Contents

[Programming Language Levels 2](#_Toc161608725)

[Compiler: 3](#_Toc161608726)

[Interpreter: 3](#_Toc161608727)

[Java execution 4](#_Toc161608728)

[What is platform independent. 4](#_Toc161608729)

[Following are the notable features of Java: 5](#_Toc161608730)

[Object Oriented 5](#_Toc161608731)

[Platform Independent 5](#_Toc161608732)

[Simple 5](#_Toc161608733)

[Secure 5](#_Toc161608734)

[Architecture-neutral 5](#_Toc161608735)

[Portable 5](#_Toc161608736)

[Robust 5](#_Toc161608737)

[Multithreaded 5](#_Toc161608738)

[Interpreted 6](#_Toc161608739)

[High Performance 6](#_Toc161608740)

[Distributed 6](#_Toc161608741)

[Dynamic 6](#_Toc161608742)

[Java installed structure 6](#_Toc161608743)

[What is JVM? 7](#_Toc161608744)

[Libraries 7](#_Toc161608745)

[What is JRE? 7](#_Toc161608746)

[4. What is JDK? 8](#_Toc161608747)

[JVM Architecture 9](#_Toc161608748)

[Class Loader 9](#_Toc161608749)

[JVM Execution Engine 10](#_Toc161608750)

[Interpreter 10](#_Toc161608751)

[JIT Compiler 10](#_Toc161608752)

[Native Method Interface 11](#_Toc161608753)

[Java Memory management 11](#_Toc161608754)

[Java Memory Structure: 11](#_Toc161608755)

[Heap : 12](#_Toc161608756)

[Heap Size 13](#_Toc161608757)

[Heap Internal 13](#_Toc161608758)

[JVM Stacks: 15](#_Toc161608759)

[Native method Stacks: 15](#_Toc161608760)

[Program counter (PC) registers: 16](#_Toc161608761)

[Method area: 16](#_Toc161608762)

[Cache Memory 17](#_Toc161608763)

# Programming Language Levels

there are three levels of programming language.

1. First one is High level
2. Second one Low level or machine level
3. And third one is Assembly level

The most of the computer programs which was written by a human is called as high level programming language. high level programing language is the language that programmers can easily understand. It contains normal English words which human can understand easily.

High level programming language is programmer friendly language , it means humans can easily understand.

For example Java , Python are called as high level programming language.

On the other hand the computer can not understand these high level language, it means computers cannot understand basic English. Computers can only understand 1s and 0s. So the code or language which was written using this 1 and 0 is called as low level language.

The language which computer understands is low level language which contains 0 and 1s.

SO low level language is machine friendly language it means it can easily understandable by computer, and very tough to understand by humans. Because we cannot read 1s and 0s.

There is one more level called as Assembly level. This assemble language is like more than low level and less than high level.

Assemble language can use symbols , numbers etc.

what ever we write in java is High level language. And computer cannot understand this high level language so, we need to convert high level language to computer understandable low level or machine language. It means 1s and 0s.

For converting this high level to low level we use compilers and interpreters. SO what are Compilers and interpreters.

Compiler and interpreter are software programs that convert a high-level language into a machine language or 0's and 1's binary form that a computer can understand and perform tasks as per the program's instructions.

## Compiler:

A [compiler](https://www.javatpoint.com/compiler-tutorial) is a software program that convert a source code to machine code. Or high level language to low level language.

A compiler **converts complete source code** into machine code at once. It means what ever we write in code , the whole code is converted into machine code not line by line.

For example in C program is a high level language and when we compile the C program the whole C program is converted in to machine level language.

## Interpreter:

An [interpreter](https://www.javatpoint.com/interpreter-pattern) is also a software program that converts a source code into a machine language.

So interpreter also converts high-level programming language into machine language but it will convert  line-by-line not whole code at once. It means when we write code and interpreter will read the code line by line and covnerts into machine code or 1s and 0s.

For example Javascript is high level programming language and by using interpreter this javascript is converted to low level language. Java script is executed line by line.

So the basic difference between compiler and interpreter is compiler will convert whle code into lowlevel at once. But interpreter will convert line by line.

One more feature of compiler is Compiler can also convert highlevel code to intermediate level.

Next we will discuss how this compiler and interpreter is used in java for executing the code.

As we discussed that High level code will be converted into low level code. As java is a high level language we need to convert it into low level language for execution. but Java is not an one step conversion it means it will not directly convert from high level to low level.

Java has a two step execution process.

In first step the while Java source code is converted to byte code and also called as class files. And in second step these byte code is converted into machine code.

Byte code is like intermediate code. It is same like assembly language but not assembly language

now how this compiler and interpreter are integrated in these execution steps.

# Java execution

In java the source code is passed through compiler and compiler will covert this source code to byte code. It means the whole java code is compiled with compiler and converts into byte code or .class files.

For Ex Room. Java is a java file and when this Room.java files compiled the compiler will generate the Room.class file

Interpreter will convert the byte code into machine code. So the class file which was generated by compiler is input for this interpreter and interpreter will interpret the bytecode line by line and executes to get the desired output.

First compiler will convert the java source code into byte code and interpreter will convert this byte code into machine language.

# What is platform independent.

Platform independent means running the code on any platform or operating systems. It means when we write code in one operating system the same code should be executed on different platforms or operating systems with out any issues.

Java is platform independent and why is it platorm independent?

In Java , First compiler will compile the code and next interpreter will execute the code. These are two level process.

First compiler will generate the class files and these class files can be transferred to any other platforms and can be executed in other platforms.

For example java code written in windows and compiled in windows, and the class files generated in windows can be transferred to other platforms like mac or linux and can be executed in those platforms.

To execute these class files in other platform those platform should contain java installed in those platforms

So the only condition to execute these .class files is the other OS which we are trying to execute the class files should have the Java installed.

For example if we consider the C program, when C program is compiled in windows it will generate the .exe and this exe file can be executed on only windows platform and it cannot be executed on other platforms. SO C is platform dependent language.

# Following are the notable features of Java:

## 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 the Virtual Machine (JVM) on whichever platform it is being run on.

## Simple

Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it 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.

## Architecture-neutral

Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Java is platform-independent because it uses a virtual machine. The Java programming language and all APIs are compiled into bytecodes. Bytecodes are effectively platform-independent. The virtual machine takes care of the differences between the bytecodes for the different platforms. We can compile code in windows and generate bytecode and use same bytecode in linux to execute. We have jvm which can understand the bytecode irrespective of OS.

## Portable

Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. The 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 perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

## 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.

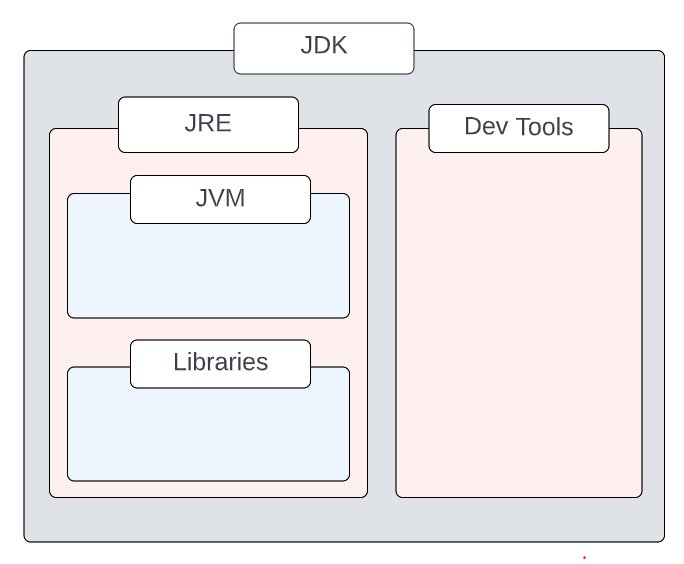
## Distributed

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 an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

# Java installed structure



JRE = JVM + libraries to run Java application

JDK = JRE + tools to develop Java Application

## What is JVM?

Java Virtual machine (JVM) is the virtual machine that runs the Java bytecodes. You get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.

JVM is responsible for converting bytecode to machine understandable code.

Java Virtual Machine, or JVM, loads, verifies and executes Java bytecode. JVM acts as an interpreter.

## Libraries

Libraries are just a collection of classes which are predefined in JAVA. And these classes we get when we install the Java.

