CHAPTER 3

First Step Towards Java Programming

Whenever we want to write a program, we should first think about writing comments. What are comments? Comments are description about the features of a program. This means that whatever we write in a program should be described using comments. Why should we write comments? When we write comments, we can understand what the program is doing as well as it helps others to easily follow our code. This means *readability* and *understandability* of a program will be more. If a program is understandable, then only can it be usable in a software. If other members of the software development team cannot understand our program, then they may not be able to use it in the project and will reject it. So writing comments is compulsory in any program. Remember, it is a good programming habit.

There are three types of comments in Java—single line, multi line, and Java documentation. Let us discuss all of them here:

- **Single line comments:** These comments are for marking a single line as a comment. These comments start with double slash symbol // and after this, whatever is written till the end of the line is taken as a comment. For example,

```
//This is my comment of one line.
```

- **Multi line comments:** These comments are used for representing several lines as comments. These comments start with /* and end with */. In between /* and */, whatever is written is treated as a comment. For example,

```
/*  This is a multi line comment. This is line one.
This is line two of the comment.
This is line three of the comment. */
```

- **Java documentation comments:** These comments start with /** and end with */. These comments are used to provide description for every feature in a Java program. This description proves helpful in the creation of a `.html` file called *API (*Application Programming Interface*)* document. Java documentation comments should be used before every feature in the program as shown here:

```
/** description about a class */
Class code

/** description about a method */
Method code
```

API Document

The API document generated from the `.java` program is similar to a help file where all the features are available with their descriptions. The user can refer to any feature in this file and get some knowledge regarding how he can use it in his program. To create an API document, we should use a special compiler called `javadoc` compiler. Figure 3.1 shows the process of creation of an API document.

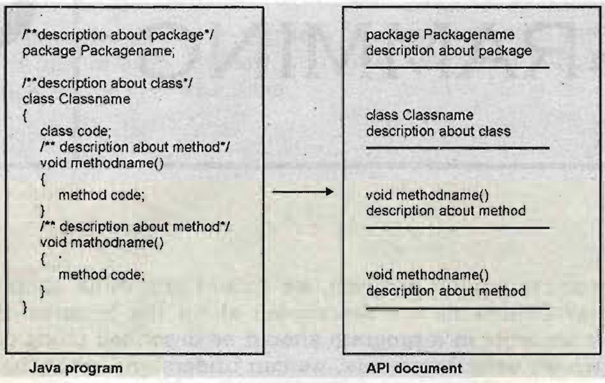

Figure 3.1 API document creation

Important Interview Question

What is an API document?

An API document is a `.html` file that contains description of all the features of a software, a product, or a technology. API document is helpful for the user to understand how to use the software or technology.

Starting a Java program

So let us start our first Java program with multi line comments:

```
/* This is my first Java program.
       To display a message.
       Author: DreamTech team
       Version: v1.0
       Project title: Dream project
       Project code: 123
*/
```

Importing classes

Suppose we are writing a C/C++ program, the first line of the program would generally be:

```
#include<stdio.h>
```

This means, a request is made to the C/C++ compiler to include the header file `<stdio.h>`. What is a header file? A *header file* is a file, which contains the code of functions that will be needed in a program. In other words, to be able to use any function in a program, we must first include the header file containing that function's code in our program. For example, `<stdio.h>` is the header file that contains functions, like `printf()`, `scanf()`, `puts()`, `gets()`, etc. So if we want to use any of these functions, we should include this header file in our C/C++ program.

What happens when we include the header file? After we include the header file, the C/C++ compiler goes to the standard library (generally it is available in `tc/lib`) and searches for the header file there. When it finds the header file, it copies the entire header file content into the program where the `#include` statement is written. Thus, if we write a program of 10 lines; then after copying code from the header file, the program size may become, say, 510 lines. From where do these additional 500 lines are coming? They have been copied physically into our program from the header file. Thus, our program size increases unnecessarily, wasting memory and processor time.

A similar but a more efficient mechanism, available in case of Java, is that of importing classes. First, we should decide which classes are needed in our program. Generally, programmers are interested in two things:

- Using the classes by creating objects in them.
- Using the methods (functions) of the classes.

In Java, methods are available in classes or interfaces. What is an interface? An *interface* is similar to a class that contains some methods. Of course, there is a lot of difference between an interface and a class, which we will discuss later. The main point to be kept in mind, at this stage, is that a class or an interface contains methods. A group of classes and interfaces are contained in a package. A *package* is a kind of directory that contains a group of related classes and interfaces and Java has several such packages in its library.

$$\begin{tikzcd}[row sep=small] \text{Java library} \arrow[d, dash] \\ \text{Packages} \arrow[d, dash] \\ \text{Classes and interfaces} \arrow[d, dash] \\ \text{Methods} \end{tikzcd}$$

Note

If a programmer wants to use a class, then that class should be imported into his program. If he wants to use a method, then that corresponding class or interface should be imported into his program.

In our first Java program, we are using two classes namely, `System` and `String`. These classes belong to a package called `java.lang` (here `lang` represents language). So these two classes must be imported into our program, as shown below:

```
        import java.lang.System;
        import java.lang.String;
```

Whenever we want to import several classes of the same package, we need not write several import statements, as shown above; instead, we can write a single statement as:

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

Here, * means all the classes and interfaces of that package, i.e. `java.lang`, are imported (made available) into our program. In import statement, the package name that we have written acts like a reference to the JVM to search for the classes there. JVM will not copy any code from the classes or packages. On the other hand, when a class name or a method name is used, JVM goes to the Java library, executes the code there, comes back, and substitutes the result in that place of the program. Thus, the Java program size will not be increased.

Important Interview Question

What is the difference between `#include` and `import` statement?

`#include` directive makes the compiler go to the C/C++ standard library and copy the code from the header files into the program. As a result, the program size increases, thus wasting memory and processor's time.

`import` statement makes the JVM go to the Java standard library, execute the code there, and substitute the result into the program. Here, no code is copied and hence no waste of memory or processor's time. So, `import` is an efficient mechanism than `#include`.

After importing the classes into the program, the next step is to write a class. Since Java is purely an object-oriented programming language, we cannot write a Java program without having at least one class or object. So, it is mandatory that every Java program should have at least one class in it. How to write a class? We should use `class` keyword for this purpose and then write the class name.

```
class First {
      statements;
}
```

A class code starts with a { and ends with a }. We know that a class or an object contains variables and methods (functions). So we can create any number of variables and methods inside the class. This is our first program, so we will create only one method, i.e. compulsory, in it—`main()` method.

Why should we write `main()` method? Because, if `main()` method is not written in a Java program, JVM will not execute it. `main()` is the starting point for JVM to start execution of a Java program.

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

Next to `main()` method's name, we have written `String args[]`. Before discussing the `main()` method, let us first see how a method works. A method, generally, performs two functions.

- ❑ It can accept some data from outside
- ❑ It can also return some result

Let us take `sqrt()` method. This method is used to calculate square root value of a given number. So, at the time of calling `sqrt()` method, we should pass a number (e.g. 16) for which we want to calculate square root. Then, it calculates square root value and returns the result (e.g. 4) to us. Similarly, `main()` method also accepts some data from us. For example, it accepts a group of strings, which is also called a *string type array*. This array is `String args[]`, which is written along with the `main()` method as:

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

Here `args[]` is the array name and it is of `String` type. This means that it can store a group of strings. Remember, this array can also store a group of numbers but in the form of strings only. The values passed to `main()` method are called *arguments*. These arguments are stored into `args[]` array, so the name `args[]` is generally used for it.

A method can return some result. If we want the method to return the result in form of an integer, then we should write `int` before the method name. Similarly, to get the result in string form or as a single character, we can write `string` or `char` before it, respectively. If a method is not meant to return any value, then we should write `void` before that method's name. `void` means *no value*. `main()` method does not return any value, so `void` should be written before that method's name.

A method is executed only when it is called. But, how to call a method? Methods are called using 2 steps in Java.

1. Create an object to the class to which the method belongs. The syntax of creating the object is:

```
Classname objectname  = new Classname();
```

2. Then the method should be called using the `objectname.methodname()`.

Since `main()` method exists in the class `First`; to call `main()` method, we should first of all create an object to `First` class, something like this:

```
First obj = new First();
```

Then call the `main()` method as:

```
obj.main();
```

So, if we write the statement:

```
First obj = new First()
```

inside the `main()` method, then JVM will execute this statement and create the object.

```
class First
{
        public static void main(String args[])
        {
                First obj = new First();  //object is created hereafter
JVM executes this.
        }
}
```

By looking at the code, we can understand that an object could be created only after calling the `main()` method. But for calling the `main()` method, first of all we require an object. Now, how is it possible to create an object before calling the `main()` method? So, we should call the `main()` method without creating an object. Such methods are called static methods and should be declared as `static`.

Static methods are the methods, which can be called and executed without creating the objects. Since we want to call `main()` method without using an object, we should declare `main()` method as `static`. Then, how is the `main()` method called and executed? The answer is by using the `classname.methodname()`. JVM calls `main()` method using its class name as `First.main()` at the time of running the program.

JVM is a program written by *JavaSoft* people (Java development team) and `main()` is the method written by us. Since, `main()` method should be available to the JVM, it should be declared as `public`. If we don't declare `main()` method as `public`, then it doesn't make itself available to JVM and JVM cannot execute it.

So, the `main()` method should always be written as shown here:

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

If at all we want to make any changes, we can interchange `public` and `static` and write it as follows:

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

Or, we can use a different name for the string type array and write it as:

```
static public void main(String x[])
```

Important Interview Question

What happens if `String args[]` is not written in `main()` method?

When `main()` method is written without `String args[]` as:

```
public static void main()
```

the code will compile but JVM cannot run the code because it cannot recognize the `main()` method as the method from where it should start execution of the Java program. Remember JVM always looks for `main()` method with string type array as parameter.

Till now, according to our discussion, our Java program looks like this:

```
/* This is my first Java program.
To display a message.
Author: DreamTech team
Version: v1.0
Project title: Dream project
Project code: 123
*/
class First
{
        public static void main(String args[])
        {
                statements;   //these are executed by JVM.
        }
}
```

Our aim of writing this program is just to display a string "Welcome to Java". In Java, `print()` method is used to display something on the monitor. So, we can write it as:

```
print("Welcome to Java");
```

But this is not the correct way of calling a method in Java. A method should be called by using `objectname.methodname()`. So, to call `print()` method, we should create an object to the class to which `print()` method belongs. `print()` method belongs to `PrintStream` class. So, we should call `print()` method by creating an object to `PrintStream` class as:

```
PrintStream obj.print("Welcome to Java");
```

But as it is not possible to create the object to `PrintStream` class directly, an alternative is given to us, i.e. `System.out`. Here, *System* is the class name and *out* is a static variable in *System* class. *out* is called a *field* in *System* class. When we call this field, a `PrintStream` class object will be created internally. So, we can call the `print()` method as shown below:

```
System.out.print("Welcome to Java");
```

`System.out` gives the `PrintStream` class object. This object, by default, represents the standard output device, i.e. the monitor. So, the string "Welcome to Java" will be sent to the monitor.

Now, let us see the final version of our Java program:

Program 1: To display a message.

```
/* This is my first Java program.
To display a message.
Author: DreamTech team
Version: v1.0
Project title: Dream project
Project code: 123
*/
class First
{
        public static void main(String args[])
        {
                System.out.print("Welcome to Java");
        }
}
```

Output:

```
C:\>javac First.java
C:\>java First
Welcome to Java
```

Save the above program in a text editor like *Notepad* with the name `First.java`. Now, go to *System* prompt and compile it using *javac* compiler as:

```
javac First.java
```

The compiler generates a file called `First.class` that contains byte code instructions. This file is executed by calling the JVM as:

```
java First
```

Then, we can see the result.

We already discussed that JVM is a program. Which program is it? By observing the command `java First`, we can say that JVM is nothing but `java.exe` program. In fact, JVM is written in C language.

