STATIC SUBROUTINES IN JAVA

EVERY SUBROUTINE IN JAVA must be defined inside some class. This makes Java rather unusual among programming languages, since most languages allow free-floating, independent subroutines. One purpose of a class is to group together related subroutines and variables. Perhaps the designers of Java felt that everything must be related to something. As a less philosophical motivation, Java's designers wanted to place firm controls on the ways things are named, since a Java program potentially has access to a huge number of subroutines created by many different programmers. The fact that those subroutines are grouped into named classes (and classes are grouped into named "packages") helps control the confusion that might result from so many different names.

4.2.1 Subroutine Definitions

A subroutine definition in Java takes the form:

```
modifiers  return-type  subroutine-name  ( parameter-list ) {
    statements
}
```

It will take us a while -- most of the chapter -- to get through what all this means in detail. Of course, you've already seen examples of subroutines in previous chapters, such as the `main()` routine of a program and the `drawFrame()` routine of the animation applets in Section 3.8. So you are familiar with the general format.

The **statements** between the braces, { and }, in a subroutine definition make up the body of the subroutine. These statements are the inside, or implementation part, of the "black box", as discussed in theprevious section. They are the instructions that the computer executes when the method is called. Subroutines can contain any of the statements discussed in Chapter 2 and Chapter 3.

The **modifiers** that can occur at the beginning of a subroutine definition are words that set certain characteristics of the subroutine, such as whether it is static or not. The modifiers that you've seen so far are "`static`" and "`public`". There are only about a half-dozen possible modifiers altogether.

If the subroutine is a function, whose job is to compute some value, then the **return-type** is used to specify the type of value that is returned by the function. We'll be looking at functions and return types in some detail in Section 4.4. If the subroutine is not a function, then the **return-type** is replaced by the special value `void`, which indicates that no value is returned. The term "void" is meant to indicate that the return value is empty or non-existent.

Finally, we come to the **parameter-list** of the method. Parameters are part of the interface of a subroutine. They represent information that is passed into the subroutine from outside, to be used by the subroutine's internal computations. For a concrete example, imagine a class named *Television* that includes a method named `changeChannel()`. The immediate question is: What channel should it change to? A parameter can be used to answer this question. Since the channel number is an integer, the type of the parameter would be int, and the declaration of the `changeChannel()` method might look like

```
public void changeChannel(int channelNum) { ... }
```

This declaration specifies that `changeChannel()` has a parameter named `channelNum` of type int. However, `channelNum` does not yet have any particular value. A value for `channelNum` is provided when the subroutine is called; for example: `changeChannel(17);`

The parameter list in a subroutine can be empty, or it can consist of one or more parameter declarations of the form **type parameter-name**. If there are several declarations, they are separated by commas. Note that each declaration can name only one parameter. For example, if you want two parameters of type double, you have to say "`double x, double y`", rather than "`double x, y`".

Parameters are covered in more detail in the next section.

Here are a few examples of subroutine definitions, leaving out the statements that define what the subroutines do:

```
public static void playGame() {
    // "public" and "static" are modifiers; "void" is the
    // return-type; "playGame" is the subroutine-name;
    // the parameter-list is empty.
    . . .  // Statements that define what playGame does go here.
}

int getNextN(int N) {
    // There are no modifiers; "int" in the return-type;
    // "getNextN" is the subroutine-name; the parameter-list
    // includes one parameter whose name is "N" and whose
    // type is "int".
    . . .  // Statements that define what getNextN does go here.
}

static boolean lessThan(double x, double y) {
    // "static" is a modifier; "boolean" is the
    // return-type; "lessThan" is the subroutine-name;
    // the parameter-list includes two parameters whose names are
    // "x" and "y", and the type of each of these parameters
    // is "double".
    . . .  // Statements that define what lessThan does go here.
}
```

In the second example given here, `getNextN` is a non-static method, since its definition does not include the modifier "`static`" -- and so it's not an example that we should be looking at in this chapter! The other modifier shown in the examples is "`public`". This modifier indicates that the method can be called from anywhere in a program, even from outside the class where the method is defined. There is another modifier, "`private`", which indicates that the method can be called **only** from inside the same class. The modifiers `public` and `private` are called access specifiers. If no access specifier is given for a method, then by default, that method can be called from anywhere in the "package" that contains the class, but not from outside that package. (Packages were introduced in Subsection 2.6.4, and you'll learn more about them later in this chapter, in Section 4.5.) There is one other access modifier, `protected`, which will only become relevant when we turn to object-oriented programming in Chapter 5.

Note, by the way, that the `main()` routine of a program follows the usual syntax rules for a subroutine. In

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

the modifiers are `public` and `static`, the return type is `void`, the subroutine name is `main`, and the parameter list is "`String[] args`". The only question might be about "`String[]`", which has to be a type if it is to match the syntax of a parameter list. In fact, `String[]` represents a so-called "array type", so the syntax is valid. We will cover arrays in Chapter 7. (The parameter, `args`, represents information provided to the program when the `main()` routine is called by the system. In case you know the term, the information consists of any "command-line arguments" specified in the command that the user typed to run the program.)

You've already had some experience with filling in the implementation of a subroutine. In this chapter, you'll learn all about writing your own complete subroutine definitions, including the interface part.

4.2.2 Calling Subroutines

When you define a subroutine, all you are doing is telling the computer that the subroutine exists and what it does. The subroutine doesn't actually get executed until it is called. (This is true even for the `main()`routine in a class -- even though **you** don't call it, it is called by the system when the system runs your program.) For example, the `playGame()` method given as an example above could be called using the following subroutine call statement:

```
playGame();
```

This statement could occur anywhere in the same class that includes the definition of `playGame()`, whether in a `main()` method or in some other subroutine. Since `playGame()` is a `public` method, it can also be called from other classes, but in that case, you have to tell the computer which class it comes from. Since `playGame()` is a `static` method, its full name includes the name of the class in which it is defined. Let's say, for example, that `playGame()` is defined in a class named `Poker`. Then to call `playGame()` from **outside** the `Poker` class, you would have to say

```
Poker.playGame();
```

The use of the class name here tells the computer which class to look in to find the method. It also lets you distinguish between `Poker.playGame()` and other potential `playGame()` methods defined in other classes, such as `Roulette.playGame()` or `Blackjack.playGame()`.

More generally, a subroutine call statement for a `static` subroutine takes the form

```
subroutine-name(parameters);
```

if the subroutine that is being called is in the same class, or

```
class-name.subroutine-name(parameters);
```

if the subroutine is defined elsewhere, in a different class. (Non-static methods belong to objects rather than classes, and they are called using object names instead of class names. More on that later.) Note that the parameter list can be empty, as in the `playGame()` example, but the parentheses must be there even if there is nothing between them. The number of parameters that you provide when you call a subroutine must match the number listed in the parameter list in the subroutine definition, and the types of the parameters in the call statement must match the types in the subroutine definition.

Source : http://math.hws.edu/javanotes/c4/s2.html