CHAPTER 9

METHODS – LOGICAL PERFORMING UNITS IN PROGRAMS

The word "method" has already been mentioned many times in this book. A method is a piece of program code that performs a well-defined task. In every program we have had a method named **`main()`**. Method **`main()`** is always executed first when the execution of a program begins. We have also encountered methods **`print()`**, **`printf()`**, and **`nextLine()`** which are standard methods to write data to the screen and read data from the keyboard. In Chapter 8 we studied many string methods (**`indexOf()`**, **`substring()`**, **`compareTo()`**, etc.) which are standard methods to manipulate strings. Now we are going to take a closer look at methods and their use. You will learn how to write your own methods. The term "calling" will be an important concept associated with methods.

With Java we can write static methods and non-static instance methods. The methods that we are going to study in this chapter are static methods. The instance methods, which are in some ways different from the static methods, will be studied in the following chapter.

9.1 Simple static methods and the concept of calling

All programs that we have studied so far are of the form

```
// SomeName.java

class SomeName
{
   public static void main( String[] not_in_use )
   {
      Statements that declare data (variables, objects, and arrays).

      Functional action statements.
   }
}
```

The source program statements that we have seen so far have been statements of the method named **main()**. Method **main()** has always the reserved words **public**, **static**, and **void** preceding its name, and **String[] not_in_use** is written inside parentheses after the method name **main**. The Java statements that dictate what method **main()** does are inside a pair of braces { }.

From now on, we will start studying programs that may contain several methods. The simplest form of a method is such that the type of the method is **void**, and the parentheses after the method name are empty.

Program **Messages.java** is an example where a simple method is called inside the method **main()**. Although the structure of **Messages.java** is such that the source code of the method **print_message()** is written first and method **main()** is at the end of program, the program execution starts from method **main()**. Method **main()** is the "main program" in the file. When a Java program is executed on a computer, the operating system first activates the Java interpreter, the Java virtual machine, and then the virtual machine starts the program by executing the method that is named **main()**.

Method **print_message()** in program **Messages.java** can be considered a subroutine because its execution is completely controlled by method **main()**. The source code of the subroutine starts

```
static void print_message()
{
   ...
```

Note that there is no semicolon (;) here.

and it is called inside method **main()** simply by writing the method name in the following way

```
print_message() ;
```

What happens in a method call is that the calling method stops running, and the statements of the method that was called are executed. When all statements of the method that was called are executed, the program execution continues in the calling method from the statement that follows the method call.

In **Messages.java**, method **print_message()** is called twice. By studying the output you can find out that **print_message()** always prints the same text lines, while method **main()** prints something else in between the message from **print_message()**. Program **Messages.java** could, of course, be written without method **print_message()**. If the statements inside method **print_message()** were copied to those two places where **print_message()** is called in method **main()**, the program would behave in the same way as it is doing now, but it would not need any method calls.

The statements that form the body of method **print_message()** are inside these braces. The structure of this method is similar to the structure of method **main()**. Generally, the name of a method can be invented by the programmer, but method **main()** must have that name.

All methods must be written inside some class declaration. In this program, class **Messages** contains two separate methods.

```
//  Messages.java (c) Kari Laitinen

class Messages
{
   static void print_message()
   {
      System.out.print( "\n   This is method named \"print_message()\"." ) ;
      System.out.print( "\n   Methods usually contain many statements. " ) ;
      System.out.print( "\n   Let us now return to the calling method." ) ;
   }

   public static void main( String[] not_in_use )
   {
      System.out.print( "\n THE FIRST STATEMENT IN METHOD \"main()\"." ) ;

      print_message() ;

      System.out.print( "\n THIS IS BETWEEN TWO METHOD CALLS." ) ;

      print_message() ;

      System.out.print( "\n END OF METHOD \"main()\".\n" ) ;
   }
}
```

Method **print_message()** is called twice inside method **main()**. A simple static method belonging to the same class as the calling method can be called by writing its name, a pair of empty parentheses, and a semicolon. Method calls are statements in Java. What happens in a method call is that the statements inside the called method are executed, and program execution continues from the statement that follows the method call in the calling method.

Messages.java - 1. Method main() calling a simple method named print_message().

```
D:\javafiles2>java Messages

 THE FIRST STATEMENT IN METHOD "main()".
   This is method named "print_message()".
   Methods usually contain many statements.
   Let us now return to the calling method.
 THIS IS BETWEEN TWO METHOD CALLS.
   This is method named "print_message()".
   Methods usually contain many statements.
   Let us now return to the calling method.
 END OF METHOD "main()".
```

Those lines containing mostly lowercase letters are printed by method **print_message()**.

Messages.java - X. Method print_message() prints always the same message.

In programming terminology, the method that calls another method is the caller, and the method that is called is the callee. In **Messages.java**, method **`main()`** is the caller and method **`print_message()`** is the callee. A caller calls a callee like an employer employs an employee. A callee is always subordinate to its caller. The caller decides when a callee is executed. The caller continues by executing the statements that follow the method call when the statements of a callee have been executed.

Methods are executed, statement by statement, from the first statement to the last statement. Although computers can execute statements extremely fast, only one statement is being executed at a time. To better understand what is happening when a program is being executed, we can think that there exists such a thing as "program control". The program control is at that statement which is currently being executed. When the current statement has been completely executed, the program control is passed to the following statement. The program control is at the first executable statement of method **`main()`** when the execution of a program begins. When the last statement of method **`main()`** is executed, the program control is passed back to the operating system of the computer.

A method call is a statement that passes the program control to the called method, the callee. Just after the execution of a method call, the program control is at the first executable statement in callee. The program control goes through every statement in callee. After the last statement in the callee has been executed, the program control is passed to the statement that follows the method call in caller.

In large computer programs there are methods that call other methods that call other methods that call other methods ... In well-designed programs there is, of course, always a last method that is called but which does not call any other methods. In large programs, methods are useful because they allow programs to be divided into manageable pieces of source code. Program **Letters.java** is an example where a called method calls two other methods. Method **`print_letters()`** is a callee in relation to method **`main()`**, but it is a caller in relation to the two other methods.

Although **Letters.java** does not do anything that could be considered as creative computing (i.e. the program is a simple textbook program), the program is an example of how a programming task can be divided into smaller programming tasks with the help of methods. What program **Letters.java** does is that it prints all letters of the English alphabet. First it prints all uppercase letters and then it prints all lowercase letters. We can imagine that **Letters.java** is the result of a software development project. A boss in a software company could have started a software project to produce a program that first prints all uppercase letters and then all lowercase letters. The software developers working on the project could have divided the programming work into the subtasks

- print uppercase letters
- print lowercase letters

which would have been implemented (i.e. programmed) as two separate methods by different people.

A method is a piece of source program that performs a certain activity. When a caller calls a method, the call is like a command to perform the activity that is programmed inside the method. Because method calls are like commands, it is usual that method names are in a commanding, imperative form. For example, the method names

```
print_uppercase_letters
print_message
```

are in the form of a command, since an imperative verb is the first word in the name. Technically, programmers are free to name methods according to the general naming rules of Java, but it is useful to name methods so that they are commands. This way method names can be easily distinguished from variable names. Inventing accurate and descriptive names for the methods you write helps you to understand your programming task better.

```
//  Letters.java (c) 2005 Kari Laitinen

class Letters
{
   static void print_uppercase_letters()
   {
      System.out.print( "\n Uppercase English letters are: \n\n" ) ;

      for ( char letter_to_print  =  'A' ;
                 letter_to_print  <= 'Z' ;
                 letter_to_print  ++  )
      {
         System.out.print(  " "  +  letter_to_print ) ;
      }
   }

   static void print_lowercase_letters()
   {
      System.out.print( "\n\n Lowercase English letters are: \n\n" ) ;

      for ( char letter_to_print  =  'a' ;
                 letter_to_print  <= 'z' ;
                 letter_to_print  ++ )
      {
         System.out.print(  " "  +  letter_to_print ) ;
      }
   }

   static void print_letters()
   {
      print_uppercase_letters() ;
      print_lowercase_letters() ;
   }

   public static void main( String[] not_in_use )
   {
      print_letters() ;
   }
}
```

These two methods are called by the method **print_letters()**.

The method called by **main()** contains two other method calls.

Method **main()** has only one statement which is a method call. These methods are in such an order that a callee is always written before the caller. In this book programs are generally written so that the method that will be called later in the program is placed before the calling method in the source program file.

