CHAPTER 2

INTRODUCTION TO JAVA

Before starting to learn Java, let us plunge into its history and see how the language originated. In 1990, Sun Microsystems Inc. (US) has conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called *Stealth Project* but later its name was changed to *Green Project*.

In January of 1991, Bill Joy, James Gosling, Mike Sheradin, Patrick Naughton, and several others met in Aspen, Colorado to discuss this project. Mike Sheradin was to focus on business development; Patrick Naughton was to begin work on the graphics system; and James Gosling was to identify the proper programming language for the project. Gosling thought C and C++ could be used to develop the project. But the problem he faced with them is that they were system dependent languages and hence could not be used on various processors, which the electronic devices might use. So he started developing a new language, which was completely system independent. This language was initially called *Oak*. Since this name was registered by some other company, later it was changed to *Java*.

Why the name Java? James Gosling and his team members were consuming a lot of tea while developing this language. They felt that they were able to develop a better language because of the good quality tea they had consumed. So the tea also had its own role in developing this language and hence, they fixed the name for the language as *Java*. Thus, the symbol for *Java* is tea cup and saucer.

By September of 1994, Naughton and Jonathan Payne started writing *WebRunner*—a Java-based Web browser, which was later renamed as *HotJava*. By October 1994, HotJava was stable and was demonstrated to Sun executives. HotJava was the first browser, having the capabilities of executing *applets*, which are programs designed to run dynamically on Internet. This time, Java's potential in the context of the World Wide Web was recognized.

Sun formally announced Java and HotJava at SunWorld conference in 1995. Soon after, Netscape Inc. announced that it would incorporate Java support in its browser Netscape Navigator. Later, Microsoft also announced that they would support Java in their Internet Explorer Web browser, further solidifying Java's role in the World Wide Web. On January 23rd 1996, JDK 1.0 version was released. Today more than 4 million developers use Java and more than 1.75 billion devices run Java. Thus, Java pervaded the world.

Features of Java

Apart from being a system independent language, there are other reasons too for the immense popularity of this language. Let us have a look at some of its features.

- **Simple:** Java is a simple programming language. Rather than saying that this is the feature of Java, we can say that this is the design aim of Java. When Java is developed, they wanted it to be simple because it has to work on electronic devices, where less memory is available. Now, the question is how Java is made simple? First of all, the difficult concepts of C and C++ have been omitted in Java. For example, the concept of *pointers*—which is very difficult for both learners and programmers—has been completely eliminated from Java. Next, JavaSoft (the team who developed Java is called with this name) people maintained the same syntax of C and C++ in Java, so that a programmer who knows C or C++ will find Java already familiar.

Important Interview Question

Why pointers are eliminated from Java?

1. *Pointers lead to confusion for a programmer.*
2. *Pointers may crash a program easily, for example, when we add two pointers, the program crashes immediately. The same thing could also happen when we forgot to free the memory allotted to a variable and reallot it to some other variable.*
3. *Pointers break security. Using pointers, harmful programs like Virus and other hacking programs can be developed.*

Because of the above reasons, pointers have been eliminated from Java.

- **Object-oriented:** Java is an object-oriented programming language. This means Java programs use objects and classes. What is an object? An object is anything that really exists in the world and can be distinguished from others. Everything that we see physically will come into this definition, for example, every human being, a book, a tree, and so on.

Now, every object has properties and exhibits certain behavior. Let us try to illustrate this point by taking an example of a dog. It got properties like name, height, color, age, etc. These properties are represented by variables. Now, the object dog will have some actions like running, barking, eating, etc. These actions are represented by various methods (functions) in our programming. In programming, various tasks are done by methods only. So, we can conclude that objects contain variables and methods.

A group of objects exhibiting same behavior (properties + actions) will come under the same group called a *class*. A class represents a group name given to several objects. For example, take the dogs: Pinky, Nancy, Tom, and Subbu. All these four dogs exhibit same behavior and hence belong to the same group, called dog. So *dog* is the class name, which contains four objects. In other words, we could define a class as a model or a blueprint for creating the objects. We write the characteristics of the objects in the class: 'dog'. This means, a class can be used as a model in creation of objects. So, just like objects, a class also contains properties and actions, i.e. variables and methods (Figure 2.1).

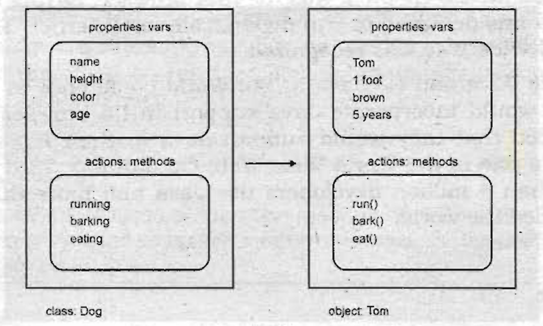


Figure 2.1 Class and object

We can use a class as a model for creating objects. To write a class, we can write all the characteristics of objects which should follow the class. These characteristics will guide us to create the objects. A class and its objects are almost the same with the difference that a class does not exist physically, while an object does. For example, if we say *dog*; it forms a picture into our mind with 4 legs, 2 ears, some length and height. This picture in our mind is *class*. If we tally this picture with the physical things around us, we can find Tom living in our house is satisfying these qualities. So Tom, which physically exists, is an object and not a class.

Let us take another example. Flower is a class but if we take Rose, Lily, Jasmine – they are all objects of *flower* class. The class *flower* does not exist physically but its objects, like Rose, Lily, Jasmine exist physically.

Important Interview Question

What is the difference between a function and a method?

A method is a function that is written in a class. We do not have functions in Java; instead we have methods. This means whenever a function is written in Java, it should be written inside the class only. But if we take C++, we can write the functions inside as well as outside the class. So in C++, they are called member functions and *not methods.*

Since Java is a purely object-oriented programming language, in order to write a program in Java, we need atleast a class or an object. This is the reason why in every Java program we write atleast one class. C++ is not a purely object-oriented language, since it is possible to write programs in C++ without using a class or an object.

- ❑ **Distributed:** Information is distributed on various computers on a network. Using Java, we can write programs, which capture information and distribute it to the clients. This is possible because Java can handle the protocols like TCP/IP and UDP.
- ❑ **Robust:** Robust means *strong*. Java programs are strong and they don't crash easily like a C or C++ program. There are two reasons for this. Firstly, Java has got excellent inbuilt exception handling features. An *exception* is an error that occurs at run time. If an exception occurs, the program terminates abruptly giving rise to problems like loss of data. Overcoming such problems is called *exception handling*. This means that even though an exception occurs in a Java program, no harm will happen.

Another reason, why Java is robust lies in its memory management features. Most of the C and C++ programs crash in the middle because of not allocating sufficient memory or forgetting the memory to be freed in a program. Such problems will not occur in Java because the user need not allocate or deallocate the memory in Java. Everything will be taken care of by JVM only. For example, JVM will allocate the necessary memory needed by a Java program.

Important Interview Question

Which part of JVM will allocate the memory for a Java program?

Class loader subsystem of JVM will allocate the necessary memory needed by the Java program.

Similarly, JVM is also capable of deallocating the memory when it is not used. Suppose a variable or an object is created in memory and is not used. Then after some time, it is automatically removed by garbage collector of JVM. *Garbage collector* is a form of memory management that checks the memory from time to time and marks the variables or objects not used by the program, automatically. After repeatedly identifying the same variable or object, garbage collector confirms that the variable or object is not used and hence can be deleted.

Important Interview Question

Which algorithm is used by garbage collector to remove the unused variables or objects from memory?

Garbage collector uses many algorithms but the most commonly used algorithm is mark and sweep.

Important Interview Question

How can you call the garbage collector?

Garbage collector is automatically invoked when the program is being run. It can be also called by calling gc() method of Runtime class or System class in Java.

- **Secure:** Security problems like eavesdropping, tampering, impersonation, and virus threats can be eliminated or minimized by using Java on Internet.
- **System independence:** Java's byte code is not machine dependent. It can be run on any machine with any processor and any operating system.
- **Portability:** If a program yields the same result on every machine, then that program is called *portable*. Java programs are portable. This is the result of Java's *System independence* nature.
- **Interpreted:** Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter in JVM. If we take any other language, only an interpreter or a compiler is used to execute the programs. But in Java, we use both compiler and interpreter for the execution.
- **High Performance:** The problem with interpreter inside the JVM is that it is slow. Because of this, Java programs used to run slow. To overcome this problem, along with the interpreter, JavaSoft people have introduced JIT (Just In Time) compiler, which enhances the speed of execution. So now in JVM, both interpreter and JIT compiler work together to run the program.
- **Multithreaded:** A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different blocks of code. Creating multiple threads is called 'multithreaded'.
- **Scalability:** Java platform can be implemented on a wide range of computers with varying levels of resources—from embedded devices to mainframe computers. This is possible because Java is compact and platform independent.
- **Dynamic:** Before the development of Java, only static text used to be displayed in the browser. But when James Gosling demonstrated an animated atomic molecule where the rays are moving and stretching, the viewers were dumbstruck. This animation was done using an *applet* program, which are the dynamically interacting programs on Internet.

