UNIT 1

Introduction

Java is one of the most popular and widely used programming language and platform. A platform is an environment that helps to develop and run programs.Java is designed by James Goslig at Sun Microsystems lab.

Java is fast, reliable and secure. From desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet, Java is used.

Buzz words

- **Simple:** Java is easy to learn. Java has removed many confusing and rarely-used features e.g. explicit pointers, operator overloading etc. Java also takes care of memory management and it also provides an automatic garbage collector. This collects the unused objects automatically.
- **Platform-independent language:** The programs written in Java language, after compilation, are converted into an intermediate level language called the **bytecode.** This makes java highly portable as its bytecodes can be run on any machine by an interpreter called the Java Virtual Machine(JVM) and thus java provides 'reusability of code'.
- **object-oriented programming language:** OOP makes the complete program simpler by dividing it into a number of objects. The objects can be used as a bridge to have data flow from one function to another.
- **Robust:**It provides strong inbuilt exception handling, garbage collection.
- **Multithreaded:** Java can perform many tasks at once by defining multiple threads. For example, a program that manages a Graphical User Interface (GUI) while waiting for input from a network connection uses another thread to perform and wait's instead of using the default GUI thread for both tasks. This keeps the GUI responsive.
- **Create applets:** Applets are programs that run in web browsers
- **Does not require any pre-processor:** It does not require inclusion of header files for creating a Java application.

Translation Process:

Java source file (i.e .java) will be given as the input to compiler and the output of the compiler is byte code or .class file because of which java is platform independent.

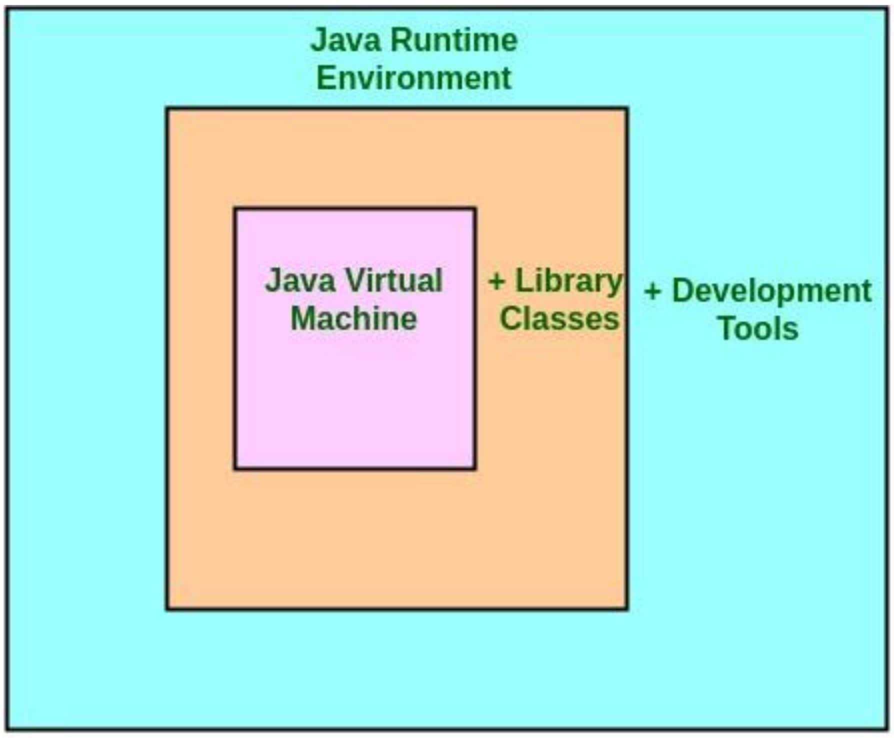
The byte code acts as input to interpreter which analyses and executed byte code and gives the desired output.

Thus java is compiled and interpreted language.

Few terminologies encountered

- **JAVA DEVELOPMENT KIT:** The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

- **JAVA RUNTIME ENVIRONMENT: Java Runtime Environment** (to say JRE) is an installation package which provides environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
- **JVM** – **Java Virtual machine**(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.



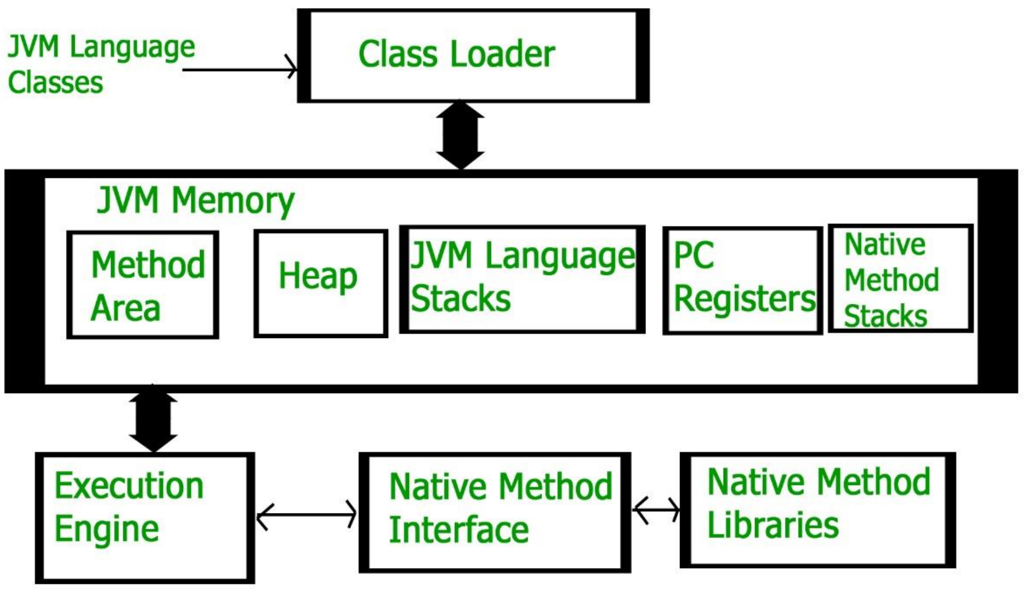
JDK = JRE + Development Tool
JRE = JVM + Library Classes

Some more information regarding JVM:

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 Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). 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, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it.



Class Loader Subsystem
It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading : The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory.

Linking : Performs verification, preparation.

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not.

- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.

Initialization : In this phase, all static variables are assigned with their values defined in the code

JVM Memory

Method area : In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap area : Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

Stack area : For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers : Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

Native method stacks : For every thread, separate native stack is created. It stores native method information.

Execution Engine

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
- *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector.

Simple program: Refer a1.java

Variables, Data Type : The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs
- Names should begin with a letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Names should start with a lowercase letter and it cannot contain whitespace
- Reserved words (like Java keywords, such as int or String) cannot be used as names

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

- String - stores text, such as "Hello". String values are surrounded by double quotes
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

  Syntax:

- *type variable* = *value*;

Display variables:

The println() method is often used to display variables.

To combine both text and a variable, use the + character:

Ex:

```
String name = "world";
System.out.println("Hello " + name);//Hello world
```

Primitive and non-primitive data types

Data type	Size
byte	1 byte
short	2 bytes

int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
boolean	1 bit
char	2 bytes

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are String,array,class,interface.

Typecasting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  byte -> short -> char -> int -> long -> float -> double

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char -> short -> byte

Refer program: a2.java

Mixed Mode operation:

All floating point values (float and double) in an arithmetic operation (+, −, *, /) are converted to double type before the arithmetic operation in performed.

All integer values (byte, short and int) in an arithmetic operations (+, −, *, /, %) are converted to int type before the arithmetic operation in performed. However, if one of the values in an arithmetic operation (+, −, *, /, %) is long, then all values are converted to long type before the arithmetic operation in performed.

Refer a3.java

Control structure:

Java supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

Loops : can execute a block of code as long as a specified condition is reached.

```
while (condition) {
  // code block to be executed
}

for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

The switch expression is evaluated once.
The value of the expression is compared with the values of each case.
If there is a match, the associated block of code is executed.

The break and default keywords are optional

Java Break

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**

Java Continue:

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

.