Letters.java - 1. Method main() calling a method that calls two other methods.

```
D:\javafiles2>java Letters

 Uppercase English letters are:

 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

 Lowercase English letters are:

 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Letters.java - X. All text is printed here by the two topmost methods.

9.2 Methods that take parameters

The methods that we have studied so far are called in the following way

```
method_name() ;
```

You may have wondered why there always has to be a pair of empty parentheses () at the end of a method's name. In your mind you may have asked: "Can a pair of empty parentheses bear any meaning?" The answer is that they can. A pair of empty parentheses means that the method in question does not take any parameters. This, then, raises yet another question: what are parameters?

Parameters, which are sometimes called arguments, are data that are transferred between a method and its caller. A method usually performs a specific activity, and it may need some data while performing the activity that is programmed inside it. With the help of parameters, a calling method can provide the necessary data for a called method.

Program **Sums.java** contains a method named `print_sum()`, and that method takes parameters. We say that it takes two parameters of type `int` because two variables of type `int` are declared inside the parentheses that have so far been empty. Method `print_sum()` is called in **Sums.java** in the following ways

```
print_sum( 555, 222 ) ;
print_sum( first_integer, second_integer ) ;
```

In the first method call, `print_sum()` calculates and prints the sum of the two integer literals 555 and 222. In the second method call, `print_sum()` calculates and prints the sum of the integers stored in the variables `first_integer` and `second_integer`.

When method `print_sum()` is called, data is transferred from the caller to the callee. When the call statement is executed, the data that is supplied as parameters is copied to the parameter variables declared inside method `print_sum()`, and after that the internal statements of the called method are executed.

To explore the calling mechanism of methods, let's compare the calling of methods that take parameters and the calling of methods that take no parameters. When a method without parameters is called, for example, in the following way

```
print_message() ;
```

the method call means "Go and execute the internal statements of method `print_message()`". When a method that takes parameters is called, for example, as follows

```
print_sum( 1122, 3344 ) ;
```

the call means "First copy value 1122 to the first parameter variable of `print_sum()`, then copy 3344 to the second parameter variable, and then go and execute the internal statements of `print_sum()`". There is much meaning embedded in the latter method call. When the method call is compiled, the compiler generates the necessary bytecode instructions to copy the parameter data to the parameter variables.

The source code of method `print_sum()` begins with the lines

```
static void print_sum( int first_integer_from_caller,
                       int second_integer_from_caller )
{
   ...
```

The first two lines before the opening brace { are said to be the declarator of the method. The declarator specifies the most necessary information for the caller:

- the return type of the method (The return type is `void` in the above declarator, but later we will see methods which have more interesting return types.),
- the method name, and
- the names of the parameters and their types.

For the caller of a method, the information given in the method declarator is sufficient to use the method. The caller must know how a method works, but a method can be called without knowing the internal implementation of the method precisely. The implementation of a method means its internal program structure: what kinds of loops are used, which variables are declared, etc. We have already used methods without knowing how they are written internally. In Chapter 8 we used the string methods **`compareTo()`**, **`substring()`**, etc. We called these methods without seeing their actual source code.

A method may be called only with those parameter types that are specified in the declarator of the method. The declarator of **`print_sum()`** says that it accepts two and only two parameters of type **`int`**. Therefore, the method calls

```
print_sum( "Hello", "world" ) ;
print_sum( 33, 44, 55 ) ;
print_sum( 6666 ) ;
```

are illegal and result in compilation error. The first call above is not accepted by the Java compiler because the two parameters are not of type **`int`**. The last two method calls are not acceptable because, although the parameters are integer literals, the number of parameters is not equal to two.

Program **Decorations.java** is another example where methods take parameters. In that program, method **`main()`** calls a method that takes a string reference as a parameter. The first method call is