The Java Virtual Machine

Java Virtual Machine (JVM) is the heart of entire Java program execution process. It is responsible for taking the `.class` file and converting each byte code instruction into the machine language instruction that can be executed by the microprocessor. Figure 2.2 shows the architecture of Java Virtual Machine.

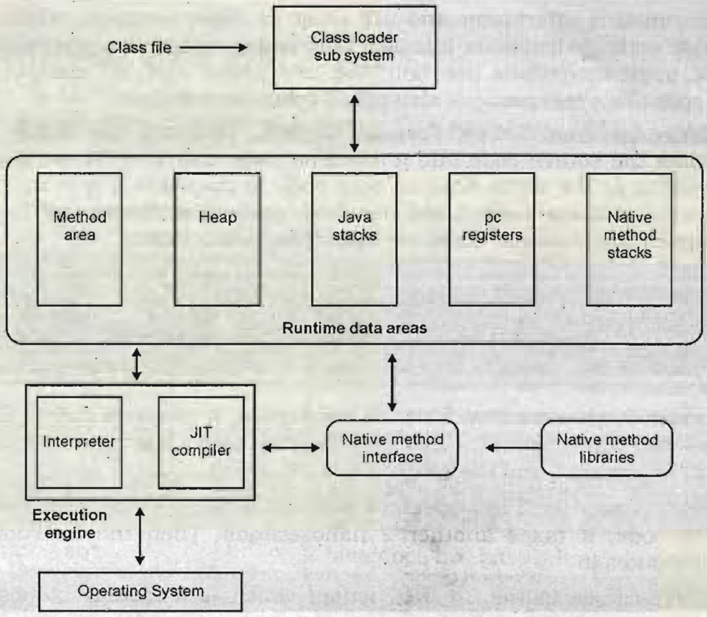


Figure 2.2 Components in JVM architecture

First of all, the `.java` program is converted into a `.class` file consisting of byte code instructions by the java compiler. Remember, this java compiler is outside the JVM. Now this `.class` file is given to the JVM. In JVM, there is a module (or program) called *class loader sub system,* which performs the following functions:

- First of all, it loads the `.class` file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program.

This memory is divided into 5 parts, called *run time data areas,* which contain the data and results while running the program. These areas are as follows:

- **Method area:** Method area is the memory block, which stores the class code, code of the variables, and code of the methods in the Java program. (Method means functions written in a class)
- **Heap:** This is the area where objects are created. Whenever JVM loads a class, a method and a heap area are immediately created in it.
- **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java stacks. So, *Java stacks* are memory areas where Java methods are executed. While executing methods, a separate frame will be created in the Java stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.
- **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.
- **Native method stacks:** Java methods are executed on Java stacks. Similarly, native methods (for example C/C++ functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called *Native method interface.*

Execution engine contains interpreter and JIT (Just In Time) compiler, which are responsible for converting the byte code instructions into machine code so that the processor will execute them. Most of the JVM implementations use both the interpreter and JIT compiler simultaneously to convert the byte code. This technique is also called *adaptive optimizer.*

Generally, any language (like C/C++, Fortran, COBOL, etc.) will use either an interpreter or a compiler to translate the source code into a machine code. But in JVM, we got interpreter and JIT compiler both working at the same time on byte code to translate it into machine code. Now, the main question is why both are needed and how both work simultaneously? To understand this, let us take some sample code. Assume these are byte code instructions:

```
print a;
print b;
Repeat the following 10 times by changing i values from 1 to 10:
print a;
```

When, the interpreter starts execution from 1st instruction, it converts `print a;` into machine code and gives it to the microprocessor. For this, say the interpreter has taken 2 nanoseconds time. The processor takes it, executes it, and the value of a is displayed.

Now, the interpreter comes back into memory and reads the 2nd instruction `print b;` To convert this into machine code, it takes another 2 nanoseconds. Then the instruction is given to the processor and it executes it.

Next, the interpreter comes to the 3rd instruction, which is a looping statement `print a;` This should be done 10 times and this is known to the interpreter. Hence, the first time it converts `print a` into machine code. It takes 2 nanoseconds for this. After giving this instruction to the processor, it comes back to memory and reads the `print a` instruction the 2nd time and converts it to machine code. This will take another 2 nanoseconds. This will be given to the processor and the interpreter comes back and reads `print a` again and converts it 3rd time taking another 2 nanoseconds. Like this, the interpreter will convert the `print a` instruction for 10 times, consuming a total of 10 x 2 = 20 nanoseconds. This procedure is not efficient in terms of time. That is the reason why JVM does not allocate this code to the interpreter. It allots this code to the JIT compiler.

