**Java Types**

**1 byte = 8 bit**

Primitive Types:

1. Byte →1 byte (Stores from -128 to 127)
2. Short → 2 bytes (Stores from -32,768 to 32,767)
3. int → 4 bytes (Stores from -2,147,483,648 to 2,147,483,647)
4. Long → 8 bytes (Stores from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
5. Float → 4 bytes (Stores fractional, 6 to 7 decimal)
6. Double → 8 bytes (Stores fractional, 15 decimal)
7. Boolean → 1 bit (true|false, 1|0)
8. Char → 2 bytes

Non-Primitive Types:

1. String
2. Arrays
3. Classes

Primitive vs Non-primitive

| Primitive | Non-Primitive |
| --- | --- |
| Already defined by Java | Defined by programmer except String |
| Always has a value | Can be null |
| Cannot be used to call methods | Can be used to call methods |
| Size depends on the type | All have same size |

Notes:

1. Double is more precise than float
2. String refers to an object

**Pass by Value vs pass by reference**

Pass by value means the actual value is passed. The variable that is passed is copied to somewhere so the original stays untouched while the parameter goes through changes.

Pass by reference means a number which indicates the object of the variable is sent. Unlike passing by value, the parameter still points to the same memory location so as the parameter goes through changes, actual value also changes.

Java is always **pass by value** not pass by reference.

**JVM (Java Virtual Machine)**

It is a specification that provides a runtime environment in which java bytecode can be executed. Acts as a run-time engine to run Java applications. In other programming languages, the compiler produces machine code for a particular system. However, Java compiler produces code for a Virtual Machine known as Java Virtual Machine.

Operations:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

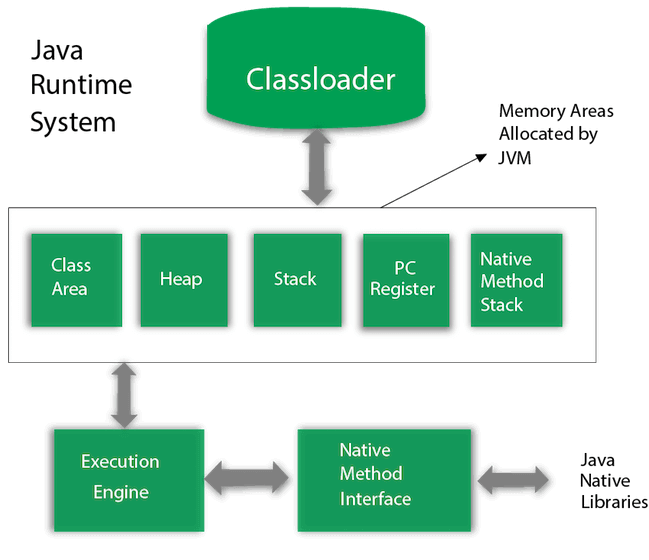
Gives definitions to:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

How it works?



Architecture:



1. Classloader: (Loading,Linking and initialization) Loads class files. 3 types:
   1. Bootstrap: Loads rt.jar file which contains all java standard class files.
   2. Extension: It loades the jar files located inside the *$JAVA\_HOME/jre/lib/ext* directory.
   3. System/Application: It loads the class files from classpath. By default, classpath is set to the current directory.
2. Class Area: Stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
3. Heap: Runtime data area in which objects are allocated.
4. Stack: Stores frames. It holds local variables and partial results, and plays a part in method invocation and return. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
5. Program Counter Register: PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.
6. Native Method Stack: It contains all the native methods used in the application.
7. Execution Engine: It has the following components:
   1. A virtual processor
   2. Interpreter**:** Read bytecode stream then execute the instructions
   3. Just in time (JIT) compiler: “Compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU. Used to reduce wasted time.
   4. Garbage Collector: It destroys un-referenced objects.
8. Java Native Libraries: Framework which provides an interface to communicate with another application written in another language.

In order to write and execute a software program, you need the following

**1) Editor** – To type your program into, a notepad could be used for this

**2) Compiler** – high language program into native machine code

**3) Linker** – To combine different program files reference in your main program together.

**4) Loader** – To load the files from your secondary storage device into RAM for execution. The loading is automatically done when you execute your code.

**5) Execution** – Actual execution of the code which is handled by your OS & processor.

Garbage Collector:

*Garbage Collection* would deal with finding and deleting the garbage from the memory. However, in reality, *Garbage Collection* tracks each and every object available in the JVM heap space, and removes the unused ones.

C, C++ programmer has to delete manually and can forget to do so. This results in dangling pointers and unreachable objects and after a while there is no space for the developer in the heap.

But in Java this process is made automatically by Garbage collector. It keeps track of the objects and **marks** the unreachable objects. Then **sweeps** the marked places which is the deletion process. ( the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects).