without providing a size declarator?

7. The following statement creates a `BankAccount` array:

```
BankAccount[] acc = new BankAccount[10];
```

Is it okay or not okay to execute the following statements?

```
acc[0].setBalance(5000.0);
acc[0].withdraw(100.0);
```

8. If a sequential search method is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to read to locate the value?
9. Look at the following array definition:

```
double[][] sales = new double[8][10];
```

   a. How many rows does the array have?
   b. How many columns does the array have?
   c. How many elements does the array have?
   d. Write a statement that stores a number in the last column of the last row in the array.

Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

1. Rainfall Class

Write a `RainFall` class that stores the total rainfall for each of 12 months into an array of `doubles`. The program should have methods that return the following:

- the total rainfall for the year
- the average monthly rainfall
- the month with the most rain
- the month with the least rain

Demonstrate the class in a complete program.

Input Validation: Do not accept negative numbers for monthly rainfall figures.

2. Payroll Class

Write a `Payroll` class that uses the following arrays as fields:

- **`employeeId`**. An array of seven integers to hold employee identification numbers. The array should be initialized with the following numbers:

  ```
  5658845 4520125 7895122 8777541
  8451277 1302850 7580489
  ```

- **`hours`**. An array of seven integers to hold the number of hours worked by each employee
- **`payRate`**. An array of seven `doubles` to hold each employee's hourly pay rate
- **`wages`**. An array of seven `doubles` to hold each employee's gross wages

The class should relate the data in each array through the subscripts. For example, the number in element 0 of the `hours` array should be the number of hours worked by the employee whose identification number is stored in element 0 of the `employeeId` array. That same employee's pay rate should be stored in element 0 of the `payRate` array.

In addition to the appropriate accessor and mutator methods, the class should have a method that accepts an employee's identification number as an argument and returns the gross pay for that employee.

Demonstrate the class in a complete program that displays each employee number and asks the user to enter that employee's hours and pay rate. It should then display each employee's identification number and gross wages.

Input Validation: Do not accept negative values for hours or numbers less than 6.00 for pay rate.

3. Charge Account Validation

Create a class with a method that accepts a charge account number as its argument. The method should determine whether the number is valid by comparing it to the following list of valid charge account numbers:

`5658845`	`4520125`	`7895122`	`8777541`	`8451277`	`1302850`
`8080152`	`4562555`	`5552012`	`5050552`	`7825877`	`1250255`
`1005231`	`6545231`	`3852085`	`7576651`	`7881200`	`4581002`

VideoNote The Charge Account Validation Problem

These numbers should be stored in an array or an `ArrayList` object. Use a sequential search to locate the number passed as an argument. If the number is in the array, the method should return `true`, indicating the number is valid. If the number is not in the array, the method should return `false`, indicating the number is invalid.

Write a program that tests the class by asking the user to enter a charge account number. The program should display a message indicating whether the number is valid or invalid.

4. Charge Account Modification

Modify the charge account validation class that you wrote for Programming Challenge 3 so it reads the list of valid charge account numbers from a file. Use Notepad or another text editor to create the file.

5. Driver's License Exam

The local Driver's License Office has asked you to write a program that grades the written portion of the driver's license exam. The exam has 20 multiple choice questions. Here are the correct answers:

1. B	6. A	11. B	16. C
2. D	7. B	12. C	17. C
3. A	8. A	13. D	18. B
4. A	9. C	14. A	19. D
5. C	10. D	15. D	20. A

A student must correctly answer 15 of the 20 questions to pass the exam.

Write a class named `DriverExam` that holds the correct answers to the exam in an array field. The class should also have an array field that holds the student's answers. The class should have the following methods:

- `passed`. Returns true if the student passed the exam, or false if the student failed
- `totalCorrect`. Returns the total number of correctly answered questions
- `totalIncorrect`. Returns the total number of incorrectly answered questions
- `questionsMissed`. An `int` array containing the question numbers of the questions that the student missed

Demonstrate the class in a complete program that asks the user to enter a student's answers, and then displays the results returned from the `DriverExam` class's methods.

Input Validation: Only accept the letters A, B, C, or D as answers.

6. Quarterly Sales Statistics

Write a program that lets the user enter four quarterly sales figures for six divisions of a company. The figures should be stored in a two-dimensional array. Once the figures are entered, the program should display the following data for each quarter:

- A list of the sales figures by division
- Each division's increase or decrease from the previous quarter (this will not be displayed for the first quarter)
- The total sales for the quarter
- The company's increase or decrease from the previous quarter (this will not be displayed for the first quarter)
- The average sales for all divisions that quarter
- The division with the highest sales for that quarter

Input Validation: Do not accept negative numbers for sales figures.

7. Grade Book

A teacher has five students who have taken four tests. The teacher uses the following grading scale to assign a letter grade to a student, based on the average of his or her four test scores:

Test Score	Letter Grade
90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Write a class that uses a `String` array or an `ArrayList` object to hold the five students' names, an array of five characters to hold the five students' letter grades, and five arrays of four `doubles` each to hold each student's set of test scores. The class should have methods that return a specific student's name, the average test score, and a letter grade based on the average.

Demonstrate the class in a program that allows the user to enter each student's name and his or her four test scores. It should then display each student's average test score and letter grade.

Input Validation: Do not accept test scores less than zero or greater than 100.

8. Grade Book Modification

Modify the grade book application in Programming Challenge 7 so that it drops each student's lowest score when determining the test score averages and letter grades.

9. Lottery Application

Write a `Lottery` class that simulates a lottery. The class should have an array of five integers named `lotteryNumbers`. The constructor should use the `Random` class (from the Java API) to generate a random number in the range of 0 through 9 for each element in the array. The class should also have a method that accepts an array of five integers that represent a person's lottery picks. The method is to compare the corresponding elements in the two arrays and return the number of digits that match. For example, the following shows the `lotteryNumbers` array and the user's array with sample numbers stored in each. There are two matching digits (elements 2 and 4).

`lotteryNumbers` array:

7	4	9	1	3

User's array:

4	2	9	7	3

In addition, the class should have a method that returns a copy of the `lotteryNumbers` array.

Demonstrate the class in a program that asks the user to enter five numbers. The program should display the number of digits that match the randomly generated lottery numbers. If all of the digits match, display a message proclaiming the user a grand prize winner.

10. Array Operations

Write a program with an array that is initialized with test data. Use any primitive data type of your choice. The program should also have the following methods:

- `getTotal`. This method should accept a one-dimensional array as its argument and return the total of the values in the array.
- `getAverage`. This method should accept a one-dimensional array as its argument and return the average of the values in the array.
- `getHighest`. This method should accept a one-dimensional array as its argument and return the highest value in the array.
- `getLowest`. This method should accept a one-dimensional array as its argument and return the lowest value in the array.

Demonstrate each of the methods in the program.

11. Number Analysis Class

Write a class with a constructor that accepts a file name as its argument. Assume the file contains a series of numbers, each written on a separate line. The class should read the contents of the file into an array, and then displays the following data:

- The lowest number in the array
- The highest number in the array
- The total of the numbers in the array
- The average of the numbers in the array

This chapter's source code folder, available at www.pearsonhighered.com/gaddis, contains a text file named *Numbers.txt*. This file contains twelve random numbers. Write a program that tests the class by using this file.

12. Name Search

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find the following files in the *Chapter 07* folder:

- *GirlNames.txt* – This file contains a list of the 200 most popular names given to girls born in the United States for the years 2000 through 2009.
- *BoyNames.txt* – This file contains a list of the 200 most popular names given to boys born in the United States for the years 2000 through 2009.

Write a program that reads the contents of the two files into two separate arrays, or `ArrayList`s. The user should be able to enter a boy's name, a girl's name, or both, and the application will display messages indicating whether the names were among the most popular.

13. Population Data

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find a file named *USPopulation.txt* in the *Chapter 07* folder. The file contains the midyear population of the United States, in thousands, during the years 1950 through 1990. The first line in the file contains the population for 1950, the second line contains the population for 1951, and so forth.

Write a program that reads the file's contents into an array. The program should display the following data:

- The average annual change in population during the time period
- The year with the greatest increase in population during the time period
- The year with the smallest increase in population during the time period

14. World Series Champions

If you have downloaded this book's source code (the companion Web site is available at www.pearsonhighered.com/gaddis), you will find a file named *WorldSeriesWinners.txt*. This file contains a chronological list of the winning teams in the World Series from 1903 through 2009. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2009. Note that the World Series was not played in 1904 or 1994, so those years are skipped in the file.)

Write a program that lets the user enter the name of a team, and then displays the number of times that team has won the World Series in the time period from 1903 through 2009.

TIP: Read the contents of the *WorldSeriesWinners.txt* file into an array, or an `ArrayList`. When the user enters the name of a team, the program should step through the array or `ArrayList`, counting the number of times the selected team appears.

15. 2D Array Operations

Write a program that creates a two-dimensional array initialized with test data. Use any primitive data type that you wish. The program should have the following methods:

- `getTotal`. This method should accept a two-dimensional array as its argument and return the total of all the values in the array.
- `getAverage`. This method should accept a two-dimensional array as its argument and return the average of all the values in the array.
- `getRowTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the total of the values in the specified row.
- `getColumnTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a column in the array. The method should return the total of the values in the specified column.
- `getHighestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the highest value in the specified row of the array.
- `getLowestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the lowest value in the specified row of the array.

Demonstrate each of the methods in this program.

16. Phone Book `ArrayList`

Write a class named `PhoneBookEntry` that has fields for a person's name and phone number. The class should have a constructor and appropriate accessor and mutator methods. Then write a program that creates at least five `PhoneBookEntry` objects and stores them in an `ArrayList`. Use a loop to display the contents of each object in the `ArrayList`.

17. Trivia Game

In this programming challenge, you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering 5 trivia questions. (There are 10 questions, 5 for each player.) When a question is displayed, four possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer, he or she earns a point.
- After answers have been selected for all of the questions, the program displays the number of points earned by each player and declares the player with the highest number of points the winner.

You are to design a `Question` class to hold the data for a trivia question. The `Question` class should have `String` fields for the following data:

- A trivia question
- Possible answer 1
- Possible answer 2
- Possible answer 3
- Possible answer 4
- The number of the correct answer (1, 2, 3, or 4)

The `Question` class should have appropriate constructor(s), accessor, and mutator methods.

The program should create an array of 10 `Question` objects, one for each trivia question. (If you prefer, you can use an `ArrayList` instead of an array.) Make up your own trivia questions on the subject or subjects of your choice for the objects.

CHAPTER 8

A Second Look at Classes and Objects

TOPICS

8.1 Static Class Members

CONCEPT: **A static class member belongs to the class, not objects instantiated from the class.**

A Quick Review of Instance Fields and Instance Methods

Recall from Chapter 6 that each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class and store different values in each instance's fields. For example, the `Rectangle` class that we created in Chapter 6 has a `length` field and a `width` field. Let's say that `box` references an instance of the `Rectangle` class and we execute the following statement:

```
box.setLength(10);
```

This statement stores the value 10 in the `length` field that belongs to the instance that is referenced by `box`. You can think of instance fields as belonging to a specific instance of a class.

You will also recall that classes may have instance methods as well. When you call an instance method, it performs an operation on a specific instance of the class. For example, assuming that box references an instance of the Rectangle class, look at the following statement:

```
x = box.getLength();
```

This statement calls the getLength method, which returns the value of the length field that belongs to a specific instance of the Rectangle class: the one referenced by box. Both instance fields and instance methods are associated with a specific instance of a class, and they cannot be used until an instance of the class is created.

Static Members

It is possible to create a field or method that does not belong to any instance of a class. Such members are known as static fields and static methods. When a value is stored in a static field, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist in order for values to be stored in the class's static fields. Likewise, static methods do not operate on the fields that belong to any instance of the class. Instead, they can operate only on static fields. You can think of static fields and static methods as belonging to the class instead of an instance of the class. In this section, we will take a closer look at static members. First we will examine static fields.

Static Fields

When a field is declared with the key word static, there will be only one copy of the field in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static field is shared by all instances of the class. For example, the Countable class shown in Code Listing 8-1 uses a static field to keep count of the number of instances of the class that are created.

Code Listing 8-1 (Countable.java)

```
/**
   This class demonstrates a static field.
*/

public class Countable
{
   private static int instanceCount = 0;

   /**
      The constructor increments the static
      field instanceCount. This keeps track
      of the number of instances of this
      class that are created.
   */

   public Countable()
   {
```

```
18          instanceCount++;
19       }
20
21       /**
22          The getInstanceCount method returns
23          the number of instances of this class
24          that have been created.
25          @return The value in the instanceCount field.
26       */
27
28       public int getInstanceCount()
29       {
30          return instanceCount;
31       }
32    }
```

First, notice in line 7 the declaration of the static field named `instanceCount` as follows:

```
private static int instanceCount = 0;
```

A static field is created by placing the key word `static` after the access specifier and before the field's data type. Notice that we have explicitly initialized the `instanceCount` field with the value 0. This initialization takes place only once, regardless of the number of instances of the class that are created.

NOTE: Java automatically stores 0 in all uninitialized static member variables. The `instanceCount` field in this class is explicitly initialized so it is clear to anyone reading the code that the field starts with the value 0.

Next, look at the constructor in lines 16 through 19. The constructor uses the `++` operator to increment the `instanceCount` field. Each time an instance of the `Countable` class is created, the constructor will be called and the `instanceCount` field will be incremented. As a result, the `instanceCount` field will contain the number of instances of the `Countable` class that have been created. The `getInstanceCount` method, in lines 28 through 31, returns the value in `instanceCount`. The program in Code Listing 8-2 demonstrates this class.

Code Listing 8-2 **(StaticDemo.java)**

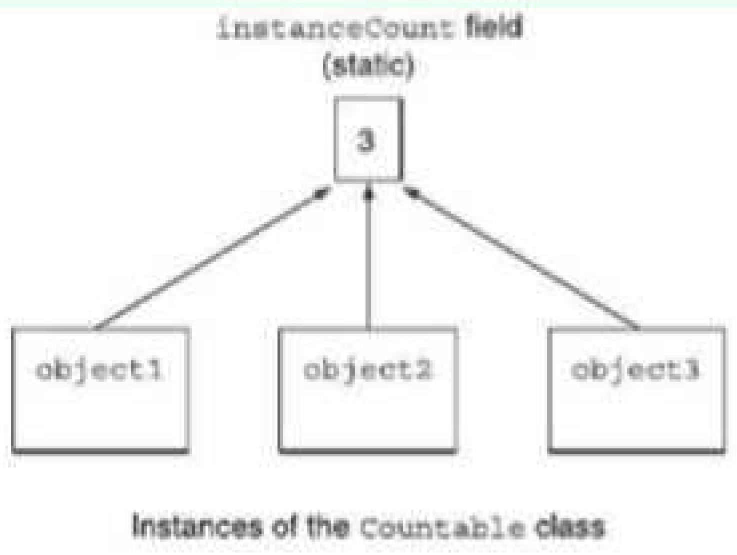
```
/**
   This program demonstrates the Countable class.
*/

public class StaticDemo
{
   public static void main(String[] args)
   {
      int objectCount;
10
11          // Create three instances of the
12          // Countable class.
13          Countable object1 = new Countable();
14          Countable object2 = new Countable();
15          Countable object3 = new Countable();
16
17          // Get the number of instances from
18          // the class's static field.
19          objectCount = object1.getInstanceCount();
20          System.out.println(objectCount +
21                         " instances of the class " +
22                         "were created.");
23       }
24    }
```

Program Output

```
3 instances of the class were created.
```

The program creates three instances of the `Countable` class, referenced by the variables `object1`, `object2`, and `object3`. Although there are three instances of the class, there is only one copy of the static field. This is illustrated in Figure 8-1.

Figure 8-1 All instances of the class share the static field



In line 19 the program calls the `getInstanceCount` method to retrieve the number of instances that have been created:

```
objectCount = object1.getInstanceCount();
```

Although the program calls the `getInstanceCount` method from `object1`, the same value would be returned from any of the objects.

Static Methods

When a class contains a static method, it isn't necessary for an instance of the class to be created in order to execute the method. The program in Code Listing 8-3 shows an example of a class with static methods.

Code Listing 8-3 (`Metric.java`)

```java
/**
   This class demonstrates static methods.
*/

public class Metric
{
   /**
      The milesToKilometers method converts a
      distance in miles to kilometers.
      @param m The distance in miles.
      @return The distance in kilometers.
   */

   public static double milesToKilometers(double m)
   {
      return m * 1.609;
   }

   /**
      The kilometersToMiles method converts
      a distance in kilometers to miles.
      @param k The distance in kilometers.
      @return The distance in miles.
   */

   public static double kilometersToMiles(double k)
   {
      return k / 1.609;
   }
}
```

A static method is created by placing the key word `static` after the access specifier in the method header. The `Metric` class has two static methods: `milesToKilometers` and `kilometersToMiles`. Because they are declared as `static`, they belong to the class and may be called without any instances of the class being in existence. You simply write the name of the class before the dot operator in the method call. Here is an example:

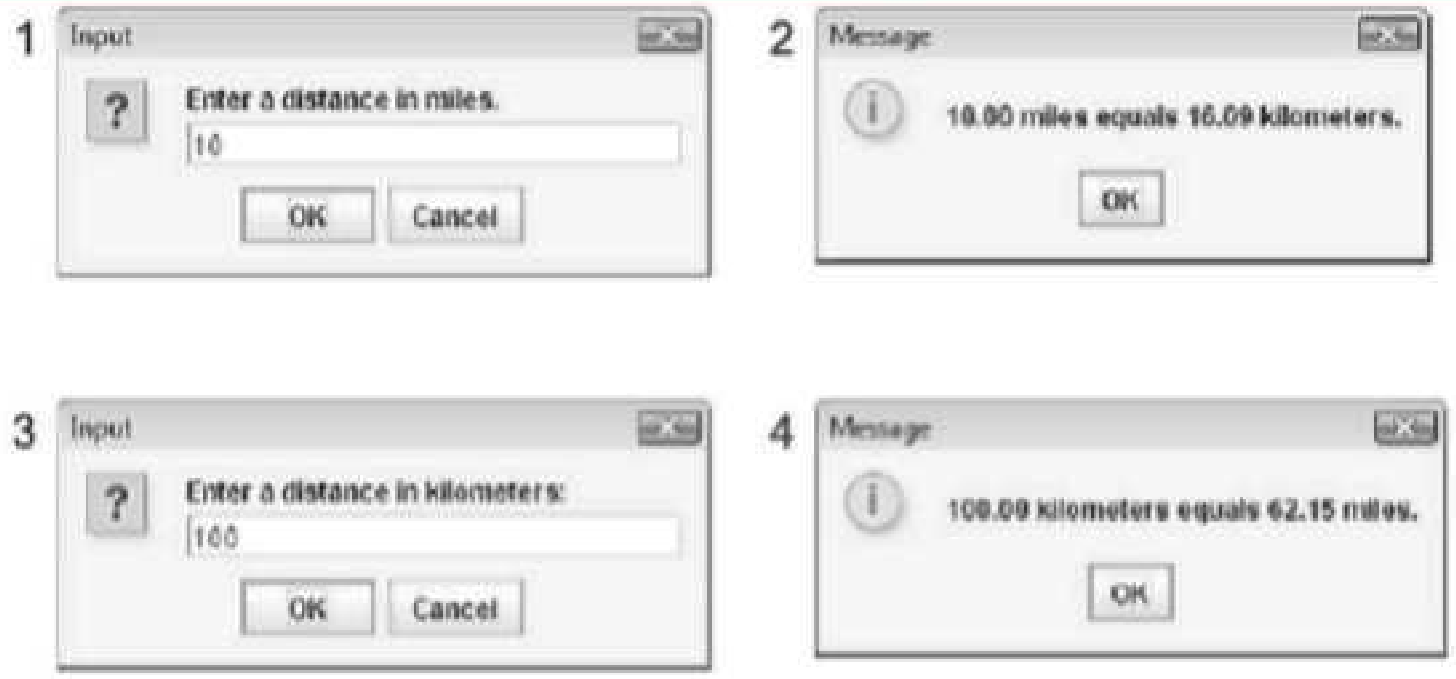
```java
kilometers = Metric.milesToKilometers(10.0);
```

This statement calls the `milesToKilometers` method, passing the value `10.0` as an argument. Notice that the method is not called from an instance of the class, but is called directly from the `Metric` class. Code Listing 8-4 shows a program that uses the `Metric` class. Figure 8-2 shows an example of interaction with the program.

Code Listing 8-4 (`MetricDemo.java`)

```java
import javax.swing.JOptionPane;
import java.text.DecimalFormat;

/**
   This program demonstrates the Metric class.
*/

public class MetricDemo
{
   public static void main(String[] args)
   {
      String input; // To hold input
      double miles; // A distance in miles
      double kilos; // A distance in kilometers

      // Create a DecimalFormat object.
      DecimalFormat fmt =
                    new DecimalFormat("0.00");

      // Get a distance in miles.
      input = JOptionPane.showInputDialog("Enter " +
                          "a distance in miles.");
      miles = Double.parseDouble(input);

      // Convert the distance to kilometers.
      kilos = Metric.milesToKilometers(miles);
      JOptionPane.showMessageDialog(null,
               fmt.format(miles) + " miles equals " +
               fmt.format(kilos) + " kilometers.");

      // Get a distance in kilometers.
      input = JOptionPane.showInputDialog("Enter " +
                         "a distance in kilometers:");
      kilos = Double.parseDouble(input);

      // Convert the distance to kilometers.
      miles = Metric.kilometersToMiles(kilos);
      JOptionPane.showMessageDialog(null,
            fmt.format(kilos) + " kilometers equals " +
            fmt.format(miles) + " miles.");

        System.exit(0);
    }
}
```

Figure 8-2 Interaction with the `MetricDemo.java` program



Enter a distance in miles.
10

10.00 miles equals 16.09 kilometers.

Enter a distance in kilometers:
100

100.00 kilometers equals 62.15 miles.

Static methods are convenient for many tasks because they can be called directly from the class, as needed. They are most often used to create utility classes that perform operations on data, but have no need to collect and store data. The `Metric` class is a good example. It is used as a container to hold methods that convert miles to kilometers and vice versa, but is not intended to store any data.

The only limitation that static methods have is that they cannot refer to non-static members of the class. This means that any method called from a static method must also be static. It also means that if the method uses any of the class's fields, they must be static as well.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

8.1 What is the difference between an instance field and a static field?

8.2 What action is possible with a static method that isn't possible with an instance method?

8.3 Describe the limitation of static methods.

8.2 Passing Objects as Arguments to Methods

CONCEPT: To pass an object as a method argument, you pass an object reference.

In Chapter 5 we discussed how primitive values, as well as references to `String` objects, can be passed as arguments to methods. You can also pass references to other types of objects as arguments to methods. For example, recall that in Chapter 6 we developed a `Rectangle` class. The program in Code Listing 8-5 creates an instance of the `Rectangle` class and then passes a reference to that object as an argument to a method.

Code Listing 8-5 `(PassObject.java)`

```java
/**
   This program passes an object as an argument.
*/

public class PassObject
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Pass a reference to the object to
      // the displayRectangle method.
      displayRectangle(box);
   }

   /**
      The displayRectangle method displays the
      length and width of a rectangle.
      @param r A reference to a Rectangle
      object.
   */

   public static void displayRectangle(Rectangle r)
   {
      // Display the length and width.
      System.out.println("Length : " + r.getLength() +
                         " Width : " + r.getWidth());
   }
}
```

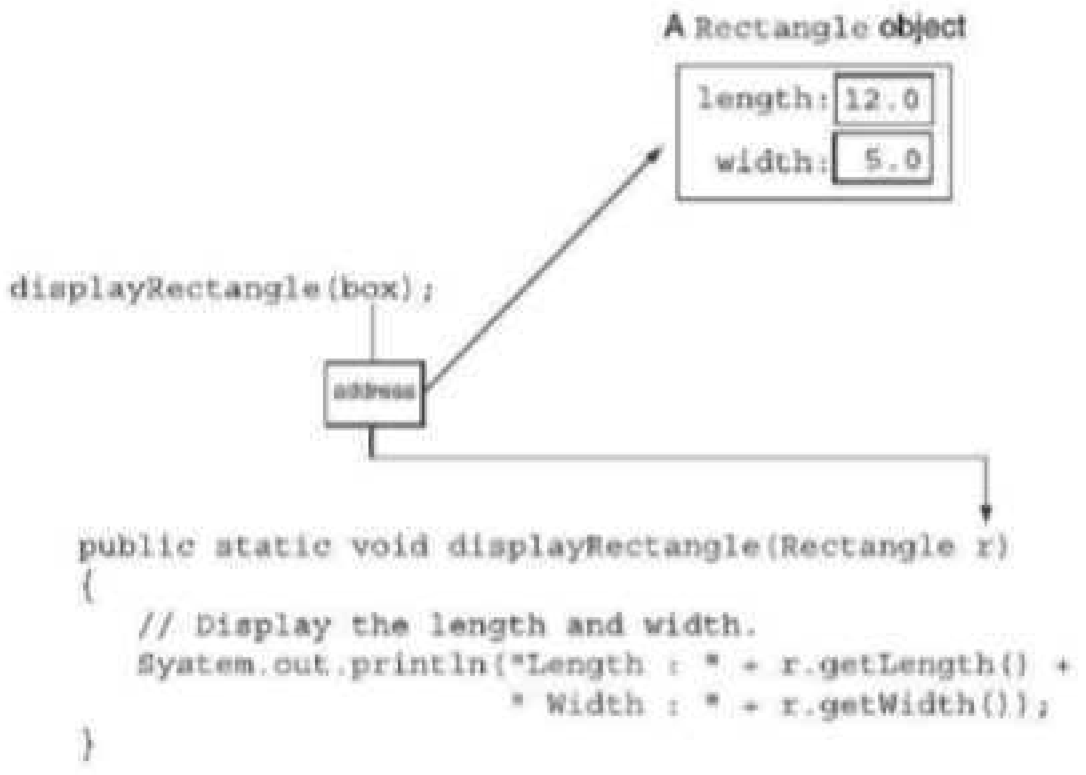
Program Output

```
Length : 12.0 Width : 5.0
```

In this program's `main` method, the `box` variable is a `Rectangle` reference variable. In line 14 its value is passed as an argument to the `displayRectangle` method. The `displayRectangle` method has a parameter variable, `r`, which is also a `Rectangle` reference variable, that receives the argument.

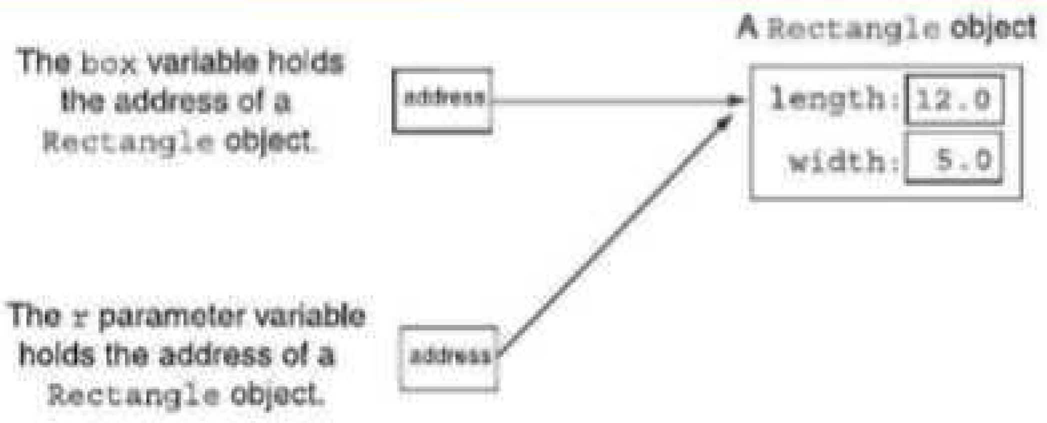
Recall that a reference variable holds the memory address of an object. When the `displayRectangle` method is called, the address that is stored in `box` is passed into the `r` parameter variable. This is illustrated in Figure 8-3. This means that when the `displayRectangle` method is executing, `box` and `r` both reference the same object. This is illustrated in Figure 8-4.

Figure 8-3 Passing a reference as an argument



```
public static void displayRectangle(Rectangle r)
{
   // Display the length and width.
   System.out.println("Length : " + r.getLength() +
                      " Width : " + r.getWidth());
}
```

Figure 8-4 Both box and `r` reference the same object


A Rectangle object
The box variable holds the address of a Rectangle object.
address
length: 12.0
width: 5.0
The r parameter variable holds the address of a Rectangle object.
address

Recall from Chapter 5 that when a variable is passed as an argument to a method, it is said to be passed by value. This means that a copy of the variable's value is passed into the method's parameter. When the method changes the contents of the parameter variable, it does not affect the contents of the original variable that was passed as an argument. When a reference variable is passed as an argument to a method, however, the method has access to the object that the variable references. As you can see from Figure 8-4, the displayRectangle method has access to the same Rectangle object that the box variable references. When a method receives an object reference as an argument, it is possible for the method to modify the contents of the object referenced by the variable. This is demonstrated in Code Listing 8-6.

Code Listing 8-6 (PassObject2.java)

```java
/**
   This program passes an object as an argument.
   The object is modified by the receiving method.
*/

public class PassObject2
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle(12.0, 5.0);

      // Display the object's contents.
      System.out.println("Contents of the box object:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());

      // Pass a reference to the object to the
      // changeRectangle method.
      changeRectangle(box);

      // Display the object's contents again.
      System.out.println("\nNow the contents of the " +
                         "box object are:");
      System.out.println("Length : " + box.getLength() +
                         " Width : " + box.getWidth());
   }

   /**
      The changeRectangle method sets a Rectangle
      object's length and width to 0.
      @param r The Rectangle object to change.
   */

   public static void changeRectangle(Rectangle r)
   {
      r.setLength(0.0);
      r.setWidth(0.0);
   }
}
```

Program Output

```
Contents of the box object:
Length : 12.0 Width : 5.0

Now the contents of the box object are:
Length : 0.0 Width : 0.0
```

When writing a method that receives the value of a reference variable as an argument, you must take care not to accidentally modify the contents of the object that is referenced by the variable.

8.3 Returning Objects from Methods

CONCEPT: A method can return a reference to an object.

VideoNote
Returning Objects from Methods

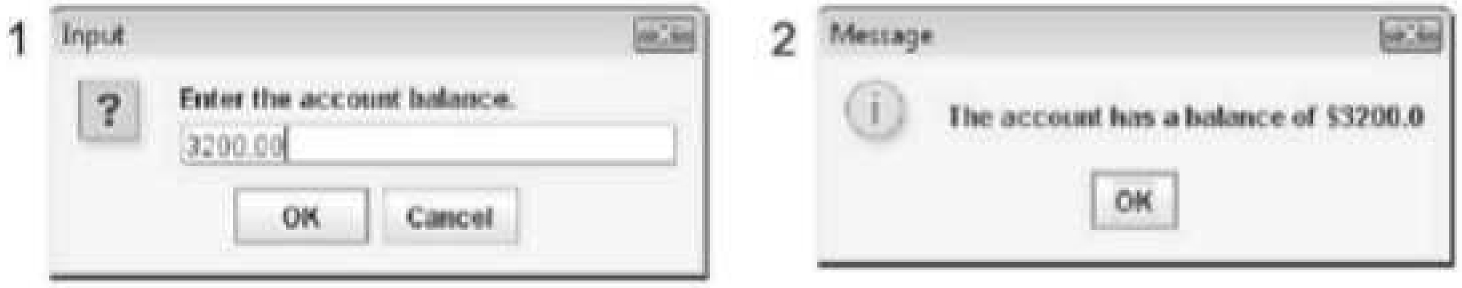
Just as methods can be written to return an `int`, `double`, `float`, or other primitive data type, they can also be written to return a reference to an object. For example, recall the `BankAccount` class that was discussed in Chapter 6. The program in Code Listing 8-7 uses a method, `getAccount`, which returns a reference to a `BankAccount` object. Figure 8-5 shows example interaction with the program.

Code Listing 8-7 **(ReturnObject.java)**

```
import javax.swing.JOptionPane;

/**
   This program demonstrates how a method
   can return a reference to an object.
*/

public class ReturnObject
{
   public static void main(String[] args)
   {
      BankAccount account;

      // Get a reference to a BankAccount object.
      account = getAccount();

      // Display the account's balance.
      JOptionPane.showMessageDialog(null,
               "The account has a balance of $" +
               account.getBalance());

```

```
         System.exit(0);
      }

      /**
         The getAccount method creates a BankAccount
         object with the balance specified by the
         user.
         @return A reference to the object.
      */

      public static BankAccount getAccount()
      {
         String input;         // To hold input
         double balance;       // Account balance

         // Get the balance from the user.
         input = JOptionPane.showInputDialog("Enter " +
                                    "the account balance.");
         balance = Double.parseDouble(input);

         // Create a BankAccount object and return
         // a reference to it.
         return new BankAccount(balance);
      }
   }
```

Figure 8-5 Interaction with the `ReturnObject.java` program



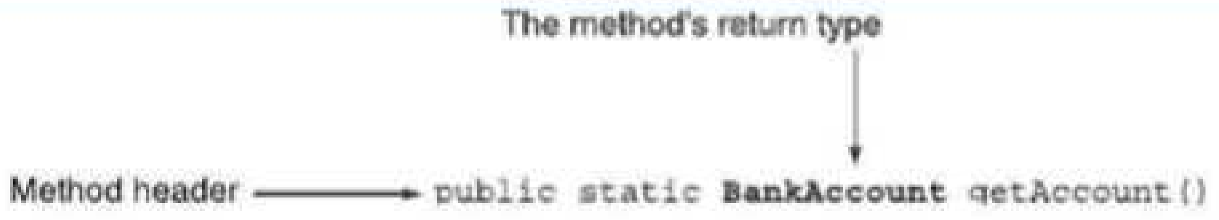
Enter the account balance.

3200.00

The account has a balance of $3200.0

Notice that the `getAccount` method has a return data type of `BankAccount`. Figure 8-6 shows the method's return type, which is listed in the method header.

Figure 8-6 The `getAccount` method header



A return type of BankAccount means the method returns a reference to a BankAccount object when it terminates. The following statement, which appears in line 15, assigns the getAccount method's return value to account:

```
account = getAccount();
```

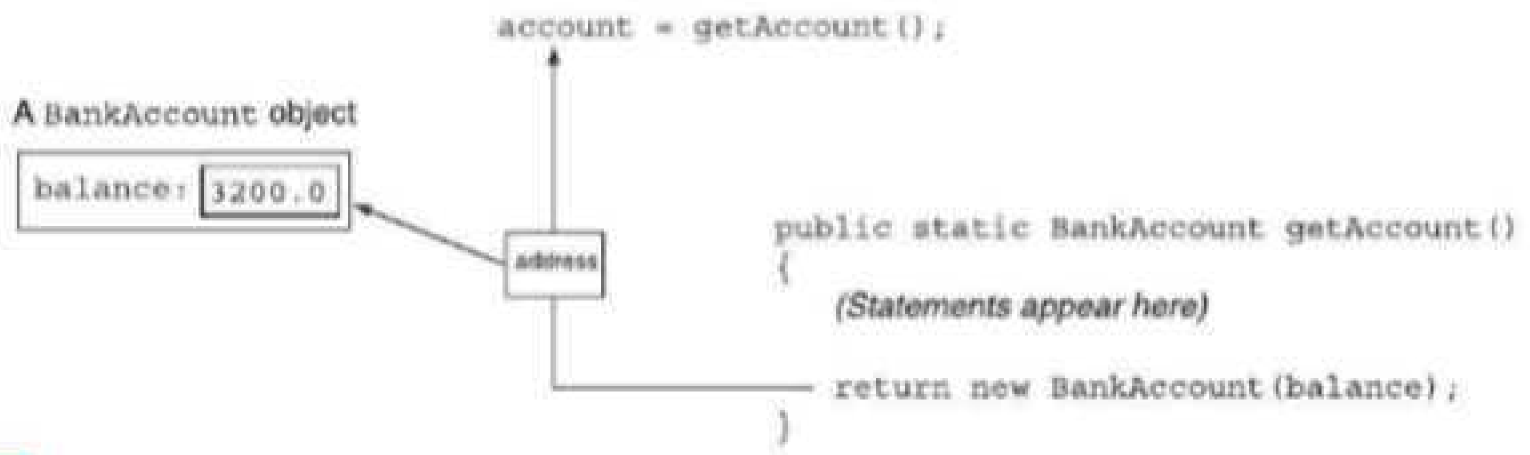
After this statement executes, the account variable will reference the BankAccount object that was returned from the getAccount method.

Now let's look at the getAccount method. In lines 38 and 39 the method uses a JOptionPane dialog box to get the account balance from the user. In line 40 the value entered by the user is converted to a double and assigned to balance, a local variable. The last statement in the method, in line 44, is the following return statement:

```
return new BankAccount(balance);
```

This statement uses the new key word to create a BankAccount object, passing balance as an argument to the constructor. The address of the object is then returned from the method, as illustrated in Figure 8-7. Back in line 15, where the method is called, the address is assigned to account.

Figure 8-7 The getAccount method returns a reference to a BankAccount object



8.4 The toString Method

CONCEPT: Most classes can benefit from having a method named toString, which is implicitly called under certain circumstances. Typically, the method returns a string that represents the state of an object.

Quite often we need to display a message that indicates an object's state. An object's *state* is simply the data that is stored in the object's fields at any given moment. For example, recall that the BankAccount class has one field: balance. At any given moment, a BankAccount object's balance field will hold some value. The value of the balance field represents the object's state at that moment. The following might be an example of code that displays a BankAccount object's state:

```
BankAccount account = new BankAccount(1500.0);
System.out.println("The account balance is $" +
                   account.getBalance());
```

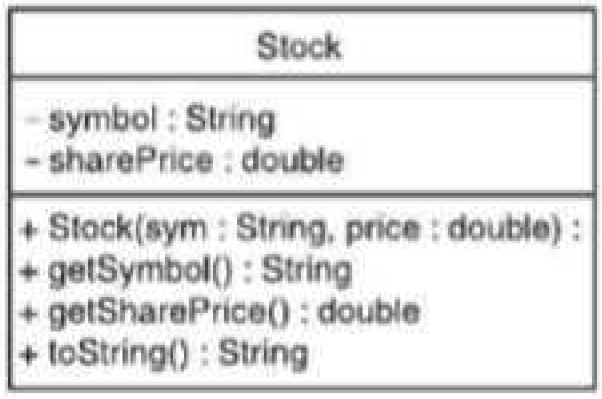
The first statement creates a `BankAccount` object, passing the value 1500.0 to the constructor. Recall that the `BankAccount` constructor stores this value in the `balance` field. After this statement executes, the `account` variable will reference the `BankAccount` object. In the second statement, the `System.out.println` method displays a string showing the value of the object's `balance` field. The output of this statement will look like this:

```
The account balance is $1500.0
```

Let's take a closer look at the second statement, which displays the state of the object. The argument that is passed to `System.out.println` is a string, which is put together from two pieces. The concatenation operator (+) joins the pieces together. The first piece is the string literal `"The account balance is $"`. To this, the value returned from the `getBalance` method is concatenated. The resulting string, which is displayed on the screen, represents the current state of the object.

Creating a string that represents the state of an object is such a common task that many programmers equip their classes with a method that returns such a string. In Java, it is standard practice to name this method `toString`. Let's look at an example of a class that has a `toString` method. Figure 8-8 shows the UML diagram for the `Stock` class, which holds data about a company's stock.

Figure 8-8 UML diagram for the `Stock` class



This class has two fields: `symbol` and `sharePrice`. The `symbol` field holds the trading symbol for the company's stock. This is a short series of characters that are used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` field holds the current price per share of the stock. Table 8-1 describes the class's methods.

Table 8-1 The `Stock` class methods

Method	Description
Constructor	This constructor accepts arguments that are assigned to the `symbol` and `sharePrice` fields.
`getSymbol`	This method returns the value in the `symbol` field.
`getSharePrice`	This method returns the value in the `sharePrice` field.
`toString`	This method returns a string representing the state of the object. The string will be appropriate for displaying on the screen.

