Compilation and Execution of a Java Program

Java, being a platform independent programming language, doesn't work on one-step-compilation. Instead, it involves a two-step execution, first through an OS independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system. The two principle stages are explained below:

I] Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode, the compiler follows the following steps:

1. Parse: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.
2. Enter: Enters symbols for the definitions into the symbol table.
3. Process annotations: If Requested, processes annotations found in the specified compilation units.
4. Attribute: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
5. Flow: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
6. ~~Desugar:~~ Debugger Rewrites the AST and translates away some syntactic sugar.
7. Generate: Generates '.Class' files.

II] Execution

The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system. To run, the main class file (the class that contains the method main) is passed to the JVM, and then goes through three main stages before the final machine code is executed. These stages are:

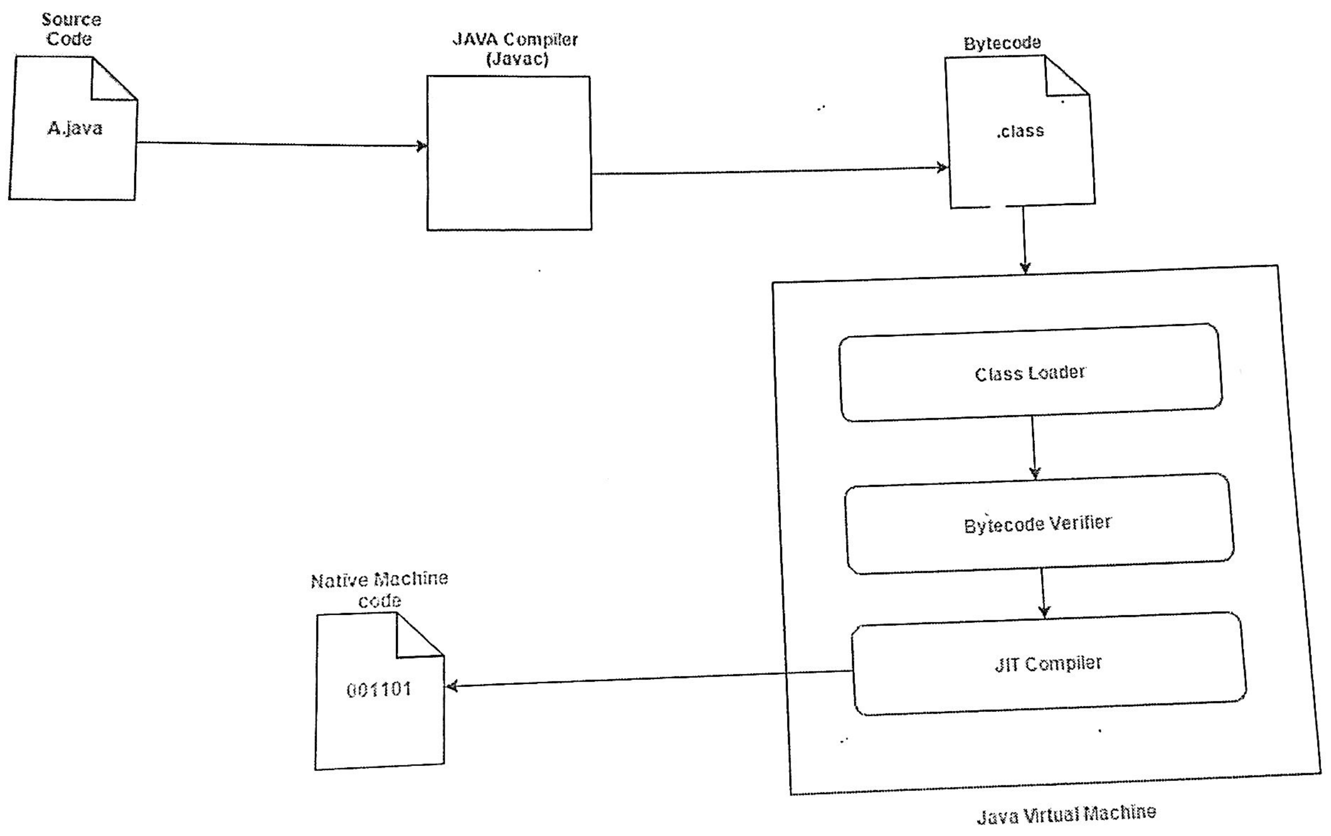
1. Class Loader : The main class is loaded into the memory by passing its '.class' file to the JVM, through invoking the latter. All the other classes referenced in the program are loaded through the class loader.
2. A class loader, itself an object, creates a flat name space of class bodies that are referenced by a string name. ~~The method definition is:~~
3. There are two types of class loaders: primordial, and non-primordial. Primordial class loader is embedded into all the JVMs, and is the default class loader. A non-primordial class loader is a user-defined class loader, which can be coded in order to customize class-loading process. Non-primordial class loader, if defined, is preferred over the default one, to load classes.
4. Bytecode Verifier : After the bytecode of a class is loaded by the class loader, it has to be inspected by the bytecode verifier, whose job is to check that the instructions don't perform damaging actions. The following are some of the checks carried out:

- Variables are initialized before they are used.

- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The run time stack does not overflow.
- If any of the above checks fails, the verifier doesn't allow the class to be loaded.

5. Just-In-Time Compiler : This is the final stage encountered by the java program, and its job is to convert the loaded bytecode into machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed, unless methods are executed less frequently.

The process can be well-illustrated by the following diagram:



Due to the two-step execution process described above, a java program is independent of the target operating system. However, because of the same, the execution time is way more than a similar program written in a compiled platform-dependent program.

Internal Details of Hello Java Program

1. Internal Details of Hello Java

In the previous page, we have learnt about the first program, how to compile and run the first java program. Here, we are going to learn, what happens while compiling and running the java program. Moreover, we will see some question based on the first program.

What happens at compile time?

```
    public static void main(String args[]) {
      int num1 = 100;
      int num2 = 20;

      System.out.println("num1 + num2: " + (num1 + num2) );
      System.out.println("num1 - num2: " + (num1 - num2) );
      System.out.println("num1 * num2: " + (num1 * num2) );
      System.out.println("num1 / num2: " + (num1 / num2) );
      System.out.println("num1 % num2: " + (num1 % num2) );
    }
}
```

Output:

```
num1 + num2: 120
num1 - num2: 80
num1 * num2: 2000
num1 / num2: 5
num1 % num2: 0
```

Checkout these java programs related to arithmetic Operators in Java:

1. Java Program to Add two numbers
2. Java Program to Multiply two Numbers

2) Assignment Operators

Assignments operators in java are: =, +=, -=, *=, /=, %=

num2 = num1 would assign value of variable num1 to the variable.

num2+=num1 is equal to num2 = num2+num1

num2-=num1 is equal to num2 = num2-num1

num2*=num1 is equal to num2 = num2*num1

num2/=num1 is equal to num2 = num2/num1

num2%=num1 is equal to num2 = num2%num1

Example of Assignment Operators

```
public class AssignmentOperatorDemo {
  public static void main(String args[]) {
    int num1 = 10;
    int num2 = 20;

    num2 = num1;
    System.out.println("= Output: "+num2);

    num2 += num1;
    System.out.println("+= Output: "+num2);

    num2 -= num1;
    System.out.println("-= Output: "+num2);

    num2 *= num1;
    System.out.println("*= Output: "+num2);

    num2 /= num1;
    System.out.println("/= Output: "+num2);

    num2 %= num1;
    System.out.println("%= Output: "+num2);
  }
}
```

Output:

```
= Output: 10
+= Output: 20
-= Output: 10
*= Output: 100
/= Output: 10
%= Output: 0
```

3) Auto-increment and Auto-decrement Operators

++ and —

num++ is equivalent to num=num+1;

num— is equivalent to num=num-1;

Example of Auto-increment and Auto-decrement Operators

```
public class AutoOperatorDemo {
        int num1=100;
        int num2=200;
        num1++;
        num2--;
        System.out.println("num1++ is: "+num1);
        System.out.println("num2-- is: "+num2);
    }
}
```

Output:

```
num1++ is: 101
num2-- is: 199
```

4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in java are: &&, ||, !

Let's say we have two boolean variables b1 and b2.

b1&&b2 will return true if both b1 and b2 are true else it would return false.

b1||b2 will return false if both b1 and b2 are false else it would return true.

!b1 would return the opposite of b1, that means it would be true if b1 is false and it would return false if b1 is true.

Example of Logical Operators

```
public class LogicalOperatorDemo {
   public static void main(String args[]) {
      boolean b1 = true;
      boolean b2 = false;

      System.out.println("b1 && b2: " + (b1&&b2));
      System.out.println("b1 || b2: " + (b1||b2));
      System.out.println("!(b1 && b2): " + !(b1&&b2));
   }
}
```

Output:

```
b1 && b2: false
b1 || b2: true
!(b1 && b2): true
```

5) Comparison(Relational) operators

We have six relational operators in Java: ==, !=, >, <, >=, <=

== returns true if both the left side and right side are equal

!= returns true if left side is not equal to the right side of operator.

> returns true if left side is greater than right.

< returns true if left side is less than right side.

>= returns true if left side is greater than or equal to right side.

<= returns true if left side is less than or equal to right side.

