

# 3

## Using Classes and Objects

### CHAPTER OBJECTIVES

- Discuss the creation of objects and the use of object reference variables.
- Explore the services provided by the `String` class.
- Describe how the Java standard class library is organized into packages.
- Explore the services provided by the `Random` and `Math` classes.
- Discuss ways to format output using the `NumberFormat` and `DecimalFormat` classes.
- Introduce enumerated types.
- Discuss wrapper classes and the concept of autoboxing.

This chapter further explores the use of predefined classes and the objects we can create from them. Using classes and objects for the services they provide is a fundamental part of object-oriented software, and it sets the stage for writing classes of our own. In this chapter, we use classes and objects to manipulate character strings, produce random numbers, perform complex calculations, and format output. This chapter also introduces the enumerated type, which is a special kind of class in Java, and discusses the concept of a wrapper class.

## 3.1 Creating Objects

At the end of Chapter 1 we presented an overview of object-oriented concepts, including the basic relationship between classes and objects. Then in Chapter 2, in addition to discussing primitive data, we provided some examples of using objects for the services they provide. This chapter explores these ideas further.

In previous examples, we've used the `println` method many times. As we mentioned in Chapter 2, the `println` method is a service provided by the `System.out` object, which represents the standard output stream. To be more precise, the identifier `out` is an object variable that is stored in the `System` class. It has been predefined and set up for us as part of the Java standard class library. We can simply use it.

In Chapter 2 we also used the `Scanner` class, which represents an object that allows us to read input from the keyboard or a file. We created a `Scanner` object using the `new` operator. Once the object was created, we were able to use it for the various services it provides. That is, we were able to invoke its methods.

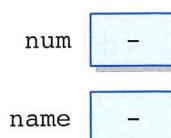
Let's carefully examine the idea of creating an object. In Java, a variable name represents either a primitive value or an object. Like variables that hold primitive types, a variable that refers to an object must be declared. The class used to define an object can be thought of as the type of an object. The declarations of object variables are similar in structure to the declarations of primitive variables.

Consider the following two declarations:

```
int num;
String name;
```

The first declaration creates a variable that holds an integer value, as we've seen many times before. The second declaration creates a `String` variable that holds a *reference* to a `String` object. An object variable doesn't hold the object itself, it holds the address of an object.

Initially, the two variables declared above don't contain any data. We say they are *uninitialized*, which can be depicted as follows:



As we pointed out in Chapter 2, it is always important to be certain that a variable is initialized before using it. For an object variable, that means we must make

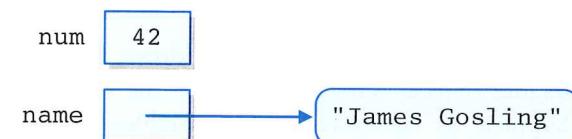
sure it refers to a valid object prior to using it. In most situations the compiler will issue an error if you attempt to use a variable before initializing it.

An object reference variable can also be set to `null`, which is a reserved word in Java. A null reference specifically indicates that a variable does not refer to an object.

Note that even though we've declared a `String` reference variable, no `String` object actually exists yet. The act of creating an object using the `new` operator is called *instantiation*. An object is said to be an *instance* of a particular class. To instantiate an object, we can use the `new` operator, which returns the address of the new object. The following two assignment statements give values to the two variables declared above:

```
num = 42;
name = new String("James Gosling");
```

After the `new` operator creates the object, a *constructor* is invoked to help set it up initially. A constructor is a special method that has the same name as the class. In this example, the parameter to the constructor is a string literal that specifies the characters that the `String` object will hold. After these assignments are executed, the variables can be depicted as follows:



### KEY CONCEPT

The `new` operator returns a reference to a newly created object.



**VideoNote**  
Creating objects

Because an object reference variable holds the address of the object, it can be thought of as a *pointer* to the location in memory where the object is held. We could show the numeric address, but the actual address value is irrelevant—what's important is that the variable refers to a particular object.

After an object has been instantiated, we use the *dot operator* to access its methods. We've used the dot operator many times already, such as in calls to `System.out.println`. The dot operator is appended directly after the object reference, followed by the method being invoked. For example, to invoke the `length` method defined in the `String` class, we can use the dot operator on the `name` reference variable:

```
count = name.length()
```

The `length` method does not take any parameters, but the parentheses are still necessary to indicate that a method is being invoked. Some methods produce a value that is *returned* when the method completes. The purpose of the `length` method of the `String` class is to determine and return the length of the string (the number of characters it contains). In this example, the returned value is assigned

to the variable count. For the string "James Gosling", the length method returns 13, which includes the space between the first and last names. Some methods do not return a value. Other String methods are discussed in the next section.

The act of declaring the object reference variable and creating the object itself can be combined into one step by initializing the variable in the declaration, just as we do with primitive types:

```
String title = new String("Java Foundations");
```

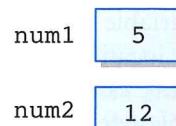
Even though they are not primitive types, character strings are so fundamental and are used so often that Java defines string literals delimited by double quotation marks, as we've seen in various examples. This is a shortcut notation. Whenever a string literal appears, a String object is created automatically. Therefore, the following declaration is valid:

```
String city = "London";
```

That is, for string objects, the explicit use of the new operator and the call to the constructor can be eliminated. In most cases, we will use this simplified syntax.

### Aliases

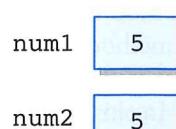
Because an object reference variable stores an address, a programmer must be careful when managing objects. First, let's review the effect of assignment on primitive values. Suppose we have two integer variables—num1, initialized to 5, and num2, initialized to 12:



In the following assignment statement, a copy of the value that is stored in num1 is stored in num2.

```
num2 = num1;
```

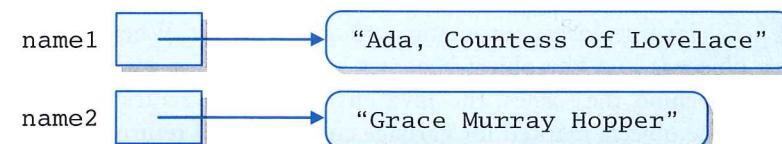
The original value of 12 in num2 is overwritten by the value 5. The variables num1 and num2 still refer to different locations in memory, and both of those locations now contain the value 5:



Now consider the following object declarations:

```
String name1 = "Ada, Countess of Lovelace";
String name2 = "Grace Murray Hopper";
```

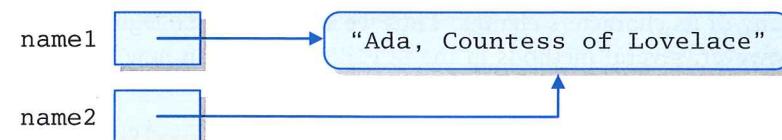
Initially, the references name1 and name2 refer to two different string objects:



Now suppose the following assignment statement is executed, copying the value in name1 into name2.

```
name2 = name1;
```

This assignment works the same as the integer assignment—a copy of the value of name1 is stored in name2. But remember, object variables hold the address of an object, and it is the address that gets copied. Originally, the two references referred to different objects. After the assignment, both name1 and name2 contain the same address and therefore refer to the same object:



The name1 and name2 reference variables are now *aliases* of each other because they are two different variables that refer to the same object. All references to the object originally referenced by name2 are now gone; that object cannot be used again in the program.