Let us see how the JIT compiler will execute the looping instruction. First of all, the JIT compiler reads the `print a` instruction and converts that into machine code. For this, say, it is taking 2 nanoseconds. Then the JIT compiler allots a block of memory and pushes this machine code instruction into that memory. For this, say, it is taking another 2 nanoseconds. This means JIT compiler has taken a total of 4 nanoseconds. Now, the processor will fetch this instruction from memory and executes it 10 times. Just observe that the JIT compiler has taken only 4 nanoseconds for execution, whereas to execute the same loop the interpreter needs 20 nanoseconds. Thus, JIT compiler increases the speed of execution. Recognize that the first two instructions will not be allotted to JIT compiler by the JVM. The reason is clear. Here it takes 4 nanoseconds to convert each instruction, whereas the interpreter actually took only 2 nanoseconds.

After loading the `.class` code into memory, JVM first of all identifies which code is to be left to interpreter and which one to JIT compiler so that the performance is better. The blocks of code allocated for JIT compiler are also called *hotspots.* Thus, both the interpreter and JIT compiler will work simultaneously to translate the byte code into machine code.

Important Interview Question

What is JIT compiler?

JIT compiler is the part of JVM which increases the speed of execution of a Java program.

Differences between C++ and Java:

By the way, C++ is also an object-oriented programming language, just like Java. But there are some important feature-wise differences, between C++ and Java. Let us have a glance at them in Table 2.1.

Table 2.1

C++	**Java**
C++ is not a purely object-oriented programming language, since it is possible to write C++ programs without using a class or an object.	Java is purely an object-oriented programming language, since it is not possible to write a Java program using atleast one class.
Pointers are available in C++.	We cannot create and use pointers in Java.
Allotting memory and deallocating memory is the responsibility of the programmer	Allocation and deallocation of memory will be taken care of by JVM.
C++ has goto statement.	Java does not have goto statement.
Automatic casting is available in C++.	In some cases, implicit casting is available. But it is advisable that the programmer should use casting wherever required.
Multiple Inheritance feature is available in C++.	No Multiple Inheritance in Java, but there are means to achieve it.
Operator overloading is available in C++.	It is not available in Java.
#define, typedef and header files are available in C++.	#define, typedef and header are not available in Java, but there are means to achieve them.
There are 3 access specifiers in C++: private, public, and protected.	Java supports 4 access specifiers: private, public, protected, and default.
There are constructors and destructors in C++.	Only constructors are there in Java. No destructors are available in this language.

Parts of Java

Sun Microsystems Inc. has divided Java into 3 parts—Java SE, Java EE, and Java ME. Let us discuss them in brief here:

- **Java SE**: It is the Java Standard Edition that contains basic core Java classes. This edition is used to develop standard applets and applications.
- **Java EE**: It is the Java Enterprise Edition and it contains classes that are beyond Java SE. In fact, we need Java SE in order to use many of the classes in Java EE. Java EE mainly concentrates on providing business solutions on a network.
- **Java ME**: It stands for Java Micro Edition. Java ME is for developers who develop code for portable devices, such as a PDA or a cellular phone. Code on these devices needs to be small in size and should take less memory.

Reference

Java software can be downloaded from the Sun Microsystems site:
http://java.sun.com/javase/downloads/index.jsp

Conclusion

In this chapter, you have learned the features of the Java and got familiar with the JVM architecture along with part of Java. By now, you must have got a glimpse of Java. Along with all this, we have also compared the C++ with Java.

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.

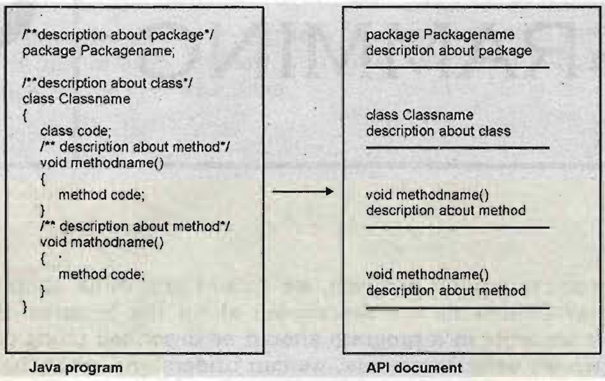


Figure 3.1 API document creation

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>
```

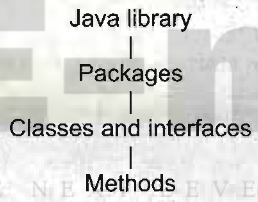
This means, a request is made to the C/C++ compiler to include the header file `<stdio.h>`. What is a header file? A *header file* is a file, which contains the code of functions that will be needed in a program. In other words, to be able to use any function in a program, we must first include the header 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.



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.