Let us write another Java program to find the sum of two numbers. We start this program with a single line comment like this:

```
//To find sum of two numbers
```

Then we should import whatever classes and interfaces we want to use in our program. In this program, we will use two classes: `System` and `String`. These classes are available in the package `java.lang`. So, we should import these classes as:

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

Here, * represents all the classes of `java.lang` package. This import statement acts like a reference for the JVM to search for classes when a particular class is used in the program. After writing the import statement, we should write a class, since it is compulsory to create at least one class in every Java program. Let us write the class as shown below:

```
class Sum
{
        statements;
        }
```

We know that a class contains variables and methods. Even if we do not create variables or methods, we should write at least one method, i.e. `main()`. The purpose of writing the `main()` method is to make JVM execute the statements within it. Let us take an example:

```
        class Sum
        {
                public static void main(String args[])
                {
                        statements;
                }
        }
```

Here, `main()` method is declared as `public`, `static`, and `void`. `public` because it should be made available to JVM, which is another program; `static` because it should be called without using any object; and `void` because it does not return any value. Also, after the method name, we write `String args[]` which is an array to store the values passed to `main()` method. These values passed to `main()` are called *arguments*. These values are stored in `args[]` array in the form of strings.

Inside the `main()` method, let us write the code to find the sum of two numbers. Observe the full program now:

Program 2: To find the sum of two numbers.

```
//To find sum of two numbers
import java.lang.*;

class Sum
{
public static void main(String args[])
{
//variables
                int x,y;

                //store values into variables
                x = 10;
                y = 25;

                //calculate sum and store result into z
                int z = x+y;

                //display result
                System.out.print(z);
}
}
```

Output:

```
C:\> javac Sum.java
C:\> java Sum
35
```

In the above program, inside `main()` method, we write the first statement as:

```
int x,y;
```

Here x,y are called variables. A variable represents memory location to store data or values. We want to store integer numbers into x,y. So before these variables, we write int. This int represents the type of data to be stored into the variables. It is also called a *data type*. By writing int x,y, we are announcing the java compiler that we are going to store integer type data into x,y variables. Remember, declaring the data type is always required before using any variable. All data types available in Java are discussed in Chapter 4.

After declaring the x,y variables as int data type, we have stored integer numbers into x,y as:

```
    x = 10;
    y = 25;
```

= represents storing the right hand side value into the left hand side variable. Thus, 10 is stored into x and 25 is stored into y. So = is a symbol that represents assignment (or storage) operation. Such symbols are also called *operators*. All the operators available in Java will be discussed in Chapter 5. The next step is to find the sum of these values. For this purpose, we write:

```
    int z = x+y;
```

At the right hand side we got x+y, which performs the sum. Here + is called *addition operator*. The result of this sum is stored into z, which we declared as another int type variable. Now, we should display the result using print() method as:

```
    System.out.print(z);
```

Here, System is a class in java.lang package and out is a field in this class. When we refer to System.out, it creates PrintStream object, which, by default, represents the standard output device, i.e. monitor. So, by writing System.out.print(z), we are passing z value to print() method, which displays that value on the monitor. So in the output, we can see the result as 35.

Formatting the Output

In Program 2, we have used the following statement to display the result:

```
    System.out.print(z);
```

This will display the result 35 on the monitor. But, this is not the proper way to display the results to the user. When the user sees 35, he would be baffled as what is this number 35?. So, it is the duty of the programmer to prompt the user with a proper message, something like this:

```
    Sum of two numbers= 35
```

This is more clear and avoids any confusion for the user. How can we display the output in the above format? For this purpose, we should add a string "Sum of two numbers=" before the z value in the print() method:

```
    System.out.print("Sum of two numbers= "+ z);
```

Here, we are using + to join the string "Sum of two numbers=" and the numeric variable z. The reason is that the print() method cannot display more than one value. So if two values, like a string and a numeric value need to be displayed using a single print() method, we should combine them using a + operator.

Now, the output will look like this:

```
        Sum of two numbers= 35
```

We know that value of `x` is `10` and `y` is `25`. Suppose, we use `print()` method to display directly `x+y` value in the place of `z`, we can write:

```
        System.out.print("Sum of two numbers="+ x+y);
```

which displays:

```
        Sum of two numbers= 1025
```

See the output is wrong. What is the reason? In the `print()` method's braces, from left to right, we got a string "Sum of two numbers=" and two numerit variables `x` and `y` as shown below:

```
        "Sum of two numbers= "+ x+y
```

Since left one is a string, the next value, i.e. of x is also converted into a string; thus the value of `x`, i.e. 10 will be treated as separate characters as `1` and `0` and are joined to the string. Thus, we get the output in the string form:

```
        Sum of two numbers= 10
```

The next variable value is 25. Since till now, at left we got a string, so this y value is also converted into a string. Thus 2 and 5 are taken as separate characters and joined to the string, thus we get the result as:

```
        Sum of two number= 1025
```

So, how to get the correct result? By using another pair of simple braces inside the `print()` method, we can get the correct result.

```
        System.out.print("Sum of two numbers= "+ (x+y));
```

Here, the execution will start from the inner braces. At left, we got `x`, which represents a number and at right we got `y`, which is also a number. Since both are numbers, addition will be done by the + operator, thus giving a result `35`. Then the result will be displayed like this:

```
        Sum of two numbers= 35
```

To eliminate the above inner braces, we can use two `print()` methods like this:

```
        System.out.print("Sum of two numbers= ");
        System.out.print(x+y);
```

This will also give the correct result as:

```
        Sum of two numbers= 35
```

Please observe that we have used two `print()` methods and still we got the result in only one line. What does it mean? First `print()` method is displaying the string: "`Sum of two numbers`" and keeping the cursor in the same line. Next `print()` method is also showing its output in the same line adjacent to the previous string. To conclude, `print()` method displays the result and then keeps the cursor in the same line.

Suppose, we want the result in two separate lines, then we can use `println()` method in the place of `print()` method as:

```
        System.out.println("Sum of two numbers= ");
        System.out.println(x+y);
```

This will display the result in two lines as:

```
        Sum of two numbers= 35
```

`println()` is also a method belonging to `PrintStream` class. It throws the cursor to the next line after displaying the result.

Important Interview Question

What is the difference between `print()` and `println()` method?

Both methods are used to display the results on the monitor. `print()` method displays the result and then retains the cursor in the same line, next to the end of the result. `println()` displays the result and then throws the cursor to the next line.

Let us take the following statement:

```
        System.out.print ("Sum of two numbers= "+ (x+y));
```

This will display the output as shown below:

```
        Sum of two numbers= 35
```

Now, suppose we want to display the result in two lines using only one `println()` method, what is the way? For this purpose, we can use a code `\n` inside the string, as shown here:

```
        System.out.println("Sum of two numbers=\n"+(x+y));
```

This will display the result like this:

```
        Sum of two numbers= 35
```

This means `\n` is throwing the cursor into the next line at that place. This is called *backslash code* or *escape sequence*. Table 3.1 lists the noteworthy backslash codes with their meanings.

Table 3.1

Backslash code	Meaning
`\n`	Next line
`\t`	Horizontal tab space
`\r`	Enter key
`\b`	Backspace

Backslash code	Meaning
\f	Form feed
\\	Displays \
\"	Displays "
\'	Displays '

For example, observe the following statements:

```
        System.out.println("Hello");
        System.out.println("\\Hello\\");    //Observe \\
        System.out.println("\"Hello\"");    //Observe \"
```

The above statements display the output as:

```
        Hello
        \Hello\
        "Hello"
```

Program 3: To understand the effect of `print()` and `println()` methods and the backslash codes.

```
        //Formatting the output
        class Format
        {
                public static void main(String args[])
                {
                        int a=1,b=2,c=3,d=4;
                        System.out.print(a+"\t"+b);
                        System.out.println(b+"\n"+b);
                        System.out.print(":"+c);
                        System.out.println();  //this throws cursor to the next line
                        System.out.println("Hello\\Hi\""+d);
                }
        }
```

Output:

```
        C:\> javac Format.java
        C:\> java Format

        1       22
        2
        :3
        Hello\Hi"4
```

By now, you must be finding Java to be very friendly.

Conclusion

In this chapter, we have learned about providing comments in the java program. We have also come across with API documents. While moving towards the end of the chapter we have come across with the creating a program in Java.