### DESIGN FOCUS

One important implication of aliases is that when we use one reference to change an object, it is also changed for the other reference, because there is really only one object. Aliases can produce undesirable effects unless they are managed carefully.

That is not to say that aliases are a bad thing. There are many situations in which it's helpful to have multiple references to objects. In fact, every time you pass an object to a method, you create an alias. It's not that you want to avoid aliases, you just want to be aware of the effect they have on the objects you manage.

All interaction with an object occurs through a reference variable, so we can use an object only if we have a reference to it. When all references to an object are lost (perhaps by reassignment), that object can no longer contribute to the program. The program can no longer invoke its methods or use its variables. At this point the object is called *garbage* because it serves no useful purpose.

**KEY CONCEPT**

Multiple reference variables can refer to the same object.

Java performs automatic *garbage collection*. When the last reference to an object is lost, the object becomes a candidate for garbage collection. Occasionally, behind the scenes, the Java environment executes a method that “collects” all the objects marked for garbage collection and returns their memory to the system for future use. The programmer does not have to worry about explicitly reclaiming memory that has become garbage.

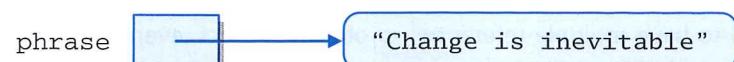
## 3.2 The String Class

Let's examine the `String` class in more detail. Figure 3.1 lists some of the more useful methods of the `String` class.

Once a `String` object is created, its value cannot be lengthened or shortened, nor can any of its characters change. Thus we say that a `String` object is *immutable*. However, several methods in the `String` class return new `String` objects that are the result of modifying the original string's value.

Note that some of the `String` methods, such as `charAt`, refer to the *index* of a particular character. An index specifies a particular position, and therefore a particular character, in a string. The index of the first character in a string is zero, the index of the next character is one, and so on. Therefore, in the string "Hello", the index of the character 'H' is zero, and the character at index four is 'o'.

Several `String` methods are exercised in the program shown in Listing 3.1 on page 78. As you examine the `StringMutation` program, keep in mind that this is not a single `String` object that changes its data; this program creates five separate `String` objects using various methods of the `String` class. Originally, the `phrase` object is set up:



```

String (String str)
  Constructor: creates a new string object with the same characters as str.

char charAt (int index)
  Returns the character at the specified index.

int compareTo (String str)
  Returns an integer indicating if this string is lexically before (a negative return value), equal to (a zero return value), or lexically after (a positive return value), the string str.

String concat (String str)
  Returns a new string consisting of this string concatenated with str.

boolean equals (String str)
  Returns true if this string contains the same characters as str (including case) and false otherwise.

boolean equalsIgnoreCase (String str)
  Returns true if this string contains the same characters as str (without regard to case) and false otherwise.

int length ()
  Returns the number of characters in this string.

String replace (char oldChar, char newChar)
  Returns a new string that is identical with this string except that every occurrence of oldChar is replaced by newChar.

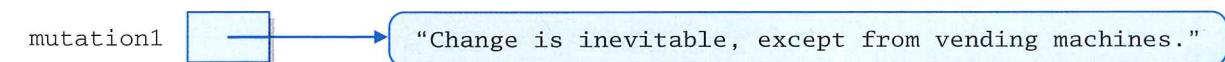
String substring (int offset, int endIndex)
  Returns a new string that is a subset of this string starting at index offset and extending through endIndex-1.

String toLowerCase ()
  Returns a new string identical to this string except all uppercase letters are converted to their lowercase equivalent.

String toUpperCase ()
  Returns a new string identical to this string except all lowercase letters are converted to their uppercase equivalent.
  
```

**FIGURE 3.1** Some methods of the `String` class

After printing the original `phrase` and its length, the `concat` method is executed to create a new `String` object referenced by the variable `mutation1`:



**LISTING 3.1**

```

//***** StringMutation.java      Java Foundations *****
// Demonstrates the use of the String class and its methods.
//***** */

public class StringMutation
{
    // Prints a string and various mutations of it.
    public static void main(String[] args)
    {
        String phrase = "Change is inevitable";
        String mutation1, mutation2, mutation3, mutation4;

        System.out.println("Original string: " + phrase);
        System.out.println("Length of string: " + phrase.length());

        mutation1 = phrase.concat(", except from vending machines.");
        mutation2 = mutation1.toUpperCase();
        mutation3 = mutation2.replace('E', 'X');
        mutation4 = mutation3.substring(3, 30);

        // Print each mutated string

        System.out.println("Mutation #1: " + mutation1);
        System.out.println("Mutation #2: " + mutation2);
        System.out.println("Mutation #3: " + mutation3);
        System.out.println("Mutation #4: " + mutation4);

        System.out.println("Mutated length: " + mutation4.length());
    }
}

```

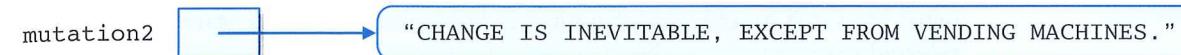
**OUTPUT**

```

Original string: "Change is inevitable"
Length of string: 20
Mutation #1: Change is inevitable, except from vending machines.
Mutation #2: CHANGE IS INEVITABLE, EXCEPT FROM VENDING MACHINES.
Mutation #3: CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
Mutation #4: NGX IS INXVITABLX, XXCXPT F
Mutated length: 27

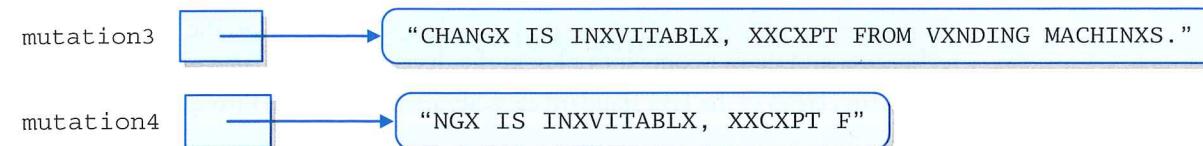
```

Then the `toUpperCase` method is executed on the `mutation1` object, and the resulting string is stored in `mutation2`:



Notice that the `length` and `concat` methods are executed on the `phrase` object, but the `toUpperCase` method is executed on the `mutation1` object. Any method of the `String` class can be executed on any `String` object, but for any given invocation, the method is executed on a particular object. The results of executing `toUpperCase` on `mutation1` would be very different from the results of executing `toUpperCase` on `phrase`. Remember that each object has its own state, which often affects the results of method calls.

Finally, the `String` object variables `mutation3` and `mutation4` are initialized by the calls to `mutation2.replace` and `mutation3.substring`, respectively:

**3.3 Packages**

We mentioned earlier that the Java language is supported by a standard class library that we can make use of as needed. Let's examine that idea further.

A *class library* is a set of classes that supports the development of programs. A compiler or development environment often comes with a class library. Class libraries can also be obtained separately through third-party vendors. The classes in a class library contain methods that are valuable to a programmer because of the special functionality they offer. In fact, programmers often become dependent on the methods in a class library and begin to think of them as part of the language. However, technically, they are not in the language itself.

The `String` class, for instance, is not an inherent part of the Java language. It is part of the Java standard class library that can be found in any Java development environment. The classes that make up the library were created by employees at Sun Microsystems, where the Java language was created.

The class library is made up of several clusters of related classes, which are sometimes called Java APIs, or *application programming interfaces*. For example,

**KEY CONCEPT**

Methods are often executed on a particular object, and that object's state usually affects the results.

**KEY CONCEPT**

A class library provides useful support when one is developing programs.

we may refer to the Java Database API when we're talking about the set of classes that helps us write programs that interact with a database. Another example of an API is the Java Swing API, a set of classes that defines special graphical components used in a graphical user interface. Sometimes the entire standard library is referred to generically as the Java API.

#### KEY CONCEPT

The Java standard class library is organized into packages.

The classes of the Java standard class library are also grouped into *packages*. Each class is part of a particular package. The `String` class, for example, is part of the `java.lang` package. The `System` class is part of the `java.lang` package as well. The `Scanner` class is part of the `java.util` package.

The package organization is more fundamental and language-based than the API organization. Although there is a general correspondence between package and API names, the groups of classes that make up a given API might cross packages. In this text, we refer to classes primarily in terms of their package organization.

Figure 3.2 describes some of the packages that are part of the Java standard class library. These packages are available on any platform that supports Java software development. Some of these packages support highly specific programming techniques and do not come into play in the development of basic programs.

Various classes of the Java standard class library are discussed throughout this book.

#### The import Declaration

The classes of the `java.lang` package are automatically available for use when writing a Java program. To use classes from any other package, however, we must either *fully qualify* the reference or use an *import declaration*. Let's consider these two options.

When you want to use a class from a class library in a program, you could use its fully qualified name, including the package name, every time it was referenced. For example, every time you wanted to refer to the `Scanner` class defined in the `java.util` package, you could write `java.util.Scanner`. However, completely specifying the package and class name every time it is needed quickly becomes tiring. Java provides the `import` declaration to simplify these references.

The `import` declaration specifies the packages and classes that will be used in a program so that the fully qualified name is not necessary with each reference. Recall that the example programs that use the `Scanner` class in Chapter 2 include an `import` declaration like this one:

```
import java.util.Scanner;
```

This declaration asserts that the `Scanner` class of the `java.util` package may be used in the program. Once this `import` declaration is made, it is sufficient to use the simple name `Scanner` when referring to that class in the program.

Package	Provides support to
<code>java.applet</code>	Create programs (applets) that are easily transported across the Web.
<code>java.awt</code>	Draw graphics and create graphical user interfaces; AWT stands for Abstract Windowing Toolkit.
<code>java.beans</code>	Define software components that can be easily combined into applications.
<code>java.io</code>	Perform a wide variety of input and output functions.
<code>java.lang</code>	General support; it is automatically imported into all Java programs.
<code>java.math</code>	Perform calculations with arbitrarily high precision.
<code>java.net</code>	Communicate across a network.
<code>java.rmi</code>	Create programs that can be distributed across multiple computers; RMI stands for Remote Method Invocation.
<code>java.security</code>	Enforce security restrictions.
<code>java.sql</code>	Interact with databases; SQL stands for Structured Query Language.
<code>java.text</code>	Format text for output.
<code>java.util</code>	General utilities.
<code>javax.swing</code>	Create graphical user interfaces with components that extend the AWT capabilities.
<code>javax.xml.parsers</code>	Process XML documents; XML stands for eXtensible Markup Language.

FIGURE 3.2 Some packages in the Java standard class library

If two classes from two different packages have the same name, `import` declarations will not suffice, because the compiler won't be able to figure out which class is being referenced in the flow of the code. When such situations arise (which occurs rarely), the fully qualified names should be used in the code.

Another form of the `import` declaration uses an asterisk (\*) to indicate that any class inside the package might be used in the program. Therefore, the following declaration allows all classes in the `java.util` package to be referenced in the program without qualifying each reference:

```
import java.util.*;
```

If only one class of a particular package will be used in a program, it is usually better to name the class specifically in the `import` declaration, because that provides more specific information to anyone reading the code. However, if two or more classes of the package will be used, the \* notation is usually fine.

**Import-Declaration**

```
Java keyword      package name      class name
import java.util.Random;
import java.awt.*;
               wildcard
```

**KEY CONCEPT**

All classes of the `java.lang` package are automatically imported for every program.

The classes of the `java.lang` package are automatically imported because they are fundamental and can be thought of as basic extensions to the language. Therefore, any class in the `java.lang` package, such as `System` and `String`, can be used without an explicit `import` declaration. It's as if all program files automatically contain the following declaration:

```
import java.lang.*;
```

## 3.4 The Random Class

The need for random numbers occurs frequently when one is writing software. Games often use random numbers to simulate the roll of a die or the shuffle of a deck of cards. A flight simulator may use random numbers to determine how often a simulated flight has engine trouble. A program designed to help high school students prepare for the SATs may use random numbers to choose the next question to ask.

The `Random` class, which is part of the `java.util` package, represents a *pseudorandom number generator*. A random number generator picks a number at random out of a range of values. Program code that plays this role is technically pseudorandom, because a program has no means to actually pick a number randomly. A pseudorandom number generator performs a series of complicated calculations, based on an initial *seed value*, and produces a number. Even though they are technically not random (because they are calculated), the values produced by a pseudorandom number generator usually appear random—at least random enough for most situations.

Figure 3.3 lists some of the methods of the `Random` class. The `nextInt` method can be called with no parameters, or we can pass it a single integer value. If no parameter is passed in, the `nextInt` method generates a random number across the entire range of `int` values, including negative numbers. Usually, though, we need a random number within a more specific range. For instance, to simulate the

**Random ()**

Constructor: creates a new pseudorandom number generator.

**float nextFloat ()**

Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

**int nextInt ()**

Returns a random number that ranges over all possible `int` values (positive and negative).

**int nextInt (int num)**

Returns a random number in the range 0 to `num`-1.

**FIGURE 3.3** Some methods of the `Random` class

roll of a die, we might want a random number in the range of 1 to 6. The `nextInt` method returns a value that's in the range from 0 to one less than its parameter. For example, if we pass 100 as a parameter to `nextInt`, we'll get a return value that is greater than or equal to 0 and less than or equal to 99.

Note that the value that we pass to the `nextInt` method is also the number of possible values we can get in return. We can shift the range as needed by adding or subtracting the proper amount. To get a random number in the range of 1 to 6, we can call `nextInt(6)` to get a value from 0 to 5, and then add 1.

The `nextFloat` method of the `Random` class returns a `float` value that is greater than or equal to 0.0 and less than 1.0. If desired, we can use multiplication to scale the result, cast it into an `int` value to truncate the fractional part, and then shift the range as we do with integers.

The program shown in Listing 3.2 produces several random numbers in various ranges.

**Generating-Random-Numbers**

Random object

shifts result by 10 to 10-29

```
num = generator.nextInt(20) + 10;
```

produces an int in the range 0-19

**KEY CONCEPT**

A pseudorandom number generator performs a complex calculation to create the illusion of randomness.

**LISTING 3.2**

```
*****
// RandomNumbers.java          Java Foundations
//
// Demonstrates the creation of pseudo-random numbers using the
// Random class.
*****
import java.util.Random;

public class RandomNumbers
{
    //-----
    // Generates random numbers in various ranges.
    //-----

    public static void main(String[] args)
    {
        Random generator = new Random();
        int num1;
        float num2;

        num1 = generator.nextInt();
        System.out.println("A random integer: " + num1);

        num1 = generator.nextInt(10);
        System.out.println("From 0 to 9: " + num1);

        num1 = generator.nextInt(10) + 1;
        System.out.println("From 1 to 10: " + num1);

        num1 = generator.nextInt(15) + 20;
        System.out.println("From 20 to 34: " + num1);

        num1 = generator.nextInt(20) - 10;
        System.out.println("From -10 to 9: " + num1);

        num2 = generator.nextFloat();
        System.out.println("A random float (between 0-1): " + num2);

        num2 = generator.nextFloat() * 6; // 0.0 to 5.999999
        num1 = (int)num2 + 1;
        System.out.println("From 1 to 6: " + num1);
    }
}
```

**LISTING 3.2***continued***OUTPUT**

A random integer: 243057344  
 From 0 to 9: 9  
 From 1 to 10: 2  
 From 20 to 34: 33  
 From -10 to 9: -4  
 A random float (between 0-1): 0.58384484  
 From 1 to 6: 3

**3.5 The Math Class**

The `Math` class provides a large number of basic mathematical functions that are often helpful in making calculations. The `Math` class is defined in the `java.lang` package of the Java standard class library. Figure 3.4 lists several of its methods.

All the methods in the `Math` class are *static methods* (also called *class methods*), which means they we can invoke them through the name of the class in which they are defined, without having to instantiate an object of the class first. Static methods are discussed further in Chapter 5.

The methods of the `Math` class return values, which can be used in expressions as needed. For example, the following statement computes the absolute value of the number stored in `total`, adds it to the value of `count` raised to the fourth power, and stores the result in the variable `value`.

```
value = Math.abs(total) + Math.pow(count, 4);
```

Note that you can pass an integer value to a method that accepts a double parameter. This is a form of assignment conversion, which was discussed in Chapter 2.

It's also interesting to note that the `Math` class contains a method called `random` that returns a random floating point value in the range 0.0 to 1.0. Therefore, this method could be used as an alternative to creating a `Random` object and calling its methods, as described in the previous section. However, the `Math` class does not have a method that returns an integer, or lets you specify the range of the result, as `Random` does. ~

**KEY CONCEPT**

All methods of the `Math` class are static, which means that they are invoked through the class name.



VideoNote

Example using the Random and Math classes

```

static int abs (int num)
    Returns the absolute value of num.

static double acos (double num)

static double asin (double num)

static double atan (double num)
    Returns the arc cosine, arc sine, or arc tangent of num.

static double cos (double angle)

static double sin (double angle)

static double tan (double angle)
    Returns the angle cosine, sine, or tangent of angle, which is measured in
    radians.

static double ceil (double num)
    Returns the ceiling of num, which is the smallest whole number greater than or
    equal to num.

static double exp (double power)
    Returns the value e raised to the specified power.

static double floor (double num)
    Returns the floor of num, which is the largest whole number less than or equal
    to num.

static double pow (double num, double power)
    Returns the value num raised to the specified power.

static double random ()
    Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

static double sqrt (double num)
    Returns the square root of num, which must be positive.

