What is Java?

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is:

- **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- **Simple:** Java is designed to bke easy to learn. If you understand the basic concept of OOP Java would be easy to master.
- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural-neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
- **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called **Oak** after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as **Java**, from a list of random words.

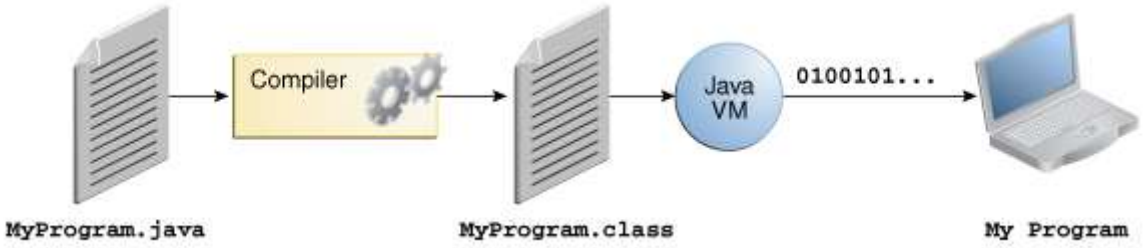
Sun released the first public implementation as Java 1.0 in **1995**. It promised **Write Once, Run Anywhere**(WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

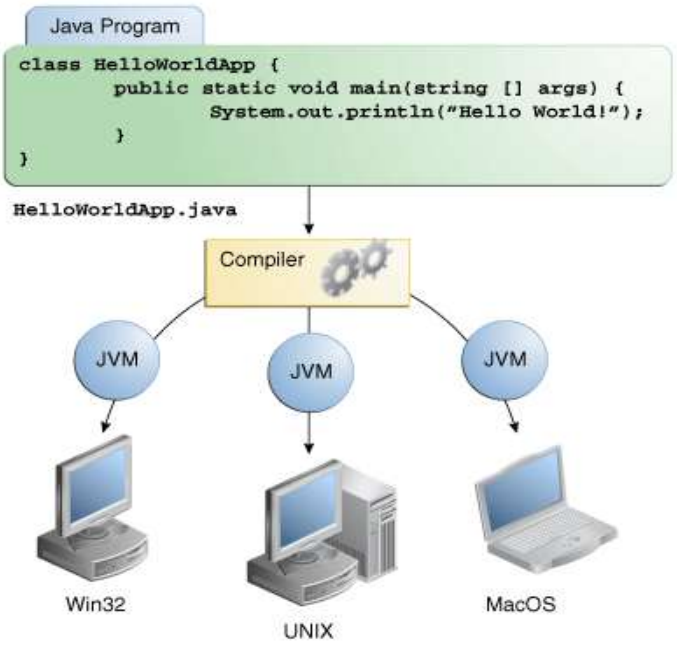
Process flow of execution

In the Java programming language, all source code is first written in plain text files ending with the `.java` extension. Those source files are then compiled into `.class` files by the `javac` compiler. A `.class` file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine[1] (Java VM). The `java` launcher tool then runs your application with an instance of the Java Virtual Machine.



An overview of the software development process.

Because the Java VM is available on many different operating systems, the same `.class` files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS



Through the Java VM, the same application is capable of running on multiple platforms.

What is JVM?

JVM stands for ***Java Virtual Machine***. It is an abstract computing machine that is **responsible for executing Java programs**. When you write a Java program, the source code is compiled into byte code which is understandable by the JVM. Upon execution, the JVM translates byte code into machine code of the target operating system.

The JVM is the cornerstone of the Java programming language. It is responsible for the very well-known feature of Java: cross-platform. That means you can write a Java program once and run it anywhere: Windows, Linux, Mac and Solaris, as long as JRE is installed on the host operating system.

Every time you run a Java program, the JVM is started to execute and manage the program's execution. The JVM is running in two modes: client (default) and server.

An Oracle's implementation for JVM is called Java HotSpot VM

What is JRE?

JRE stands for ***Java Runtime Environment***. It provides the **libraries**, JVM and other components necessary for you to run applets and applications written in the Java programming language.

The JRE contains standard tools such as java, keytool, policytoo, but it doesn't contain compilers or debuggers for developing applets and applications.

When you deploy your Java applications on client's computer, the client needs a JRE to be installed.

What is JDK?

JDK stands for ***Java Development Kit***. It's a superset of JRE. The JDK includes the JRE plus command-line development tools such as compilers (javac) and debuggers (jdb) and others (jar, javadoc, etc) that are necessary or useful for developing applets and applications.

Therefore, as a Java programmer, you have to install JDK as a minimum requirement for theh development environment.

Summary:

- **JVM** = JVM is the Virtual Machine that runs Java applications. The JVM makes Java platform independence
- **JRE** = JVM + standard libraries: provides environment for executing Java applications
- **JDK** = JRE + development tools for compiling and debugging Java applications

Tips:

- You should have both JRE and JDK installations (setup) on your computer. You will need both during the development process.
- You should have multiple versions of JDK and JRE installed: 1.5, 1.6, 1.7 and 1.8 for different testing purposes in the future.

- You should install both 32-bit and 64-bit versions.
- When installing the JDK, remember to check 'Install Demos and Samples'. Then you can explore various interesting examples in the **demo** directory under JDK's installation path.
- Only the JDK includes source code of the Java runtime libraries. You can discover the source code in the **src.zip** file which can be found under JDK's installation directory.

First Java Program:

```
public class MyFirstJavaProgram {
   /* This is my first java program.
    * This will print 'Hello World' as the output
    */

    public static void main(String []args) {
       System.out.println("Hello World"); // prints Hello World
    }
}
```

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and add the code as above.
2. Save the file as: MyFirstJavaProgram.java.
3. Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
4. Type ' javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
5. Now, type ' java MyFirstJavaProgram ' to run your program.
6. You will be able to see ' Hello World ' printed on the window.

```
C:\> javac MyFirstJavaProgram.java

C:\> java MyFirstJavaProgram

Hello World
```

Basic Syntax:

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case.
  If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
  Example *class MyFirstJavaClass*
- **Method Names -** All method names should start with a Lower Case letter.
  If several words are used to form the name of the method, then each inner word's first letter

should be in Upper Case.
Example *public void myMethodName()*

- **Program File Name -** Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile).
  Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
- **public static void main(String args[]) -** Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character identifiers, can have any combination of characters.
- A keyword cannot be used as an identifier.
- Most importantly identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value
- Examples of illegal identifiers: 123abc, -salary

Java Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	Assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	Final	finally	float
for	Goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	Static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	While		

Comments in Java

Java supports single-line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{
   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
```

```
      */

      public static void main(String []args){
         // This is an example of single line comment
         /* This is also an
                    example of single line comment. */
         System.out.println("Hello World");
      }
   }
```

Datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

The Java language has 8 primitive types: **boolean**, **byte**, **char**,**double**, **float**, **int**, **long**, and **short**.

A **boolean** type represents either *true* or *false* value.

A **char** type represents a single character, such as 'a', 'B', 'c', ...Actually char type is 16-bit integer number (un-signed).

The others are numeric types. The following table lists the primitive types which represent numbers in the Java language (the char type is also included because it is actually a number type):

	Type	Bits	Bytes	Minimum value	Maximum value
Integer numbers	**byte**	8	1	**-128** (-2^7)	**127** (2^7-1)
	char	16	2	**0**	**65,535**
	short	16	2	**-32,768** (-2^{15})	**32,767** (2^{15}-1)
	int	32	4	**-2,147,483,648** (-2^{31})	**2,147,483,647** (2^{31}-1)
	long	64	8	approx. -9,2 billions of billion (-2^{63})	approx. 9,2 billions of billio 1)
Floating point numbers	**float**	32	4	N/A	N/A
	Double	64	8	N/A	N/A

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. The basic form of a variable declaration is shown here:

```
data type variable [ = value][, variable [= value] ...] ;
```

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
```

```
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

Program N0. 2

Local Variable Demo 1

```
public class LocalVarDemo_2
{
   public static void main(String[] args)
   {
        int age=21;    //local variables
        System.out.println("\n\t Value of age = " + age);
   }
}
```

Program N0. 3

Local Variable Demo 2

```
public class LocalVarDemo_3
{
   public static void main(String[] args)
   {
        char ch;
        float avg;
        String str;

        ch = 'A';
        avg = 45.67f;
        str = "Welcome";

        System.out.println("\n\t Value of ch = " + ch);
        System.out.println("\n\t Value of avg = " + avg);
        System.out.println("\n\t Value of str = " + str);
   }
}
```

Program N0. 4

Local Variable Demo 3

```
 public class LocalVarDemo_4
{
   static void showvalue()
   {
        int age = 32; //local variable
        System.out.println("\n\t Value of age = " + age);
```

```
    }
    public static void main(String[] args)
    {
        showvalue();
    }
}
```

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name
  . *ObjectReference.VariableName*.

Example:

```
import java.io.*;

public class Employee{
   // this instance variable is visible for any child class.
   public String name;

   // salary  variable is visible in Employee class only.
   private double salary;

   // The name variable is assigned in the constructor.
   public Employee (String empName){
      name = empName;
   }

   // The salary variable is assigned a value.
   public void setSalary(double empSal){
      salary = empSal;
   }

   // This method prints the employee details.
   public void printEmp(){
      System.out.println("name  : " + name );
```

```
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

Program N0. 5

Instance Variable Class Student

```
class Student
{
        public int roll;      // instance varialbe
        String name;   // data members / fields

        public void showStud()
        {
          System.out.println("\n\t Roll No :" + roll);
          System.out.println("\n\t Name :" + name);
        }
}

public class InstanceVarDemo_5
{
   public static void main(String[] args)
   {
        Student stud = new Student();
        stud.roll = 101;
        stud.name = "Chetan";
        stud.showStud();
   }
}
```

Program N0. 6

Instance Variable Default Values

```
class Demo
{
       int i;
       byte b;
       String str;
       boolean bl;
}

public class InstanceVarDemo_6
{
   public static void main(String[] args)
```

```
    {
        Demo d1 = new Demo();
        System.out.println("Default value of int i = " +
d1.i);
        System.out.println("Default value of byte b = " +
d1.b);
        System.out.println("Default value of String str = "
+ d1.str);
        System.out.println("Default value of boolean bl = "
+ d1.bl);
    }
}
```

Program N0. 7

Class marketing dept

```
class MktDept
{
    int empid;
    int sal;

    static String deptName;
}
public class StaticVarDemo_7
{
        public static void main(String[] args)
        {
                //MktDept m1 = new MktDept();
                //m1.empid = 101;

                MktDept.deptName = "Marketing Department";
                System.out.println("\n\t Dept name = " +
MktDept.deptName);
        }
}
```

Assignment

Program N0. 8

Class Product

Class/static variables:

- Class variables also known as static variables are declared with the ***static*** keyword in a class, but outside a method, constructor or a block.

- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name*ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
public class Employee{
   // salary  variable is a private static variable
   private static double salary;
   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" + salary);
   }
}
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT