Introduction to Computer Programming

Using the Java Programming Language

5. Methods

A method in Java is a block of statements that has a name and can be executed by calling (also called invoking) it from some other place in your program. Along with fields, methods are one of the two elements that are considered members of a class. (Constructors and initializers are not considered class members.)

Every program must have at least one method for the program to accomplish any work. And every program must have a method named main, which is the method first invoked when the program is run.

Creating a Method

All methods — including the main method — must begin with a method declaration. Here's the basic form of a method declaration:

```
visibility [static] return-type method-name (parameter-list)
{
        statements...
}
```

The following list describes the method declaration piece by piece:

visibility: The visibility of a method determines whether the method is available to other classes. The options are

public: Allows any other class to access the method

private: Hides the method from other classes

protected: Lets subclasses use the method but hides the method from other classes

static: This optional keyword declares that the method is a static method, which means that you can call it without first creating an instance of the class in which it's defined. The main method must always be static, and any other methods in the class that contains the main method usually should be static as well.

return-type: After the word static comes the return type, which indicates whether the method returns a value when it is called — and if so, what type the value is. If the method doesn't return a value, specify void.

If you specify a return type other than void, the method must end with a return statement that returns a value of the correct type. For more information, see return Statement.

method-name: Now comes the name of your method. The rules for making up method names are the same as the rules for creating other identifiers: Use any combination of letters and numbers, but start with a letter.

parameter list: You can pass one or more values to a method by listing the values in parentheses following the method name. The parameter list in the method declaration lets Java know what types of parameters a method should expect to receive and provides names so that the statements in the method's body can access the parameters as local variables.

Note: If the method doesn't accept parameters, you must still code the parentheses that surround the parameter list. You just leave the parentheses empty.

statements: One or more Java statements that comprise the method body, enclosed in a set of braces. Unlike Java statements such as if, while, and for, the method body requires you to use the braces even if the body consists of only one statement.

Let's take a look at a simple method that does not take any arguments and does not return any values.

```
public class MethodExample1 {

	public static void main(String[] args) {
		System.out.println("Calling My Sample Method");

		SampleMethod();
	}

	public static void SampleMethod() {
		System.out.println("Displaying a line from my method");
	}

}
```

In this example you can see that the main method made a call to my SampleMethod method. The method only performed one action, to display something to the console. The output from this program can be seen here:

```
Calling My Sample Method
Displaying a line from my method
```

Passing Values to a Method

To understand how to pass values to methods in Java, think about sending someone to the supermarket to buy bread. When you do this, you say, "Go to the supermarket and buy some bread." Some other time, you send that same person to the supermarket to buy bananas. You say, "Go to the supermarket and buy some bananas." And what's the point of all this?

Well, you have a method, and you have some on-the-fly information that you pass to the method when you call it. The method is named "Go to the supermarket and buy some…." The on-the-fly information is either "bread" or "bananas," depending on your culinary needs. In Java, the method calls would look like this:

```
GoToTheSupermarketAndBuySome(bread);
GoToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are called parameters or parameter lists. With parameters, your methods become much more versatile. When you call your goToTheSupermarketAndBuySome method, you decide right there and then what you're going to ask your pal to buy.

Let's take an example of a method that will add two numbers and print the result. Clearly this is a silly example of a method as we can just add two numbers using the plus operand, but it's a good example to illustrate passing values to a method.

```
public class MethodExample2 {

	public static void main(String[] args) {
		System.out.println("I'm going to call my method now.");

		AddTwoNumbers (2, 7);
	}

	public static void AddTwoNumbers (int num1, int num2) {
		int sum = num1 + num2;

		System.out.println("The sum of the numbers is: " + sum);
	}
}
```

As you can see, the method AddTwoNumbers accepts two values passed to it. It is mandatory that you declare the variables in the parameter list of the method.

The output from this method would be:

```
I'm going to call my method now.
The sum of the numbers is: 9
```

Returning Values from a Method

It is very useful for a method to perform some actions and then return a value back to the method that called it (the main method in our examples). To carry our example of the method that added two numbers together one step further, we could create a method to add two numbers and return the result back to the caller. This takes a couple of adjustments to our method.

First, in the method declaration we need to change the `void` to the type of value that will be returned. For our example it will be an integer value or `int`. We also must use the `return` statement to return the value from our method. These modifications can be seen here:

```
public class MethodExample3 {

	public static void main(String[] args) {
		int sum;

		System.out.println("I'm going to call my method now.");

		sum = AddTwoNumbers (2, 7);

		System.out.println("The sum of the two numbers is: " + sum);
	}

	public static int AddTwoNumbers (int num1, int num2) {
		int total = num1 + num2;

		return total;
	}
}
```

The output of this example would be the same as our previous example. The only difference is that the sum value is being displayed in the main method, not the `AddTwoNumbers` method. The sum is being returned from the `AddTwoNumbers` method.

Pass by Value vs. Pass by Reference

When we pass values to a method, it is important to understand that the value of the variable we pass is getting passed to the method, not the actual variable. Let me illustrate with an example. In this example I am passing a number to a method that I create. In the method I change the value of the variable. Let's see what happens when I return back to the main method.

```
public class MethodExample4 {

	public static void main(String[] args) {
		int num = 5;

		System.out.println("I'm going to call my method now.");

		//num is originally 5
		ChangeNumber (num);

		//What is num now?
		System.out.println("The value of num is: " + num);
	}

	public static void ChangeNumber (int number) {

		number = 10;
	}
}
```

In this example we passed `num` to the method `ChangeNumber`. The important concept is to understand that the variable `num` did not get passed to the method. Only the value that `num` held at the time it was passed got sent to `ChangeNumber`. So, no matter what `ChangeNumber` does to that variable, the original value will still be held in the calling method.

So, the output of this program would be:

```
I'm going to call my method now.
The value of num is: 5
```

The only exception to this is when we pass an array to a method. This will be discussed once arrays are introduced in the next chapter. For now it is important to remember that we cannot change the values of variables that are passed into a method. If we do change their value, the new value only applies while we are in the method.