```

**FIGURE 3.4** Some methods of the Math class

The Quadratic program, shown in Listing 3.3, uses the Math class to compute the roots of a quadratic equation. Recall that a quadratic equation has the following general form:

$$ax^2 + bx + c = 0$$

### LISTING 3.3

```

//*****
// Quadratic.java          Java Foundations
// Demonstrates the use of the Math class to perform a calculation
// based on user input.
//*****

import java.util.Scanner;

public class Quadratic
{
    //-----
    // Determines the roots of a quadratic equation.
    //-----
    public static void main(String[] args)
    {
        int a, b, c; // ax^2 + bx + c
        double discriminant, root1, root2;

        Scanner scan = new Scanner(System.in);

        System.out.print("Enter the coefficient of x squared: ");
        a = scan.nextInt();

        System.out.print("Enter the coefficient of x: ");
        b = scan.nextInt();

        System.out.print("Enter the constant: ");
        c = scan.nextInt();

        // Use the quadratic formula to compute the roots.
        // Assumes a positive discriminant.

        discriminant = Math.pow(b, 2) - (4 * a * c);
        root1 = ((-1 * b) + Math.sqrt(discriminant)) / (2 * a);
        root2 = ((-1 * b) - Math.sqrt(discriminant)) / (2 * a);

        System.out.println("Root #1: " + root1);
        System.out.println("Root #2: " + root2);
    }
}

```

**LISTING 3.3***continued***OUTPUT**

```
Enter the coefficient of x squared: 3
Enter the coefficient of x: 8
Enter the constant: 4
Root #1: -0.6666666666666666
Root #2: -2.0
```

The Quadratic program reads values that represent the coefficients in a quadratic equation ( $a$ ,  $b$ , and  $c$ ) and then evaluates the quadratic formula to determine the roots of the equation. The quadratic formula is

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that this program assumes that the discriminant (the value under the square root) is positive. If it's not, the result will not be a valid number, which Java represents as `NAN`, which stands for Not A Number. In Chapter 5 we discuss how such situations can be avoided.

## 3.6 Formatting Output

The `NumberFormat` class and the `DecimalFormat` class are used to format information so that it looks appropriate when printed or displayed. They are both part of the Java standard class library and are defined in the `java.text` package.

### The `NumberFormat` Class

The `NumberFormat` class provides generic formatting capabilities for numbers. We don't instantiate a `NumberFormat` object by using the `new` operator. Instead, we request an object from one of its static methods that we invoke through the class name itself. Figure 3.5 lists some of the methods of the `NumberFormat` class.

Two of the methods in the `NumberFormat` class, `getCurrencyInstance` and `getPercentInstance`, return an object that is used to format numbers. The `getCurrencyInstance` method returns a formatter for monetary values, and the `getPercentInstance` method returns an object that formats a percentage.

`String format (double number)`

Returns a string containing the specified number formatted according to this object's pattern.

`static NumberFormat getCurrencyInstance()`

Returns a `NumberFormat` object that represents a currency format for the current locale.

`static NumberFormat getPercentInstance()`

Returns a `NumberFormat` object that represents a percentage format for the current locale.

**FIGURE 3.5** Some methods of the `NumberFormat` class

The `format` method is invoked through a formatter object and returns a `String` that contains the number formatted in the appropriate manner.

The `Purchase` program shown in Listing 3.4 uses both types of formatters. It reads in a sales transaction and computes the final price, including tax.

The `getCurrencyInstance` and `getPercentInstance` methods are called *factory methods*, because they produce and return an instance of an object set up in a particular manner. Essentially, a `NumberFormat` factory method uses the new

**LISTING 3.4**

```
//*****
// Purchase.java          Java Foundations
//
// Demonstrates the use of the NumberFormat class to format output.
//*****
```

```
import java.util.Scanner;
import java.text.NumberFormat;

public class Purchase
{
    //-----
    // Calculates the final price of a purchased item using values
    // entered by the user.
    //-----

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

**LISTING 3.4***continued*

```

final double TAX_RATE = 0.06; // 6% sales tax

int quantity;
double subtotal, tax, totalCost, unitPrice;

Scanner scan = new Scanner(System.in);

NumberFormat fmt1 = NumberFormat.getCurrencyInstance();
NumberFormat fmt2 = NumberFormat.getPercentInstance();

System.out.print("Enter the quantity: ");
quantity = scan.nextInt();

System.out.print("Enter the unit price: ");
unitPrice = scan.nextDouble();

subtotal = quantity * unitPrice;
tax = subtotal * TAX_RATE;
totalCost = subtotal + tax;

// Print output with appropriate formatting

System.out.println("Subtotal: " + fmt1.format(subtotal));
System.out.println("Tax: " + fmt1.format(tax) + " at "
    + fmt2.format(TAX_RATE));
System.out.println("Total: " + fmt1.format(totalCost));
}
}

```

**OUTPUT**

```

Enter the quantity: 6
Enter the unit price: 1.69
Subtotal: $10.14
Tax: $0.61 at 6%
Total: $10.75

```

operator to create a `NumberFormat` object and then sets up the object to format values in a particular way and returns it so that it can be used.

**The DecimalFormat Class**

Unlike the `NumberFormat` class, the `DecimalFormat` class is instantiated in the traditional way using the `new` operator. Its constructor takes a `String` parameter

`DecimalFormat (String pattern)`

Constructor: creates a new `DecimalFormat` object with the specified pattern.

`void applyPattern (String pattern)`

Applies the specified pattern to this `DecimalFormat` object.

`String format (double number)`

Returns a string containing the specified number formatted according to the current pattern.

**FIGURE 3.6** Some methods of the `DecimalFormat` class

that represents the pattern that will guide the formatting process. We can then use the `format` method to format a particular value. At a later point, if we want to change the pattern that the formatter object uses, we can invoke the `applyPattern` method. Figure 3.6 describes these methods.

The pattern defined by the string that is passed to the `DecimalFormat` constructor can get fairly elaborate. Various symbols are used to represent particular formatting guidelines. The pattern defined by the string "`0.###`", for example, indicates that at least one digit should be printed to the left of the decimal point and should be a zero if the integer portion of the value is zero. It also indicates that the fractional portion of the value should be rounded to three digits.

This pattern is used in the `CircleStats` program, shown in Listing 3.5, which reads the radius of a circle from the user and computes the circle's area and circumference. Trailing zeros, such as in the circle's area of `78.540`, are not printed using this pattern.

**LISTING 3.5**

```

*****
// CircleStats.java          Java Foundations
//
// Demonstrates the formatting of decimal values using the
// DecimalFormat class.
*****

import java.util.Scanner;
import java.text.DecimalFormat;

public class CircleStats
{

```

**LISTING 3.5***continued*

```

-----  

// Calculates the area and circumference of a circle given its  

// radius.  

-----  

public static void main(String[] args)  

{  

    int radius;  

    double area, circumference;  

    Scanner scan = new Scanner(System.in);  

    System.out.print("Enter the circle's radius: ");  

    radius = scan.nextInt();  

    area = Math.PI * Math.pow(radius, 2);  

    circumference = 2 * Math.PI * radius;  

    // Round the output to three decimal places  

    DecimalFormat fmt = new DecimalFormat("0.###");  

    System.out.println("The circle's area: " + fmt.format(area));  

    System.out.println("The circle's circumference: "  

        + fmt.format(circumference));
}

```

**OUTPUT**

```

Enter the circle's radius: 5
The circle's area: 78.54
The circle's circumference: 31.416

```

**The printf Method**

In addition to print and println, the System class has another output method called printf, which allows the user to print a formatted string containing data values. The first parameter to the method represents the format string, and the remaining parameters specify the values that are inserted into the format string.

For example, the following line of code prints an ID number and a name:

```
System.out.printf("ID: %5d\tName: %s", id, name);
```

The first parameter specifies the format of the output and includes literal characters that label the output values as well as escape characters such as \t. The pattern %5d indicates that the corresponding numeric value (`id`) should be printed in a field of five characters. The pattern %s matches the string parameter name. The values of `id` and `name` are inserted into the string, producing a result such as

```
ID: 24036 Name: Larry Flagelhopper
```

The `printf` method was added to Java to mirror a similar function used in programs written in the C programming language. This makes it easier for a programmer to translate (or *migrate*) an existing C program into Java.

Older software that still has value is called a *legacy system*. Maintaining a legacy system is often a costly effort because, among other things, it is based on older technologies. But in many cases, maintaining a legacy system is still more cost-effective than migrating it to new technology, such as writing it in a newer language. Adding the `printf` method is an attempt to make such migrations easier, and therefore less costly, by providing the same kind of output statement that C programmers have come to rely on.

However, using the `printf` method is not a particularly clean object-oriented solution to the problem of formatting output, so we avoid its use in this text.

**KEY CONCEPT**

The `printf` method was added to Java to support the migration of legacy systems.

**3.7 Enumerated Types**

Java provides the ability to define an *enumerated type*, which can then be used as the type of a variable when it is declared. An enumerated type establishes all possible values of a variable of that type by listing, or enumerating, them. The values are identifiers and can be anything desired.

For example, the following declaration defines an enumerated type called `Season` whose possible values are `winter`, `spring`, `summer`, and `fall`.

```
enum Season {winter, spring, summer, fall}
```

There is no limit to the number of values that you can list for an enumerated type. Once the type is defined, a variable can be declared of that type:

```
Season time;
```

The variable `time` is now restricted in the values it can take on. It can hold one of the four `Season` values, but nothing else. Java enumerated types are considered to be *type-safe*, which means that any attempt to use a value other than one of the enumerated values will result in a compile-time error.

**KEY CONCEPT**

Enumerated types are type-safe, ensuring that invalid values will not be used.

The values are accessed through the name of the type—for example,

```
time = Season.spring;
```

Enumerated types can be quite helpful in situations in which you have a relatively small number of distinct values that a variable can assume. For example, suppose we wanted to represent the various letter grades a student could earn. We might declare the following enumerated type:

```
enum Grade {A, B, C, D, F}
```

Any initialized variable that holds a `Grade` is guaranteed to have one of those valid grades. That's better than using a simple character or `String` variable to represent the grade, which could take on any value.

Suppose we also wanted to represent plus and minus grades, such as A– and B+. We couldn't use A– or B+ as values, because they are not valid identifiers (the characters '-' and '+' cannot be part of an identifier in Java). However, the same values could be represented using the identifiers `Aminus`, `Bplus`, and so on.

Internally, each value in an enumerated type is stored as an integer, which is referred to as its *ordinal value*. The first value in an enumerated type has an ordinal value of 0, the second has an ordinal value of 1, the third has an ordinal value of 2, and so on. The ordinal values are used internally only. You cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value.

An enumerated type is a special kind of class, and the variables of an enumerated type are object variables. Thus there are a few methods associated with all enumerated types. The `ordinal` method returns the numeric value associated with a particular enumerated type value. The `name` method returns the name of the value, which is the same as the identifier that defines the value.

Listing 3.6 shows a program called `IceCream` that declares an enumerated type and exercises some of its methods. Because enumerated types are special types of classes, they are not defined within a method. They can be defined either at the class level (within the class but outside a method), as in this example, or at the outermost level.

### LISTING 3.6

```
/*
 * IceCream.java          Java Foundations
 *
 * Demonstrates the use of enumerated types.
 */
```

### LISTING 3.6

*continued*

```
public class IceCream
{
    enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                 rockyRoad, mintChocolateChip, cookieDough}

    //-----// Creates and uses variables of the Flavor type.//-----
    public static void main(String[] args)
    {
        Flavor cone1, cone2, cone3;

        cone1 = Flavor.rockyRoad;
        cone2 = Flavor.chocolate;

        System.out.println("cone1 value: " + cone1);
        System.out.println("cone1 ordinal: " + cone1.ordinal());
        System.out.println("cone1 name: " + cone1.name());

        System.out.println();
        System.out.println("cone2 value: " + cone2);
        System.out.println("cone2 ordinal: " + cone2.ordinal());
        System.out.println("cone2 name: " + cone2.name());

        cone3 = cone1;

        System.out.println();
        System.out.println("cone3 value: " + cone3);
        System.out.println("cone3 ordinal: " + cone3.ordinal());
        System.out.println("cone3 name: " + cone3.name());
    }
}
```

### OUTPUT

```
cone1 value: rockyRoad
cone1 ordinal: 5
cone1 name: rockyRoad

cone2 value: chocolate
cone2 ordinal: 1
cone2 name: chocolate

cone3 value: rockyRoad
cone3 ordinal: 5
cone3 name: rockyRoad
```

## 3.8 Wrapper Classes

As we've discussed previously, Java represents data by using primitive types (such as `int`, `double`, `char`, and `boolean`) in addition to classes and objects. Having two categories of data to manage (primitive values and object references) can present a challenge in some circumstances. There are times when you may want to treat primitive data as though they were objects. In these cases we need to "wrap" a primitive value into an object.

A *wrapper class* represents a particular primitive type. For instance, the `Integer` class represents a simple integer value. An object created from the `Integer` class stores a single `int` value. The constructors of the wrapper classes accept the primitive value to store. Here is an example:

```
Integer ageObj = new Integer(40);
```

### KEY CONCEPT

A wrapper class allows a primitive value to be managed as an object.

Once this declaration and instantiation are performed, the `ageObj` object effectively represents the integer 40 as an object. It can be used wherever an object is needed in a program rather than a primitive type.

For each primitive type in Java there exists a corresponding wrapper class in the Java class library. All wrapper classes are defined in the `java.lang` package. Figure 3.7 shows the wrapper class that corresponds to each primitive type.

Note that there is even a wrapper class that represents the type `void`. However, unlike the other wrapper classes, the `Void` class cannot be instantiated. It simply represents the concept of a void reference.

Primitive Type	Wrapper Class
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>
<code>void</code>	<code>void</code>

FIGURE 3.7 Wrapper classes in the `java.lang` package

```
Integer (int value)
Constructor: creates a new Integer object storing the specified value.
```

```
byte byteValue ()
double doubleValue ()
float floatValue ()
int intValue ()
long longValue ()
Return the value of this Integer as the corresponding primitive type.
```

```
static int parseInt (String str)
Returns the int corresponding to the value stored in the specified string.
```

```
static String toBinaryString (int num)
static String toHexString (int num)
static String toOctalString (int num)
Returns a string representation of the specified integer value in the
corresponding base.
```

FIGURE 3.8 Some methods of the `Integer` class

Wrapper classes also provide various methods related to the management of the associated primitive type. For example, the `Integer` class contains methods that return the `int` value stored in the object and that convert the stored value to other primitive types. Figure 3.8 lists some of the methods found in the `Integer` class. The other wrapper classes have similar methods.

Note that the wrapper classes also contain static methods that can be invoked independent of any instantiated object. For example, the `Integer` class contains a static method called `parseInt` that converts an integer that is stored in a `String` to its corresponding `int` value. If the `String` object `str` holds the string "987", the following line of code converts the string into the integer value 987 and stores that value in the `int` variable `num`:

```
num = Integer.parseInt(str);
```

The Java wrapper classes often contain static constants that are helpful as well. For example, the `Integer` class contains two constants, `MIN_VALUE` and `MAX_VALUE`, that hold the smallest and largest `int` values, respectively. The other wrapper classes contain similar constants for their types.

## Autoboxing

*Autoboxing* is the automatic conversion between a primitive value and a corresponding wrapper object. For example, in the following code, an int value is assigned to an Integer object reference variable.

```
Integer obj1;
int num1 = 69;
obj1 = num1; // automatically creates an Integer object
```

The reverse conversion, called *unboxing*, also occurs automatically when needed. For example,

```
Integer obj2 = new Integer(69);
int num2;
num2 = obj2; // automatically extracts the int value
```

### KEY CONCEPT

Autoboxing provides automatic conversions between primitive values and corresponding wrapper objects.

Assignments between primitive types and object types are generally incompatible. The ability to autobox occurs only between primitive types and corresponding wrapper classes. In any other case, attempting to assign a primitive value to an object reference variable, or vice versa, will cause a compile-time error.

## Summary of Key Concepts

- The new operator returns a reference to a newly created object.
- Multiple reference variables can refer to the same object.
- Methods are often executed on a particular object, and that object's state usually affects the results.
- A class library provides useful support when one is developing programs.
- The Java standard class library is organized into packages.
- All classes of the java.lang package are automatically imported for every program.
- A pseudorandom number generator performs a complex calculation to create the illusion of randomness.
- All methods of the Math class are static, which means they are invoked through the class name.
- The printf method was added to Java to support the migration of legacy systems.
- Enumerated types are type-safe, ensuring that invalid values will not be used.
- A wrapper class allows a primitive value to be managed as an object.
- Autoboxing provides automatic conversions between primitive values and corresponding wrapper objects.

## Summary of Terms

**application programming interface (API)** A set of related classes that supports a particular aspect of programming.

**autoboxing** The automatic conversion of a primitive type to an object of its corresponding wrapper class.

**class library** A set of classes that supports the development of programs.

**constructor** A special method that has the same name as the class and is called when a object is created to set up the object initially.

**enumerated type** A Java data type in which all values of the type are explicitly listed.

**garbage collection** The process of reclaiming memory space that can no longer be used by a program.

**immutable** An object whose data (state) cannot be modified once it is created.

- import declaration** A programming statement used to specify which external classes (from which packages) are used in a program.
- instance** An object. An object is an instance of a class.
- instantiation** The process of creating a new object.
- package** A language-level organization mechanism for classes. Each class in the Java API belongs to a particular package.
- pseudorandom number generator** A program element that performs calculations to produce a series of numbers in seemingly random order.
- wrapper class** A class that corresponds to a particular primitive data type.

### Self-Review Questions

- SR 3.1 What does the new operator accomplish?
- SR 3.2 What is a null reference?
- SR 3.3 What is an alias? How is it related to garbage collection?
- SR 3.4 Write a declaration for a String variable called author and initialize it to the string "Fred Brooks". Draw a graphical representation of the variable and its value.
- SR 3.5 Write a statement that prints the value of a String object called title in all uppercase letters.
- SR 3.6 Write a declaration for a String variable called front and initialize it to the first 10 characters of another String object called description.
- SR 3.7 What is a Java package?
- SR 3.8 What does the java.net package contain? The java.swing package?
- SR 3.9 What package contains the Scanner class? The String class? The Random class? The Math class?
- SR 3.10 What does an import declaration accomplish?
- SR 3.11 Why doesn't the String class have to be specifically imported into our programs?
- SR 3.12 Given a Random object called rand, what does the call rand.nextInt() return?
- SR 3.13 Given a Random object called rand, what does the call rand.nextInt(20) return?

- SR 3.14 What is a class method (also called a static method)?
- SR 3.15 Write a statement that prints the sine of an angle measuring 1.23 radians.
- SR 3.16 Write a declaration for a double variable called result and initialize it to 5 raised to the power 2.5.
- SR 3.17 What are the steps to output a floating point value as a percentage using Java's formatting classes?
- SR 3.18 Write the declaration of an enumerated type that represents movie ratings.
- SR 3.19 How can we represent a primitive value as an object?

### Exercises

- EX 3.1 Write a statement that prints the number of characters in a String object called overview.
- EX 3.2 Write a statement that prints the eighth character of a String object called introduction.
- EX 3.3 Write a declaration for a String variable called change and initialize it to the characters stored in another String object called original with all 'e' characters changed to 'j'.
- EX 3.4 What output is produced by the following code fragment?
- ```
String m1, m2, m3;
m1 = "Quest for the Holy Grail";
m2 = m1.toLowerCase();
m3 = m1 + " " + m2;
System.out.println(m3.replace('h', 'z'));
```
- EX 3.5 What is the effect of the following import declaration?
- ```
import java.awt.*;
```
- EX 3.6 Assuming that a Random object called generator has been created, what is the range of the result of each of the following expressions?
- generator.nextInt(20)
  - generator.nextInt(8) + 1
  - generator.nextInt(45) + 10
  - generator.nextInt(100) - 50

- EX 3.7** Write code to declare and instantiate an object of the `Random` class (call the object reference variable `rand`). Then write a list of expressions using the `nextInt` method that generates random numbers in the following specified ranges, including the end points. Use the version of the `nextInt` method that accepts a single integer parameter.
- 0 to 10
  - 0 to 500
  - 1 to 10
  - 1 to 500
  - 25 to 50
  - 10 to 15
- EX 3.8** Write an assignment statement that computes the square root of the sum of `num1` and `num2` and assigns the result to `num3`.
- EX 3.9** Write a single statement that computes and prints the absolute value of `total`.
- EX 3.10** Write code statements to create a `DecimalFormat` object that will round a formatted value to four decimal places. Then write a statement that uses that object to print the value of the result, properly formatted.
- EX 3.11** Write code statements that prompt for and read a double value from the user and then print the result of raising that value to the fourth power. Output the results to three decimal places.
- EX 3.12** Write a declaration for an enumerated type that represents the days of the week.

### Programming Projects

- PP 3.1** Write an application that prompts for and reads the user's first name and last name (separately). Then print a string composed of the first letter of the user's first name, followed by the first five characters of the user's last name, followed by a random number in the range 10 to 99. Assume that the last name is at least five letters long. Similar algorithms are sometimes used to generate usernames for new computer accounts.
- PP 3.2** Write an application that prints the sum of cubes. Prompt for and read two integer values and print the sum of each value raised to the third power.

- PP 3.3** Write an application that creates and prints a random phone number of the form XXX-XXX-XXXX. Include the dashes in the output. Do not let the first three digits contain an 8 or 9 (but don't be more restrictive than that), and make sure that the second set of three digits is not greater than 742. *Hint:* Think through the easiest way to construct the phone number. Each digit does not have to be determined separately.

- PP 3.4** Write an application that reads the  $(x, y)$  coordinates for two points. Compute the distance between the two points using the following formula:

$$\text{Distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- PP 3.5** Write an application that reads the radius of a sphere and prints its volume and surface area. Use the following formulas, in which  $r$  represents the sphere's radius. Print the output to four decimal places.

$$\text{Volume} = \frac{4}{3}\pi r^3$$

$$\text{Surface area} = 4\pi r^2$$

- PP 3.6** Write an application that reads the lengths of the sides of a triangle from the user. Compute the area of the triangle using Heron's formula (below), in which  $s$  represents half of the perimeter of the triangle, and  $a$ ,  $b$ , and  $c$  represent the lengths of the three sides. Print the area to three decimal places.

$$\text{Area} = \sqrt{s(s - a)(s - b)(s - c)}$$

### Answers to Self-Review Questions

- SRA 3.1** The `new` operator creates a new instance (an object) of the specified class. The constructor of the class is then invoked to help set up the newly created object.
- SRA 3.2** A null reference is a reference that does not refer to any object. The reserved word `null` can be used to check for null references to avoid following them.
- SRA 3.3** Two references are aliases of each other if they refer to the same object. Changing the state of the object through one reference changes it for the other, because there is actually only one object. An object is marked for garbage collection only when there are no valid references to it.

- SRA 3.4 The following declaration creates a `String` variable called `author` and initializes it.