```
print_text_in_decorated_box( first_text ) ;
```

which means roughly the following: "Here is the string reference **`first_text`**. Make the local string reference **`text_from_caller`** reference the **`String`** object that is referenced by **`first_text`**. Then execute the internal statements of method **`print_text_in_decorated_box()`**". Figure 9-1 describes the situation when the above method call has been executed. As you can see by studying the output of **Decorations.java**, the text that is referenced by **`first_text`** is printed inside a kind of decorated box. Method **`print_text_in_decorated_box()`** can read the characters of the **`String`** object referenced by **`first_text`**, but no copy of the **`String`** object is made. When variables are method parameters, they are copied to the parameter variables of the called method, the callee. Also when **`String`** objects are used as parameters, references to the objects are passed to the callee, but no copies of the objects are made. Also arrays and other types of objects, which we'll study later, are passed to the callee in the same way as **`String`** objects.

When a called method receives a copy of a parameter, we say that the parameter is passed by value. In Java, parameters are always passed by value. When method parameters are objects, the passed values are references to the objects. In some other programming languages (e.g. C++ and C#) parameters can be passed by reference, which means that the parameters of a method can be declared so that when the method is called it can refer to data items declared in the calling method. The developers of Java wanted to create a simple programming language, and, hence, there is only one way to pass parameters to a called method.

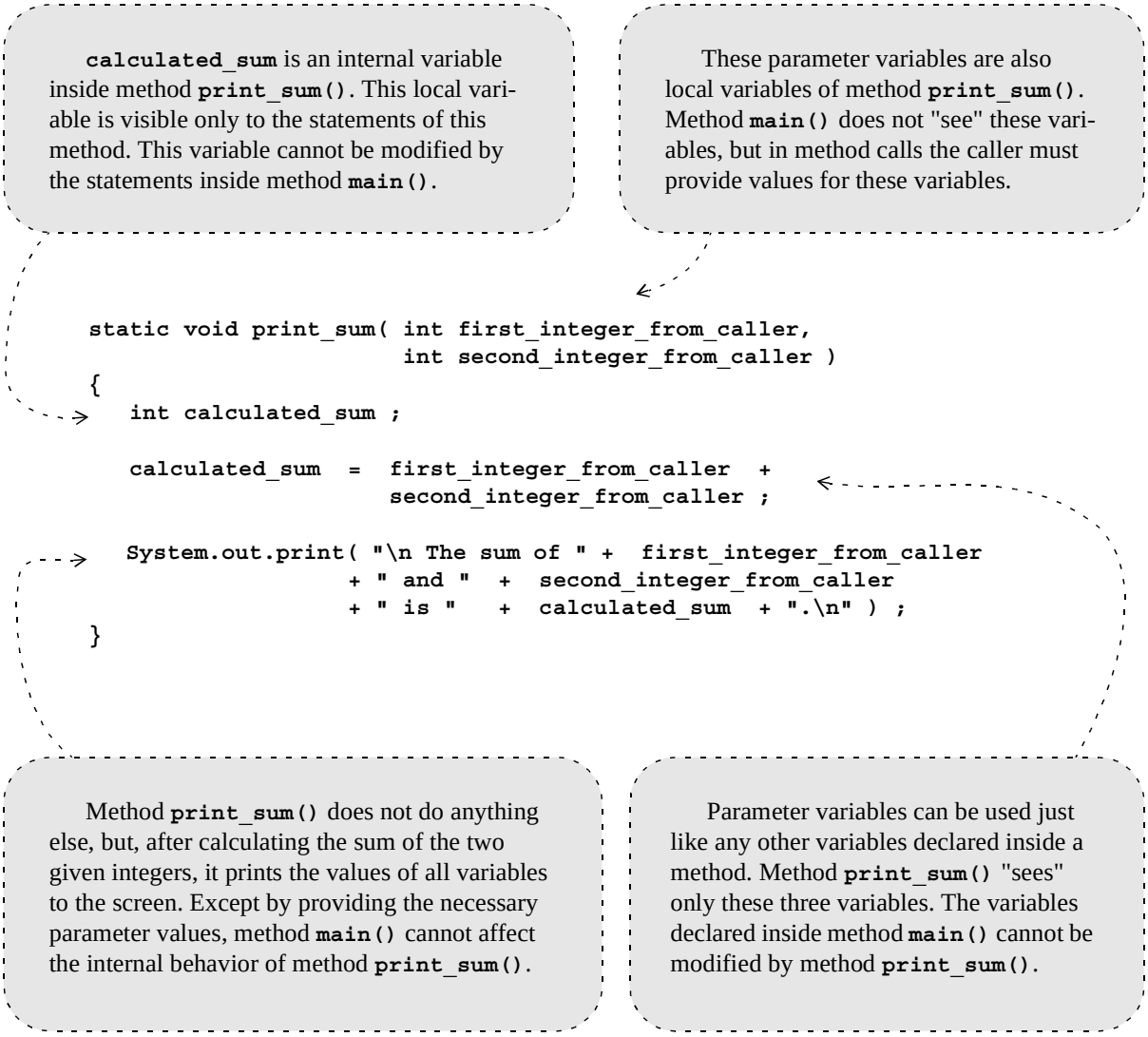
In the method declaration, the parameters that the method takes are declared inside parentheses after the method name. Parameter declarations are separated by commas. Parameters are declared in the same way as variables, except that there are no semicolons (;) to terminate the declarations of parameter data.

```
//  Sums.java

import java.util.* ;

class Sums
{
   static void print_sum( int first_integer_from_caller,
                          int second_integer_from_caller )
   {
      int calculated_sum ;

      calculated_sum  =  first_integer_from_caller  +
                         second_integer_from_caller ;

      System.out.print( "\n The sum of " +  first_integer_from_caller
                      + " and "  +  second_integer_from_caller
                      + " is "   +  calculated_sum  + ".\n" ) ;
   }

   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      print_sum( 555, 222 ) ;

      System.out.print( "\n As you can see, this program can print"
                      +  "\n the sum of two integers. Please, type in"
                      +  "\n two integers separated with a space:\n\n  " ) ;

      int first_integer   =  keyboard.nextInt() ;

      int second_integer  =  keyboard.nextInt() ;

      print_sum( first_integer, second_integer ) ;
   }
}
```

When method **print_sum()** is called here, value 555 is copied to variable **first_integer_from_caller**, and value 222 is copied to **second_integer_from_caller** before the statements inside **print_sum()** are executed. This method call thus prints the sum of 555 and 222.

In this method call, before the internal statements of method **print_sum()** are executed, the contents of variable **first_integer** are copied to variable **first_integer_from_caller**, and the contents of variable **second_integer** are copied to **second_integer_from_caller**. The data stored in variables inside the method **main()** is thus copied to parameter variables declared inside method **print_sum()**.

Sums.java - 1.+ Method main() calls a method that takes two parameters of type int.



Sums.java - 1 - 1. The method print_sum().

```
D:\javafiles2>java Sums

 The sum of 555 and 222 is 777.

 As you can see, this program can print
 the sum of two integers. Please, type in
 two integers separated with a space:

  3344 11122

 The sum of 3344 and 11122 is 14466.
```

Sums.java - X. Two sums of integers calculated and printed with method print_sum().

This method prints the character given as the first parameter as many times as the second parameter specifies.

```
//  Decorations.java  (c) Kari Laitinen

class Decorations
{
   static void multiprint_character( char character_from_caller,
                                     int  number_of_times_to_repeat )
   {
      int  repetition_counter  =  0 ;

      while ( repetition_counter  <  number_of_times_to_repeat )
      {
         System.out.print( character_from_caller ) ;
         repetition_counter  ++ ;
      }
   }

   static void print_text_in_decorated_box( String text_from_caller )
   {
      int text_length  =  text_from_caller.length() ;

      System.out.print( "\n " ) ;
      multiprint_character( '=',  text_length  +  8  ) ;
      System.out.print( "\n " ) ;
      multiprint_character( '*',  text_length  +  8  ) ;
      System.out.print( "\n **" ) ;
      multiprint_character( ' ',  text_length  +  4  ) ;

      System.out.print( "**\n **  "  + text_from_caller + "  **\n **" ) ;

      multiprint_character( ' ',  text_length  +  4  ) ;
      System.out.print( "**\n " ) ;
      multiprint_character( '*',  text_length  +  8  ) ;
      System.out.print( "\n " ) ;
      multiprint_character( '=',  text_length  +  8  ) ;
      System.out.print( "\n " ) ;
   }

   public static void main( String[] not_in_use )
   {
      String  first_text  =  "Hello, world." ;

      print_text_in_decorated_box( first_text ) ;

      print_text_in_decorated_box(
                  "I am a computer program written in Java." ) ;
   }
}
```

The parameter for method **print_text_in_-decorated_box()** can be either a reference to a **String** object containing the text to be printed or a string literal.

Decorations.java - 1.+ Method main() calls a method that takes a string as a parameter.

This method "decorates" the text with a box that consist of characters = and *. Because texts can be of varied length, the width of the decoration box must be adjusted to correspond to the text length. For this reason, the string method **`length()`** is used here to find out how many characters there are in the **`String`** object referenced by **`text_from_caller`**. After this statement has been executed, **`text_length`** contains the character count of the text.

Here, reference to a **`String`** object is passed as a parameter. In this kind of a method call, the object is not copied for the called method, but only a reference to the object is passed to the callee. In practice this means that the physical memory address of the object is passed to the callee. Methods that have objects as parameters use the objects that have been created by the calling method.

```
static void print_text_in_decorated_box( String text_from_caller )
{
   int text_length  =  text_from_caller.length() ;

   System.out.print( "\n " ) ;
   multiprint_character( '=',  text_length  +  8  ) ;
   System.out.print( "\n " ) ;
   multiprint_character( '*',  text_length  +  8  ) ;
   System.out.print( "\n **" ) ;
   multiprint_character( ' ',  text_length  +  4  ) ;
```

The caller of a method can give an arithmetic expression as a parameter. In this call to **`multiprint_character()`**, the space character is printed as many times as is the value of **`text_length`** plus 4. The value of the arithmetic expression is calculated first, and that value is then passed as a parameter.

Decorations.java - 1 - 1. Part of the method that prints text inside a decorative border.

```
D:\javafiles2>java Decorations

 ======================
 **********************
 **                    **
 **  Hello, world.     **
 **                    **
 **********************
 ======================

 ================================================
 ************************************************
 **                                            **
 **  I am a computer program written in Java.  **
 **                                            **
 ************************************************
 ================================================
```

Decorations.java - X. Texts printed inside decorative frames.

A method which takes no parameters behaves always in the same way, regardless of when and where it is called. But the behavior of a method that does take parameters depends on what kinds of parameters are given. A method declared with parameters must be given a correct number of parameters of the correct type. A method that takes parameters is a kind of incomplete program until it is given the necessary parameter data.

The terminology of programming languages makes a distinction between those method parameters that are declared in a method declarator and those parameters that are given in a method call. The term *formal parameters* means the parameters in a method declarator. The term *actual parameters* refers to the parameters given in a method call. For example, the method declarator

```
static void
multiprint_character( char character_from_caller,
                      int  number_of_times_to_repeat )
```

specifies formal parameters **character_from_caller** and **number_of_times_to_repeat**. These parameters are formal because, although they are declared, they have no values until method **multiprint_character()** is called in some other method. The formal parameters of a method specify how the method can be called. You can imagine that the above method declarator says: "Hi! I am method **multiprint_character()**. You can call me by giving first an actual parameter of type **char**, and then an actual parameter of type **int**. Before my internal statements will be executed, the first actual parameter is stored in variable **character_from_caller**, and the second actual parameter is stored in variable **number_of_times_to_repeat**."

The actual parameters are given in method calls. For example, in the method call

```
multiprint_character( '=',  text_length  +  8  ) ;
```

'=' and **text_length** + 8 are actual parameters which are evaluated, and the values are copied to the formal parameters. Actual parameters may be variables, arithmetic or other expressions, literal constants, or even calls to other methods. In the above method call, '=' is a character literal meaning the character code of the equal sign, and **text_length** + 8 is an arithmetic expression. Any expression used as an actual parameter is evaluated to find a value to be copied to the corresponding formal parameter. Evaluation means that the current value of the expression is calculated. In the above method call, the value of **text_length** + 8 is calculated, and that value is copied to the formal parameter **number_of_times_to_repeat**.

Those things which are called parameters in this book are called arguments in some other books, especially in the context of other programming languages. The word "argument" is also used in the context of Java, for example, in the electronic Java documentation. To my knowledge, however, "parameter" is the more official word in the Java world. Therefore, I try to avoid saying argument when meaning parameter. However, if somebody speaks/writes about arguments, formal arguments, or actual arguments, it is very likely that what is meant are parameters, formal parameters, and actual parameters, respectively. To make things even more complex, I must warn you that some people use the word "argument" to refer to actual parameters and the word "parameter" to refer to formal parameters.