Strings, Classes, Objects, and Subroutines

Built-in Subroutines and Functions

Recall that a subroutine is a set of program instructions that have been chunked together and given a name. A subroutine is designed to perform some task. To get that task performed in a program, you can "call" the subroutine using a subroutine call statement. In Java, every subroutine is contained either in a class or in an object. Some classes that are standard parts of the Java language contain predefined subroutines that you can use. A value of type *String*, which is an object, contains subroutines that can be used to manipulate that string. These subroutines are "built into" the Java language. You can call all these subroutines without understanding how they were written or how they work. Indeed, that's the whole point of subroutines: A subroutine is a "black box" which can be used without knowing what goes on inside.

Let's first consider subroutines that are part of a class. One of the purposes of a class is to group together some variables and subroutines, which are contained in that class. These variables and subroutines are called static members of the class. You've seen one example: In a class that defines a program, the main() routine is a static member of the class. The parts of a class definition that define static members are marked with the reserved word "static", such as the word "static" in public static void main...

When a class contains a static variable or subroutine, the name of the class is part of the full name of the variable or subroutine. For example, the standard class named *System* contains a subroutine named exit. To use that subroutine in your program, you must refer to it as System.exit. This full name consists of the name of the class that contains the subroutine, followed by a period, followed by the name of the subroutine. This subroutine requires an integer as its parameter, so you would actually use it with a subroutine call statement such as

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

Calling System.exit will terminate the program and shut down the Java Virtual Machine. You could use it if you had some reason to terminate the program before the end of the main routine. (The parameter tells the computer why the program was terminated. A parameter value of 0 indicates that the program ended normally. Any other value indicates that the program was terminated because an error was detected, so you could call System.exit(1) to indicate that the program is ending because of an error. The parameter is sent back to the operating system; in practice, the value is usually ignored by the operating system.)

System is just one of many standard classes that come with Java. Another useful class is called *Math*. This class gives us an example of a class that contains static variables: It includes the variables Math.PI and Math.E whose values are the mathematical constants π and e. *Math* also contains a large number of mathematical "functions." Every subroutine performs some specific task. For some subroutines, that task is to compute or retrieve some data value. Subroutines of this type are called functions. We say that a function returns a value. Generally, the returned value is meant to be used somehow in the program that calls the function.

You are familiar with the mathematical function that computes the square root of a number. The corresponding function in Java is called Math.sqrt. This function is a static member subroutine of the class named *Math*. If x is any numerical value, then Math.sqrt(x) computes and returns the square root of that value.
Since Math.sqrt(x) represents a value, it doesn't make sense to put it on a line by itself in a subroutine call statement such as

```
Math.sqrt(x);   // This doesn't make sense!
```

What, after all, would the computer do with the value computed by the function in this case? You have to tell the computer to do something with the value. You might tell the computer to display it:

```
System.out.print( Math.sqrt(x) );  // Display the square root of x.
```

or you might use an assignment statement to tell the computer to store that value in a variable:

```
lengthOfSide = Math.sqrt(x);
```

The function call Math.sqrt(x) represents a value of type double, and it can be used anyplace where a numeric literal of type double could be used. The x in this formula represents the parameter to the subroutine; it could be a variable named "x", or it could be replaced by any expression that represents a numerical value. For example, Math.sqrt(2) computes the square root of 2, and Math.sqrt(a*a+b*b) would be legal as long as a and b are numeric variables.

The *Math* class contains many static member functions. Here is a list of some of the more important of them:

- Math.abs(x), which computes the absolute value of x.
- The usual trigonometric functions, Math.sin(x), Math.cos(x), and Math.tan(x). (For all the trigonometric functions, angles are measured in radians, not degrees.)

