Once you have created an enumerated data type in your program, you can declare variables of that type. For example, the following statement declares `workDay` as a variable of the `Day` type:

```
Day workDay;
```

Because `workDay` is a `Day` variable, the only values that we can legally assign to it are the `enum` constants `Day.SUNDAY`, `Day.MONDAY`, `Day.TUESDAY`, `Day.WEDNESDAY`, `Day.THURSDAY`, `Day.FRIDAY`, and `Day.SATURDAY`. If we try to assign any value other than one of the `Day` type's `enum` constants, a compiler error will result. For example, the following statement assigns the value `Day.WEDNESDAY` to the `workDay` variable:

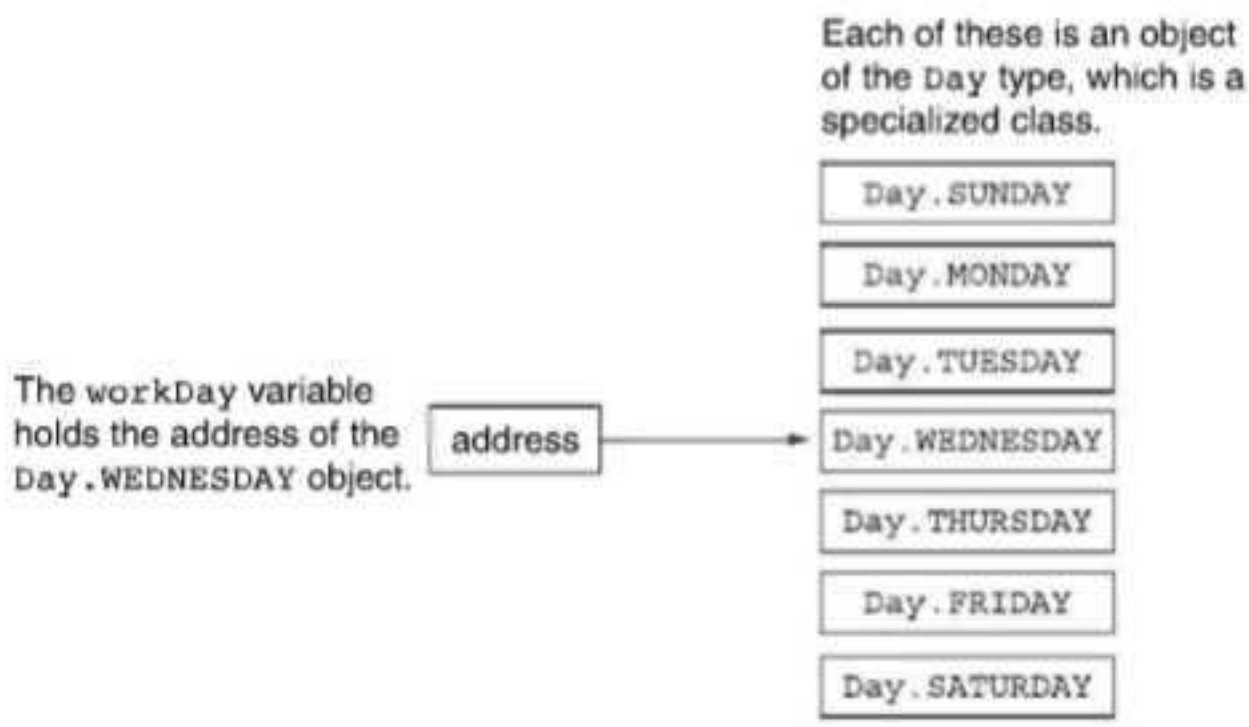
```
Day workDay = Day.WEDNESDAY;
```

Notice that we assigned `Day.WEDNESDAY` instead of just `WEDNESDAY`. The name `Day.WEDNESDAY` is the *fully qualified name* of the `Day` type's `WEDNESDAY` constant. Under most circumstances you must use the fully qualified name of an `enum` constant.

Enumerated Types Are Specialized Classes

When you write an enumerated type declaration, you are actually creating a special kind of class. In addition, the `enum` constants that you list inside the braces are actually objects of the class. In the previous example, `Day` is a class, and the `enum` constants `Day.SUNDAY`, `Day.MONDAY`, `Day.TUESDAY`, `Day.WEDNESDAY`, `Day.THURSDAY`, `Day.FRIDAY`, and `Day.SATURDAY` are all instances of the `Day` class. When we assigned `Day.WEDNESDAY` to the `workDay` variable, we were assigning the address of the `Day.WEDNESDAY` object to the variable. This is illustrated in Figure 8-15.

Figure 8-15 The `workDay` variable references the `Day.WEDNESDAY` object



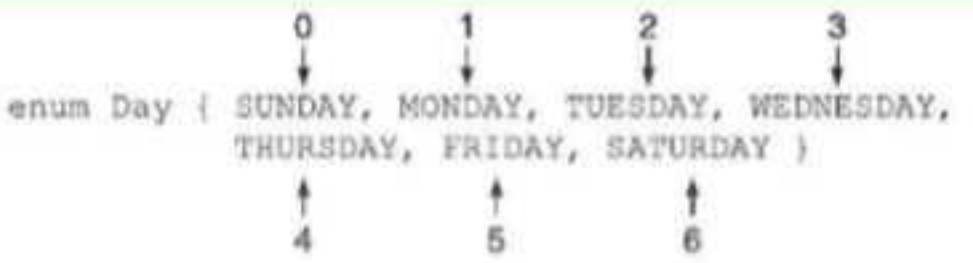
`enum` constants, which are actually objects, come automatically equipped with a few methods. One of them is the `toString` method. The `toString` method simply returns the name of

the calling enum constant as a string. For example, assuming that the Day type has been declared as previously shown, both of the following code segments display the string WEDNESDAY (recall that the toString method is implicitly called when an object is passed to System.out.println):

```
// This code displays WEDNESDAY.
Day workDay = Day.WEDNESDAY;
System.out.println(workDay);

// This code also displays WEDNESDAY.
System.out.println(Day.WEDNESDAY);
```

enum constants also have a method named ordinal. The ordinal method returns an integer value representing the constant's ordinal value. The constant's *ordinal value* is its position in the enum declaration, with the first constant being at position 0. Figure 8-16 shows the ordinal value of each of the constants declared in the Day data type.

Figure 8-16 The Day enumerated data type and the ordinal positions of its enum constants



For example, assuming that the Day type has been declared as previously shown, look at the following code segment:

```
Day lastWorkDay = Day.FRIDAY;
System.out.println(lastWorkDay.ordinal());
System.out.println(Day.MONDAY.ordinal());
```

The ordinal value for Day.FRIDAY is 5 and the ordinal value for Day.MONDAY is 1, so this code will display:

```
5
1
```

The last enumerated data type methods that we will discuss here are equals and compareTo. The equals method accepts an object as its argument and returns true if that object is equal to the calling enum constant. For example, assuming that the Day type has been declared as previously shown, the following code segment will display "The two are the same":

```
Day myDay = Day.TUESDAY;
if (myDay.equals(Day.TUESDAY))
   System.out.println("The two are the same.");
```

The compareTo method is designed to compare enum constants of the same type. It accepts an object as its argument and returns the following:

- a negative integer value if the calling `enum` constant's ordinal value is less than the argument's ordinal value
- zero if the calling `enum` constant is the same as the argument
- a positive integer value if the calling `enum` constant's ordinal value is greater than the argument's ordinal value

For example, assuming that the `Day` type has been declared as previously shown, the following code segment will display `"FRIDAY is greater than MONDAY"`:

```
Day myDay = Day.FRIDAY;
if (myDay.compareTo(Day.MONDAY) > 0)
   System.out.println(myDay + " is greater than "
                      + Day.MONDAY);
```

One place to declare an enumerated type is inside a class. If you declare an enumerated type inside a class, it cannot be inside a method. Code Listing 8-18 shows an example. It demonstrates the `Day` enumerated type.

Code Listing 8-18 **(EnumDemo.java)**

```
1 /**
2    This program demonstrates an enumerated type.
3 */
4
5 public class EnumDemo
6 {
7    // Declare the Day enumerated type.
8    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
9               THURSDAY, FRIDAY, SATURDAY }
10
11   public static void main(String[] args)
12   {
13      // Declare a Day variable and assign it a value.
14      Day workDay = Day.WEDNESDAY;
15
16      // The following statement displays WEDNESDAY.
17      System.out.println(workDay);
18
19      // The following statement displays the ordinal
20      // value for Day.SUNDAY, which is 0.
21      System.out.println("The ordinal value for " +
22                         Day.SUNDAY + " is " +
23                         Day.SUNDAY.ordinal());
24
25      // The following statement displays the ordinal
26      // value for Day.SATURDAY, which is 6.
27      System.out.println("The ordinal value for " +
28                         Day.SATURDAY + " is " +
29                         Day.SATURDAY.ordinal());
```

```
30
31       // The following statement compares two enum constants.
32       if (Day.FRIDAY.compareTo(Day.MONDAY) > 0)
33          System.out.println(Day.FRIDAY + " is greater than " +
34                             Day.MONDAY);
35       else
36          System.out.println(Day.FRIDAY + " is NOT greater than " +
37                             Day.MONDAY);
38    }
39 }
```

Program Output

```
WEDNESDAY
The ordinal value for SUNDAY is 0
The ordinal value for SATURDAY is 6
FRIDAY is greater than MONDAY
```

You can also write an enumerated type declaration inside its own file. If you do, the filename must match the name of the type. For example, if we stored the `Day` type in its own file, we would name the file *Day.java*. This makes sense because enumerated data types are specialized classes. For example, look at Code Listing 8-19. This file, *CarType.java*, contains the declaration of an enumerated data type named `CarType`. When it is compiled, a byte code file named *CarType.class* will be generated.

Code Listing 8-19 (CarType.java)

```
1 /**
2    CarType enumerated data type
3 */
4
5 enum CarType { PORSCHE, FERRARI, JAGUAR }
```

Also look at Code Listing 8-20. This file, *CarColor.java*, contains the declaration of an enumerated data type named `CarColor`. When it is compiled, a byte code file named *CarColor.class* will be generated.

Code Listing 8-20 (CarColor.java)

```
1 /**
2    CarColor enumerated data type
3 */
4
5 enum CarColor { RED, BLACK, BLUE, SILVER }
```

Code Listing 8-21 shows the `SportsCar` class, which uses these enumerated types. Code Listing 8-22 demonstrates the class.

Code Listing 8-21 (SportsCar.java)

```
1  import java.text.DecimalFormat;
2
3  /**
4     SportsCar class
5  */
6
7  public class SportsCar
8  {
9     private CarType make;     // The car's make
10    private CarColor color;   // The car's color
11    private double price;     // The car's price
12
13    /**
14       The constructor initializes the car's make,
15       color, and price.
16       @param aMake The car's make.
17       @param aColor The car's color.
18       @param aPrice The car's price.
19    */
20
21    public SportsCar(CarType aMake, CarColor aColor,
22                     double aPrice)
23    {
24       make = aMake;
25       color = aColor;
26       price = aPrice;
27    }
28
29    /**
30       getMake method
31       @return The car's make.
32    */
33
34    public CarType getMake()
35    {
36       return make;
37    }
38
39    /**
40       getColor method
41       @return The car's color.
42    */
```

```
43
44    public CarColor getColor()
45    {
46       return color;
47    }
48
49    /**
50       getPrice method
51       @return The car's price.
52     */
53
54    public double getPrice()
55    {
56       return price;
57    }
58
59    /**
60       toString method
61       @return A string indicating the car's make,
62       color, and price.
63    */
64
65    public String toString()
66    {
67       // Create a DecimalFormat object for
68       // dollar formatting.
69       DecimalFormat dollar = new DecimalFormat("#,##0.00");
70
71       // Create a string representing the object.
72       String str = "Make: " + make +
73                    "\nColor: " + color +
74                    "\nPrice: $" + dollar.format(price);
75
76       // Return the string.
77       return str;
78    }
79 }
```

Code Listing 8-22 (SportsCarDemo.java)

```
1 /**
2    This program demonstrates the SportsCar class.
3 */
4
5 public class SportsCarDemo
6 {
```

```
 7    public static void main(String[] args)
 8    {
 9       // Create a SportsCar object.
10       SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
11                                            CarColor.RED, 100000);
12
13       // Display the object's values.
14       System.out.println(yourNewCar);
15    }
16 }
```

Program Output

```
Make: PORSCHE
Color: RED
Price: $100,000.00
```

Switching On an Enumerated Type

Java allows you to test an `enum` constant with a `switch` statement. For example, look at the program in Code Listing 8-23. It creates a `SportsCar` object, and then uses a `switch` statement to test the object's `make` field.

Code Listing 8-23 (SportsCarDemo2.java)

```
1 /**
2    This program shows that you can switch on an
3    enumerated type.
4 */
5
6 public class SportsCarDemo2
7 {
8    public static void main(String[] args)
9    {
10      // Create a SportsCar object.
11      SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
12                                           CarColor.RED, 100000);
13
14      // Get the car make and switch on it.
15      switch (yourNewCar.getMake())
16      {
17         case PORSCHE :
18            System.out.println("Your car was made in Germany.");
19            break;
20         case FERRARI :
```

```
21            System.out.println("Your car was made in Italy.");
22            break;
23         case JAGUAR :
24            System.out.println("Your car was made in England.");
25            break;
26         default:
27            System.out.println("I'm not sure where that car "
28                               + "was made.");
29      }
30   }
31 }
```

Program Output

```
Your car was made in Germany.
```

In line 15 the `switch` statement tests the value returned from the `yourNewCar.getMake()` method. This method returns a `CarType` enumerated constant. Based upon the value returned from the method, the program then branches to the appropriate `case` statement. Notice in the `case` statements that the enumerated constants are not fully qualified. In other words, we had to write `PORSCHE`, `FERRARI`, and `JAGUAR` instead of `CarType.PORSCHE`, `CarType.FERRARI`, and `CarType.JAGUAR`. If you give a fully qualified `enum` constant name as a `case` expression, a compiler error will result.

TIP: Notice that the `switch` statement in Code Listing 8-23 has a `default` section, even though it has a `case` statement for every `enum` constant in the `CarType` type. This will handle things in the event that more `enum` constants are added to the *CarType* file. This type of planning is an example of "defensive programming."

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

8.5 Look at the following statement, which declares an enumerated data type:

```
enum Flower { ROSE, DAISY, PETUNIA }
```

a) What is the name of the data type?
b) What is the ordinal value for the `enum` constant `ROSE`? For `DAISY`? For `PETUNIA`?
c) What is the fully qualifed name of the `enum` constant `ROSE`? Of `DAISY`? Of `PETUNIA`?
d) Write a statement that declares a variable of this enumerated data type. The variable should be named `flora`. Initialize the variable with the `PETUNIA` constant.

8.6 Assume that the following enumerated data type has been declared:

```
enum Creatures{ HOBBIT, ELF, DRAGON }
```

What will the following code display?

```
System.out.println(Creatures.HOBBIT + " "
                   + Creatures.ELF + " "
                   + Creatures.DRAGON);
```

8.7 Assume that the following enumerated data type has been declared:

```
enum Letters { Z, Y, X }
```

What will the following code display?

```
if (Letters.Z.compareTo(Letters.X) > 0)
   System.out.println("Z is greater than X.");
else
   System.out.println("Z is not greater than X.");
```

8.10 Garbage Collection

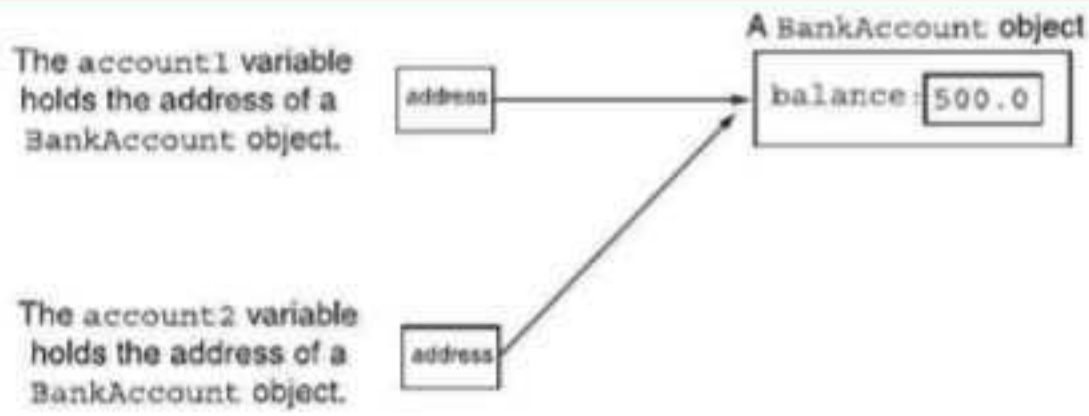
CONCEPT: The Java Virtual Machine periodically runs a process known as the garbage collector, which removes unreferenced objects from memory.

When an object is no longer needed, it should be destroyed so the memory it uses can be freed for other purposes. Fortunately, you do not have to destroy objects after you are finished using them. The Java Virtual Machine periodically performs a process known as garbage collection, which automatically removes unreferenced objects from memory. For example, look at the following code:

```
// Declare two BankAccount reference variables.
BankAccount account1, account2;

// Create an object and reference it with account1.
account1 = new BankAccount(500.0);

// Reference the same object with account2.
account2 = account1;

// Store null in account1 so it no longer
// references the object.
account1 = null;

// The object is still referenced by account2, though.
// Store null in account2 so it no longer references
// the object.
account2 = null;

// Now the object is no longer referenced, so it
// can be removed by the garbage collector.
```

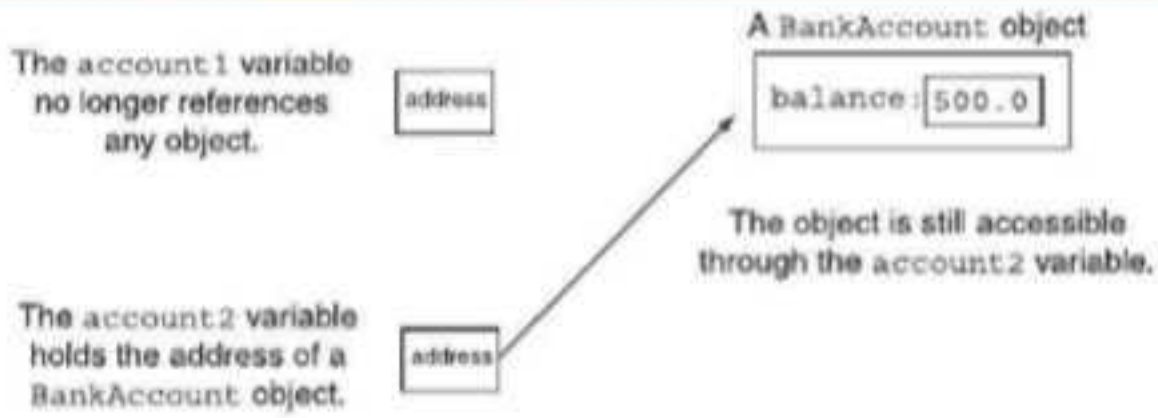
This code uses two reference variables, `account1` and `account2`. A `BankAccount` object is created and referenced by `account1`. Then, `account1` is assigned to `account2`, which causes `account2` to reference the same object as `account1`. This is illustrated in Figure 8-17.

Figure 8-17 Both `account1` and `account2` reference the same object



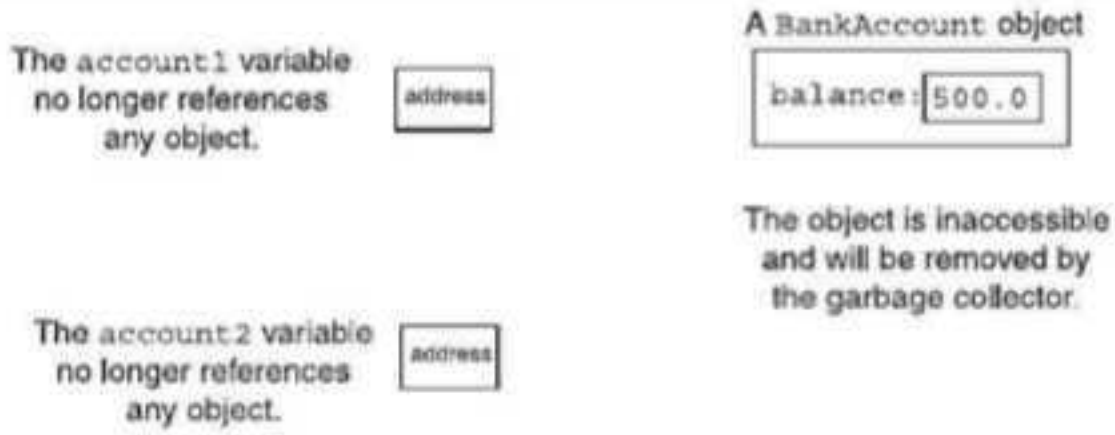
Next, the `null` value is assigned to `account1`. This removes the address of the object from the `account1` variable, causing it to no longer reference the object. Figure 8-18 illustrates this.

Figure 8-18 The object is only referenced by the `account2` variable



The object is still accessible, however, because it is referenced by the `account2` variable. The next statement assigns `null` to `account2`. This removes the object's address from `account2`, causing it to no longer reference the object. Figure 8-19 illustrates this. Because the object is no longer accessible, it will be removed from memory the next time the garbage collector process runs.

Figure 8-19 The object is no longer referenced



The `finalize` Method

If a class has a method named `finalize`, it is called automatically just before an instance of the class is destroyed by the garbage collector. If you wish to execute code just before an object is destroyed, you can create a `finalize` method in the class and place the code there. The `finalize` method accepts no arguments and has a `void` return type.

NOTE: The garbage collector runs periodically, and you cannot predict exactly when it will execute. Therefore, you cannot know exactly when an object's finalize method will execute.

8.11 Focus on Object-Oriented Design: Class Collaboration

CONCEPT: **It is common for classes to interact, or collaborate, with each other to perform their operations. Part of the object-oriented design process is identifying the collaborations among classes.**

In an object-oriented application it is common for objects of different classes to collaborate. This simply means that objects interact with each other. Sometimes one object will need the services of another object in order to fulfill its responsibilities. For example, let's say an object needs to read a number from a file and then format the number to appear as a dollar amount, so it can be displayed in a message dialog. The object might use the services of a `Scanner` object to read the number from the file, and then use the services of a `DecimalFormat` object to format the number. In this example, the object is collaborating with objects created from classes in the Java API. The objects that you create from your own classes can also collaborate with each other.

If one object is to collaborate with another object, then it must know something about the other object's class methods and how to call them. For example, suppose we were to write a class named `StockPurchase`, which uses an object of the `Stock` class (presented earlier in this chapter) to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, it must know how to call the `Stock` class's `getSharePrice` method to get the price per share of the stock. Code Listing 8-24 shows an example of the `StockPurchase` class. (This file is in the source code folder *Chapter 08\StockPurchase Class*.)

Code Listing 8-24 (`StockPurchase.java`)

```
1 /**
2    The StockPurchase class represents a stock purchase.
3 */
4
5 public class StockPurchase
6 {
7    private Stock stock;  // The stock that was purchased
```

```
8     private int shares; // Number of shares owned
9
10    /**
11       Constructor
12       @param stockObject The stock to purchase.
13       @param numShares The number of shares.
14    */
15
16    public StockPurchase(Stock stockObject, int numShares)
17    {
18       // Create a copy of the object referenced by
19       // stockObject.
20       stock = new Stock(stockObject);
21       shares = numShares;
22    }
23
24    /**
25       getStock method
26       @return A copy of the Stock object for the stock
27               being purchased.
28    */
29
30    public Stock getStock()
31    {
32       // Return a copy of the object referenced by stock.
33       return new Stock(stock);
34    }
35
36    /**
37       getShares method
38       @return The number of shares being purchased.
39    */
40
41    public int getShares()
42    {
43       return shares;
44    }
45
46    /**
47       getCost method
48       @return The cost of the stock purchase.
49    */
50
51    public double getCost()
52    {
53       return shares * stock.getSharePrice();
54    }
55 }
```

The constructor for this class accepts a `Stock` object representing the stock that is being purchased, and an `int` representing the number of shares to purchase. In line 20 we see the first collaboration: The `StockPurchase` constructor makes a copy of the `Stock` object by using the `Stock` class's copy constructor. The copy constructor is used again in the `getStock` method, in line 33, to return a copy of the `Stock` object.

The next collaboration takes place in the `getCost` method. This method calculates and returns the cost of the stock purchase. In line 53 it calls the `Stock` class's `getSharePrice` method to determine the stock's price per share. The program in Code Listing 8-25 demonstrates this class. (This file is also stored in the source code folder *Chapter 08\StockPurchase Class.*)

Code Listing 8-25 (StockTrader.java)

```
1  import java.util.Scanner;
2
3  /**
4     This program allows you to purchase shares of XYZ
5     company's stock.
6  */
7
8  public class StockTrader
9  {
10    public static void main(String[] args)
11    {
12       int sharesToBuy;  // Number of shares to buy.
13
14       // Create a Stock object for the company stock.
15       // The trading symbol is XYZ and the stock is
16       // currently $9.62 per share.
17       Stock xyzCompany = new Stock("XYZ", 9.62);
18
19       // Create a Scanner object for keyboard input.
20       Scanner keyboard = new Scanner(System.in);
21
22       // Display the current share price.
23       System.out.printf("XYZ stock is currently $%,.2f.\n",
24                         xyzCompany.getSharePrice());
25
26       // Get the number of shares to purchase.
27       System.out.print("How many shares do you want to buy? ");
28       sharesToBuy = keyboard.nextInt();
29
30       // Create a StockPurchase object for the transaction.
31       StockPurchase buy =
32                new StockPurchase(xyzCompany, sharesToBuy);
```

```
33
34       // Display the cost of the transaction.
35       System.out.printf("Cost of the stock: $%,.2f",
36                         buy.getCost());
37    }
38 }
```

Program Output with Example Input Shown in Bold

```
XYZ stock is currently $9.62.
How many shares do you want to buy? 100 [Enter]
Cost of the stock: $962.00
```

Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary among classes by examining the responsibilities of the classes. In Chapter 6, Section 6.9, we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are as follows:

- Things that the class is responsible for knowing
- Actions that the class is responsible for doing

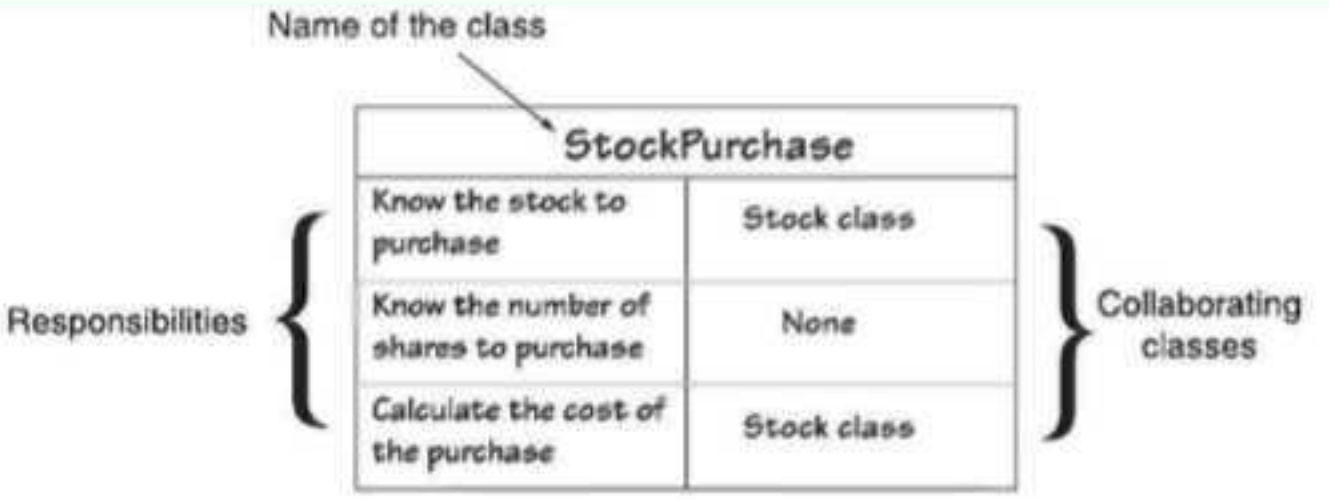
Often you will determine that the class must collaborate with another class in order to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. CRC stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which is discussed in Chapter 6, Section 6.9), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as the following:

- Will an object of this class need to get data from another object in order to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Figure 8-20 shows an example CRC card for the `StockPurchase` class.

Figure 8-20 CRC card



From the CRC card shown in the figure, we can see that the `StockPurchase` class has the following responsibilities and collaborations:

- **Responsibility:** To know the stock to purchase
  Collaboration: The `Stock` class
- **Responsibility:** To know the number of shares to purchase
  Collaboration: None
- **Responsibility:** To calculate the cost of the purchase
  Collaboration: The `Stock` class

When you have completed a CRC card for each class in the application, you will have a good idea of each class's responsibilities and how the classes must interact.

8.12 Common Errors to Avoid

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

- **Attempting to refer to an instance field or instance method in a static method.** Static methods can refer only to other class members that are static.
- **In a method that accepts an object as an argument, writing code that accidentally modifies the object.** When a reference variable is passed as an argument to a method, the method has access to the object that the variable references. When writing a method that receives 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.
- **Allowing a `null` reference to be used.** 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. This can happen when a class has a reference variable as a field, and it is not properly initialized with the address of an object.
- **Forgetting to use the fully qualified name of an `enum` constant.** Under most circumstances you must use the fully qualified name of an `enum` constant. One exception to this is when the `enum` constant is used as a case expression in a switch statement.

Review Questions and Exercises

Multiple Choice and True/False

1. This type of method cannot access any non-static member variables in its own class.
   a. instance
   b. `void`
   c. static
   d. non-static
2. When an object is passed as an argument to a method, this is actually passed.
   a. a copy of the object
   b. the name of the object
   c. a reference to the object
   d. none of these; you cannot pass an object
3. If you write this method for a class, Java will automatically call it any time you concatenate an object of the class with a string.
   a. `toString`
   b. `plusString`
   c. `stringConvert`
   d. `concatString`
4. Making an instance of one class a field in another class is called _____.
   a. nesting
   b. class fielding
   c. aggregation
   d. concatenation
5. This is the name of a reference variable that is always available to an instance method and refers to the object that is calling the method.
   a. `callingObject`
   b. `this`
   c. `me`
   d. `instance`
6. This `enum` method returns the position of an `enum` constant in the declaration.
   a. `position`
   b. `location`
   c. `ordinal`
   d. `toString`
7. Assuming the following declaration exists:

   ```
   enum Seasons { SPRING, WINTER, SUMMER, FALL }
   ```

   what is the fully qualified name of the `FALL` constant?
   a. `FALL`
   b. `enum.FALL`
   c. `FALL.Seasons`
   d. `Seasons.FALL`

8. You cannot use the fully qualified name of an `enum` constant for this.
   a. a `switch` expression
   b. a `case` expression
   c. an argument to a method
   d. all of these
9. The Java Virtual Machine periodically performs this process, which automatically removes unreferenced objects from memory.
   a. memory cleansing
   b. memory deallocation
   c. garbage collection
   d. object expungement
10. If a class has this method, it is called automatically just before an instance of the class is destroyed by the Java Virtual Machine.
    a. `finalize`
    b. `destroy`
    c. `remove`
    d. `housekeeper`
11. CRC stands for
    a. Class, Return value, Composition
    b. Class, Responsibilities, Collaborations
    c. Class, Responsibilities, Composition
    d. Compare, Return, Continue
12. **True or False:** A static member method may refer to non-static member variables of the same class, but only after an instance of the class has been defined.
13. **True or False:** All static member variables are initialized to –1 by default.
14. **True or False:** When an object is passed as an argument to a method, the method can access the argument.
15. **True or False:** A method cannot return a reference to an object.
16. **True or False:** You can declare an enumerated data type inside a method.
17. **True or False:** Enumerated data types are actually special types of classes.
18. **True or False:** `enum` constants have a `toString` method.

Find the Error

The following class definition has an error. What is it?

```
1.  public class MyClass
    {
          private int x;
          private double y;

          public static void setValues(int a, double b)
          {
```

```
            x = a;
            y = b;
        }
    }
```

2. Assume the following declaration exists :

```
enum Coffee { MEDIUM, DARK, DECAF }
```

Find the error(s) in the following switch statement:

```
// This code has errors!
Coffee myCup = DARK;
switch (myCup)
{
    case Coffee.MEDIUM :
          System.out.println("Mild flavor.");
          break;
    case Coffee.DARK :
          System.out.println("Strong flavor.");
          break;
    case Coffee.DECAF :
          System.out.println("Won't keep you awake.");
          break;
    default:
          System.out.println("Never heard of it.");
}
```

Algorithm Workbench

1. Consider the following class declaration:

```
public class Circle
{
   private double radius;

   public Circle(double r)
   {
      radius = r;
   }

   public double getArea()
   {
      return Math.PI * radius * radius;
   }

   public double getRadius()
   {
      return radius;
   }
}
```

a. Write a `toString` method for this class. The method should return a string containing the radius and area of the circle.
b. Write an `equals` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object contains the same data as the calling object, or `false` otherwise.
c. Write a `greaterThan` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object has an area that is greater than the area of the calling object, or `false` otherwise.

2. Consider the following class declaration:

```
public class Thing
{
     private int x;
     private int y;
     private static int z = 0;

     public Thing()
     {
          x = z;
          y = z;
     }
     static void putThing(int a)
     {
          z = a;
     }
}
```

Assume a program containing the class declaration defines three `Thing` objects with the following statements:

```
Thing one = new Thing();
Thing two = new Thing();
Thing three = new Thing();
```

a. How many separate instances of the `x` member exist?
b. How many separate instances of the `y` member exist?
c. How many separate instances of the `z` member exist?
d. What value will be stored in the `x` and `y` members of each object?
e. Write a statement that will call the `putThing` method.

3. A pet store sells dogs, cats, birds, and hamsters. Write a declaration for an enumerated data type that can represent the types of pets the store sells.

Short Answer

1. Describe one thing you cannot do with a static method.
2. Why are static methods useful in creating utility classes?
3. Describe the difference in the way variables and class objects are passed as arguments to a method.

4. Even if you do not write an `equals` method for a class, Java provides one. Describe the behavior of the `equals` method that Java automatically provides.
5. A "has a" relationship can exist between classes. What does this mean?
6. What happens if you attempt to call a method using a reference variable that is set to `null`?
7. Is it advisable or not advisable to write a method that returns a reference to an object that is a private field? What is the exception to this?
8. What is the `this` key word?
9. Look at the following declaration:

```
enum Color { RED, ORANGE, GREEN, BLUE }
```

   a. What is the name of the data type declared by this statement?
   b. What are the `enum` constants for this type?
   c. Write a statement that defines a variable of this type and initializes it with a valid value.

10. Assuming the following `enum` declaration exists:

```
enum Dog { POODLE, BOXER, TERRIER }
```

what will the following statements display?

```
a. System.out.println(Dog.POODLE + "\n" +
                      Dog.BOXER + "\n" +
                      Dog.TERRIER);

b. System.out.println(Dog.POODLE.ordinal() + "\n" +
                      Dog.BOXER.ordinal() + "\n" +
                      Dog.TERRIER.ordinal());

c. Dog myDog = Dog.BOXER;
   if (myDog.compareTo(Dog.TERRIER) > 0)
       System.out.println(myDog + " is greater than " +
                          Dog.TERRIER);
   else
      System.out.println(myDog + " is NOT greater than " +
                         Dog.TERRIER);
```

11. Under what circumstances does an object become a candidate for garbage collection?

Programming Challenges

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

1. Area Class

Write a class that has three overloaded static methods for calculating the areas of the following geometric shapes:

- circles
- rectangles
- cylinders

Here are the formulas for calculating the area of the shapes.

Area of a circle:	$Area = \pi r^2$
where	π is `Math.PI` and r is the circle's radius
Area of a rectangle:	$Area = Width \times Length$
Area of a cylinder:	$Area = \pi r^2 h$
where	π is `Math.PI`, r is the radius of the cylinder's base, and h is the cylinder's height

Because the three methods are to be overloaded, they should each have the same name, but different parameter lists. Demonstrate the class in a complete program.

VideoNote

The `BankAccount` Class Copy Constructor Problem

2. `BankAccount` Class Copy Constructor

Add a copy constructor to the `BankAccount` class. This constructor should accept a `BankAccount` object as an argument. It should assign to the `balance` field the value in the argument's `balance` field. As a result, the new object will be a copy of the argument object.

3. Carpet Calculator

The Westfield Carpet Company has asked you to write an application that calculates the price of carpeting for rectangular rooms. To calculate the price, you multiply the area of the floor (width times length) by the price per square foot of carpet. For example, the area of floor that is 12 feet long and 10 feet wide is 120 square feet. To cover that floor with carpet that costs $8 per square foot would cost $960. ($12 \times 10 \times 8 = 960$.)

First, you should create a class named `RoomDimension` that has two fields: one for the length of the room and one for the width. The `RoomDimension` class should have a method that returns the area of the room. (The area of the room is the room's length multiplied by the room's width.)

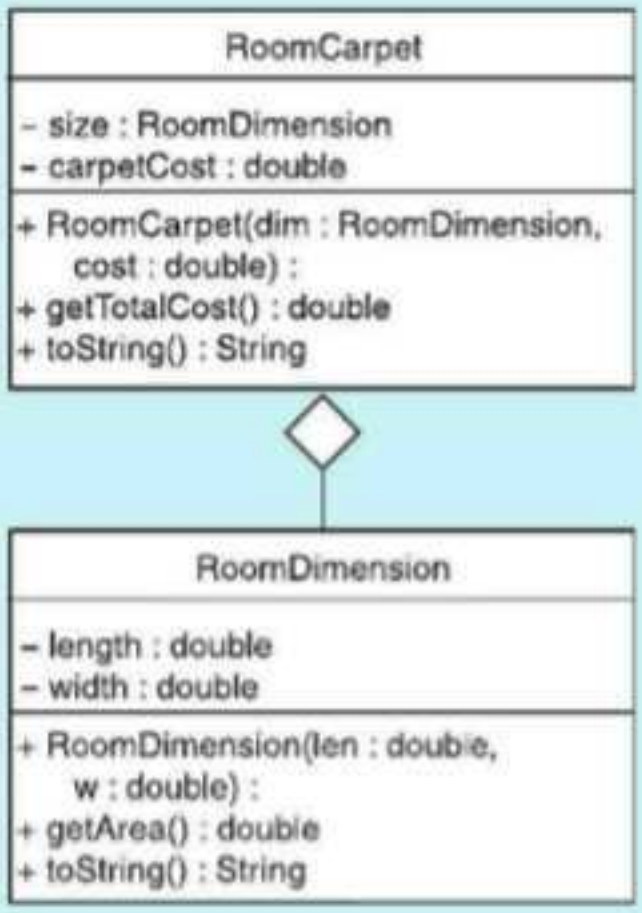
Next you should create a `RoomCarpet` class that has a `RoomDimension` object as a field. It should also have a field for the cost of the carpet per square foot. The `RoomCarpet` class should have a method that returns the total cost of the carpet.

Figure 8-21 is a UML diagram that shows possible class designs and the relationships among the classes. Once you have written these classes, use them in an application that asks the user to enter the dimensions of a room and the price per square foot of the desired carpeting. The application should display the total cost of the carpet.

4. `LandTract` Class

Make a `LandTract` class that has two fields: one for the tract's length and one for the width. The class should have a method that returns the tract's area, as well as an `equals` method and a `toString` method. Demonstrate the class in a program that asks the user to enter the dimensions for two tracts of land. The program should display the area of each tract of land and indicate whether the tracts are of equal size.

Figure 8-21 UML diagram for Programming Challenge 3



5. `Month` Class

Write a class named `Month`. The class should have an `int` field named `monthNumber` that holds the number of the month. For example, January would be 1, February would be 2, and so forth. In addition, provide the following methods:

- A no-arg constructor that sets the `monthNumber` field to 1.
- A constructor that accepts the number of the month as an argument. It should set the `monthNumber` field to the value passed as the argument. If a value less than 1 or greater than 12 is passed, the constructor should set `monthNumber` to 1.
- A constructor that accepts the name of the month, such as "January" or "February" as an argument. It should set the `monthNumber` field to the correct corresponding value.
- A `setMonthNumber` method that accepts an `int` argument, which is assigned to the `monthNumber` field. If a value less than 1 or greater than 12 is passed, the method should set `monthNumber` to 1.
- A `getMonthNumber` method that returns the value in the `monthNumber` field.
- A `getMonthName` method that returns the name of the month. For example, if the `monthNumber` field contains 1, then this method should return "January".
- A `toString` method that returns the same value as the `getMonthName` method.
- An `equals` method that accepts a `Month` object as an argument. If the argument object holds the same data as the calling object, this method should return `true`. Otherwise, it should return `false`.

- A `greaterThan` method that accepts a `Month` object as an argument. If the calling object's `monthNumber` field is greater than the argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.
- A `lessThan` method that accepts a `Month` object as an argument. If the calling object's `monthNumber` field is less than the argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.

6. `CashRegister` Class

Write a `CashRegister` class that can be used with the `RetailItem` class that you wrote in Chapter 6's Programming Challenge 4. The `CashRegister` class should simulate the sale of a retail item. It should have a constructor that accepts a `RetailItem` object as an argument. The constructor should also accept an integer that represents the quantity of items being purchased. In addition, the class should have the following methods:

- The `getSubtotal` method should return the subtotal of the sale, which is the quantity multiplied by the price. This method must get the price from the `RetailItem` object that was passed as an argument to the constructor.
- The `getTax` method should return the amount of sales tax on the purchase. The sales tax rate is 6 percent of a retail sale.
- The `getTotal` method should return the total of the sale, which is the subtotal plus the sales tax.

Demonstrate the class in a program that asks the user for the quantity of items being purchased, and then displays the sale's subtotal, amount of sales tax, and total.

7. Sales Receipt File

Modify the program you wrote in Programming Challenge 6 to create a file containing a sales receipt. The program should ask the user for the quantity of items being purchased, and then generate a file with contents similar to the following:

```
SALES RECEIPT
Unit Price: $10.00
Quantity: 5
Subtotal: $50.00
Sales Tax: $ 3.00
Total: $53.00
```

8. Parking Ticket Simulator

For this assignment you will design a set of classes that work together to simulate a police officer issuing a parking ticket. You should design the following classes:

- **The `ParkedCar` Class:** This class should simulate a parked car. The class's responsibilities are as follows:
  - To know the car's make, model, color, license number, and the number of minutes that the car has been parked.
- **The `ParkingMeter` Class:** This class should simulate a parking meter. The class's only responsibility is as follows:
  - To know the number of minutes of parking time that has been purchased.

- **The `ParkingTicket` Class:** This class should simulate a parking ticket. The class's responsibilities are as follows:
  - To report the make, model, color, and license number of the illegally parked car
  - To report the amount of the fine, which is $25 for the first hour or part of an hour that the car is illegally parked, plus $10 for every additional hour or part of an hour that the car is illegally parked
  - To report the name and badge number of the police officer issuing the ticket
- **The `PoliceOfficer` Class:** This class should simulate a police officer inspecting parked cars. The class's responsibilities are as follows:
  - To know the police officer's name and badge number
  - To examine a `ParkedCar` object and a `ParkingMeter` object, and determine whether the car's time has expired
  - To issue a parking ticket (generate a `ParkingTicket` object) if the car's time has expired

Write a program that demonstrates how these classes collaborate.

9. Geometry Calculator

Design a `Geometry` class with the following methods:

- A static method that accepts the radius of a circle and returns the area of the circle. Use the following formula:
  $Area = \pi r^2$
  Use `Math.PI` for π and the radius of the circle for r.
- A static method that accepts the length and width of a rectangle and returns the area of the rectangle. Use the following formula:
  $Area = Length \times Width$
- A static method that accepts the length of a triangle's base and the triangle's height. The method should return the area of the triangle. Use the following formula:
  $Area = Base \times Height \times 0.5$

The methods should display an error message if negative values are used for the circle's radius, the rectangle's length or width, or the triangle's base or height.

Next, write a program to test the class, which displays the following menu and responds to the user's selection:

```
Geometry Calculator
1. Calculate the Area of a Circle
2. Calculate the Area of a Rectangle
3. Calculate the Area of a Triangle
4. Quit

Enter your choice (1-4):
```

Display an error message if the user enters a number outside the range of 1 through 4 when selecting an item from the menu.

10. Car Instrument Simulator

For this assignment, you will design a set of classes that work together to simulate a car's fuel gauge and odometer. The classes you will design are the following:

- **The `FuelGauge` Class:** This class will simulate a fuel gauge. Its responsibilities are as follows:
  - To know the car's current amount of fuel, in gallons.
  - To report the car's current amount of fuel, in gallons.
  - To be able to increment the amount of fuel by 1 gallon. This simulates putting fuel in the car. (The car can hold a maximum of 15 gallons.)
  - To be able to decrement the amount of fuel by 1 gallon, if the amount of fuel is greater than 0 gallons. This simulates burning fuel as the car runs.
- **The `Odometer` Class:** This class will simulate the car's odometer. Its responsibilities are as follows:
  - To know the car's current mileage.
  - To report the car's current mileage.
  - To be able to increment the current mileage by 1 mile. The maximum mileage the odometer can store is 999,999 miles. When this amount is exceeded, the odometer resets the current mileage to 0.
  - To be able to work with a `FuelGauge` object. It should decrease the `FuelGauge` object's current amount of fuel by 1 gallon for every 24 miles traveled. (The car's fuel economy is 24 miles per gallon.)

Demonstrate the classes by creating instances of each. Simulate filling the car up with fuel, and then run a loop that increments the odometer until the car runs out of fuel. During each loop iteration, print the car's current mileage and amount of fuel.

CHAPTER 9

Text Processing and More about Wrapper Classes

TOPICS

9.1 Introduction to Wrapper Classes

CONCEPT: Java provides wrapper classes for the primitive data types. The wrapper class for a given primitive type contains not only a value of that type, but also methods that perform operations related to the type.

Recall from Chapter 2 that the primitive data types are called "primitive" because they are not created from classes. Instead of instantiating objects, you create variables from the primitive data types, and variables do not have attributes or methods. They are designed simply to hold a single value in memory.

Java also provides wrapper classes for all of the primitive data types. A *wrapper class* is a class that is "wrapped around" a primitive data type and allows you to create objects instead of variables. In addition, these wrapper classes provide methods that perform useful operations on primitive values. For example, you have already used the wrapper class "parse" methods to convert strings to primitive values.

Although these wrapper classes can be used to create objects instead of variables, few programmers use them that way. One reason is because the wrapper classes are immutable, which means that once you create an object, you cannot change the object's value. Another reason is because they are not as easy to use as variables for simple operations. For example, to get the value stored in an object you must call a method, whereas variables can be used directly in assignment statements, used in mathematical operations, passed as arguments to methods, and so forth.

Although it is not normally useful to create objects from the wrapper classes, they do provide static methods that are very useful. We examine several of Java's wrapper classes in this chapter. We begin by looking at the `Character` class, which is the wrapper class for the `char` data type.

9.2 Character Testing and Conversion with the `Character` Class

CONCEPT: The `Character` class is a wrapper class for the `char` data type. It provides numerous methods for testing and converting character data.

The `Character` class is part of the `java.lang` package, so no `import` statement is necessary to use this class. The class provides several static methods for testing the value of a `char` variable. Some of these methods are listed in Table 9-1. Each of the methods accepts a single `char` argument and returns a `boolean` value.

Table 9-1 Some static `Character` class methods for testing `char` values

Method	Description
`boolean isDigit(char ch)`	Returns `true` if the argument passed into *ch* is a digit from 0 through 9. Otherwise returns `false`.
`boolean isLetter(char ch)`	Returns `true` if the argument passed into *ch* is an alphabetic letter. Otherwise returns `false`.
`boolean isLetterOrDigit(char ch)`	Returns `true` if the character passed into *ch* contains a digit (0 through 9) or an alphabetic letter. Otherwise returns `false`.
`boolean isLowerCase(char ch)`	Returns `true` if the argument passed into *ch* is a lowercase letter. Otherwise returns `false`.
`boolean isUpperCase(char ch)`	Returns `true` if the argument passed into *ch* is an uppercase letter. Otherwise returns `false`.
`boolean isSpaceChar(char ch)`	Returns `true` if the argument passed into *ch* is a space character. Otherwise returns `false`.
`boolean isWhiteSpace(char ch)`	Returns `true` if the argument passed into *ch* is a whitespace character (a space, tab, or newline character). Otherwise returns `false`.

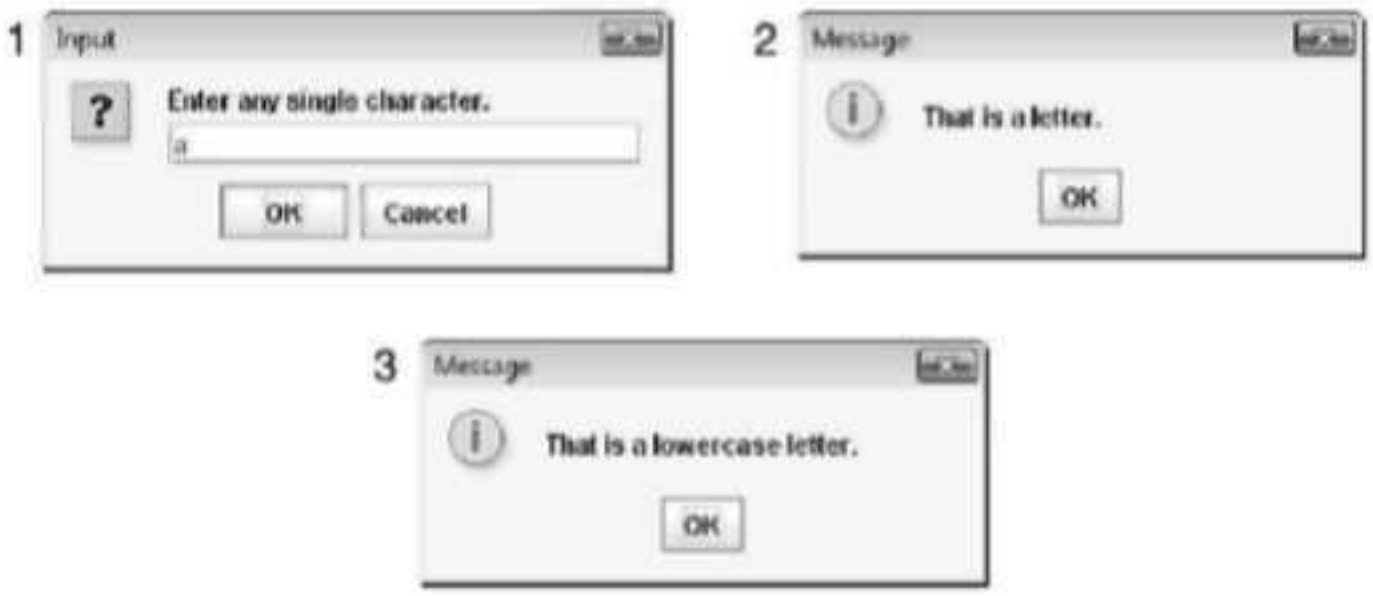
The program in Code Listing 9-1 demonstrates many of these methods. Figures 9-1 and 9-2 show example interactions with the program.

Code Listing 9-1 (CharacterTest.java)

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program demonstrates some of the Character
5     class's character testing methods.
6  */
7
8  public class CharacterTest
9  {
10    public static void main(String[] args)
11    {
12       String input;  // To hold the user's input
13       char ch;       // To hold a single character
14
15       // Get a character from the user and store
16       // it in the ch variable.
17       input = JOptionPane.showInputDialog("Enter " +
18                                  "any single character.");
19       ch = input.charAt(0);
20
21       // Test the character.
22       if (Character.isLetter(ch))
23       {
24          JOptionPane.showMessageDialog(null,
25                             "That is a letter.");
26       }
27
28       if (Character.isDigit(ch))
29       {
30          JOptionPane.showMessageDialog(null,
31                              "That is a digit.");
32       }
33
34       if (Character.isLowerCase(ch))
35       {
36          JOptionPane.showMessageDialog(null,
37                "That is a lowercase letter.");
38       }
39
40       if (Character.isUpperCase(ch))
41       {
42          JOptionPane.showMessageDialog(null,
43               "That is an uppercase letter.");
44       }
45
```

```
46        if (Character.isSpaceChar(ch))
47        {
48           JOptionPane.showMessageDialog(null,
49                                "That is a space.");
50        }
51
52        if (Character.isWhitespace(ch))
53        {
54           JOptionPane.showMessageDialog(null,
55            "That is a whitespace character.");
56        }
57
58        System.exit(0);
59     }
60 }
```

Figure 9-1 Interaction with the `CharacterTest.java` program



Code Listing 9-2 shows a more practical application of the character testing methods. It tests a string to determine whether it is a seven-character customer number in the proper format. Figures 9-3 and 9-4 show example interactions with the program.

Figure 9-2 Interaction with the `CharacterTest.java` program

Code Listing 9-2 (CustomerNumber.java)

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program tests a customer number to
5     verify that it is in the proper format.
6  */
7
8  public class CustomerNumber
9  {
10    public static void main(String[] args)
11    {
12       String input;  // To hold the user's input
13
14       // Get a customer number.
15       input = JOptionPane.showInputDialog("Enter " +
16          "a customer number in the form LLLNNNN\n" +
17          "(LLL = letters and NNNN = numbers)");
18
19       // Validate the input.
20       if (isValid(input))
21       {
22          JOptionPane.showMessageDialog(null,
23                   "That's a valid customer number.");
24       }
25       else
26       {
27          JOptionPane.showMessageDialog(null,
28              "That is not the proper format of a " +
29              "customer number.\nHere is an " +
30              "example: ABC1234");
31       }
32
33       System.exit(0);
34    }
35
36    /**
37       The isValid method determines whether a
38       String is a valid customer number. If so, it
39       returns true.
40       @param custNumber The String to test.
41       @return true if valid, otherwise false.
42    */
43
44    private static boolean isValid(String custNumber)
45    {
```

```
46        boolean goodSoFar = true;    // Flag
47        int i = 0;                   // Control variable
48
49        // Test the length.
50        if (custNumber.length() != 7)
51           goodSoFar = false;
52
53        // Test the first three characters for letters.
54        while (goodSoFar && i < 3)
55        {
56           if (!Character.isLetter(custNumber.charAt(i)))
57              goodSoFar = false;
58           i++;
59        }
60
61        // Test the last four characters for digits.
62        while (goodSoFar && i < 7)
63        {
64           if (!Character.isDigit(custNumber.charAt(i)))
65              goodSoFar = false;
66           i++;
67        }
68
69        return goodSoFar;
70     }
71  }
```

Figure 9-3 Interaction with the `CustomerNumber.java` program

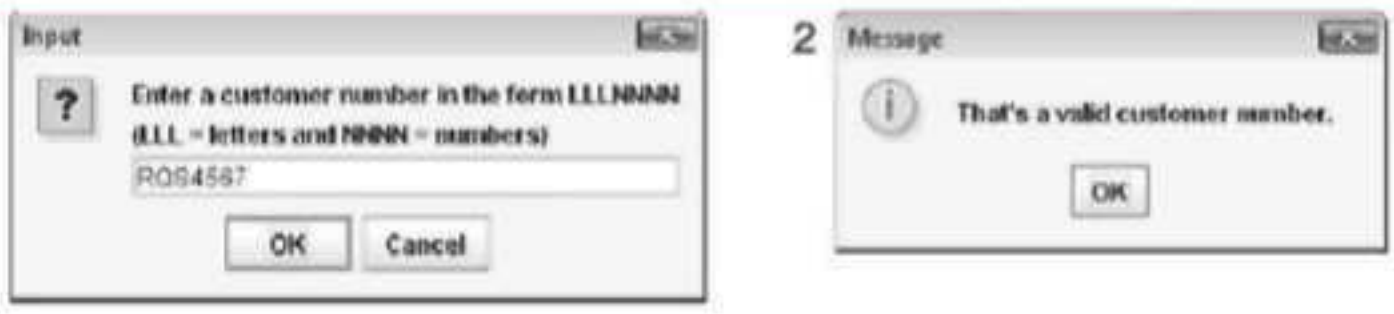


Figure 9-4 Interaction with the `CustomerNumber.java` program

In this program, the customer number is expected to be seven characters long and consist of three alphabetic letters followed by four numeric digits. The `isValid` method accepts a `String` argument, which will be tested. The method uses the following local variables, which are declared in lines 46 and 47:

```
boolean goodSoFar = true;  // Flag
int i = 0;                 // Control variable
```

The `goodSoFar` variable is a flag variable that is initialized with `true`, but will be set to `false` immediately when the method determines the customer number is not in a valid format. The `i` variable is a loop control variable.

The first test is to determine whether the string is the correct length. In line 50 the method tests the length of the `custNumber` argument. If the argument is not seven characters long, it is not valid and the `goodSoFar` variable is set to `false` in line 51.

Next, the method uses the following loop, in lines 54 through 59, to validate the first three characters:

```
while (goodSoFar && i < 3)
{
   if (!Character.isLetter(custNumber.charAt(i)))
      goodSoFar = false;
   i++;
}
```

Recall from Chapter 2 that the `String` class's `charAt` method returns a character at a specific position in a string (position numbering starts at 0). This code uses the `Character.isLetter` method to test the characters at positions 0, 1, and 2 in the `custNumber` string. If any of these characters are not letters, the `goodSoFar` variable is set to `false` and the loop terminates. Next, the method uses the following loop, in lines 62 through 67, to validate the last four characters:

```
while (goodSoFar && i < 7)
{
   if (!Character.isDigit(custNumber.charAt(i)))
      goodSoFar = false;
   i++;
}
```

This code uses the `Character.isDigit` method to test the characters at positions 3, 4, 5, and 6 in the `custNumber` string. If any of these characters are not digits, the `goodSoFar` variable is set to `false` and the loop terminates. Last, the method returns the value of the `goodSoFar` method.

Character Case Conversion

The `Character` class also provides the static methods listed in Table 9-2 for converting the case of a character. Each method accepts a `char` argument and returns a `char` value.

Table 9-2 Some `Character` class methods for case conversion

Method	Description
`char toLowerCase(char ch)`	Returns the lowercase equivalent of the argument passed to *ch*.
`char toUpperCase(char ch)`	Returns the uppercase equivalent of the argument passed to *ch*.

If the `toLowerCase` method's argument is an uppercase character, the method returns the lowercase equivalent. For example, the following statement will display the character `a` on the screen:

```
System.out.println(Character.toLowerCase('A'));
```

If the argument is already lowercase, the `toLowerCase` method returns it unchanged. The following statement also causes the lowercase character `a` to be displayed:

```
System.out.println(Character.toLowerCase('a'));
```

If the `toUpperCase` method's argument is a lowercase character, the method returns the uppercase equivalent. For example, the following statement will display the character `A` on the screen:

```
System.out.println(Character.toUpperCase('a'));
```

If the argument is already uppercase, the `toUpperCase` method returns it unchanged.

Any non-letter argument passed to `toLowerCase` or `toUpperCase` is returned as it is. Each of the following statements displays the method argument without any change:

```
System.out.println(Character.toLowerCase('*'));
System.out.println(Character.toLowerCase('$'));
System.out.println(Character.toUpperCase('&'));
System.out.println(Character.toUpperCase('%'));
```

The program in Code Listing 9-3 demonstrates the `toUpperCase` method in a loop that asks the user to enter Y or N. The program repeats as long as the user enters Y or y in response to the question.

Code Listing 9-3 (`CircleArea.java`)

```
1 import java.util.Scanner;
2
3 /**
4    This program demonstrates the Character
5    class's toUpperCase method.
6 */
7
```

```
 8 public class CircleArea
 9 {
10    public static void main(String[] args)
11    {
12       double radius; // The circle's radius
13       double area;   // The circle's area
14       String input;  // To hold a line of input
15       char choice;   // To hold a single character
16
17       // Create a Scanner object to read keyboard input.
18       Scanner keyboard = new Scanner(System.in);
19
20       do
21       {
22          // Get the circle's radius.
23          System.out.print("Enter the circle's radius: ");
24          radius = keyboard.nextDouble();
25
26          // Consume the remaining newline character.
27          keyboard.nextLine();
28
29          // Calculate and display the area.
30          area = Math.PI * radius * radius;
31          System.out.printf("The area is %.2f.\n", area);
32
33          // Repeat this?
34          System.out.print("Do you want to do this " +
35                           "again? (Y or N) ");
36          input = keyboard.nextLine();
37          choice = input.charAt(0);
38
39       } while (Character.toUpperCase(choice) == 'Y');
40    }
41 }
```

Program Output with Example Input Shown in Bold

```
Enter the circle's radius: 10 [Enter]
The area is 314.16.
Do you want to do this again? (Y or N) y [Enter]
Enter the circle's radius: 15 [Enter]
The area is 706.86.
Do you want to do this again? (Y or N) n [Enter]
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.1 Write a statement that converts the contents of the `char` variable `big` to lowercase. The converted value should be assigned to the variable `little`.

9.2 Write an `if` statement that displays the word "digit" if the `char` variable `ch` contains a numeric digit. Otherwise, it should display "Not a digit."

9.3 What is the output of the following statement?

```
System.out.println(Character.toUpperCase(Character.toLowerCase('A')));
```

9.4 Write a loop that asks the user, "Do you want to repeat the program or quit? (R/Q)". The loop should repeat until the user has entered an R or Q (either uppercase or lowercase).

9.5 What will the following code display?

```
char var = '$';
System.out.println(Character.toUpperCase(var));
```

9.6 Write a loop that counts the number of uppercase characters that appear in the `String` object `str`.

9.3 More `String` Methods

CONCEPT: The `String` class provides several methods for searching and working with `String` objects.

Searching for Substrings

The `String` class provides several methods that search for a string inside of a string. The term *substring* commonly is used to refer to a string that is part of another string. Table 9-3 summarizes some of these methods. Each of the methods in Table 9-3 returns a `boolean` value indicating whether the string was found.

Let's take a closer look at each of these methods.

The `startsWith` and `endsWith` Methods

The `startsWith` method determines whether the calling object's string begins with a specified substring. For example, the following code determines whether the string "Four score and seven years ago" begins with "Four". The method returns `true` if the string begins with the specified substring, or `false` otherwise.

```
String str = "Four score and seven years ago";
if (str.startsWith("Four"))
   System.out.println("The string starts with Four.");
else
   System.out.println("The string does not start with Four.");
```

Table 9-3 String methods that search for a substring

Method	Description
`boolean startsWith(String str)`	This method returns `true` if the calling string begins with the string passed into *`str`*.
`boolean endsWith(String str)`	This method returns `true` if the calling string ends with the string passed into *`str`*.
`boolean regionMatches(int start, String str, int start2, int n)`	This method returns `true` if a specified region of the calling string matches a specified region of the string passed into *`str`*. The *`start`* parameter indicates the starting position of the region within the calling string. The *`start2`* parameter indicates the starting position of the region within *`str`*. The *`n`* parameter indicates the number of characters in both regions.
`boolean regionMatches(Boolean ignoreCase, int start, String str, int start2, int n)`	This overloaded version of the *`regionMatches`* method has an additional parameter, *`ignoreCase`*. If `true` is passed into this parameter, the method ignores the case of the calling string and *`str`* when comparing the regions. If `false` is passed into the *`ignoreCase`* parameter, the comparison is case-sensitive.

In the code, the method call `str.startsWith("Four")` returns `true` because the string does begin with "Four". The `startsWith` method performs a case-sensitive comparison, so the method call `str.startsWith("four")` would return `false`.

The `endsWith` method determines whether the calling string ends with a specified substring. For example, the following code determines whether the string "Four score and seven years ago" ends with "ago". The method returns `true` if the string does end with the specified substring or `false` otherwise.

```
String str = "Four score and seven years ago";
if (str.endsWith("ago"))
   System.out.println("The string ends with ago.");
else
   System.out.println("The string does not end with ago.");
```

In the code, the method call `str.endsWith("ago")` returns `true` because the string does end with "ago". The `endsWith` method also performs a case-sensitive comparison, so the method call `str.endsWith("Ago")` would return `false`.

The program in Code Listing 9-4 demonstrates a search algorithm that uses the `startsWith` method. The program searches an array of strings for an element that starts with a specified string.

Code Listing 9-4 (PersonSearch.java)

```
1 import java.util.Scanner;
2
3 /**
4    This program uses the startsWith method to search using
5    a partial string.
6 */
7
8 public class PersonSearch
9 {
10    public static void main(String[] args)
11    {
12       String lookUp;  // To hold a lookup string
13
14       // Create an array of names.
15       String[] people = { "Cutshaw, Will", "Davis, George",
16                           "Davis, Jenny", "Russert, Phil",
17                           "Russell, Cindy", "Setzer, Charles",
18                           "Smathers, Holly", "Smith, Chris",
19                           "Smith, Brad", "Williams, Jean" };
20
21       // Create a Scanner object for keyboard input.
22       Scanner keyboard = new Scanner(System.in);
23
24       // Get a partial name to search for.
25       System.out.print("Enter the first few characters of " +
26                        "the last name to look up: ");
27       lookUp = keyboard.nextLine();
28
29       // Display all of the names that begin with the
30       // string entered by the user.
31       System.out.println("Here are the names that match:");
32       for (String person : people)
33       {
34          if (person.startsWith(lookUp))
35             System.out.println(person);
36       }
37    }
38 }
```

Program Output with Example Input Shown in Bold

```
Enter the first few characters of the last name to look up: Davis [Enter]
```

```
Here are the names that match:
Davis, George
Davis, Jenny
```

Program Output with Example Input Shown in Bold

```
Enter the first few characters of the last name to look up: Russ [Enter]
Here are the names that match:
Russert, Phil
Russell, Cindy
```

The regionMatches Methods

The String class provides overloaded versions of the regionMatches method, which determines whether specified regions of two strings match. The following code demonstrates:

```
String str = "Four score and seven years ago";
String str2 = "Those seven years passed quickly";
if (str.regionMatches(15, str2, 6, 11))
   System.out.println("The regions match.");
else
   System.out.println("The regions do not match.");
```

This code will display "The regions match." The specified region of the str string begins at position 15, and the specified region of the str2 string begins at position 6. Both regions consist of 11 characters. The specified region in the str string is "seven years" and the specified region in the str2 string is also "seven years". Because the two regions match, the regionMatches method in this code returns true. This version of the regionMatches method performs a case-sensitive comparison. An overloaded version accepts an additional argument indicating whether to perform a case-insensitive comparison. The following code demonstrates:

```
String str = "Four score and seven years ago";
String str2 = "THOSE SEVEN YEARS PASSED QUICKLY";

if (str.regionMatches(true, 15, str2, 6, 11))
   System.out.println("The regions match.");
else
   System.out.println("The regions do not match.");
```

This code will also display "The regions match." The first argument passed to this version of the regionMatches method can be true or false, indicating whether a case-insensitive comparison should be performed. In this example, true is passed, so case will be ignored when the regions "seven years" and "SEVEN YEARS" are compared.

Each of these methods indicates by a boolean return value whether a substring appears within a string. The String class also provides methods that not only search for items within a string, but also report the location of those items. Table 9-4 describes overloaded versions of the indexOf and lastIndexOf methods.

Table 9-4 `String` methods for getting a character or substring's location

Method	Description
`int indexOf(char ch)`	Searches the calling `String` object for the character passed into *ch*. If the character is found, the position of its first occurrence is returned. Otherwise, −1 is returned.
`int indexOf(char ch, int start)`	Searches the calling `String` object for the character passed into *ch*, beginning at the position passed into *start* and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, −1 is returned.
`int indexOf(String str)`	Searches the calling `String` object for the string passed into *str*. If the string is found, the beginning position of its first occurrence is returned. Otherwise, −1 is returned.
`int indexOf(String str, int start)`	Searches the calling `String` object for the string passed into *str*. The search begins at the position passed into *start* and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, −1 is returned.
`int lastIndexOf(char ch)`	Searches the calling `String` object for the character passed into *ch*. If the character is found, the position of its last occurrence is returned. Otherwise, −1 is returned.
`int lastIndexOf(char ch, int start)`	Searches the calling `String` object for the character passed into *ch*, beginning at the position passed into *start*. The search is conducted backward through the string, to position 0. If the character is found, the position of its last occurrence is returned. Otherwise, −1 is returned.
`int lastIndexOf(String str)`	Searches the calling `String` object for the string passed into *str*. If the string is found, the beginning position of its last occurrence is returned. Otherwise, −1 is returned.
`int lastIndexOf(String str, int start)`	Searches the calling `String` object for the string passed into *str*, beginning at the position passed into *start*. The search is conducted backward through the string, to position 0. If the string is found, the beginning position of its last occurrence is returned. Otherwise, −1 is returned.

Finding Characters with the indexOf and lastIndexOf Methods

The indexOf and lastIndexOf methods can search for either a character or a substring within the calling string. If the item being searched for is found, its position is returned. Otherwise -1 is returned. Here is an example of code using two of the methods to search for a character:

```
String str = "Four score and seven years ago";
int first, last;

first = str.indexOf('r');
last = str.lastIndexOf('r');

System.out.println("The letter r first appears at " +
                   "position " + first);

System.out.println("The letter r last appears at " +
                   "position " + last);
```

This code produces the following output:

```
The letter r first appears at position 3
The letter r last appears at position 24
```

The following code shows another example. It uses a loop to show the positions of each letter 'r' in the string.

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
                   "following locations:");
position = str.indexOf('r');
while (position != -1)
{
   System.out.println(position);
   position = str.indexOf('r', position + 1);
}
```

This code will produce the following output:

```
The letter r appears at the following locations:
3
8
24
```

The following code is very similar, but it uses the lastIndexOf method and shows the positions in reverse order:

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
                   "following locations.");
```

```
position = str.lastIndexOf('r');
while (position != -1)
{
   System.out.println(position);
   position = str.lastIndexOf('r', position - 1);
}
```

This code will produce the following output:

```
The letter r appears at the following locations.
24
8
3
```

Finding Substrings with the `indexOf` and `lastIndexOf` Methods

The `indexOf` and `lastIndexOf` methods can also search for substrings within a string. The following code shows an example. It displays the starting positions of each occurrence of the word "and" within a string.

```
String str = "and a one and a two and a three";
int position;
System.out.println("The word and appears at the " +
                   "following locations.");
position = str.indexOf("and");
while (position != -1)
{
   System.out.println(position);
   position = str.indexOf("and", position + 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
0
10
20
```

The following code also displays the same results, but in reverse order:

```
String str = "and a one and a two and a three";
int position;

System.out.println("The word and appears at the " +
                   "following locations.");
position = str.lastIndexOf("and");
while (position != -1)
{
   System.out.println(position);
   position = str.lastIndexOf("and", position - 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
20
10
0
```

Extracting Substrings

The `String` class provides several methods that allow you to retrieve a substring from a string. The methods we will examine are listed in Table 9-5.

Table 9-5 String methods for extracting substrings

Method	Description
`String substring(int start)`	This method returns a copy of the substring that begins at *start* and goes to the end of the calling object's string.
`String substring(int start, int end)`	This method returns a copy of a substring. The argument passed into *start* is the substring's starting position, and the argument passed into *end* is the substring's ending position. The character at the *start* position is included in the substring, but the character at the *end* position is not included.
`void getChars(int start, int end, char[] array, int arrayStart)`	This method extracts a substring from the calling object and stores it in a `char` array. The argument passed into *start* is the substring's starting position, and the argument passed into *end* is the substring's ending position. The character at the *start* position is included in the substring, but the character at the *end* position is not included. (The last character in the substring ends at *end* − 1.) The characters in the substring are stored as elements in the array that is passed into the *array* parameter. The *arrayStart* parameter specifies the starting subscript within the array where the characters are to be stored.
`char[] toCharArray()`	This method returns all of the characters in the calling object as a `char` array.

The `substring` Methods

The `substring` method returns a copy of a substring from the calling object. There are two overloaded versions of this method. The first version accepts an `int` argument that is the starting position of the substring. The method returns a reference to a `String` object

containing all of the characters from the starting position to the end of the string. The character at the starting position is part of the substring. Here is an example of the method's use:

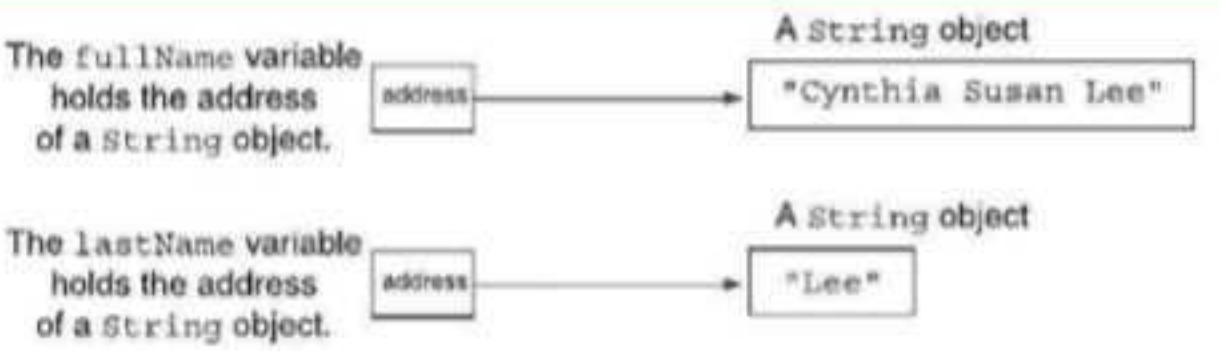
```
String fullName = "Cynthia Susan Lee";
String lastName = fullName.substring(14);
System.out.println("The full name is " + fullName);
System.out.println("The last name is " + lastName);
```

This code will produce the following output:

```
The full name is Cynthia Susan Lee
The last name is Lee
```

Keep in mind that the `substring` method returns a new `String` object that holds a copy of the substring. When this code executes, the `fullName` and `lastName` variables will reference two different `String` objects, as shown in Figure 9-5.

Figure 9-5 The `fullName` and `lastName` variables reference separate objects



The second version of the method accepts two `int` arguments. The first specifies the substring's starting position and the second specifies the substring's ending position. The character at the starting position is included in the substring, but the character at the ending position is not. Here is an example of how the method is used:

```
String fullName = "Cynthia Susan Lee";
String middleName = fullName.substring(8, 13);
System.out.println("The full name is " + fullName);
System.out.println("The middle name is " + middleName);
```

The code will produce the following output:

```
The full name is Cynthia Susan Lee
The middle name is Susan
```

The `getChars` and `toCharArray` Methods

The `getChars` and `toCharArray` methods convert the calling `String` object to a `char` array. The `getChars` method can be used to convert a substring, while the `toCharArray` method converts the entire string. Here is an example of how the `getChars` method might be used:

```
String fullName = "Cynthia Susan Lee";
char[] nameArray = new char[5];
```

```
fullName.getChars(8, 13, nameArray, 0);
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
    System.out.print(nameArray[i] + " ");
```

This code stores the individual characters of the substring "Susan" in the elements of the nameArray array, beginning at element 0. The code will produce the following output:

```
The full name is Cynthia Susan Lee
The values in the array are:
S u s a n
```

The toCharArray method returns a reference to a char array that contains all of the characters in the calling object. Here is an example:

```
String fullName = "Cynthia Susan Lee";
char[] nameArray;
nameArray = fullName.toCharArray();
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
    System.out.print(nameArray[i] + " ");
```

This code will produce the following output:

```
The full name is Cynthia Susan Lee
The values in the array are:
C y n t h i a   S u s a n   L e e
```

These methods can be used when you want to use an array processing algorithm on the contents of a String object. The program in Code Listing 9-5 converts a String object to an array and then uses the array to determine the number of letters, digits, and whitespace characters in the string. Figure 9-6 shows an example of interaction with the program.

Code Listing 9-5 (StringAnalyzer.java)

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program displays the number of letters,
5     digits, and whitespace characters in a string.
6  */
7
8  public class StringAnalyzer
9  {
10    public static void main(String [] args)
11    {
12       String input;          // To hold input
13       char[] array;          // Array for input
```

Methods That Return a Modified `String`

The `String` class methods listed in Table 9-6 return a modified copy of a `String` object.

Table 9-6 Methods that return a modified copy of a `String` object

Method	Description
`String concat(String str)`	This method returns a copy of the calling `String` object with the contents of *`str`* concatenated to it.
`String replace(char oldChar, char newChar)`	This method returns a copy of the calling `String` object, in which all occurrences of the character passed into *`oldChar`* have been replaced by the character passed into *`newChar`*.
`String trim()`	This method returns a copy of the calling `String` object, in which all leading and trailing whitespace characters have been deleted.

The `concat` method performs the same operation as the `+` operator when used with strings. For example, look at the following code, which uses the `+` operator:

```
String fullName;
String firstName = "Timothy ";
String lastName = "Haynes";
fullName = firstName + lastName;
```

Equivalent code can also be written with the `concat` method. Here is an example:

```
String fullName;
String firstName = "Timothy ";
String lastName = "Haynes";
fullName = firstName.concat(lastName);
```

The `replace` method returns a copy of a `String` object, where every occurrence of a specified character has been replaced with another character. For example, look at the following code:

```
String str1 = "Tom Talbert Tried Trains";
String str2;
str2 = str1.replace('T', 'D');
System.out.println(str1);
System.out.println(str2);
```

In this code, the `replace` method will return a copy of the `str1` object with every occurrence of the letter 'T' replaced with the letter 'D'. The code will produce the following output:

```
Tom Talbert Tried Trains
Dom Dalbert Dried Drains
```

Table 9-7 Some of the `String` class's `valueOf` methods

Method	Description
`String valueOf(boolean b)`	If the `boolean` argument passed to *b* is `true`, the method returns the string `"true"`. If the argument is `false`, the method returns the string `"false"`.
`String valueOf(char c)`	This method returns a `String` containing the character passed into *c*.
`String valueOf(char[] array)`	This method returns a `String` that contains all of the elements in the `char` array passed into *array*.
`String valueOf(char[] array, int subscript, int count)`	This method returns a `String` that contains part of the elements in the `char` array passed into *array*. The argument passed into *subscript* is the starting subscript and the argument passed into *count* is the number of elements.
`String valueOf(double number)`	This method returns the `String` representation of the `double` argument passed into *number*.
`String valueOf(float number)`	This method returns the `String` representation of the `float` argument passed into *number*.
`String valueOf(int number)`	This method returns the `String` representation of the `int` argument passed into *number*.
`String valueOf(long number)`	This method returns the `String` representation of the `long` argument passed into *number*.

This code will produce the following output:

```
true
abcde
bcd
2.4981567
7
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

9.7 Write a method that accepts a reference to a `String` object as an argument and returns `true` if the argument ends with the substring "ger". Otherwise, the method should return `false`.

9.8 Modify the method you wrote for Checkpoint 9.7 so it performs a case-insensitive test. The method should return `true` if the argument ends with "ger" in any possible combination of uppercase and lowercase letters.

9.9 Look at the following declaration:

```
String cafeName = "Broadway Cafe";
String str;
```

Which of the following methods would you use to make `str` reference the string "Broadway"?

The first two constructors create empty `StringBuilder` objects of a specified size. The first constructor makes the `StringBuilder` object large enough to hold 16 characters, and the second constructor makes the object large enough to hold *`length`* characters. Remember, `StringBuilder` objects automatically resize themselves, so it is not a problem if you later want to store a larger string in the object. The third constructor accepts a `String` object as its argument and assigns the object's contents to the `StringBuilder` object. Here is an example of its use:

```
StringBuilder city = new StringBuilder("Charleston");
System.out.println(city);
```

This code creates a `StringBuilder` object and assigns its address to the `city` variable. The object is initialized with the string "Charleston". As demonstrated by this code, you can pass a `StringBuilder` object to the `println` and `print` methods.

One limitation of the `StringBuilder` class is that you cannot use the assignment operator to assign strings to `StringBuilder` objects. For example, the following code will not work:

```
StringBuilder city = "Charleston";  // ERROR!!! Will not work!
```

Instead of using the assignment operator, you must use the `new` key word and a constructor, or one of the `StringBuilder` methods, to store a string in a `StringBuilder` object.

Other `StringBuilder` Methods

The `StringBuilder` class provides many of the same methods as the `String` class. Table 9-9 lists several of the `StringBuilder` methods that work exactly like their `String` class counterparts.

Table 9-9 Methods that are common to the `String` and `StringBuilder` classes

`char charAt(int position)`
`void getChars(int start, int end, char[] array, int arrayStart)`
`int indexOf(String str)`
`int indexOf(String str, int start)`
`int lastIndexOf(String str)`
`int lastIndexOf(String str, int start)`
`int length()`
`String substring(int start)`
`String substring(int start, int end)`

In addition, the `StringBuilder` class provides several methods that the `String` class does not have. Let's look at a few of them.

The `append` Methods

The `StringBuilder` class has several overloaded versions of a method named `append`. These methods accept an argument, which may be of any primitive data type, a `char` array,

or a String object. They append a string representation of their argument to the calling object's current contents. Because there are so many overloaded versions of append, we will examine the general form of a typical call to the method as follows:

```
object.append(item);
```

After the method is called, a string representation of *item* will be appended to *object*'s contents. The following code shows some of the append methods being used:

```
StringBuilder str = new StringBuilder();

// Append values to the object.
str.append("We sold ");          // Append a String object.
str.append(12);                  // Append an int.
str.append(" doughnuts for $"); // Append another String.
str.append(15.95);               // Append a double.

// Display the object's contents.
System.out.println(str);
```

This code will produce the following output:

```
We sold 12 doughnuts for $15.95
```

The insert Methods

The StringBuilder class also has several overloaded versions of a method named insert, which inserts a value into the calling object's string. These methods accept two arguments: an int that specifies the position in the calling object's string where the insertion should begin, and the value to be inserted. The value to be inserted may be of any primitive data type, a char array, or a String object. Because there are so many overloaded versions of insert, we will examine the general form of a typical call to the method as follows:

```
object.insert(start, item);
```

In the general form, *start* is the starting position of the insertion and *item* is the item to be inserted. The following code shows an example:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

The first statement creates a StringBuilder object initialized with the string "New City". The second statement inserts the string "York " into the StringBuilder object, beginning at position 4. The characters that are currently in the object beginning at position 4 are moved to the right. In memory, the StringBuilder object is automatically expanded in size to accommodate the inserted characters. If these statements were in a complete program and we ran it, we would see New York City displayed on the screen.

The following code shows how a char array can be inserted into a StringBuilder object:

```
char cArray[] = { '2', '0', ' ' };
StringBuilder str = new StringBuilder("In July we sold cars.");
str.insert(16, cArray);
System.out.println(str);
```

The first statement declares a `char` array named `cArray`, containing the characters `'2'`, `'0'`, and `' '`. The second statement creates a `StringBuilder` object initialized with the string `"In July we sold cars."` The third statement inserts the characters in `cArray` into the `StringBuilder` object, beginning at position 16. The characters that are currently in the object beginning at position 16 are moved to the right. If these statements were in a complete program and we ran it, we would see `In July we sold 20 cars.` displayed on the screen.

The `replace` Method

The `StringBuilder` class has a `replace` method that differs slightly from the `String` class's `replace` method. While the `String` class's `replace` method replaces the occurrences of one character with another character, the `StringBuilder` class's `replace` method replaces a specified substring with a string. Here is the general form of a call to the method:

```
object.replace(start, end, str);
```

In the general form, *`start`* is an `int` that specifies the starting position of a substring in the calling object, and *`end`* is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The `str` parameter is a `String` object. After the method executes, the substring will be replaced with `str`. Here is an example:

```
StringBuilder str =
     new StringBuilder("We moved from Chicago to Atlanta.");
str.replace(14, 21, "New York");
System.out.println(str);
```

The `replace` method in this code replaces the word "Chicago" with "New York". The code will produce the following output:

```
We moved from New York to Atlanta.
```

The `delete`, `deleteCharAt`, and `setCharAt` Methods

The `delete` and `deleteCharAt` methods are used to delete a substring or a character from a `StringBuilder` object. The `setCharAt` method changes a specified character to another value. Table 9-10 describes these methods.

Table 9-10 The `StringBuilder` class's `delete`, `deleteCharAt`, and `setCharAt` methods

Method	Description
`StringBuilder delete(int start, int end)`	The *`start`* parameter is an `int` that specifies the starting position of a substring in the calling object, and the *`end`* parameter is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The method will delete the substring.
`StringBuilder deleteCharAt (int position)`	The *`position`* parameter specifies the location of a character that will be deleted.
`void setCharAt(int position, char ch)`	This method changes the character at *`position`* to the value passed into `ch`.

Although the parentheses and the hyphen make the number easier for people to read, those characters are unnecessary for processing by a computer. In a computer system, a telephone number is commonly stored as an unformatted series of digits, as shown here:

```
9195551212
```

A program that works with telephone numbers usually needs to unformat numbers that have been entered by the user. This means that the parentheses and the hyphen must be removed before the number is stored in a file or processed in some other way. In addition, such a program needs the ability to format the digits so that the number contains the parentheses and the hyphen when it appears on the screen or is printed on paper.

Code Listing 9-6 shows a class named `Telephone` that contains the following static methods:

- `isFormatted`–This method accepts a `String` argument and returns `true` if the argument is formatted as `(XXX)XXX-XXXX`. If the argument is not formatted this way, the method returns `false`.
- `unformat`–This method accepts a `String` argument. If the argument is formatted as `(XXX)XXX-XXXX`, the method returns an unformatted version of the argument with the parentheses and the hyphen removed. Otherwise, the method returns the original argument.
- `format`–This method's purpose is to format a sequence of digits as `(XXX)XXX-XXXX`. The sequence of digits is passed as a `String` argument. If the argument is 10 characters in length, then the method returns the argument with parentheses and a hyphen inserted. Otherwise, the method returns the original argument.

The program in Code Listing 9-7 demonstrates the `Telephone` class.

Code Listing 9-6 (Telephone.java)

```
1 /**
2    The Telephone class provides static methods
3    for formatting and unformatting U.S. telephone
4    numbers.
5 */
6
7 public class Telephone
8 {
9    // These constant fields hold the valid lengths of
10   // strings that are formatted and unformatted.
11   public final static int FORMATTED_LENGTH = 13;
12   public final static int UNFORMATTED_LENGTH = 10;
13
14   /**
15      The isFormatted method determines whether a
16      string is properly formatted as a U.S. telephone
17      number in the following manner:
18      (XXX)XXX-XXXX
19      @param str The string to test.
```

```
20       @return true if the string is properly formatted,
21               or false otherwise.
22    */
23
24    public static boolean isFormatted(String str)
25    {
26       boolean valid;   // Flag to indicate valid format
27
28       // Determine whether str is properly formatted.
29       if (str.length() == FORMATTED_LENGTH &&
30           str.charAt(0) == '(' &&
31           str.charAt(4) == ')' &&
32           str.charAt(8) == '-')
33           valid = true;
34       else
35          valid = false;
36
37       // Return the value of the valid flag.
38       return valid;
39    }
40
41    /**
42       The unformat method accepts a string containing
43       a telephone number formatted as:
44       (XXX)XXX-XXXX.
45       If the argument is formatted in this way, the
46       method returns an unformatted string where the
47       parentheses and hyphen have been removed. Otherwise,
48       it returns the original argument.
49       @param str The string to unformat.
50       @return An unformatted string.
51    */
52
53    public static String unformat(String str)
54    {
55       // Create a StringBuilder initialized with str.
56       StringBuilder strb = new StringBuilder(str);
57
58       // If the argument is properly formatted, then
59       // unformat it.
60       if (isFormatted(str))
61       {
62          // First, delete the left paren at position 0.
63          strb.deleteCharAt(0);
64
65          // Next, delete the right paren. Because of the
66          // previous deletion it is now located at
67          // position 3.
```

```
68          strb.deleteCharAt(3);
69
70          // Next, delete the hyphen. Because of the
71          // previous deletions it is now located at
72          // position 6.
73          strb.deleteCharAt(6);
74       }
75
76       // Return the unformatted string.
77       return strb.toString();
78    }
79
80    /**
81       The format method formats a string as:
82       (XXX)XXX-XXXX.
83       If the length of the argument is UNFORMATTED_LENGTH
84       the method returns the formatted string. Otherwise,
85       it returns the original argument.
86       @param str The string to format.
87       @return A string formatted as a U.S. telephone number.
88    */
89
90    public static String format(String str)
91    {
92       // Create a StringBuilder initialized with str.
93       StringBuilder strb = new StringBuilder(str);
94
95       // If the argument is the correct length, then
96       // format it.
97       if (str.length() == UNFORMATTED_LENGTH)
98       {
99          // First, insert the left paren at position 0.
100         strb.insert(0, "(");
101
102         // Next, insert the right paren at position 4.
103         strb.insert(4, ")");
104
105         // Next, insert the hyphen at position 8.
106         strb.insert(8, "-");
107      }
108
109      // Return the formatted string.
110      return strb.toString();
111   }
112 }
```

Code Listing 9-7 (TelephoneTester.java)

```
1 import java.util.Scanner;
2
3 /**
4    This program demonstrates the Telephone
5    class's static methods.
6 */
7
8 public class TelephoneTester
9 {
10    public static void main(String[] args)
11    {
12       String phoneNumber;  // To hold a phone number
13
14       // Create a Scanner object for keyboard input.
15       Scanner keyboard = new Scanner(System.in);
16
17       // Get an unformatted telephone number.
18       System.out.print("Enter an unformatted telephone number: ");
19       phoneNumber = keyboard.nextLine();
20
21       // Format the telephone number.
22       System.out.println("Formatted: " +
23             Telephone.format(phoneNumber));
24
25       // Get a formatted telephone number.
26       System.out.println("Enter a telephone number formatted as");
27       System.out.print("(XXX)XXX-XXXX : ");
28       phoneNumber = keyboard.nextLine();
29
30       // Unformat the telephone number.
31       System.out.println("Unformatted: " +
32             Telephone.unformat(phoneNumber));
33    }
34 }
```

Program Output with Example Input Shown in Bold

```
Enter an unformatted telephone number: 9195551212 [Enter]
Formatted: (919)555-1212
Enter a telephone number formatted as
(XXX)XXX-XXXX : (828)555-1212 [Enter]
Unformatted: 8285551212
```