Code Listing 8-8 shows the code for the Stock class. (This file is in the source code folder *Chapter 08\Stock Class Phase 1.*)

Code Listing 8-8 (Stock.java)

```
/**
   The Stock class holds data about a stock.
*/

public class Stock
{
   private String symbol;        // Trading symbol of stock
   private double sharePrice;    // Current price per share

   /**
      Constructor
      @param sym The stock's trading symbol.
      @param price The stock's share price.
   */

   public Stock(String sym, double price)
   {
      symbol = sym;
      sharePrice = price;
   }

   /**
      getSymbol method
      @return The stock's trading symbol.
   */

   public String getSymbol()
   {
      return symbol;
   }

   /**
      getSharePrice method
      @return The stock's share price
   */

   public double getSharePrice()
   {
      return sharePrice;
   }

   /**
43       toString method
44       @return A string indicating the object's
45               trading symbol and share price.
46    */
47
48    public String toString()
49    {
50       // Create a string describing the stock.
51       String str = "Trading symbol: " + symbol +
52                    "\nShare price: " + sharePrice;
53
54       // Return the string.
55       return str;
56    }
57 }
```

The `toString` method appears in lines 48 through 56. The method creates a string listing the stock's trading symbol and price per share. This string is then returned from the method. A call to the method can then be passed to `System.out.println`, as shown in the following code:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

This code would produce the following output:

```
Trading symbol: XYZ
Share price: 9.62
```

In actuality, it is unnecessary to explicitly call the `toString` method in this example. If you write a `toString` method for a class, Java will automatically call the method when the object is passed as an argument to `print` or `println`. The following code would produce the same output as that previously shown:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

Java also implicitly calls an object's `toString` method any time you concatenate an object of the class with a string. For example, the following code would implicitly call the `xyzCompany` object's `toString` method:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" + xyzCompany);
```

This code would produce the following output:

```
The stock data is:
Trading symbol: XYZ
Share price: 9.62
```

Code Listing 8-9 shows a complete program demonstrating the `Stock` class's `toString` method. (This file is in the source code folder *Chapter 08\Stock Class Phase 1*.)

Code Listing 8-9 (StockDemo1.java)

```
/**
   This program demonstrates the Stock class's
   toString method.
*/

public class StockDemo1
{
   public static void main(String[] args)
   {
      // Create a Stock object for the XYZ Company.
      // The trading symbol is XYZ and the current
      // price per share is $9.62.
      Stock xyzCompany = new Stock ("XYZ", 9.62);

      // Display the object's values.
      System.out.println(xyzCompany);
   }
}
```

Program Output

```
Trading symbol: XYZ
Share price: 9.62
```

NOTE: Every class automatically has a `toString` method that returns a string containing the object's class name, followed by the `@` symbol, followed by an integer that is usually based on the object's memory address. This method is called when necessary if you have not provided your own `toString` method. You will learn more about this in Chapter 10.

8.5 Writing an equals Method

CONCEPT: You cannot determine whether two objects contain the same data by comparing them with the == operator. Instead, the class must have a method such as `equals` for comparing the contents of objects.

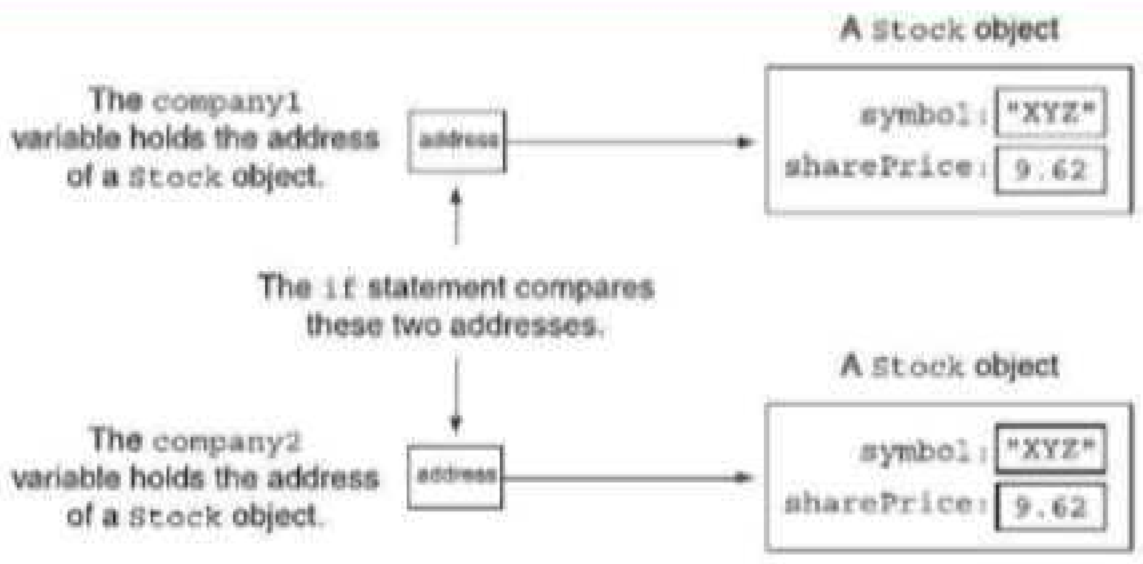
Recall from Chapter 3 that the `String` class has a method named `equals`, which determines whether two strings are equal. You can write an `equals` method for any of your own classes as well.

In fact, you must write an `equals` method (or one that works like it) for a class in order to determine whether two objects of the class contain the same values. This is because you cannot use the `==` operator to compare the contents of two objects. For example, the following code might appear to compare the contents of two Stock objects, but in reality does not:

```
// Create two Stock objects with the same values.
Stock company1 = new Stock("XYZ", 9.62);
Stock company2 = new Stock("XYZ", 9.62);

// Use the == operator to compare the objects.
// (This is a mistake.)
if (company1 == company2)
   System.out.println("Both objects are the same.");
else
   System.out.println("The objects are different.");
```

When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. This is illustrated in Figure 8-9.

Figure 8-9 The if statement tests the contents of the reference variables, not the contents of the objects the variables reference


A Stock object
The company1 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62
The if statement compares these two addresses.
A Stock object
The company2 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62

Because the two variables reference different objects in memory, they will contain different addresses. Therefore, the result of the boolean expression company1 == company2 is false and the code reports that the objects are not the same. Instead of using the == operator to compare the two Stock objects, we should write an equals method that compares the contents of the two objects.

In the source code folder *Chapter 08\Stock Class Phase 2*, you will find a revision of the Stock class. This version of the class has an equals method. The code for the method follows (no other part of the class has changed, so only the equals method is shown):

```
public boolean equals(Stock object2)
{
   boolean status;

   // Determine whether this object's symbol and
   // sharePrice fields are equal to object2's
```

```
      // symbol and sharePrice fields.
      if (symbol.equals(object2.symbol) &&
         sharePrice == object2.sharePrice)
         status = true;   // Yes, the objects are equal.
      else
         status = false;  // No, the objects are not equal.

      // Return the value in status.
      return status;
   }
```

The `equals` method accepts a `Stock` object as its argument. The parameter variable `object2` will reference the object that was passed as an argument. The `if` statement performs the following comparison: If the `symbol` field of the calling object is equal to the `symbol` field of `object2`, *and* the `sharePrice` field of the calling object is equal to the `sharePrice` field of `object2`, then the two objects contain the same values. In this case, the local variable `status` (a `boolean`) is set to `true`. Otherwise, `status` is set to `false`. Finally, the method returns the value of the `status` variable.

Notice that the method can access `object2`'s `symbol` and `sharePrice` fields directly. Because `object2` references a `Stock` object, and the `equals` method is a member of the `Stock` class, the method is allowed to access `object2`'s private fields.

The program in Code Listing 8-10 demonstrates the `equals` method. (This file is also stored in the source code folder *Chapter 08\Stock Class Phase 2*.)

Code Listing 8-10 (`StockCompare.java`)

```
/**
   This program uses the Stock class's equals
   method to compare two Stock objects.
*/

public class StockCompare
{
   public static void main(String[] args)
   {
      // Create two Stock objects with the same values.
      Stock company1 = new Stock("XYZ", 9.62);
      Stock company2 = new Stock("XYZ", 9.62);

      // Use the equals method to compare the objects.
      if (company1.equals(company2))
         System.out.println("Both objects are the same.");
      else
         System.out.println("The objects are different.");
   }
}
```

Program Output

```
Both objects are the same.
```

If you want to be able to compare the objects of a given class, you should always write an `equals` method for the class.

NOTE: Every class automatically has an `equals` method, which works the same as the `==` operator. This method is called when necessary if you have not provided your own `equals` method. You will learn more about this in Chapter 10.

8.6 Methods That Copy Objects

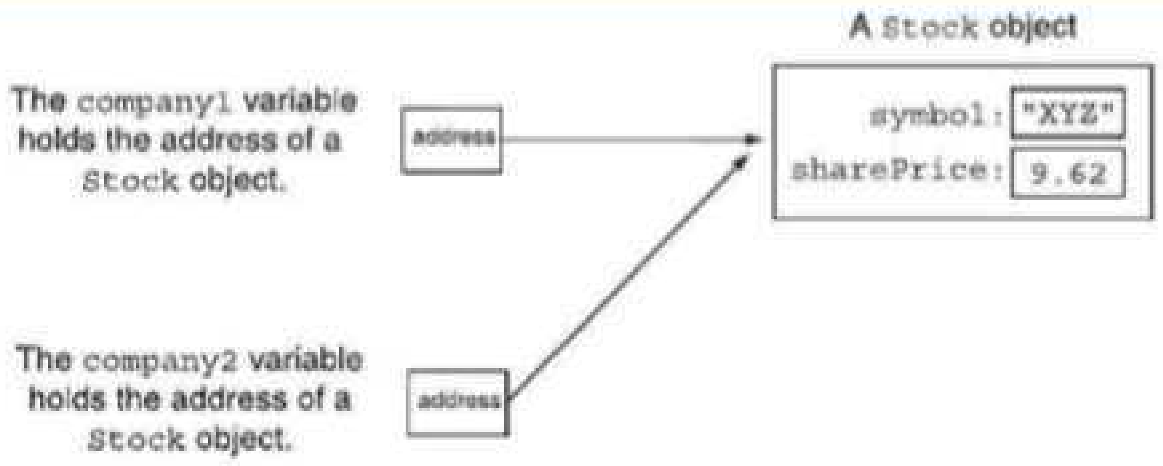
CONCEPT: You can simplify the process of duplicating objects by equipping a class with a method that returns a copy of an object.

You cannot make a copy of an object with a simple assignment statement, as you would with a primitive variable. For example, look at the following code:

```
Stock company1 = new Stock("XYZ", 9.62);
Stock company2 = company1;
```

The first statement creates a `Stock` object and assigns its address to the `company1` variable. The second statement assigns `company1` to `company2`. This does not make a copy of the object referenced by `company1`. Rather, it makes a copy of the address that is stored in `company1` and stores that address in `company2`. After this statement executes, both the `company1` and `company2` variables will reference the same object. This is illustrated in Figure 8-10.

Figure 8-10 Both variables reference the same object


A Stock object
The company1 variable holds the address of a Stock object.
address
symbol: "XYZ"
sharePrice: 9.62
The company2 variable holds the address of a Stock object.
address

This type of assignment operation is called a *reference copy* because only the object's address is copied, not the actual object itself. To copy the object itself, you must create a new object and then set the new object's fields to the same values as the fields of the object that is being copied. This process can be simplified by equipping the class with a method that performs this operation. The method then returns a reference to the duplicate object.

In the source code folder *Chapter 08\Stock Class Phase 3*, you will find a revision of the `Stock` class. This version of the class has a method named `copy`, which returns a copy of a `Stock` object. The code for the method follows (no other part of the class has changed so only the `copy` method is shown):

```
public Stock copy()
{
   // Create a new Stock object and initialize it
   // with the same data held by the calling object.
   Stock copyObject = new Stock(symbol, sharePrice);

   // Return a reference to the new object.
   return copyObject;
}
```

The `copy` method creates a new `Stock` object and passes the calling object's `symbol` and `sharePrice` fields as arguments to the constructor. This makes the new object a copy of the calling object. The program in Code Listing 8-11 demonstrates the `copy` method. (This file is also stored in the source code folder *Chapter 08\Stock Class Phase 3*.)

Code Listing 8-11 (`ObjectCopy.java`)

```
/**
   This program uses the Stock class's copy method
   to create a copy of a Stock object.
*/

public class ObjectCopy
{
   public static void main(String[] args)
   {
      // Create a Stock object.
      Stock company1 = new Stock("XYZ", 9.62);

      // Declare a Stock variable
      Stock company2;

      // Make company2 reference a copy of the object
      // referenced by company1.
      company2 = company1.copy();

      // Display the contents of both objects.
      System.out.println("Company 1:\n" + company1);
      System.out.println();
      System.out.println("Company 2:\n" + company2);

      // Confirm that we actually have two objects.
      if (company1 == company2)
      {
         System.out.println("The company1 and company2 " +
                     "variables reference the same object.");
      }
      else
      {
         System.out.println("The company1 and company2 " +
                  "variables reference different objects.");
      }
   }
}
```

Program Output

```
Company 1:
Trading symbol: XYZ
Share price: 9.62

Company 2:
Trading symbol: XYZ
Share price: 9.62
The company1 and company2 variables reference different objects.
```

Copy Constructors

Another way to create a copy of an object is to use a copy constructor. A *copy constructor* is simply a constructor that accepts an object of the same class as an argument. It makes the object that is being created a copy of the object that was passed as an argument.

In the source code folder *Chapter 08\Stock Class Phase 4*, you will find another revision of the `Stock` class. This version of the class has a copy constructor. The code for the copy constructor follows (no other part of the class has changed, so only the copy constructor is shown):

```
public Stock(Stock object2)
{
   symbol = object2.symbol;
   sharePrice = object2.sharePrice;
}
```

Notice that the constructor accepts a `Stock` object as an argument. The parameter variable `object2` will reference the object that was passed as an argument. The constructor copies the values that are in `object2`'s `symbol` and `sharePrice` fields to the `symbol` and `sharePrice` fields of the object that is being created.

The following code segment demonstrates the copy constructor. It creates a `Stock` object referenced by the variable `company1`. Then it creates another `Stock` object referenced by the variable `company2`. The object referenced by `company2` is a copy of the object referenced by `company1`.

```
// Create a Stock object.
Stock company1 = new Stock("XYZ", 9.62);
// Create another Stock object that is a copy of the company1 object.
Stock company2 = new Stock(company1);
```

8.7 Aggregation

CONCEPT: Aggregation occurs when an instance of a class is a field in another class.

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and much more. It is the combination of all these objects that makes a house object.

VideoNote Aggregation

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

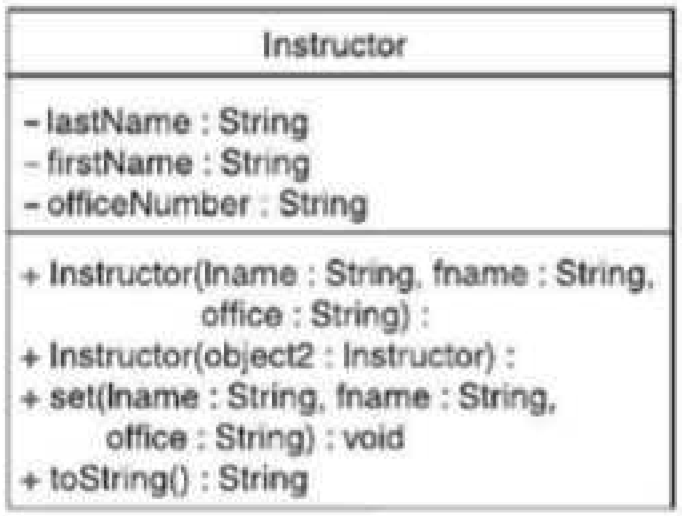
- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put fields for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as fields in the `Course` class.

Let's take a closer look at how this might be done. Figure 8-11 shows a UML diagram for the `Instructor` class. To keep things simple, the class has only the following methods:

- A constructor, which accepts arguments for the instructor's last name, first name, and office number
- A copy constructor
- A `set` method, which can be used to set all of the class's fields
- A `toString` method

Figure 8-11 UML diagram for the `Instructor` class


Instructor
- lastName : String
- firstName : String
- officeNumber : String
+ Instructor(lname : String, fname : String, office : String) :
+ Instructor(object2 : Instructor) :
+ set(lname : String, fname : String, office : String) : void
+ toString() : String

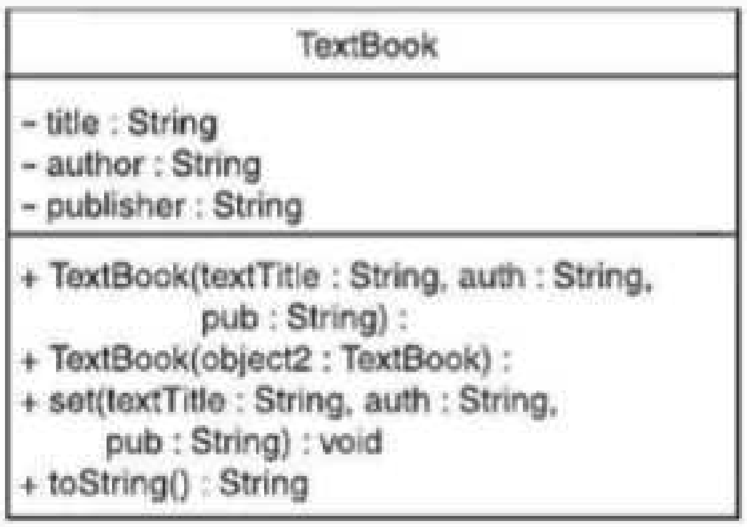
The code for the `Instructor` class is shown in Code Listing 8-12.

Code Listing 8-12 (`Instructor.java`)

```java
/**
   This class stores data about an instructor.
*/

public class Instructor
{
   private String lastName;       // Last name
   private String firstName;      // First name
   private String officeNumber;   // Office number

   /**
      This constructor initializes the last name,
      first name, and office number.
      @param lname The instructor's last name.
      @param fname The instructor's first name.
      @param office The office number.
   */

   public Instructor(String lname, String fname,
                     String office)
   {
      lastName = lname;
      firstName = fname;
      officeNumber = office;
   }

   /**
      The copy constructor initializes the object
      as a copy of another Instructor object.
      @param object2 The object to copy.
   */

   public Instructor(Instructor object2)
   {
      lastName = object2.lastName;
      firstName = object2.firstName;
      officeNumber = object2.officeNumber;
   }

   /**
      The set method sets a value for each field.
      @param lname The instructor's last name.
      @param fname The instructor's first name.
      @param office The office number.
   */

   public void set(String lname, String fname,
                   String office)
   {
      lastName = lname;
      firstName = fname;
      officeNumber = office;
   }

   /**
      toString method
      @return A string containing the instructor
              information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Last Name: " + lastName +
                   "\nFirst Name: " + firstName +
                   "\nOffice Number: " + officeNumber;

      // Return the string.
      return str;
   }
}
```

Figure 8-12 shows a UML diagram for the `TextBook` class. As before, we want to keep the class simple. The only methods it has are a constructor, a copy constructor, a `set` method, and a `toString` method. The code for the `TextBook` class is shown in Code Listing 8-13.

Figure 8-12 UML diagram for the `TextBook` class


TextBook
- title : String
- author : String
- publisher : String
+ TextBook(textTitle : String, auth : String, pub : String) :
+ TextBook(object2 : TextBook) :
+ set(textTitle : String, auth : String, pub : String) : void
+ toString() : String

Code Listing 8-13 (`TextBook.java`)

```java
/**
   This class stores data about a textbook.
*/

public class TextBook
{
   private String title;          // Title of the book
   private String author;         // Author's last name
   private String publisher;      // Name of publisher

   /**
      This constructor initializes the title,
      author, and publisher fields
      @param textTitle The book's title.
      @param auth The author's name.
      @param pub The name of the publisher.
   */

   public TextBook(String textTitle, String auth,
                   String pub)
   {
      title = textTitle;
      author = auth;
      publisher = pub;
   }

   /**
      The copy constructor initializes the object
      as a copy of another TextBook object.
      @param object2 The object to copy.
   */

   public TextBook(TextBook object2)
   {
      title = object2.title;
      author = object2.author;
      publisher = object2.publisher;
   }

   /**
      The set method sets a value for each field.
      @param textTitle The book's title.
      @param auth The author's name.
      @param pub The name of the publisher.
   */

   public void set(String textTitle, String auth,
                   String pub)
   {
      title = textTitle;
      author = auth;
      publisher = pub;
   }

   /**
      toString method
      @return A string containing the textbook
              information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Title: " + title +
                   "\nAuthor: " + author +
                   "\nPublisher: " + publisher;

      // Return the string.
      return str;
   }
}
```

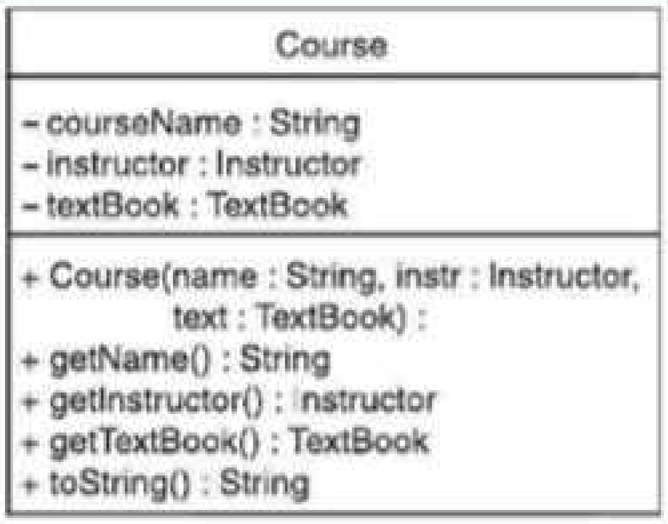
Figure 8-13 shows a UML diagram for the `Course` class. Notice that the `Course` class has an `Instructor` object and a `TextBook` object as fields. Making an instance of one class a field in another class is called *object aggregation*. The word *aggregate* means "a whole which is made of constituent parts." In this example, the `Course` class is an aggregate class because it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a "has a" relationship between the classes. For example, the relationships that exist among the `Course`, `Instructor`, and `TextBook` classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The "has a" relationship is sometimes called a *whole-part relationship* because one object is part of a greater whole. The code for the `Course` class is shown in Code Listing 8-14.

Figure 8-13 UML diagram for the `Course` class


Course
- courseName : String
- instructor : Instructor
- textBook : TextBook
+ Course(name : String, instr : Instructor, text : TextBook) :
+ getName() : String
+ getInstructor() : Instructor
+ getTextBook() : TextBook
+ toString() : String

Code Listing 8-14 (`Course.java`)

```
/**
   This class stores data about a course.
*/

public class Course
{
   private String courseName;      // Name of the course
   private Instructor instructor;  // The instructor
   private TextBook textBook;      // The textbook

   /**
      This constructor initializes the courseName,
      instructor, and text fields.
      @param name The name of the course.
      @param instructor An Instructor object.
      @param text A TextBook object.
   */

   public Course(String name, Instructor instr,
                 TextBook text)
   {
      // Assign the courseName.
      courseName = name;

      // Create a new Instructor object, passing
      // instr as an argument to the copy constructor.
      instructor = new Instructor(instr);

      // Create a new TextBook object, passing
      // text as an argument to the copy constructor.
      textBook = new TextBook(text);
   }

   /**
      getName method
      @return The name of the course.
   */

   public String getName()
   {
      return courseName;
   }

   /**
      getInstructor method
      @return A reference to a copy of this course's
              Instructor object.
   */

   public Instructor getInstructor()
   {
      // Return a copy of the instructor object.
      return new Instructor(instructor);
   }

   /**
      getTextBook method
      @return A reference to a copy of this course's
              TextBook object.
   */

   public TextBook getTextBook()
   {
      // Return a copy of the textBook object.
      return new TextBook(textBook);
   }

   /**
      toString method
      @return A string containing the course information.
   */

   public String toString()
   {
      // Create a string representing the object.
      String str = "Course name: " + courseName +
                   "\nInstructor Information:\n" +
                   instructor +
                   "\nTextbook Information:\n" +
                   textBook;

      // Return the string.
      return str;
   }
}
```

The program in Code Listing 8-15 demonstrates the `Course` class.

Code Listing 8-15 (`CourseDemo.java`)

```
/**
   This program demonstrates the Course class.
*/

public class CourseDemo
{
   public static void main(String[] args)
   {
      // Create an Instructor object.
      Instructor myInstructor =
          new Instructor("Kramer", "Shawn", "RH3010");

      // Create a TextBook object.
      TextBook myTextBook =
          new TextBook("Starting Out with Java",
                       "Gaddis", "Addison-Wesley");

      // Create a Course object.
      Course myCourse =
          new Course("Intro to Java", myInstructor,
                     myTextBook);

      // Display the course information.
      System.out.println(myCourse);
   }
}
```

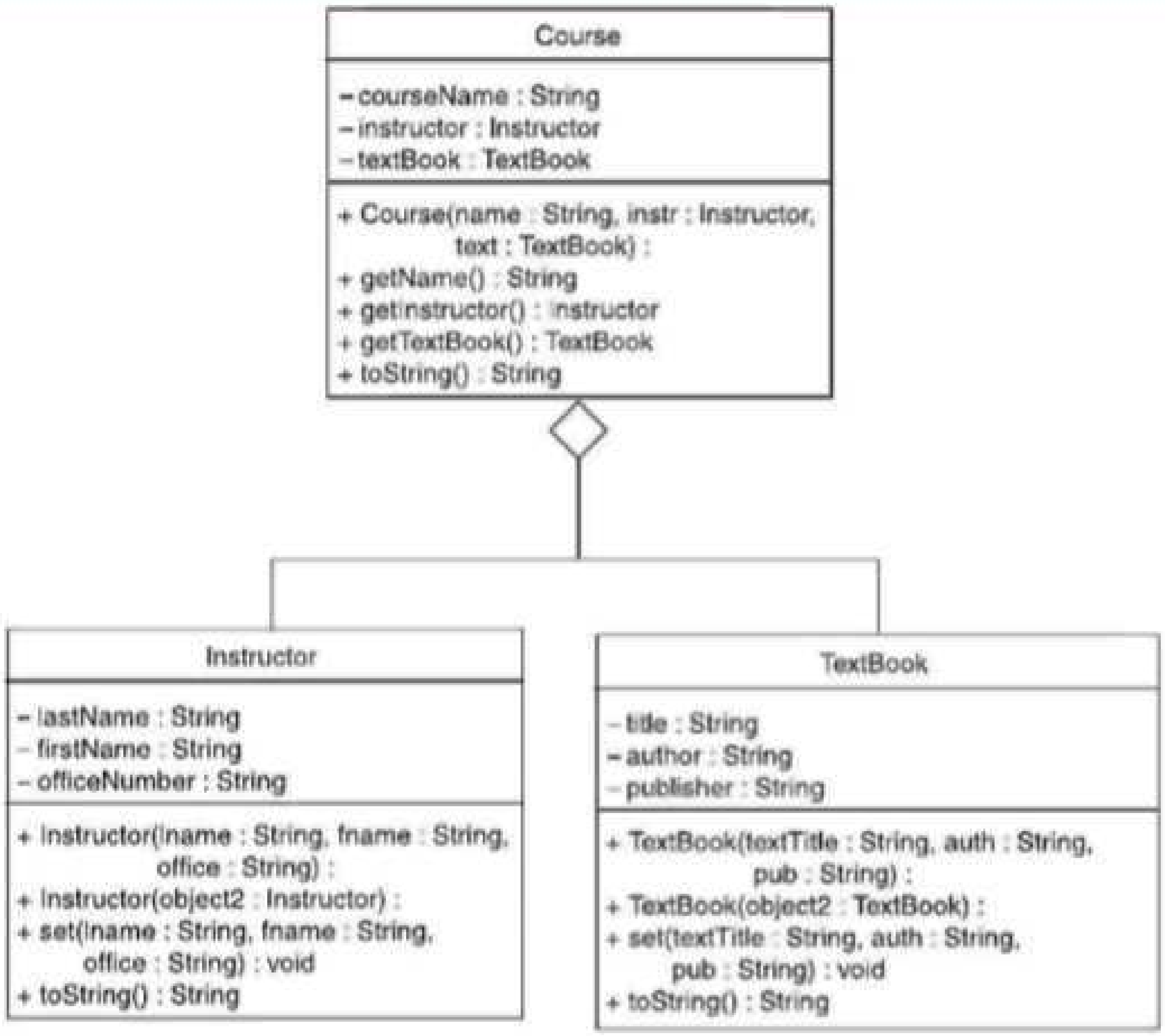
Program Output

```
Course name: Intro to Java
Instructor Information:
Last Name: Kramer
First Name: Shawn
Office Number: RH3010
Textbook Information:
Title: Starting Out with Java
Author: Gaddis
Publisher: Addison-Wesley
```

Aggregation in UML Diagrams

You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate. Figure 8-14 is a UML diagram that shows the relationship among the `Course`, `Instructor`, and `TextBook` classes. The open diamond is closest to the `Course` class because it is the aggregate (the whole).

Figure 8-14 UML diagram showing aggregation


Course
- courseName : String
- instructor : Instructor
- textBook : TextBook
+ Course(name : String, instr : Instructor, text : TextBook) :
+ getName() : String
+ getInstructor() : Instructor
+ getTextBook() : TextBook
+ toString() : String
Instructor
- lastName : String
- firstName : String
- officeNumber : String
+ Instructor(lname : String, fname : String, office : String) :
+ Instructor(object2 : Instructor) :
+ set(lname : String, fname : String, office : String) : void
+ toString() : String
TextBook
- title : String
- author : String
- publisher : String
+ TextBook(textTitle : String, auth : String, pub : String) :
+ TextBook(object2 : TextBook) :
+ set(textTitle : String, auth : String, pub : String) : void
+ toString() : String

Security Issues with Aggregate Classes

When writing an aggregate class, you should be careful not to unintentionally create "security holes" that can allow code outside the class to modify private data inside the class. We will focus on the following two specific practices that can help prevent security holes in your classes:

- **Perform Deep Copies When Creating Field Objects**
  An aggregate object contains references to other objects. When you make a copy of the aggregate object, it is important that you also make copies of the objects it references. This is known as a *deep copy*. If you make a copy of an aggregate object, but only make a reference copy of the objects it references, then you have performed a *shallow copy*.

- **Return Copies of Field Objects, Not the Originals**
  When a method in the aggregate class returns a reference to a field object, return a reference to a copy of the field object.

Let's discuss each of these practices in more depth.

Perform Deep Copies When Creating Field Objects

Let's take a closer look at the `Course` class. First, notice the arguments that the constructor accepts in lines 19 and 20 as follows:

- A reference to a `String` containing the name of the course is passed into the `name` parameter.
- A reference to an `Instructor` object is passed into the `instr` parameter.
- A reference to a `TextBook` object is passed into the `text` parameter.

Next, notice that the constructor does not merely assign `instr` to the `instructor` field. Instead, in line 27 it creates a new `Instructor` object for the `instructor` field and passes `instr` to the copy constructor. Here is the statement:

```
instructor = new Instructor(instr);
```

This statement creates a copy of the object referenced by `instr`. The `instructor` field will reference the copy.

When a class has a field that is an object, it is possible that a shallow copy operation will create a security hole. For example, suppose the `Course` constructor had been written as follows:

```
// Bad constructor!
public Course(String name, Instructor instr, TextBook text)
{
   // Assign the courseName.
   courseName = name;

   // Assign the instructor (shallow copy)
   instructor = instr;  // Causes security hole!

   // Assign the textBook (shallow copy)
   textBook = text;     // Causes security hole!
}
```

In this example, the `instructor` and `textBook` fields are merely assigned the addresses of the objects passed into the constructor. This can cause problems because there may be variables outside the `Course` object that also contain references to these `Instructor` and `TextBook` objects. These outside variables would provide direct access to the `Course` object's private data.

At this point you might be wondering why a deep copy was not also done for the `courseName` field. In line 23 the `Course` constructor performs a shallow copy, simply assigning the address of the `String` object referenced by `name` to the `courseName` field. This is permissible because `String` objects are immutable. An *immutable* object does not provide a way to change its contents. Even if variables outside the `Course` class reference the same object that `courseName` references, the object cannot be changed.

Return Copies of Field Objects, Not the Originals

When a method in an aggregate class returns a reference to a field object, it should return a reference to a copy of the field object, not the field object itself. For example, look at the `getInstructor` method in the `Course` class. The code is shown here:

```
public Instructor getInstructor()
{
   // Return a copy of the instructor object.
   return new Instructor(instructor);
}
```

Notice that the `return` statement uses the `new` key word to create a new `Instructor` object, passing the `instructor` field to the copy constructor. The object that is created is a copy of the object referenced by `instructor`. The address of the copy is then returned. This is preferable to simply returning a reference to the field object itself. For example, suppose the method had been written this way:

```
// Bad method
public Instructor getInstructor()
{
   // Return a reference to the instructor object.
   return instructor;  // WRONG! Causes a security hole.
}
```

This method returns the value stored in the `instructor` field, which is the address of an `Instructor` object. Any variable that receives the address can then access the `Instructor` object. This means that code outside the `Course` object can change the values held by the `Instructor` object. This is a security hole because the `Instructor` object is a private field! Only code inside the `Course` class should be allowed to access it.

NOTE: It is permissible to return a reference to a `String` object, even if the `String` object is a private field. This is because `String` objects are immutable.

Avoid Using `null` References

By default, a reference variable that is an instance field is initialized to the value `null`. This indicates that the variable does not reference an object. Because a `null` reference variable does not reference an object, you cannot use it to perform an operation that would require the existence of an object. For example, a `null` reference variable cannot be used to call a method. If you attempt to perform an operation with a `null` reference variable, the program will terminate. For example, look at the `FullName` class shown in Code Listing 8-16.

Code Listing 8-16 `(FullName.java)`

```java
/**
   This class stores a person's first, last, and middle
   names. The class is dangerous because it does not
   prevent operations on null reference fields.
*/

public class FullName
{
   private String lastName;         // Last name
   private String firstName;        // First name
   private String middleName;       // Middle name

   /**
      The setLastName method sets the lastName field.
      @param str The String to set lastName to.
   */

   public void setLastName(String str)
   {
      lastName = str;
   }

   /**
      The setFirstName method sets the firstName field.
      @param str The String to set firstName to.
   */

   public void setFirstName(String str)
   {
      firstName = str;
   }

   /**
      The setMiddleName method sets the middleName field.
      @param str The String to set middleName to.
   */

   public void setMiddleName(String str)
   {
      middleName = str;
   }

   /**
      The getLength method returns the length of the
      full name.
46        @return The length.
47     */
48
49    public int getLength()
50    {
51       return lastName.length() + firstName.length()
52              + middleName.length();
53    }
54
55    /**
56       The toString method returns the full name.
57       @return A reference to a String.
58     */
59
60    public String toString()
61    {
62       return firstName + " " + middleName + " "
63              + lastName;
64    }
65 }
```

First, notice that the class has three `String` reference variables as fields: `lastName`, `firstName`, and `middleName`. Second, notice that the class does not have a programmer-defined constructor. When an instance of this class is created, the `lastName`, `firstName`, and `middleName` fields will be initialized to `null` by the default constructor. Third, notice that the `getLength` method uses the `lastName`, `firstName`, and `middleName` variables to call the `String` class's `length` method in lines 51 and 52. Nothing is preventing the `length` method from being called while any or all of these reference variables are set to `null`. The program in Code Listing 8-17 demonstrates this.

Code Listing 8-17 **(NameTester.java)**

```
/**
   This program creates a FullName object, and then
   calls the object's getLength method before values
   are established for its reference fields. As a
   result, this program will crash.
*/

public class NameTester
{
   public static void main(String[] args)
   {
      int len;  // To hold the name length

      // Create a FullName object.
      FullName name = new FullName();

      // Get the length of the full name.
      len = name.getLength();
   }
}
```

This program will crash when you run it because the `getLength` method is called before the `name` object's fields are made to reference `String` objects. One way to prevent the program from crashing is to use `if` statements in the `getLength` method to determine whether any of the fields are set to `null`. Here is an example:

```
public int getLength()
{
   int len = 0;

   if (lastName != null)
      len += lastName.length();

   if (firstName != null)
      len += firstName.length();

   if (middleName != null)
      len += middleName.length();

   return len;
}
```

Another way to handle this problem is to write a no-arg constructor that assigns values to the reference fields. Here is an example:

```
public FullName()
{
   lastName = "";
   firstName = "";
   middleName = "";
}
```

8.8 The `this` Reference Variable

CONCEPT: The `this` key word is the name of a reference variable that an object can use to refer to itself. It is available to all non-static methods.

The key word `this` is the name of a reference variable that an object can use to refer to itself. For example, recall the `Stock` class presented earlier in this chapter. The class has the following `equals` method that compares the calling `Stock` object to another `Stock` object that is passed as an argument:

```
public boolean equals(Stock object2)
{
   boolean status;

   // Determine whether this object's symbol and
   // sharePrice fields are equal to object2's
   // symbol and sharePrice fields.
   if (symbol.equals(object2.symbol) &&
       sharePrice == object2.sharePrice)
      status = true;   // Yes, the objects are equal.
   else
      status = false;  // No, the objects are not equal.

   // Return the value in status.
   return status;
}
```

When this method is executing, the `this` variable contains the address of the calling object. We could rewrite the `if` statement as follows, and it would perform the same operation (the changes appear in bold):

```
if (this.symbol.equals(object2.symbol) &&
    this.sharePrice == object2.sharePrice)
```

The `this` reference variable is available to all of a class's non-static methods.

Using this to Overcome Shadowing

One common use of the `this` key word is to overcome the shadowing of a field name by a parameter name. Recall from Chapter 6 that if a method's parameter has the same name as a field in the same class, then the parameter name shadows the field name. For example, look at the constructor in the `Stock` class:

```
public Stock(String sym, double price)
{
   symbol = sym;
   sharePrice = price;
}
```

This method uses the parameter `sym` to accept an argument that is assigned to the `symbol` field, and the parameter `price` to accept an argument that is assigned to the `sharePrice` field. Sometimes it is difficult (and even time-consuming) to think of a good parameter name that is different from a field name. To avoid this problem, many programmers give parameters the same names as the fields to which they correspond, and then use the `this` key word to refer to the field names. For example, the `Stock` class's constructor could be written as follows:

```
public Stock(String symbol, double sharePrice)
{
   this.symbol = symbol;
   this.sharePrice = sharePrice;
}
```

Although the parameter names `symbol` and `sharePrice` shadow the field names `symbol` and `sharePrice`, the `this` key word overcomes the shadowing. Because `this` is a reference to the calling object, the expression `this.symbol` refers to the calling object's `symbol` field, and the expression `this.sharePrice` refers to the calling object's `sharePrice` field.

Using `this` to Call an Overloaded Constructor from Another Constructor

You already know that a constructor is automatically called when an object is created. You also know that you cannot call a constructor explicitly, as you do other methods. However, there is one exception to this rule: You can use the `this` key word to call one constructor from another constructor in the same class.

To illustrate this, recall the `Stock` class that was presented earlier in this chapter. It has the following constructor:

```
public Stock(String sym, double price)
{
   symbol = sym;
   sharePrice = price;
}
```

This constructor accepts arguments that are assigned to the `symbol` and `sharePrice` fields. Let's suppose we also want a constructor that only accepts an argument for the `symbol` field, and assigns 0.0 to the `sharePrice` field. Here's one way to write the constructor:

```
public Stock(String sym)
{
   this(sym, 0.0);
}
```

This constructor simply uses the `this` variable to call the first constructor. It passes the value in `sym` as the first argument, and 0.0 as the second argument. The result is that the `symbol` field is assigned the value in `sym` and the `sharePrice` field is assigned 0.0.

Remember the following rules about using `this` to call a constructor:

- `this` can only be used to call a constructor from another constructor in the same class.
- It *must* be the first statement in the constructor that is making the call. If it is not the first statement, a compiler error will result.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

8.4 Look at the following code. (You might want to review the `Stock` class presented earlier in this chapter.)

```
Stock stock1 = new Stock("XYZ", 9.65);
Stock stock2 = new Stock("SUNW", 7.92);
```

While the equals method is executing as a result of the following statement, what object does this reference?

```
if (stock2.equals(stock1))
   System.out.println("The stocks are the same.");
```

8.9 Enumerated Types

CONCEPT: An enumerated data type consists of a set of predefined values. You can use the data type to create variables that can hold only the values that belong to the enumerated data type.

You've already learned the concept of data types and how they are used with primitive variables. For example, a variable of the int data type can hold integer values within a certain range. You cannot assign floating-point values to an int variable because only int values may be assigned to int variables. A data type defines the values that are legal for any variable of that data type.

Sometimes it is helpful to create your own data type that has a specific set of legal values. For example, suppose you wanted to create a data type named Day, and the legal values in that data type were the names of the days of the week (Sunday, Monday, and so forth). When you create a variable of the Day data type, you can only store the names of the days of the week in that variable. Any other values would be illegal. In Java, such a type is known as an *enumerated data type*.

You use the enum key word to create your own data type and specify the values that belong to that type. Here is an example of an enumerated data type declaration:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
           THURSDAY, FRIDAY, SATURDAY }
```

An enumerated data type declaration begins with the key word enum, followed by the name of the type, followed by a list of identifiers inside braces. The example declaration creates an enumerated data type named Day. The identifiers SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY, which are listed inside the braces, are known as *enum constants*. They represent the values that belong to the Day data type. Here is the general format of an enumerated type declaration:

```
enum TypeName { One or more enum constants }
```

Note that the enum constants are not enclosed in quotation marks; therefore, they are not strings. enum constants must be legal Java identifiers.

TIP: When making up names for enum constants, it is not required that they be written in all uppercase letters. We could have written the Day type's enum constants as sunday, monday, and so forth. Because they represent constant values, however, the standard convention is to write them in all uppercase letters.