- The inverse trigonometric functions arcsin, arccos, and arctan, which are written as: Math.asin(x), Math.acos(x), and Math.atan(x). The return value is expressed in radians, not degrees.
- The exponential function Math.exp(x) for computing the number e raised to the power x, and the natural logarithm function Math.log(x) for computing the logarithm of x in the base e.
- Math.pow(x,y) for computing x raised to the power y.
- Math.floor(x), which rounds x down to the nearest integer value that is less than or equal to x. Even though the return value is mathematically an integer, it is returned as a value of type double, rather than of type int as you might expect. For example, Math.floor(3.76) is 3.0, and Math.floor(-4.2) is -5. The function Math.round(x) returns the integer that is closest to x, and Math.ceil(x) rounds x up to an integer. ("Ceil" is short for "ceiling", the opposite of "floor.")
- Math.random(), which returns a randomly chosen double in the range 0.0 <= Math.random() < 1.0. (The computer actually calculates so-called "pseudorandom" numbers, which are not truly random but are effectively random enough for most purposes.) We will find a lot of uses for Math.random in future examples.

For these functions, the type of the parameter—the x or y inside the parentheses—can be any value of any numeric type. For most of the functions, the value returned by the function is of type double no matter what the type of the parameter. However, for Math.abs(x), the value returned will be the same type as x; if x is of type int, then so is Math.abs(x). So, for example, while Math.sqrt(9) is the double value 3.0, Math.abs(9) is the int value 9.

Note that Math.random() does not have any parameter. You still need the parentheses, even though there's nothing between them. The parentheses let the computer know that this is a subroutine rather than a variable. Another example of a subroutine that has no parameters is the function System.currentTimeMillis(), from the *System* class. When this function is executed, it retrieves the current time, expressed as the number of milliseconds that have passed since a standardized base time (the start of the year 1970, if you care). One millisecond is one-thousandth of a second. The return value of System.currentTimeMillis() is of type long (a 64-bit integer). This function can be used to measure the time that it takes the computer to perform a task. Just record the time at which the task is begun and the time at which it is finished and take the difference. For more accurate timing, you can use System.nanoTime() instead. System.nanoTime() returns the number of nanoseconds since some arbitrary starting time, where one nanosecond is one-

billionth of a second. However, you should not expect the time to be truly accurate to the nanosecond.

Here is a sample program that performs a few mathematical tasks and reports the time that it takes for the program to run.

```
/**
 * This program performs some mathematical computations and displays the
 * results.  It also displays the value of the constant Math.PI.  It then
 * reports the number of seconds that the computer spent on this task.
 */
public class TimedComputation {

    public static void main(String[] args) {

        long startTime; // Starting time of program, in nanoseconds.
        long endTime;   // Time when computations are done, in nanoseconds.
        long compTime;  // Run time in nanoseconds.
        double seconds; // Time difference, in seconds.

        startTime = System.nanoTime();

        double width, height, hypotenuse;  // sides of a triangle
        width = 42.0;
        height = 17.0;
        hypotenuse = Math.sqrt( width*width + height*height );
        System.out.print("A triangle with sides 42 and 17 has hypotenuse ");
        System.out.println(hypotenuse);

        System.out.println("\nMathematically, sin(x)*sin(x) + "
                + "cos(x)*cos(x) - 1 should be 0.");
        System.out.println("Let's check this for x = 100:");
        System.out.print("      sin(100)*sin(100) + cos(100)*cos(100) - 1 is:
");
        System.out.println( Math.sin(100)*Math.sin(100)
                + Math.cos(100)*Math.cos(100) - 1 );
        System.out.println("(There can be round-off errors when"
                + " computing with real numbers!)");

        System.out.print("\nHere is a random number:  ");
        System.out.println( Math.random() );

        System.out.print("\nThe value of Math.PI is ");
        System.out.println( Math.PI );

        endTime = System.nanoTime();
        compTime = endTime - startTime;
        seconds = compTime / 1000000000.0;

        System.out.print("\nRun time in nanoseconds was: ");
        System.out.println(compTime);
        System.out.println("(This is probably not perfectly accurate!");
        System.out.print("\nRun time in seconds was:  ");
        System.out.println(seconds);

    } // end main()

} // end class TimedComputation
```

Classes and Objects

Classes can be containers for static variables and subroutines. However classes also have another purpose. They are used to describe objects. In this role, the class is a **type**, in the same way that int and double are types. That is, the class name can be used to declare variables. Such variables can only hold one type of value. The values in this case are objects. An object is a collection of variables and subroutines. Every object has an associated class that tells what "type" of object it is. The class of an object specifies what subroutines and variables that object contains. All objects defined by the same class are similar in that they contain similar collections of variables and subroutines. For example, an object might represent a point in the plane, and it might contain variables named x and y to represent the coordinates of that point. Every point object would have an x and a y, but different points would have different values for these variables. A class, named *Point* for example, could exist to define the common structure of all point objects, and all such objects would then be values of type *Point*.

As another example, let's look again at System.out.println. *System* is a class, and out is a static variable within that class. However, the value of System.out is an **object**, and System.out.println is actually the full name of a subroutine that is contained in the object System.out. You don't need to understand it at this point, but the object referred to by System.out is an object of the class *PrintStream*. *PrintStream* is another class that is a standard part of Java. **Any** object of type *PrintStream* is a destination to which information can be printed; **any** object of type *PrintStream* has a println subroutine that can be used to send information to that destination. The object System.out is just one possible destination, and System.out.println is a subroutine that sends information to that particular destination. Other objects of type *PrintStream* might send information to other destinations such as files or across a network to other computers. This is object-oriented programming: Many different things which have something in common—they can all be used as destinations for output—can all be used in the same way—through a println subroutine. The *PrintStream* class expresses the commonalities among all these objects.

The dual role of classes can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. Since class names and variable names are used in similar ways, it might be hard to tell which is which. Remember that all the built-in, predefined names in Java follow the rule that class names begin with an upper case letter while variable names begin with a lower case letter. While this is not a formal syntax rule, I strongly recommend that you follow it in your own programming. Subroutine names should also begin with lower case letters. There is no possibility of confusing a variable with a subroutine, since a subroutine name in a program is always followed by a left parenthesis.

As one final general note, you should be aware that subroutines in Java are often referred to as methods. Generally, the term "method" means a subroutine that is contained in a class or in an object. Since this is true of every subroutine in Java, every subroutine in Java is a method.

Operations on Strings

String is a class, and a value of type *String* is an object. That object contains data, namely the sequence of characters that make up the string. It also contains subroutines. All of these subroutines are in fact functions. For example, every string object contains a function named length that computes the number of characters in that string. Suppose that advice is a variable that refers to a *String*. For example, advice might have been declared and assigned a value as follows:

```
String advice;
advice = "Seize the day!";
```

Then advice.length() is a function call that returns the number of characters in the string "Seize the day!". In this case, the return value would be 14. In general, for any variable str of type *String*, the value of str.length() is an int equal to the number of characters in the string. Note that this function has no parameter; the particular string whose length is being computed is the value of str. The length subroutine is defined by the class *String*, and it can be used with any value of type *String*. It can even be used with *String* literals, which are, after all, just constant values of type *String*. For example, you could have a program count the characters in "Hello World" for you by saying

```
System.out.print("The number of characters in ");
System.out.print("the string \"Hello World\" is ");
System.out.println( "Hello World".length() );
```

The *String* class defines a lot of functions. Here are some that you might find useful. Assume that s1 and s2 are variables of type *String*:

- s1.equals(s2) is a function that returns a boolean value. It returns true if s1 consists of exactly the same sequence of characters as s2, and returns false otherwise.
- s1.equalsIgnoreCase(s2) is another boolean-valued function that checks whether s1 is the same string as s2, but this function considers upper and lower case letters to be equivalent. Thus, if s1 is "cat", then s1.equals("Cat") is false, while s1.equalsIgnoreCase("Cat") is true.

- s1.length(), as mentioned above, is an integer-valued function that gives the number of characters in s1.
- s1.charAt(N), where N is an integer, returns a value of type char. It returns the Nth character in the string. Positions are numbered starting with 0, so s1.charAt(0) is actually the first character, s1.charAt(1) is the second, and so on. The final position is s1.length() - 1. For example, the value of "cat".charAt(1) is 'a'. An error occurs if the value of the parameter is less than zero or is greater than or equal to s1.length().
- s1.substring(N,M), where N and M are integers, returns a value of type *String*. The returned value consists of the characters of s1 in positions N, N+1,..., M-1. Note that the character in position M is not included. The returned value is called a substring of s1. The subroutine s1.substring(N) returns the substring of s1 consisting of characters starting at position N up until the end of the string.
- s1.indexOf(s2) returns an integer. If s2 occurs as a substring of s1, then the returned value is the starting position of that substring. Otherwise, the returned value is -1. You can also use s1.indexOf(ch) to search for a char, ch, in s1. To find the first occurrence of x at or after position N, you can use s1.indexOf(x,N). To find the last occurrence of x in s1, use s1.lastIndexOf(x).
- s1.compareTo(s2) is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If s1 is less than s2, the value returned is a number less than zero, and if s1 is greater than s2, the value returned is some number greater than zero. There is also a function s1.compareToIgnoreCase(s2). (If both of the strings consist entirely of lower case letters, or if they consist entirely of upper case letters, then "less than" and "greater than" refer to alphabetical order. Otherwise, the ordering is more complicated.)
- s1.toUpperCase() is a *String*-valued function that returns a new string that is equal to s1, except that any lower case letters in s1 have been converted to upper case. For example, "Cat".toUpperCase() is the string "CAT". There is also a function s1.toLowerCase().
- s1.trim() is a *String*-valued function that returns a new string that is equal to s1 except that any non-printing characters such as spaces and tabs have been trimmed from the beginning and from the end of the string. Thus, if s1 has the value "fred ", then s1.trim() is the string "fred", with the spaces at the end removed.

For the functions s1.toUpperCase(), s1.toLowerCase(), and s1.trim(), note that the value of s1 is **not** changed. Instead a new string is created and returned as the value of the function. The returned value could be used, for example, in an assignment

statement such as "smallLetters = s1.toLowerCase();". To change the value of s1, you could use an assignment "s1 = s1.toLowerCase();".

Here is another extremely useful fact about strings: You can use the plus operator, +, to concatenate two strings. The concatenation of two strings is a new string consisting of all the characters of the first string followed by all the characters of the second string. For example, "Hello" + "World" evaluates to "HelloWorld". (Gotta watch those spaces, of course—if you want a space in the concatenated string, it has to be somewhere in the input data, as in "Hello " + "World".)

Let's suppose that name is a variable of type *String* and that it already refers to the name of the person using the program. Then, the program could greet the user by executing the statement:

```
System.out.println("Hello, "  +  name  +  ".  Pleased to meet you!");
```

Even more surprising is that you can actually concatenate values of **any** type onto a *String* using the + operator. The value is converted to a string, just as it would be if you printed it to the standard output, and then that string is concatenated with the other string. For example, the expression "Number" + 42 evaluates to the string "Number42". And the statements

```
System.out.print("After ");
System.out.print(years);
System.out.print(" years, the value is ");
System.out.print(principal);
```

can be replaced by the single statement:

```
System.out.print("After " + years +
                    " years, the value is " + principal);
```

Obviously, this is very convenient. It would have shortened some of the examples presented earlier in this chapter.

Introduction to Enums

Java comes with eight built-in primitive types and a huge collection of types that are defined by classes, such as *String*. But even this large collection of types is not sufficient to cover all the possible situations that a programmer might have to deal

with. So, an essential part of Java, just like almost any other programming language, is the ability to create **new** types.

Technically, an enum is considered to be a special kind of class, but that is not important for now. We will look at enums in a simplified form. In practice, most uses of enums will only need the simplified form that is presented here.

An enum is a type that has a fixed list of possible values, which is specified when the enum is created. In some ways, an enum is similar to the boolean data type, which has true and false as its only possible values. However, boolean is a primitive type, while an enum is not.

The definition of an enum type has the (simplified) form:

```
enum enum-type-name { list-of-enum-values }
```

This definition cannot be inside a subroutine. You can place it **outside** the main() routine of the program (or it can be in a separate file). The **enum-type-name** can be any simple identifier. This identifier becomes the name of the enum type, in the same way that "boolean" is the name of the boolean type and "String" is the name of the *String* type. Each value in the **list-of-enum-values** must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named *Season* whose values are the names of the four seasons of the year:

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

By convention, enum values are given names that are made up of upper case letters, but that is a style guideline and not a syntax rule. An enum value is a constant; that is, it represents a fixed value that cannot be changed. The possible values of an enum type are usually referred to as enum constants.

Note that the enum constants of type *Season* are considered to be "contained in" *Season*, which means—following the convention that compound identifiers are used for things that are contained in other things—the names that you actually use in your program to refer to them are Season.SPRING, Season.SUMMER, Season.FALL, and Season.WINTER.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named vacation of type Season with the statement:

```
Season vacation;
```

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type Season. Remember to use the full name of the constant, including "Season"! For example:

```
vacation = Season.SUMMER;
```

You can print out an enum value with an output statement such as System.out.print(vacation). The output value will be the name of the enum constant (without the "Season."). In this case, the output would be "SUMMER".

Because an enum is technically a class, the enum values are technically objects. As objects, they can contain subroutines. One of the subroutines in every enum value is named ordinal(). When used with an enum value, it returns the ordinal number of the value in the list of values of the enum. The ordinal number simply tells the position of the value in the list. That is, Season.SPRING.ordinal() is the int value 0, Season.SUMMER.ordinal() is 1, Season.FALL.ordinal() is 2, and Season.WINTER.ordinal() is 3. (You will see over and over again that computer scientists like to start counting at zero!) You can, of course, use the ordinal() method with a variable of type *Season*, such as vacation.ordinal().

Using enums can make a program more readable, since you can use meaningful names for the values. And it can prevent certain types of errors, since a compiler can check that the values assigned to an enum variable are in fact legal values for that variable. For now, you should just appreciate them as the first example of an important concept: creating new types. Here is a little example that shows enums being used in a complete program:

```
public class EnumDemo {

       // Define two enum types -- remember that the definitions
       // go OUTSIDE the main() routine!

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

    enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }

    public static void main(String[] args) {

         Day tgif;     // Declare a variable of type Day.
         Month libra;  // Declare a variable of type Month.

         tgif = Day.FRIDAY;    // Assign a value of type Day to tgif.
         libra = Month.OCT;    // Assign a value of type Month to libra.

         System.out.print("My sign is libra, since I was born in ");
         System.out.println(libra);   // Output value will be:  OCT
         System.out.print("That's the ");
         System.out.print( libra.ordinal() );
```

```
            System.out.println("-th month of the year.");
            System.out.println("   (Counting from 0, of course!)");

            System.out.print("Isn't it nice to get to ");
            System.out.println(tgif);   // Output value will be:  FRIDAY

            System.out.println( tgif + " is the " + tgif.ordinal()
                                                  + "-th day of the week.");
    }

}
```

Text Blocks: Multiline Strings

Java 15 introduced a new kind of string literal to represent multiline strings. (Recall that a literal is something you type in program to represent a constant value.) The new literals are called text blocks. A text block starts with a string of three double-quote characters, followed by optional white space and then a new line. The white space and newline are not part of the string constant that is represented by the text block. The text block is terminated by another string of three double-quote characters. A text block can be used anywhere an ordinary string literal could be used. For example,

```
String poem = """
    As I was walking down the stair,
       I met a man who wasn't there.
    He wasn't there again today.
       I wish, I wish he'd go away!""";
```

This is easier to write and to read than the following equivalent code, which builds up the multiline string using concatenation:

```
String poem = "As I was walking down the stair,\n"
    + "   I met a man who wasn't there.\n"
    + "He wasn't there again today.\n"
    + "   I wish, I wish he'd go away!\n";
```

Note that extra white space at the beginning of each line of the text block is removed from the string that is represented by the literal, but that newlines are preserved.

A textblock can include escaped characters such as \t or \\, but aside from the backslash character, '\', nothing in the text block has special meaning. For example, something in the text block that looks like a Java comment is not actually a comment; it is just ordinary characters that are part of the string.