Example of Relational operators

Note: This example is using if-else statement which is our next tutorial, if you are finding it difficult to understand then refer if-else in Java.

```
public class RelationalOperatorDemo {
   public static void main(String args[]) {
      int num1 = 10;
      int num2 = 50;
      if (num1==num2) {
                System.out.println("num1 and num2 are equal");
      }
      else{
                System.out.println("num1 and num2 are not equal");
      }

      if( num1 != num2 ){
                System.out.println("num1 and num2 are not equal");
      }
      else{
                System.out.println("num1 and num2 are equal");
      }

      if( num1 > num2 ){
                System.out.println("num1 is greater than num2");
      }
      else{
            System.out.println("num1 is not greater than num2");
    }

    if( num1 >= num2 ){
            System.out.println("num1 is greater than or equal to num2");
    }
    else{
            System.out.println("num1 is less than num2");
    }

    if( num1 < num2 ){
            System.out.println("num1 is less than num2");
    }
    else{
            System.out.println("num1 is not less than num2");
    }

    if( num1 <= num2){
            System.out.println("num1 is less than or equal to num2");
    }
    else{
            System.out.println("num1 is greater than num2");
    }
  }
}
```

Output:

```
num1 and num2 are not equal
num1 and num2 are not equal
num1 is not greater than num2
num1 is less than num2
num1 is less than num2
num1 is less than or equal to num2
```

Check out these related java programs related to relational operators:

1. Java Program to check if number is positive or negative
2. Java Program to check whether number is even or odd

6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>

num1 = 11; /* equal to 00001011*/

num2 = 22; /* equal to 00010110 */

Bitwise operator performs bit by bit processing.

num1 & num2 compares corresponding bits of num1 and num2 and generates 1 if both bits are equal, else it returns 0. In our case it would return: 2 which is 00000010 because in the binary form of num1 and num2 only second last bits are matching.

num1 | num2 compares corresponding bits of num1 and num2 and generates 1 if either bit is 1, else it returns 0. In our case it would return 31 which is 00011111

num1 ^ num2 compares corresponding bits of num1 and num2 and generates 1 if they are not equal, else it returns 0. In our example it would return 29 which is equivalent to 00011101

~num1 is a complement operator that just changes the bit from 0 to 1 and 1 to 0. In our example it would return -12 which is signed 8 bit equivalent to 11110100

num1 << 2 is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. In our case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

num1 >> 2 is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. In our case output is 2 which is equivalent to 00000010

Example of Bitwise Operators

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

    int num1 = 11;  /* 11 = 00001011 */
    int num2 = 22;  /* 22 = 00010110 */
    int result = 0;

    result = num1 & num2;
    System.out.println("num1 & num2: "+result);
    result = num1 | num2;
    System.out.println("num1 | num2: "+result);

    result = num1 ^ num2;
    System.out.println("num1 ^ num2: "+result);

    result = ~num1;
    System.out.println("~num1: "+result);

    result = num1 << 2;
    System.out.println("num1 << 2: "+result); result = num1 >> 2;
    System.out.println("num1 >> 2: "+result);
  }
}
```

Output:

```
num1 & num2: 2
num1 | num2: 31
num1 ^ num2: 29
~num1: -12
num1 << 2: 44 num1 >> 2: 2
```

Check out this program: Java Program to swap two numbers using bitwise operator

7) Ternary Operator

This operator evaluates a boolean expression and assign the value based on the result.

Syntax:

variable num1 = (expression) ? value if true : value if false

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

Example of Ternary Operator

```
public class TernaryOperatorDemo {

   public static void main(String args[]) {
      int num1, num2;
      num1 = 25;
      /* num1 is not equal to 10 that's why
           * the second value after colon is assigned
           * to the variable num2
           */
           num2 = (num1 == 10) ? 100: 200;
           System.out.println( "num2: "+num2);

           /* num1 is equal to 25 that's why
            * the first value is assigned
            * to the variable num2
            */
           num2 = (num1 == 25) ? 100: 200;
           System.out.println( "num2: "+num2);
   }
}
```

Output:

```
num2: 200
num2: 100
```

Check out these related java programs:

1. Java Program to find Largest of three numbers using Ternary Operator
2. Java Program to find the smallest of three numbers using Ternary Operator

Operator Precedence in Java

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

Unary Operators

++ – – ! ~

Multiplicative

* / %

Additive

+ –

Shift

<< >> >>>

Relational

> >= < <=

Equality
== !=
Bitwise AND
&
Bitwise XOR
^
Bitwise OR
|
Logical AND
&&
Logical OR
||
Ternary
?:
Assignment
= += -= *= /= %= > >= < <= &= ^= |=

First Example: Sum of two numbers

```
public class AddTwoNumbers {

   public static void main(String[] args) {

      int num1 = 5, num2 = 15, sum;
      sum = num1 + num2;

      System.out.println("Sum of these numbers: "+sum);
   }
}
```

Output:

```
Sum of these numbers: 20
```

Second Example: Sum of two numbers using Scanner

The scanner allows us to capture the user input so that we can get the values of both the numbers from user. The program then calculates the sum and displays it.

