Unit 1 – Introduction to Java

Programming language Types and Paradigms :

A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely

Types:

1. Procedural Programming Languages
2. Structured Programming Languages
3. Object-Oriented Programming Languages

Procedural Programming Languages:

Procedural programming specifies a list of operations that the program must complete to reach the desired state. Procedural programming can be compared to unstructured programming,where all of the code resides in a single large block.
Eg: FORTRAN and BASIC

Structured Programming Languages:

Structured programming is a special type of procedural programming. It provides additional tools to manage the problems that larger programs were creating. Structured programming requires that programmers break program structure into small pieces of code that are easily understood.
Eg: C, Ada, and Pascal

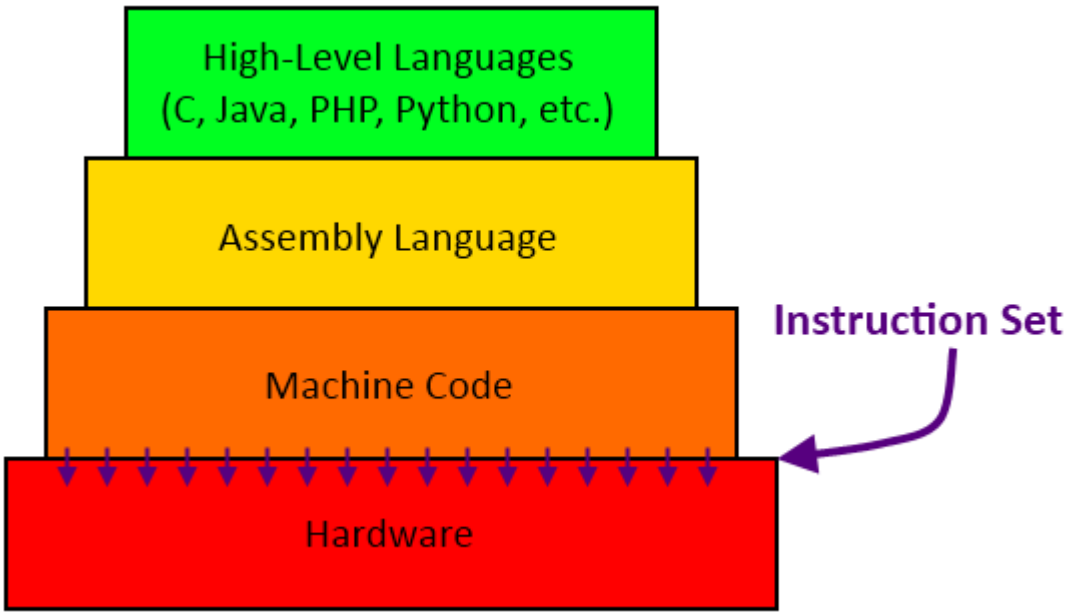
Object-Oriented Programming Languages:

Object-oriented programming is one the newest and most powerful paradigms. In object oriented programs, the designer specifies both the data structures and the types of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object. A program thus becomes a collection of cooperating objects, rather than a list of instructions.

Programming Hierarchy

High-level programming languages, while simple compared to human languages, are more complex than the languages the computer actually understands, called machine languages. Each different type of CPU has its own unique machine language.

Lying between machine languages and high-level languages are languages called assembly languages. Assembly languages are similar to machine languages, but they are much easier to program in because they allow a programmer to substitute names for numbers. Machine languages consist of numbers only.



Lying above high-level languages are languages called fourth-generation languages (usually abbreviated 4GL). 4GLs are far removed from machine languages and represent the class of computer languages closest to human languages. Regardless of what language you use, you eventually need to convert your program into machine language so that the computer can understand it.

There are two ways to do this:

–compile the program
–interpret the program

The question of which language is best is one that consumes a lot of time and energy among computer professionals. Every language has its strengths and weaknesses. For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features, but it is complex and difficult to learn.

How Computer architecture affects a language?

The basic architecture of computers has had a profound effect on language design. Most of the popular languages of the past 50 years have been designed around the prevalent computer architecture, called the von Neumann architecture, after one of its originators, john von Neumann.

These languages are called imperative languages. In a von Neumann computer, both data and programs are stored in the same memory. The CPU< which executes instructions, is separate from the memory. Therefore, instructions and data must be piped or transmitted, from memory to the CPU. Results of operations in the CPU must be moved back to memory. Nearly all digital computers built since the 1940s have been based on the von Neumann architecture. Because of the von Neumann architecture, the central features of imperative languages are variables, which model the memory cell, assignment statements, which are based on piping operation and the iterative form of repetition, which is the most efficient way to implement repetition on this architecture. Operands in expressions are piped from memory to the CPU and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment. Iteration is fast on von Neumann computers because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a simple branch instruction. This efficiency discourages the use of recursion for repetition, although recursion is sometimes more natural.

The execution of a machine code program on a von Neumann architecture computer occurs in a process called fetch-execute cycle. As stated earlier, programs reside in memory but are executed in the CPU. Each instruction to be executed must be moved from memory to the processor. The address of the next instruction to be executed in maintained in a register called the program counter. The fetch execute cycle can be simply described by the following algorithm –

Initialize the program counter

Repeat forever

Fetch the instruction pointed by the program counter,

Increment the program counter to point at the next instruction,

Decode the instruction.

Execute the instruction.

End repeat

The ‘decode the instruction’ step in the algorithm means the instruction is examined to determine what action it specifies. Program execution terminates when a stop instruction is encountered, although on a actual computer a stop instruction is rarely executed. Rather, control transfers from the operating systems to a user program execution and then back to the operating system when the user program execution is complete. In a computer system in which more than one user program may be in memory at a given time, this process is far more complex. In a functional language programming can be done without the kind of variables that are used in imperative languages, without assignment statements and without iteration. The structure of imperative programming languages is modeled on a machine architecture rather than on the abilities and inclinations of the users of programming languages, some believe that using imperative languages is somehow more natural than using a functional language.

Features of Java (Why Java?)

Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

Simple:

The simplicity of Java means that a programmer could learn it quickly. The number of language constructs has been kept small and it has a look and feel familiar to C, C++ and Objective-C. Therefore programmers can easily migrate to Java. Java does not support pointers which are a notorious source of bugs. Eliminating them simplifies the language and eliminates many potential bugs. Memory is automatically allocated and deallocation is done by the garbage collector and must not be explicitly programmed.

Object-Oriented:

Java is an Object-Oriented language from the ground up. Java includes numbers, booleans and characters as ground types. In Java, Classes are not first class citizens, i.e. classes are not objects and therefore the metaclass concept is not supported. In this way, classes can be defined only statically. Java supports single inheritance but some aspects of multiple inheritance can be achieved using the interface concept. That is, multiple inheritance exists only for abstract methods and static-final variables (i.e. class constants).

Distributed:

Java has been designed to support applications on networks. It supports different levels of connectivity through classes in the java.net package. The URL class allows a Java application to open and access remote objects on the internet. In Java, it is transparent if a file is local or remote. Using the Socket class, one can create distributed clients and servers. New upgrades of Java support *Remote Method Invocation* (RMI). Approaches which execute code on other machines across a network have been confusing as well as tedious to debug. The nicest way to think about them is that some objects happen to live in another machine, and then you can send a message to that object and get a result as if the object lived on your local machine. This simplification is exactly what Java Remote Method Invocation has implemented since version 1.1.

Interpreted:

The Java compiler generates byte-code instead of machine-dependent code. To run a program one has to load it into the Java virtual machine. This machine has been implemented for the most popular operating systems.

Robust:

A main source of errors in the C programming language is pointer manipulation. The explicit use of pointers has been removed from Java and in this way Java programs are easier to debug. On the other hand Java is a strongly typed language and therefore extensive compile-time checking for potential type errors is done. Java requires explicit method declarations and there are no implicit

declarations (as in C, for example). The exception handling model of Java allows to group all the error handling code in one place via the try/catch/finally construct.

Secure:

As already mentioned, Java has no pointers and therefore a program cannot get out of its program segment. Also the Java compiler does not handle memory layout decisions and one cannot write dynamic code. In other words one cannot evaluate a string which contains a piece of correct code. Java has neither pointers nor pointer arithmetic which could be used to sidestep Java's run-time checks and security mechanism. Another layer of security protection is commonly referred to as the ``sandbox model'': untrusted code is placed in a ``sandbox'' where it can play safely without doing any damage to the ``real world'' or full Java environment. When an applet or other untrusted code is running in the sandbox, there are a number of restrictions on what it can do. The most obvious of these restrictions is that it has no access whatsoever to the local file system. There are a number of other restrictions in the sandbox as well. These restrictions are enforced by a SecurityManager class. The model works because all of the core Java classes that perform sensitive operations, such as filesystem access, first ask permission of the currently installed SecurityManager. If the call is being made, directly or indirectly, by untrusted code, the security manager throws an exception, and the operation is not permitted. These things make Java ``secure'' or, at least, more secure than other programming languages. It is also worth mentioning that some security ``holes'' have been discovered in early versions of Java. These ``holes'' have been repaired in current versions. A chronology of security-related bugs is available. Also a Frequently Asked Questions file on Java Security is available. As one can expect, security is a very ``hot'' theme when implementing distributed systems in open networks.

Architecture Neutral:

As Java programs are compiled to byte-code machine, compiled programs will run on every architecture which implements the Java virtual machine. In this way programmers do not need to maintain software versions for different platforms; they have only to maintain one version. The compiled version will run with the same look and feel, or at least similar, on any platform. It is assumed that any implementation of the Java virtual machine fulfils the intended semantics of the original one. This is not always the case. Moreover, different semantics for the scheduling have been detected under different implementations, specially those running under Windows 95 and Windows NT. It is expected in the near future, that any new version of the Java virtual machine behaves as expected. I.e. it is expected that Java behaves trully architecture neutral.

Portable:

Following the point above, portability is achieved for free. There are no implementation dependent features. For example, Java explicitly defines the size of basic types and therefore adouble has the same size in any platform. A pure Java program is one that relies only on the documented and specified Java platform. This sentence implies that Java programs are not trully portable. This is true; there are some aspects of Java (operating system calls, file names, paths to files, events, etc. for example) which are not portable to any platform. There is a critical distinction - and connection - between portability and purity. Most people think of a portable program as ``one that produces the same results on any platform''. This is actually a very unprecise definition.

We might be tempted to define a portable program as ``one that fulfills its function on any platform''. However, this definition is very hard to apply because the ``function'' of each program is specific to that program. Using this definition, we cannot evaluate the portability of a program without (ultimately) understanding the requirements of its users; this definition blurs the distinction between portability and quality.

However, we can determine a simple specification without understanding the requirements of the user: normal termination. The Java language defines two exit paths for any procedure: a normal termination, expressed as an exit or return, and an abrupt termination, expressed as a throw of an exception. Clearly, a program has failed to operate normally when the program as a whole terminates abruptly due to an uncaught exception.

For testing the portability and ``purity'' of a Java code, a 100% purity test has been implemented by SUN. This test is a program which gives some hints of which parts of the input code is not portable. Of course termination of programs cannot be evaluated.

High-performance:

As Java is an interpreted language, one cannot expect the performance of a compiled one. Nevertheless, Java code is faster than other codes of interpreted languages in UNIX environments such as perl, tcl, etc. or script languages as bash, sh, csh, etc. Some experiments have been done comparing similar systems implemented in languages which compile to native code and Java. The result is not surprisingly: native code is faster than Java code in factors between 100 and 1000. New technologies are under development such as ``Just in Time Compiling'' (JITC). JITC takes Java compiled classes (byte code machine) and generates native code for that platform. It is expected that the generated native code will run as fast as native code generated by other compilers. Statistics about this point are still missing.

Multithreaded:

Java has built-in constructs for multithreading. These primitives are based on the monitor and condition variable model. Using the synchronized keyword, one can specify that certain methods within a class cannot run concurrently. Java makes programming with threads much easier by providing built-in language support for threads. The java.lang package provides a Thread class that supports methods to start and stop threads and set thread priorities, among other things.

Dynamic:

Java manipulates memory in a dynamic way. Classes are loaded by demand even across a network. The distributed nature of Java really shines when combined with its dynamic class loading capabilities. Together, these features make it possible for a Java interpreter to download and run code from across a network.

We can summarize that Java is not perfect, but compared to other popular programming languages one can conclude that all the features make Java a programming language with aptitudes to implement software engineering projects and distributed systems. Moreover, for a short time ISO members have approved SUN'S pas application. I.e. Java is not more a de facto standard but also a

ISO standard. It is expected that in the mean future to have implementations of the Java virtual machine compliant with the standard.

Flavors of Java

Java development can be categorized in three part according to the three flavours of Java Technology. These three flavors are
1. JSE: It covers the core java part of the java language. Also called Standard edition and is used for developing Standalone Applications.
2. JEE: It is also known as advance java. It includes JSP, Servlet, and EJB. Enterprise application can be developed by used these part of the java technology. Also called Enterprise edition and is used for developing web applications.
3. JME: This part of the java technology covers the micro edition. This technology is used to develop the mobile applications. Also known as Macro edition and is used for developing device programs and mobile application

---

Java Design Goals

Before Java, C++ was the language of choice for modern development of applications and it was perfect for a long time. But the growth of the Internet, the World-Wide Web, and “electronic commerce” has introduced new dimensions of complexity into the development process for new applications. Now the application is not only restricted to desktop, it should be available in the web in a distributed environment. And the ever changing demand cannot be met using old languages. So the thinking process started to develop a completely new programming environment to meet all the needs to modern application development.

The design and development of Java started as part of a research project to develop some advanced software for a wide variety of distributed networked devices and embedded systems. The purpose of this project was to develop a small, reliable, flexible, portable and distributed real-time development environment.

The design requirements of Java environment are mainly guided by the nature of the computing and deployment environment where the application will run.

In the internet edge, it was an absolute requirement that the application should run in the web and accessible from anywhere at any point of time. So the massive growth of World-Wide Web leads us to a completely new paradigm of looking at development and distribution of software. Another important feature is to develop a secured application which will live in the web yet secured enough to use in electronic commerce transactions. So java was designed as a platform to support high performance, robust and scalable applications running in a distributed networked environment.

The traditional schemes of binary distribution, release, upgrade, patch are no longer valid for applications running on multiple platforms on heterogeneous networked environment, So to survive in this new environment, Java must be architecture neutral, platform independent, portable and dynamically adaptable in different environment.

The Java system developed to meet these internet age requirements is simple so that the developers can easily learn and apply it. So it is made object oriented so that it can take advantage of modern software development methodologies and comfortable into distributed client-server applications. It is made multithreaded for high performance in applications that need to perform multiple concurrent activities simultaneously. It is interpreted which makes it portable and dynamic.

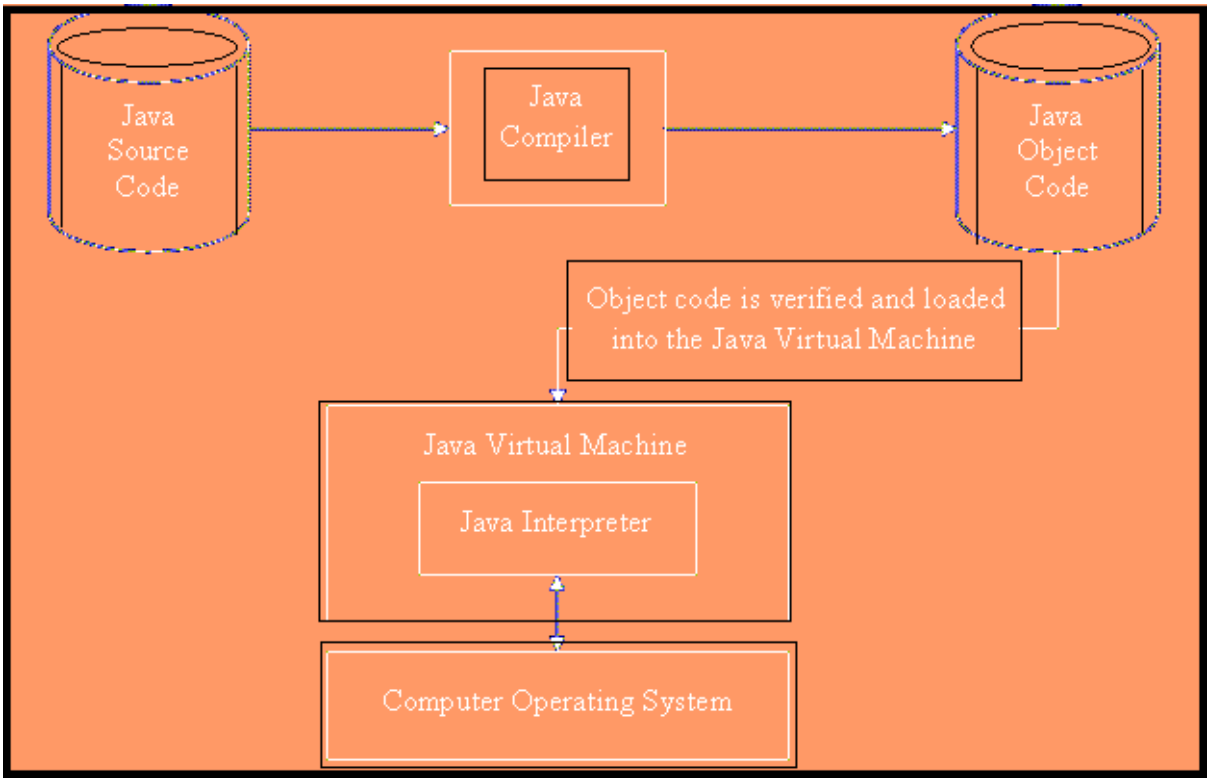


Figure : Design goals of Java

Java designers kept design goals in mind while developing Java –

Simple and Object Oriented

Primary thinking was to make a simple language which the developers can learn easily without extensive programmer training and also make it compatible to current software practices. After learning fundamental concepts programmers can be productive from the very beginning.

Java environment was designed to be object oriented in nature. The Object technology has finally found its independent way into the programming world after a long time of research and development. The requirement of distributed application development and client-server based systems coincide with the encapsulated, message driven paradigms of object oriented software. To function properly in the ever increasing complex, network-based environments, the application must be developed on object-oriented concepts. Java follows a clean and efficient object-based programming environment.

As a result the developers can access existing libraries of tested java objects that provide wide range of functionalities ranging from basics to complex network and UI programming. Flexibility is that the libraries can also be extended to provide new functionalities.

For easy migration java follows C++ features excluding the unnecessary complexities. So the developers found it easy to learn and implement.

Robust and Secure

In the age of internet and World Wide Web it is an absolute necessity to develop robust and reliable application especially in electronic commerce environment.

So Java is designed to create highly reliable and robust application. It provides double level checking starting with compile-time and followed by a second level of run-time checking. The highly optimized and proven language features guide developers towards reliable programming practices. The memory management controlled by JVM removes the fear of pointer management and memory leaks. The developers can develop Java applications with confidence that the system will take care of errors quickly and that major issues will not surface at the production time.

Another objective of Java design was to make it operate in distributed environments, so the security concerns are of high importance. As a result the security features of java are designed into the language and run-time system itself so it is secured from the foundation. In the networked environment also java applications are equally secured from intrusion by unauthorized code attempting to get into the system and making it vulnerable.

Architecture Neutral and Portable

Another main objective of java design was to make it architecture neutral so that it can be run and deployed on heterogeneous platform and networked environments. In this type of environment, the application must be capable of executing on different hardware and operating system.

To make java capable of running on different operating environments, the Java compiler generates byte codes which is an architecture neutral intermediate format designed to make it compatible on multiple hardware and software platforms. As a result the interpreted nature of Java environment solves both the binary distribution and version problem.

The Java Virtual Machine (JVM) is a combination of architecture-neutral and portable feature of java environment.JVM is basically the specification of an abstract machine for which Java compilers can generate code and it runs within the environment. The JVM specification is based on POSIX interface specification which is an industry standard definition of Portable System Interface. So the implementation of JVM for any target architecture is simple if the target system supports the basic requirement of the virtual machine.

High Performance

For any application especially which are distributed and internet enabled, performance is always a major consideration. The Java environment achieves high performance by adopting a mechanism which helps it to run in full speed without checking the run time environment. The garbage collectors runs in a separate thread ensuring timely release of memory and hence improve the performance.

What is JVM?

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your *.java files to obtain *.class files that contain the bytecodes understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

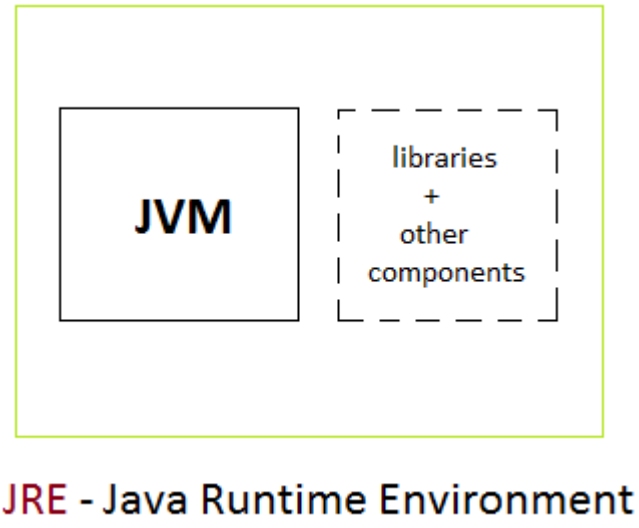
JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Run Environment).

Java applications are called WORA (Write Once Run Everywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, a *.class* file(contains byte-code) with the same filename is generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

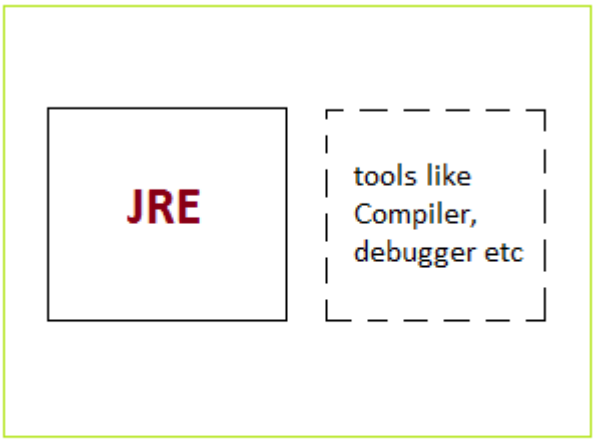
Difference between JDK and JRE

JRE : The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.



JRE - Java Runtime Environment

JDK : The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



JDK - Java Development Kit

What is Java Bytecode?

Java bytecode is the result of the compilation of a Java program, an intermediate representation of that program which is machine independent.

The Java bytecode gets processed by the Java virtual machine (JVM) instead of the processor. It is the job of the JVM to make the necessary resource calls to the processor in order to run the bytecode.

Java bytecode is the resulting compiled object code of a Java program. This bytecode can be run in any platform which has a Java installation in it.

This machine independence is because of the Java virtual machine that runs the bytecode in proxy of the processor which means that a Java programmer does not have to be knowledgeable about the quirks and nuances about specific operating systems and processors that the program will be run on because the virtual machine takes care of those specifics.

The Java bytecode is not completely compiled, but rather just an intermediate code sitting in the middle because it still has to be interpreted and executed by the JVM installed on the specific platform such as Windows, Mac or Linux.

Upon compile, the Java source code is converted into the .class bytecode.