CHAPTER

INTRODUCTION TO JAVA

2

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).

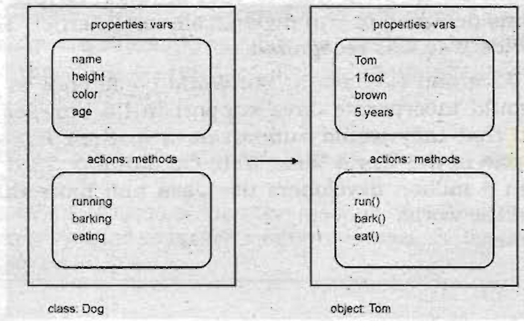

Figure 2.1 Class and object

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.

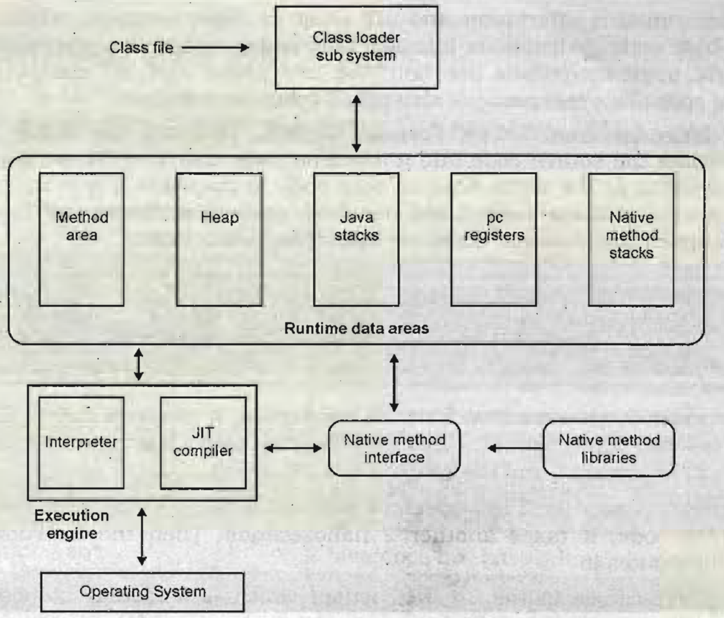

Figure 2.2 Components in JVM architecture

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.