**Java** is a general-purpose computer programming language that is concurrent, class-based, object-oriented etc.

Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. The latest version is Java 8.

There are few things which must be clear before setting up the environment

1.JDK(Java Development Kit) : JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

2.JRE(Java Runtime Environment) : JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.

3.JVM: JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.

**main method**: In Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

public: So that JVM can execute the method from anywhere.

static: Main method is to be called without object. The modifiers public and static can be written in either order.

void: The main method doesn't return anything.

main(): Name configured in the JVM.

String[]: The main method accepts a single argument: an array of elements of type String.

**System.out.println()**

System is a class in the java.lang package.

out is a static member of the System class, and is an instance of java.io.PrintStream.

println is a method of java.io.PrintStream . This method is overloaded to print message to output destination, which is typically a console or file.

/\* This is a simple Java program.

FileName : "HelloWorld.java". \*/

class HelloWorld

{

// Your program begins with a call to main().

// Prints "Hello, World" to the terminal window.

public static void main(String args[])

{

System.out.println("Hello World");

}

}

Output : Hello World

**Compiling the program :**

After successfully setting up the environment, we can open terminal in both Windows/Unix and can go to directory where the

file – HelloWorld.java is present.

Now, to compile the HelloWorld program, execute the compiler – javac , specifying the name of the source file on the command line, as shown:

javac HelloWorld.java

The compiler creates a file called HelloWorld.class (in present working directory) that contains the bytecode version of the program. Now, to execute our program, JVM(Java Virtual Machine) needs to be called using java, specifying the name of the class file on the command line, as shown:

java HelloWorld

This will print “Hello, World” to the terminal screen.

==========================================================================================================================

**How JVM Works – JVM Architecture?**

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.

Image Source : https://en.wikipedia.org/wiki/Java\_virtual\_machine

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.

For each .class file, JVM stores following information in method area.

Fully qualified name of the loaded class and its immediate parent class.

Whether .class file is related to Class or Interface or Enum

Modifier, Variables and Method information etc.

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

Please note that this object is of type Class predefined in java.lang package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc.

To get this object reference we can use getClass() method of Object class.

// A Java program to demonstrate working of a Class type

// object created by JVM to represent .class file in

// memory.

import java.lang.reflect.Field;

import java.lang.reflect.Method;

// Java code to demonstrate use of Class object

// created by JVM

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student();

// Getting hold of Class object created

// by JVM.

Class c1 = s1.getClass();

// Printing type of object using c1.

System.out.println(c1.getName());

// getting all methods in an array

Method m[] = c1.getDeclaredMethods();

for (Method method : m)

System.out.println(method.getName());

// getting all fields in an array

Field f[] = c1.getDeclaredFields();

for (Field field : f)

System.out.println(field.getName());

}

}

// A sample class whose information is fetched above using

// its Class object.

class Student

{

private String name;

private int roll\_No;

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public int getRoll\_no() { return roll\_No; }

public void setRoll\_no(int roll\_no) {

this.roll\_No = roll\_no;

}

}

Output:

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

Note : For every loaded .class file, only one object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking :** Performs verification, preparation, and (optionally) resolution.

**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. If verification fails, we get run-time exception java.lang.VerifyError.

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

**Resolution :** It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization :** In this phase, all static variables are assigned with their values defined in the code and static block(if any).

This is executed executed from top to bottom in a class and from parent to child in class hierarchy.

In general there are three class loaders :

**Bootstrap class loader :** Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in JAVA\_HOME/jre/lib directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.

**Extension class loader :** It is child of bootstrap class loader. It loads the classes present in the extensions directories JAVA\_HOME/jre/lib/ext(Extension path) or any other directory specified by the java.ext.dirs system property.

It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.

**System/Application class loader :** It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java the sun.misc.Launcher$ExtClassLoader class.

// Java code to demonstrate Class Loader subsystem

public class Test

{

public static void main(String[] args)

{

// String class is loaded by bootstrap loader, and

// bootstrap loader is not Java object, hence null

System.out.println(String.class.getClassLoader());

// Test class is loaded by Application loader

System.out.println(Test.class.getClassLoader());

}

}

**Output:**

null

sun.misc.Launcher$AppClassLoader@73d16e93

Note : JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to

extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.

Image Source: http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html

=====================================================================================================================================

**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.

Image Source : http://java.scjp.jobs4times.com/fund/fund2.png

**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, refer Garbage Collector.

**Java Native Interface (JNI) :**

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

====================================================================================================================

**Does JVM create an object of class Main?**

The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

// Not Main is abstract

abstract class Main {

public static void main(String args[])

{

System.out.println("Hello");

}

}

Since we can’t create object of abstract classes in Java, it is guaranteed that object of class with main() is not created by JVM.

==============================================================================================================================

**Is main method compulsory in Java?**

The answer to this question depends on version of java you are using. Prior to JDK 5, main method was not mandatory in a java program.

You could write your full code under static block and it ran normally.

The static block is first executed as soon as the class is loaded before the main(); method is invoked and therefore before the main() is called. main is usually declared as static method and hence Java doesn’t need an object to call main method.

However, From JDK6 main method is mandatory. If your program doesn’t contain main method, then you will get a run-time error “main method not found in the class”. Note that your program will successfully compile in this case, but at run-time, it will throw error.

// This program will successfully run

// prior to JDK 5

public class Test

{

// static block

static

{

System.out.println("program is running without main() method");

}

}

=============================================================================================================================

public class Test

{

public static void main(String[] args)

{

System.out.print("Y" + "O");

System.out.print('L' + 'O');

}

}

**Actual Output:**

“YO155”.

public class Test

{

public static void main(String[] args)

{

System.out.print("Y" + "O");

System.out.print('L');

System.out.print('O');

}

}

**Output:** YOLO

Explanation: This will now print “YOLO” instead of “YO7679”. It is because the widening primitive conversion happens only when ‘+’ operator is present.

Widening primitive conversion is applied to convert either or both operands as specified by the following rules. The result of adding Java chars, shorts or bytes is an int:

If either operand is of type double, the other is converted to double.

Otherwise, if either operand is of type float, the other is converted to float.

Otherwise, if either operand is of type long, the other is converted to long.

Otherwise, both operands are converted to type int

=============================================================================================================================

**Interesting facts about null in Java**

Almost all the programming languages are bonded with null. There is hardly a programmer, who is not troubled by null.

In Java, null is associated java.lang.NullPointerException. As it is a class in java.lang package, it is called when we try to perform some operations with or without null and sometimes we don’t even know where it has happened.

Below are some important points about null in java which every Java programmer should know:

**1.** null is Case sensitive: null is literal in Java and because keywords are case-sensitive in java, we can’t write NULL or 0 as in C language.

public class Test

{

public static void main (String[] args) throws java.lang.Exception

{

// compile-time error : can't find symbol 'NULL'

Object obj = NULL;

//runs successfully

Object obj1 = null;

}

}

**Output:**

5: error: cannot find symbol

can't find symbol 'NULL'

^

variable NULL

class Test

1 error

**2.** Reference Variable value: Any reference variable in Java has default value null.

public class Test

{

private static Object obj;

public static void main(String args[])

{

// it will print null;

System.out.println("Value of object obj is : " + obj);

}

}

**Output:**

Value of object obj is : null

**3.** Type of null: Unlike common misconception, null is not Object or neither a type. It’s just a special value, which can be assigned to any reference type and you can type cast null to any type

Examples:

// null can be assigned to String

String str = null;

// you can assign null to Integer also

Integer itr = null;

// null can also be assigned to Double

Double dbl = null;

// null can be type cast to String

String myStr = (String) null;

// it can also be type casted to Integer

Integer myItr = (Integer) null;

// yes it's possible, no error

Double myDbl = (Double) null;

**4.** Autoboxing and unboxing : During auto-boxing and unboxing operations, compiler simply throws Nullpointer exception error if a null value is assigned to primitive boxed data type.

public class Test

{

public static void main (String[] args) throws java.lang.Exception

{

//An integer can be null, so this is fine

Integer i = null;

//Unboxing null to integer throws NullpointerException

int a = i;

}

}

**Output:**

Exception in thread "main" java.lang.NullPointerException at Test.main(Test.java:6)

**5.** instanceof operator: The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.

This is an important property of instanceof operation which makes it useful for type casting checks.

public class Test

{

public static void main (String[] args) throws java.lang.Exception

{

Integer i = null;

Integer j = 10;

//prints false

System.out.println(i instanceof Integer);

//Compiles successfully

System.out.println(j instanceof Integer);

}

}

**Output:**

false

true

**6.** Static vs Non static Methods: We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won’t throw Null pointer Exception.

public class Test

{

public static void main(String args[])

{

Test obj= null;

obj.staticMethod();

obj.nonStaticMethod();

}

private static void staticMethod()

{

//Can be called by null reference

System.out.println("static method, can be called by null reference");

}

private void nonStaticMethod()

{

//Can not be called by null reference

System.out.print(" Non-static method- ");

System.out.println("cannot be called by null reference");

}

}

**Output:**

static method, can be called by null referenceException in thread "main" java.lang.NullPointerException at Test.main(Test.java:5)

**7.** == and !=The comparision and not equal to operators are allowed with null in Java. This can made useful in checking of null with objects in java.

public class Test

{

public static void main(String args[])

{

//return true;

System.out.println(null==null);

//return false;

System.out.println(null!=null);

}

}

**Output:**

true

false

=====================================================================================================================================

**Association, Composition and Aggregation in Java**

**Association**

Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

// Java program to illustrate the

// concept of Association

import java.io.\*;

// class bank

class Bank

{

private String name;

// bank name

Bank(String name)

{

this.name = name;

}

public String getBankName()

{

return this.name;

}

}

// employee class

class Employee

{

private String name;

// employee name

Employee(String name)

{

this.name = name;

}

public String getEmployeeName()

{

return this.name;

}

}

// Association between both the

// classes in main method

class Association

{

public static void main (String[] args)

{

Bank bank = new Bank("Axis");

Employee emp = new Employee("Neha");

System.out.println(emp.getEmployeeName() + " is employee of " + bank.getBankName());

}

}

Run on IDE

**Output:**

Neha is employee of Axis

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

**Aggregation**

It is a special form of Association where:

1. It represents Has-A relationship.

2. It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.

3. In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity.

**Example:** There is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).

It represents a Has-A relationship.

**When do we use Aggregation ?**

Code reuse is best achieved by aggregation.

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

1. It represents part-of relationship.

2. In composition, both the entities are dependent on each other.

3. When there is a composition between two entities, the composed object cannot exist without the other entity.

In above example a library can have no. of books on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book cannot exist without library. That’s why it is composition.

**Aggregation vs Composition**

**1. Dependency:** Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human

**2. Type of Relationship:** Aggregation relation is “has-a” and composition is “part-of” relation.

**3. Type of association:** Composition is a strong Association whereas Aggregation is a weak Assocation.

====================================================================================================================================

Useful and/or Advanced Features

**Generics in Java**

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

**Generic Class**

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we cannot use primitives like

'int','char' or 'double'.

// A Simple Java program to show working of user defined

// Generic classes

// We use < > to specify Parameter type

class Test<T>

{

// An object of type T is declared

T obj;

Test(T obj) { this.obj = obj; } // constructor

public T getObject() { return this.obj; }

}

// Driver class to test above

class Main

{

public static void main (String[] args)

{

// instance of Integer type

Test <Integer> iObj = new Test<Integer>(15);

System.out.println(iObj.getObject());

// instance of String type

Test <String> sObj =

new Test<String>("GeeksForGeeks");

System.out.println(sObj.getObject());

}

}

**Output:**

15

GeeksForGeeks

We can also pass multiple Type parameters in Generic classes.

// A Simple Java program to show multiple

// type parameters in Java Generics

// We use < > to specify Parameter type

class Test<T, U>

{

T obj1; // An object of type T

U obj2; // An object of type U

// constructor

Test(T obj1, U obj2)

{

this.obj1 = obj1;

this.obj2 = obj2;

}

// To print objects of T and U

public void print()

{

System.out.println(obj1);

System.out.println(obj2);

}

}

// Driver class to test above

class Main

{

public static void main (String[] args)

{

Test <String, Integer> obj =

new Test<String, Integer>("GfG", 15);

obj.print();

}

}

**Output:**

GfG

15

**Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

// A Simple Java program to show working of user defined

// Generic functions

class Test

{

// A Generic method example

static <T> void genericDisplay (T element)

{

System.out.println(element.getClass().getName() + " = " + element);

}

// Driver method

public static void main(String[] args)

{

// Calling generic method with Integer argument

genericDisplay(11);

// Calling generic method with String argument

genericDisplay("GeeksForGeeks");

// Calling generic method with double argument

genericDisplay(1.0);

}

}

Run on IDE

**Output :**

java.lang.Integer = 11

java.lang.String = GeeksForGeeks

java.lang.Double = 1.0

**Advantages of Generics:**

Programs that uses Generics has got many benefits over non-generic code.

Code Reuse: We can write a method/class/interface once and use for any type we want.

**Type Safety :** Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

// A Simple Java program to demonstrate that NOT using

// generics can cause run time exceptions

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creatinga an ArrayList without any type specified

ArrayList al = new ArrayList();

al.add("Sachin");

al.add("Rahul");

al.add(10); // Compiler allows this

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

// Causes Runtime Exception

String s3 = (String)al.get(2);

}

}

**Output :**

Exception in thread "main" java.lang.ClassCastException:

java.lang.Integer cannot be cast to java.lang.String at Test.main(Test.java:19)

**How generics solve this problem?**

At the time of defining ArrayList, we can specify that this list can take only String objects.

// Using generics converts run time exceptions into

// compile time exception.

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Now Compiler doesn't allow this

al.add(10);

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

String s3 = (String)al.get(2);

}

}

**Output:**

15: error: no suitable method found for add(int)

al.add(10);

^

Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

// We don't need to typecast individual members of ArrayList

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Typecasting is not needed

String s1 = al.get(0);

String s2 = al.get(1);

}

}

Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.