```
String author = new String("Fred Brooks");
```

For strings, this declaration could have been abbreviated as follows:

```
String author = "Fred Brooks";
```

This object reference variable and its value can be depicted as follows:



- SRA 3.5 The following statement prints the value of a `String` object in all uppercase letters.

```
System.out.println(title.toUpperCase());
```

- SRA 3.6 The following declaration creates a `String` object and sets it equal to the first 10 characters of the `String` called `description`.

```
String front = description.substring(0, 10);
```

- SRA 3.7 A Java package is a collection of related classes. The Java standard class library is a group of packages that supports common programming tasks.

- SRA 3.8 Each package contains a set of classes that supports particular programming activities. The classes in the `java.net` package support network communication, and the classes in the `javax.swing` class support the development of graphical user interfaces.

- SRA 3.9 The `Scanner` class and the `Random` class are part of the `java.util` package. The `String` and `Math` classes are part of the `java.lang` package.

- SRA 3.10 An import declaration establishes the fact that a program uses a particular class, specifying the package that the class is a part of. This allows the programmer to use the class name (such as `Random`) without having to fully qualify the reference (such as `java.util.Random`) every time.

- SRA 3.11 The `String` class is part of the `java.lang` package, which is automatically imported into any Java program. Therefore, no separate import declaration is needed.

- SRA 3.12 A call to the `nextInt` method of a `Random` object returns a random integer in the range of all possible `int` values, both positive and negative.

- SRA 3.13 Passing a positive integer parameter  $x$  to the `nextInt` method of a `Random` object returns a random number in the range of 0 to  $x - 1$ . Thus a call to `nextInt(20)` will return a random number in the range 0 to 19, inclusive.

- SRA 3.14 A class (or static) method can be invoked through the name of the class that contains it, such as `Math.abs`. If a method is not static, it can be executed only through an instance (an object) of the class.

- SRA 3.15 The following statement prints the sine of 1.23 radians.

```
System.out.println(Math.sin(1.23));
```

- SRA 3.16 The following declaration creates a double variable and initializes it to 5 raised to the power 2.5.

```
double result = Math.pow(5, 2.5);
```

- SRA 3.17 To output a floating point value as a percentage, you first obtain a formatter object using a call to the static method `getPercentageInstance` of the `NumberFormat` class. Then you pass the value to be formatted to the `format` method of the formatter object, which returns a properly formatted string. Here is an example:

```
NumberFormat fmt = NumberFormat.getPercentageInstance();
System.out.println(fmt.format(value));
```

- SRA 3.18 The following is a declaration of an enumerated type for movie ratings.

```
enum Ratings {G, PG, PG13, R, NC17}
```

- SRA 3.19 A wrapper class is defined in the Java standard class library for each primitive type. In situations where objects are called for, an object created from a wrapper class may suffice.