```
import java.util.Scanner;
public class AddTwoNumbers2 {

   public static void main(String[] args) {

      int num1, num2, sum;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter First Number: ");
      num1 = sc.nextInt();

      System.out.println("Enter Second Number: ");
      num2 = sc.nextInt();

      sc.close();
      sum = num1 + num2;
      System.out.println("Sum of these numbers: "+sum);
   }
}
```

Output:

```
Enter First Number:
121
Enter Second Number:
19
Sum of these numbers: 140
```

Example 1: Program to check whether the given number is positive or negative

In this program we have specified the value of number during declaration and the program checks whether the specified number is positive or negative. To understand this program you should have the basic knowledge of if-else-if statement in Core Java Programming.

```
public class Demo
{
   public static void main(String[] args)
   {
      int number=109;
      if(number > 0)
```

JAVA was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton. It is a simple programming language. Writing, compiling and debugging a program is easy in java. It helps to create modular programs and reusable code.

Java terminology

Before we start learning Java, lets get familiar with common java terms.

Java Virtual Machine (JVM)

This is generally referred as JVM. Before, we discuss about JVM lets see the phases of program execution. Phases are as follows: we write the program, then we compile the program and at last we run the program.

1) Writing of the program is of course done by java programmer like you and me.

2) Compilation of program is done by javac compiler, javac is the primary java compiler included in java development kit (JDK). It takes java program as input and generates java bytecode as output.

3) In third phase, JVM executes the bytecode generated by compiler. This is called program run phase.

So, now that we understood that the primary function of JVM is to execute the bytecode produced by compiler. **Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems.** That is why we call java as platform independent language.

bytecode

As discussed above, javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. The bytecode is saved in a .class file by compiler.

Java Development Kit(JDK)

While explaining JVM and bytecode, I have used the term JDK. Let's discuss about it. As the name suggests this is complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc.

In order to create, compile and run Java program you would need JDK installed on your computer.

Java Runtime Environment(JRE)

JRE is a part of JDK which means that JDK includes JRE. When you have JRE installed on your system, you can run a java program however you won't be able to compile it. JRE includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.

These are the basic java terms that confuses beginners in java. For complete java glossary refer this link: https://docs.oracle.com/javase/tutorial/information/glossary.html

Main Features of JAVA

Java is a platform independent language

Compiler(javac) converts source code (.java file) to the byte code(.class file). As mentioned above, JVM executes the bytecode produced by compiler. This byte code can run on any platform such as Windows, Linux, Mac OS etc. Which means a program that is compiled on windows can run on Linux and vice-versa. Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems. That is why we call java as platform independent language.

Java is an Object Oriented language

Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class.

4 main concepts of Object Oriented programming are:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Simple

Java is considered as one of simple language because it does not have complex features like Operator overloading, Multiple inheritance, pointers and Explicit memory allocation.

Robust Language

Robust means reliable. Java programming language is developed in a way that puts a lot of emphasis on early checking for possible errors, that's why java compiler is able to detect errors that are not easy to detect in other programming languages. The main features of java that makes it robust are garbage collection, Exception Handling and memory allocation.

Secure

We don't have pointers and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Java is distributed

Using java programming language we can create distributed applications. RMI(Remote Method Invocation) and EJB(Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one systems that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

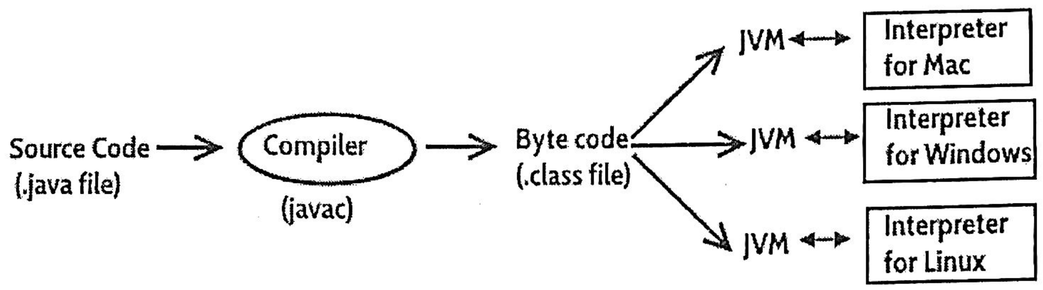
Multithreading

Java supports multithreading. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilisation of CPU.

Portable

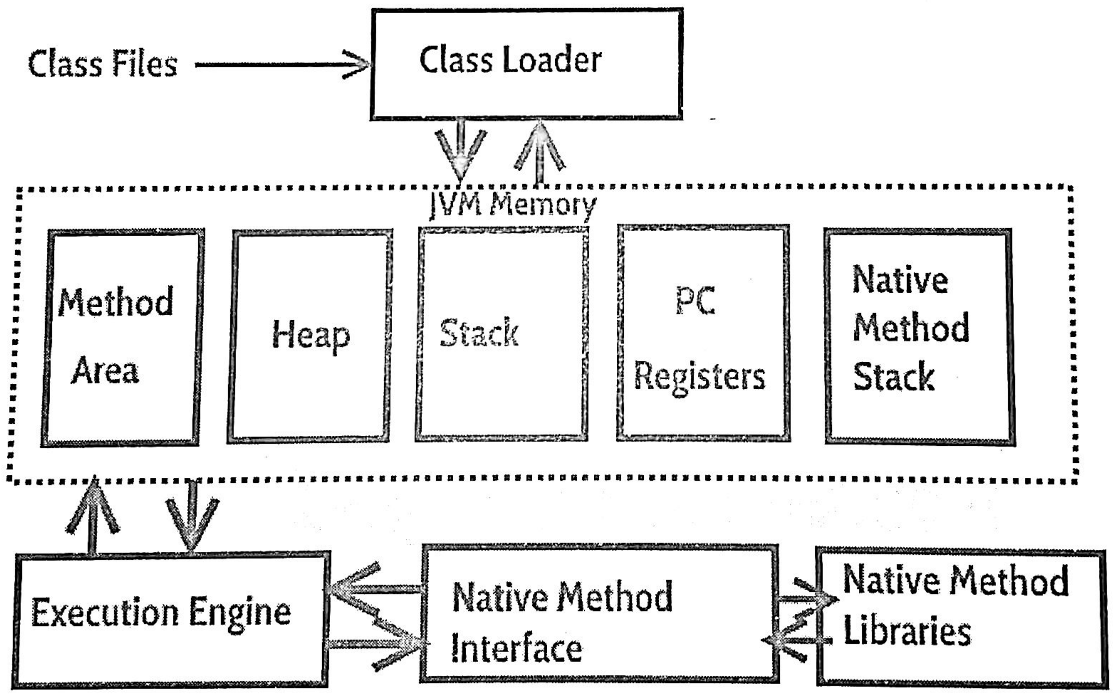
As discussed above, java code that is written on one machine can run on another machine. The platform independent byte code can be carried to any platform for execution that makes java code portable.

Java is a high level programming language. A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file). Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code.** This makes it easier for compiler as it has to generate byte code for JVM rather than different machine code for each type of machine. JVM executes the byte code generated by compiler and produce output. **JVM is the one that makes java platform independent.**
So, now we understood that the primary function of JVM is to execute the byte code produced by compiler. **Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.** Which means that the byte code generated on Windows can be run on Mac OS and vice versa. That is why we call java as platform independent language. The same thing can be seen in the diagram below:



So to summarise everything: The Java Virtual machine (JVM) is the virtual machine that runs on actual machine (your computer) and executes Java byte code. The JVM doesn't understand Java source code, that's why we need to have javac compiler that compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM. JVM makes java portable (write once, run anywhere). Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.

JVM Architecture



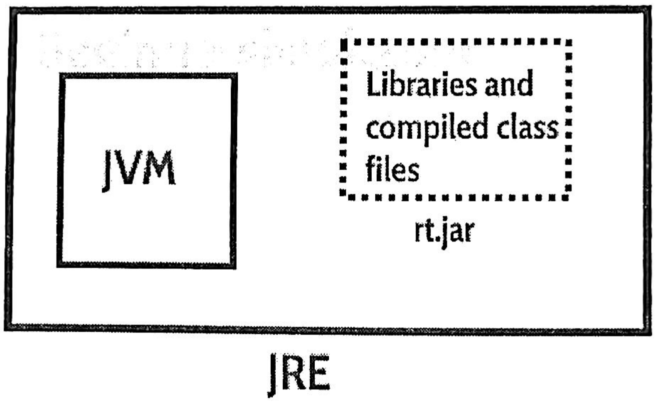
Lets see how JVM works:
Class Loader: The class loader reads the .class file and save the byte code in the **method area.**
Method Area: There is only one method area in a JVM which is shared among all the classes. This holds the class level information of each .class file.
Heap: Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.
Stack: Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.
PC Registers: This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.
Native Method stack: A native method can access the runtime data areas of the virtual machine.
Native Method interface: It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.
Garbage collection: A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.

JVM Vs JRE Vs JDK

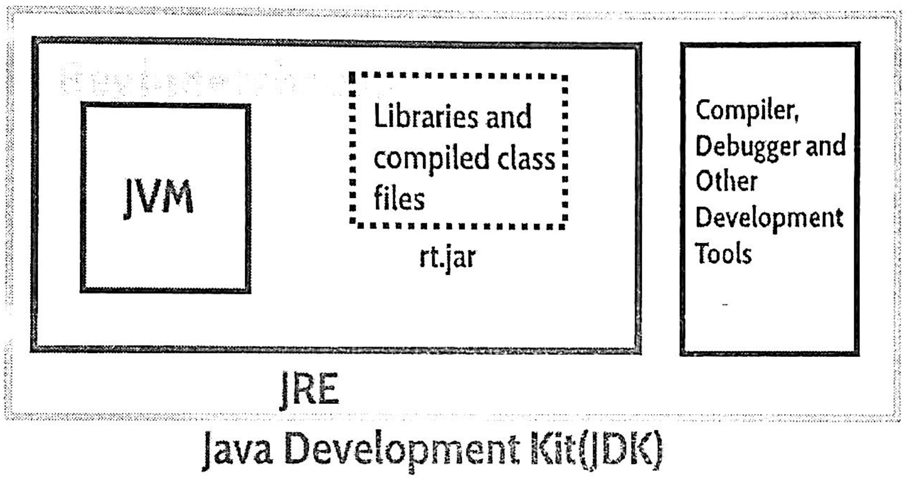
JRE: JRE is the environment within which the java virtual machine runs. JRE contains Java virtual Machine(JVM), class libraries, and other files excluding development tools such as compiler and debugger.

Which means you can run the code in JRE but you can't develop and compile the code in JRE.

JVM: As we discussed above, JVM runs the program by using class, libraries and files provided by JRE.



JDK: JDK is a superset of JRE, it contains everything that JRE has along with development tools such as compiler, debugger etc.


JVM
Libraries and compiled class files
rt.jar
Compiler, Debugger and Other Development Tools
JRE
Java Development Kit(JDK)

How to Compile and Run your First Java Program

BY CHAITANYA SINGH | FILED UNDER: LEARN JAVA

In this tutorial, we will see how to write, compile and run a java program. I will also cover java syntax, code conventions and several ways to run a java program.

Simple Java Program:

```
public class FirstJavaProgram {
  public static void main(String[] args){
    System.out.println("This is my first program in java");
  }//End of main
}//End of FirstJavaProgram Class
```

Output: This is my first program in java

How to compile and run the above program

Prerequisite: You need to have java installed on your system. You can get the java from here.

Step 1: Open a text editor, like Notepad on windows and TextEdit on Mac. Copy the above program and paste it in the text editor.

You can also use IDE like Eclipse to run the java program but we will cover that part later in the coming tutorials. For the sake of simplicity, I will only use text editor and command prompt (or terminal) for this tutorial

Step 2: Save the file as **FirstJavaProgram.java**. You may be wondering why we have named the file as FirstJavaProgram, the thing is that we should always name the file same as the public class name. In our program, the public class name is FirstJavaProgram, that's why our file name should be **FirstJavaProgram.java.**

Step 3: In this step, we will compile the program. For this, open **command prompt (cmd) on Windows,** if you are **Mac OS then open Terminal.**

To compile the program, type the following command and hit enter.

```
javac FirstJavaProgram.java
```

You may get this error when you try to compile the program: "**javac' is not recognized as an internal or external command, operable program or batch file**". This error occurs when the java path is not set in your system

If you get this error then you first need to set the path before compilation.

Set Path in Windows:

Open command prompt (cmd), go to the place where you have installed java on your system and locate the bin directory, copy the complete path and write it in the command like this.

```
set path=C:\Program Files\Java\jdk1.8.0_121\bin
```

Note: Your jdk version may be different. Since I have java version 1.8.0_121 installed on my system, I mentioned the same while setting up the path.

Set Path in Mac OS X

Open Terminal, type the following command and hit return.

```
export JAVA_HOME=/Library/Java/Home
```

Type the following command on terminal to confirm the path.

```
echo $JAVA_HOME
```

That's it.

The steps above are for setting up the path temporary which means when you close the command prompt or terminal, the path settings will be lost and you will have to set the path again next time you use it. I will share the permanent path setup guide in the coming tutorial.

Step 4: After compilation the .java file gets translated into the .class file(byte code). Now we can run the program. To run the program, type the following command and hit enter:

```
java FirstJavaProgram
```

Note that you should not append the .java extension to the file name while running the program.

Closer look to the First Java Program

Now that we have understood how to run a java program, let have a closer look at the program we have written above.

```
public class FirstJavaProgram {
```

This is the first line of our java program. Every java application must have at least one class definition that consists of class keyword followed by class name. When I say keyword, it means that it should not be changed, we should use it as it is. However the class name can be anything.

I have made the class public by using public access modifier, I will cover access modifier in a separate post, all you need to know now that a java file can have any number of classes but it can have only one public class and the file name should be same as public class name.

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

This is our next line in the program, lets break it down to understand it:

public: This makes the main method public that means that we can call the method from outside the class.

static: We do not need to create object for static methods to run. They can run itself.

void: It does not return anything.

main: It is the method name. This is the entry point method from which the JVM can run your program.

(String[] args): Used for command line arguments that are passed as strings. We will cover that in a separate post.

```
System.out.println("This is my first program in java");
```

This method prints the contents inside the double quotes into the console and inserts a newline after.

Variables in Java

BY CHAITANYA SINGH | FILED UNDER: LEARN JAVA

A variable is a name which is associated with a value that can be changed. For example when I write int i=10; here variable name is **i** which is associated with value 10, int is a data type that represents that this variable can hold integer values. We will cover the data types in the next tutorial. In this tutorial, we will discuss about variables.

How to Declare a variable in Java

To declare a variable follow this syntax:

```
data_type variable_name = value;
```

here value is optional because in java, you can declare the variable first and then later assign the value to it.

For example: Here num is a variable and int is a data type. We will discuss the data type in next tutorial so do not worry too much about it, just understand that int data type allows this num variable to hold integer values. You can read data types here but I would recommend you to finish reading this guide before proceeding to the next one.

```
int num;
```

Similarly we can assign the values to the variables while declaring them, like this:

```
char ch = 'A';
int number = 100;
```

or we can do it like this:

```
char ch;
int number;
...
ch = 'A';
number  = 100;
```

Variables naming convention in java

1) Variables naming cannot contain white spaces, for example: int num ber = 100; is invalid because the variable name has space in it.

2) Variable name can begin with special characters such as $ and _

3) As per the java coding standards the variable name should begin with a lower case letter, for example int number; For lengthy variables names that has more than one words do it like this: int smallNumber; int bigNumber; (start the second word with capital letter).

4) Variable names are case sensitive in Java.

Types of Variables in Java

There are **three types of variables** in Java.

1) Local variable 2) Static (or class) variable 3) Instance variable

Static (or class) Variable

Static variables are also known as class variable because they are associated with the class and common for all the instances of class. For example, If I create three objects of a class and access this static variable, it would be common for all, the changes made to the variable using one of the object would reflect when you access it through other objects.

Example of static variable

```
public class StaticVarExample {
  public static String myClassVar="class or static variable";

  public static void main(String args[]){
    StaticVarExample obj = new StaticVarExample();
    StaticVarExample obj2 = new StaticVarExample();
    StaticVarExample obj3 = new StaticVarExample();

    //All three will display "class or static variable"
    System.out.println(obj.myClassVar);
    System.out.println(obj2.myClassVar);
    System.out.println(obj3.myClassVar);

    //changing the value of static variable using obj2
    obj2.myClassVar = "Changed Text";

    //All three will display "Changed Text"
    System.out.println(obj.myClassVar);
    System.out.println(obj2.myClassVar);
    System.out.println(obj3.myClassVar);
  }
}
```

Output:

```
class or static variable
class or static variable
class or static variable
Changed Text
Changed Text
Changed Text
```

As you can see all three statements displayed the same output irrespective of the instance through which it is being accessed. That's is why we can access the static variables without using the objects like this:

```
System.out.println(myClassVar);
```

Do note that only static variables can be accessed like this. This doesn't apply for instance and local variables.

Instance variable

Each instance(objects) of class has its own copy of instance variable. Unlike static variable, instance variables have their own separate copy of instance variable. We have changed the instance variable value using object obj2 in the following program and when we displayed the variable using all three objects, only the obj2 value got changed, others remain unchanged. This shows that they have their own copy of instance variable.

Example of Instance variable

```
public class InstanceVarExample {
  String myInstanceVar="instance variable";

  public static void main(String args[]){
            InstanceVarExample obj = new InstanceVarExample();
            InstanceVarExample obj2 = new InstanceVarExample();
            InstanceVarExample obj3 = new InstanceVarExample();

            System.out.println(obj.myInstanceVar);
            System.out.println(obj2.myInstanceVar);
            System.out.println(obj3.myInstanceVar);


            obj2.myInstanceVar = "Changed Text";


            System.out.println(obj.myInstanceVar);
            System.out.println(obj2.myInstanceVar);
            System.out.println(obj3.myInstanceVar);
  }
}
```

Output:

```
instance variable
instance variable
instance variable
instance variable
Changed Text
instance variable
```
Local Variable
These variables are declared inside method of the class. Their scope is limited to the method which means that You can't change their values and access them outside of the method.
In this example, I have declared the instance variable with the same name as local variable, this is to demonstrate the scope of local variables.
Example of Local variable
```
public class VariableExample {
  // instance variable
  public String myVar="instance variable";

  public void myMethod(){
            // local variable
            String myVar = "Inside Method";
            System.out.println(myVar);
  }
  public static void main(String args[]){
    // Creating object
    VariableExample obj = new VariableExample();

    /* We are calling the method, that changes the
     * value of myVar. We are displaying myVar again after
     * the method call, to demonstrate that the local
     * variable scope is limited to the method itself.
     */
    System.out.println("Calling Method");
    obj.myMethod();
    System.out.println(obj.myVar);
  }
}
```
Output:
```
Calling Method
Inside Method
instance variable
```
If I hadn't declared the instance variable and only declared the local variable inside method then the statement System.out.println(obj.myVar); would have thrown compilation error. As you cannot change and access local variables outside the method.
Data Types in Java
BY CHAITANYA SINGH | FILED UNDER: LEARN JAVA
Data type defines the values that a variable can take, for example if a variable has int data type, it can only take integer values. In java we have two categories of data type: 1) Primitive data types 2) Non-primitive data types – Arrays and Strings are non-primitive data types, we will discuss them later in the coming tutorials. Here we will discuss primitive data types and literals in Java.
Java is a statically typed language. A language is statically typed, if the data type of a variable is known at compile time. This means that you must specify the type of the variable (Declare the variable) before you can use it.
In the last tutorial about Java Variables, we learned how to declare a variable, lets recall it:
```
int num;
```
So in order to use the variable num in our program, we must declare it first as shown above. It is a good programming practice to declare all the variables (that you are going to use) in the beginning of the program.
1) Primitive data types
In Java, we have eight primitive data types: boolean, char, byte, short, int, long, float and double. Java developers included these data types to maintain the portability of java as the size of these primitive data types do not change from one operating system to another.
byte, **short**, **int** and **long** data types are used for storing whole numbers.
float and **double** are used for fractional numbers.
char is used for storing characters(letters).
boolean data type is used for variables that holds either true or false.
byte:
This can hold whole number between -128 and 127. Mostly used to save memory and when you are certain that the numbers would be in the limit specified by byte data type.
Default size of this data type: 1 byte.
Default value: 0
Example:

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

                byte num;

                num = 113;
                System.out.println(num);
    }
}
```
Output:
```
113
```
Try the same program by assigning value assigning 150 value to variable num, you would get **type mismatch** error because the value 150 is out of the range of byte data type. The range of byte as I mentioned above is -128 to 127.

short:
This is greater than byte in terms of size and less than integer. Its range is -32,768 to 32767.
Default size of this data type: 2 byte
short num = 45678;

int: Used when short is not large enough to hold the number, it has a wider range: -2,147,483,648 to 2,147,483,647
Default size: 4 byte
Default value: 0
Example:
```
class JavaExample {
    public static void main(String[] args) {

                short num;

                num = 150;
                System.out.println(num);
    }
}
```
Output:
```
150
```
The byte data type couldn't hold the value 150 but a short data type can because it has a wider range.

long:
Used when int is not large enough to hold the value, it has wider range than int data type, ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
size: 8 bytes
Default value: 0
Example:
```
class JavaExample {
    public static void main(String[] args) {

                long num = -12332252626L;
                System.out.println(num);
    }
}
```
Output:
```
-12332252626
```
double: Sufficient for holding 15 decimal digits
size: 8 bytes
Example:
```
class JavaExample {
    public static void main(String[] args) {

                double num = -42937737.9d;
                System.out.println(num);
    }
}
```
Output:
```
-4.29377379E7
```
float: Sufficient for holding 6 to 7 decimal digits
size: 4 bytes
```
class JavaExample {
    public static void main(String[] args) {

                float num = 19.98f;
                System.out.println(num);
    }
}
```
Output:
```
19.98
```
boolean: holds either true of false.
```
class JavaExample {
   public static void main(String[] args) {

              boolean b = false;
              System.out.println(b);
   }
}
```
Output:
```
false
```
char: holds characters.
size: 2 bytes
```
class JavaExample {
   public static void main(String[] args) {

              char ch = 'Z';
              System.out.println(ch);
   }
}
```
Output:
```
Z
```
Literals in Java
A literal is a fixed value that we assign to a variable in a Program.
```
int num=10;
```
Here value 10 is a Integer literal.
```
char ch = 'A';
```
Here A is a char literal
Integer Literal
Integer literals are assigned to the variables of data type byte, short, int and long.
```
byte b = 100;
short s = 200;
int num = 13313131;
long l = 928389283L;
```
Float Literals
Used for data type float and double.
```
double num1 = 22.4;
float num2 = 22.4f;
```
Note: Always suffix float value with the "f" else compiler will consider it as double.
Char and String Literal
Used for char and String type.
```
char ch = 'Z';
String str = "BeginnersBook";
```
Operators in Java
BY CHAITANYA SINGH | FILED UNDER: LEARN JAVA
An operator is a character that **represents an action**, for example + is an arithmetic operator that represents addition.
Types of Operator in Java
1) Basic Arithmetic Operators
2) Assignment Operators
3) Auto-increment and Auto-decrement Operators
4) Logical Operators
5) Comparison (relational) operators
6) Bitwise Operators
7) Ternary Operator
1) Basic Arithmetic Operators
Basic arithmetic operators are: +, -, *, /, %
+ is for addition.
– is for subtraction.
* is for multiplication.
/ is for division.
% is for modulo.
Note: Modulo operator returns remainder, for example 10 % 5 would return 0
Example of Arithmetic Operators
```
public class ArithmeticOperatorDemo {
```