So there might be some functionalities which programmer will use. And these functionalities are already coded in Java and provided as libraries for us. So when we install, we will also get these libraries and can include in our java code as an depdendency. So that there is no need of writing the same functionality in our java code .

For example ADD SUB these are common functionalities which most of the programmers will use and these are already available in java and provided as libraries and we can just import those into our code and use it.

Examples for java libraries are

java. lang, java. util, and java. math

## What is JRE?

The Java Runtime Environment (JRE) is a software package which bundles the libraries (jars) and the Java Virtual Machine, and other components to run applications written in the Java. JVM is just a part of JRE distributions.

To execute any Java application, you need JRE installed in the machine. It’s minimum requirement to execute Java applications on any machine.

So to execute any java program, we need JRE to be installed in the machine in which code is executed. With out JRE we can not run t he program.

JRE bundles the following components –

* DLL files used by the Java HotSpot Client Virtual Machine.
* DLL files used by the Java HotSpot Server Virtual Machine.
* Code libraries, property settings, and resource files used by the Java runtime environment. e.g. rt.jar and charsets.jar.
* Java extension files such as localedata.jar.
* Contains files used for security management. These include the security policy (java.policy) and security properties (java.security) files.
* Jar files containing support classes for applets.
* Contains TrueType font files for use by the platform.

# 4. What is JDK?

JDK is a superset of JRE. JDK contains everything that JRE has along with development tools for developing, debugging, and monitoring Java applications. You need JDK when you need to develop Java applications.

Development tools are used for developing, debugging Java applications. So to write the java code, and generate the executables, we need this developing and debugging tools.

For example Compiler , jar creation are some of the tools available in development tools.

Compiler is used to compile the code and once code is written and compiles we will generate the jar so that it can be transferred for deployment.

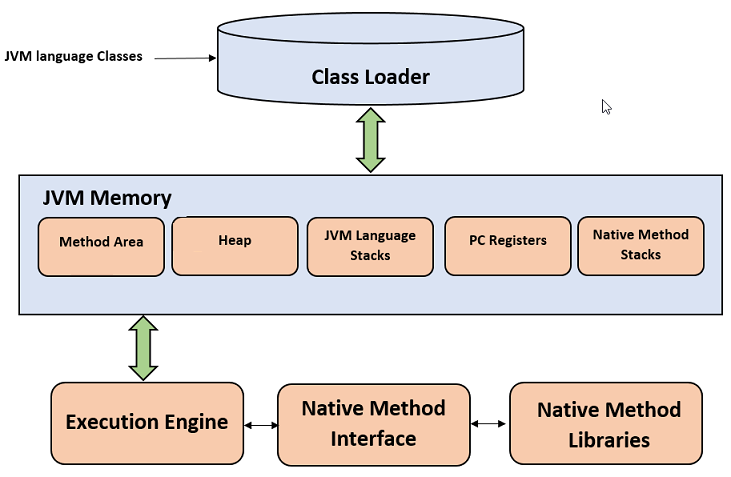
SO For developing the java program we need JDK. And for running the java program we need only JRE.

Same as JREs, JDKs are also platform dependent. So take care when you download the JDK package for your machine.

Few important components shipped with JDKs are as follows:

* appletviewer – this tool can be used to run and debug Java applets without a web browser
* apt – the annotation-processing tool
* extcheck – a utility that detects JAR file conflicts
* javadoc – the documentation generator, which automatically generates documentation from source code comments
* jar – the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files
* jarsigner – the jar signing and verification tool
* javap – the class file disassembler
* javaws – the Java Web Start launcher for JNLP applications
* JConsole – Java Monitoring and Management Console
* jhat – Java Heap Analysis Tool
* jrunscript – Java command-line script shell
* jstack – utility that prints Java stack traces of Java threads
* keytool – tool for manipulating the keystore
* policytool – the policy creation and management tool
* xjc – Part of the Java API for XML Binding (JAXB) API. It accepts an XML schema and generates Java classes

# JVM Architecture



## Class Loader

The class loader is a subsystem used for loading class files. It performs three major functions i.e. class loading, linking, and initialization.

When java program is executed the java class file will be loaded and all the dependencies present in that class files will be loaded. So JVM will try to load all the class files and there dependencies.

The dependencies which were defined in a class file will not present at one place. They might be from different modules. Some dependencies might be the one which user was created and some might be the default java dependencies. So Based on the dependencies, loading has again three phases.

For example IF JVM tries to load some abc.class dependency

##### Loading

* To load classes, JVM has 3 kind of class loaders. Bootstrap, extension and application class loader.
* When loading a class file, JVM finds out a dependency for some arbitrary class XYZ.class.
* First bootstrap class loader tries to find the class. It scans the rt.jar file in JRE lib folder.
* If class is not found then extension class loader searches the class file in inside jre\lib\ext folder.
* Again if class is not found then application classloader searches all the Jar files and classes in CLASSPATH environment variable of system.
* If class is found by any loader then class is loaded by class loader; else ClassNotFoundException is thrown.

##### Linking

After class is loaded by the classloader, linking is performed. A bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get a verification error. It also performs the memory allocation to static variables and methods found in the class.

##### Initialization

This is the final phase of class loading, here all static variable will be assigned with the original values and the static blocks will be executed.

## JVM Execution Engine

All code assigned to JVM is executed by an execution engine. The execution engine reads the byte code and executes one by one. It uses two inbuilt interpreter and JIT compiler to convert the bytecode to machine code and execute it.

As we have discussed that the class loader will load the classes, now the loaded byte code will be executed in execution engine. This execution engine will communicate with all the memory modules and execute the code.

Execution engine uses interpreter and Jit compiler to execute the code.

### Interpreter

A JVM interpreter pretty much converts each byte-code instruction to corresponding native instruction by looking up a predefined JVM-instruction to machine instruction mapping. It directly executes the bytecode and does not perform any optimization.

### JIT Compiler

To improve performance, JIT compilers interact with the JVM at runtime and compile appropriate bytecode sequences into native machine code. Typically, the JIT compiler takes a block of code (not one statement at a time as interpreter), optimize the code and then translates it to an optimized machine code.

The JIT compiler is enabled by default. You can disable the JIT compiler, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or workaround JIT compilation problems.

Interpreter will read the byte code and convert to machine code and executes in a line by line manner. Here as it is executed line by line, the same and common code will be executed multiple times and performance of the system will be reduced.

Jit Compiler- jit compiler will consider the block of byte code and convert it into machine code and execute the code. It will not consider line by line as interpreter does and because of this performance of JIT compiler is better than interpreter.

For example we have a for loop in byte code, to execute this loop JIT compiler will first check the byte code and will compile that into machine code at once in which system can understand. As first time it converted into machine understandable language for next time or for every loop it will not again check the byte code and convert . from second loop on wards it will directly execute the code.

But incase of interpreter it is not like this. As it is executed line by line, for every loop it will consider one line in bytecode loop and converts into machine code and executes and same thing happens for every line in loop and for all the loops. Because of this performance will be reduced. When compare to JIT compiler.

In java Jit compiler is enabled by default. We can also disable the JIT compiler. IF JIT compiler is disabled then jVM will use the interpreter to execute the code. But disabling jit compiler is not recommended.

## Native Method Interface

Native methods are basically the code which is not in JAVA. So If you want to run the native methods in JVM then we can use JNI and can run those in JVM.

JNI is an interface that allows Java to interact with code written in another language.

A native method in Java is a method whose implementation is written in other languages such as c and c++.

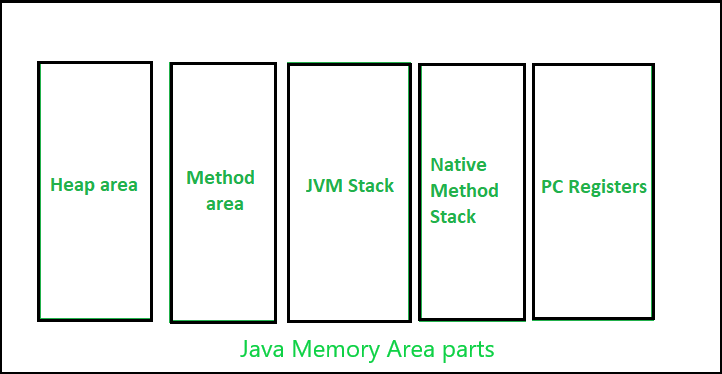
The ‘native’ keyword is used before a method to indicate that it is implemented in other language.

JNI is an interface that allows Java to interact with code written in another language. Motivation for JNI is code reusability and performance. WIth JNI, you can reuse existing/legacy code with Java (mostly C/C++). In terms of performance, native code used to be up to 20 times faster than Java, when running in interpreted mode. Modern JIT compilers (HotSpot) make this a moot point.

# Java Memory management

## Java Memory Structure:

The memory area inside JVM is divided into multiple parts to store specific parts of application data.





In every programming language, the memory is a vital resource and is also scarce in nature. Hence it’s essential that the memory is managed thoroughly without any leaks. Allocation and deallocation of memory is a critical task and requires a lot of care and consideration. However in Java, unlike other programming language, the JVM and to be specific Garbage Collector has the role of managing memory allocation so that the programmer needs not to. Whereas in other programming languages such as C the programmer has direct access to the memory who allocates memory in his code, thereby creating a lot of scope for leaks.

JVM defines various run time data area which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of thread are created during instantiation and destroyed when the thread exits.

## Heap :

Heap stores all objects that are created during application execution.

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create an object, it’s always created in the Heap space. Any object created in the heap space has global access and can be referenced from anywhere of the application.

* It is a shared runtime data area and stores the actual object in a memory. Heap space is instantiated during the virtual machine startup.
* This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.
* Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space
* There will be one and only one heap for a running JVM process.

Scanner sc = new Scanner(System.in);

The above statement creates the object of Scanner class and this object memory is allocated in heap whereas the reference ‘sc’ gets pushed to the stack and in stack sc has address of object in heap.

Create one pictorial

### Heap Size

The default maximum heap size for the Java heap is determined by the amount of physical memory available on the system.

The default initial heap size is 1/64 of physical memory.

The default maximum heap size is half of the physical memory up to a physical memory size of 192 megabytes

Other wise Default max heap size is 1/4th of the physical memory.

Below java options are used to set the java heap size

-Xms : To set an initial java heap size

-Xmx : To set maximum java heap size

-Xss : To set the Java thread stack size

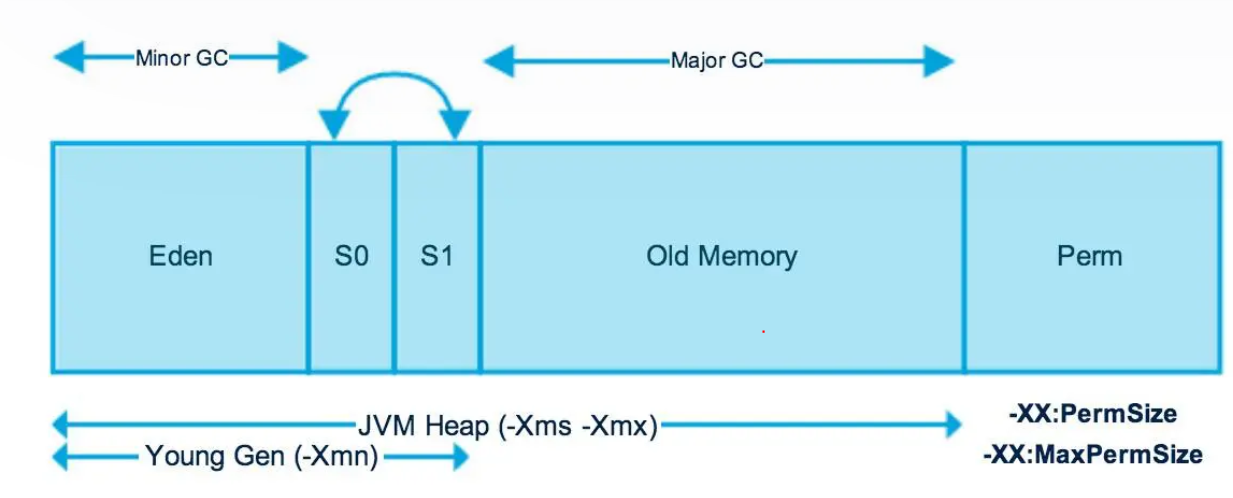
-Xmn : For setting the size of young generation, rest of the space goes for old generation.

-Xms64m or -Xms64M // 64 megabytes

-Xmx1g or -Xmx1G // 1 gigabytes

java -Xms=1M -XmX=2M "Class Name"

## Heap Internal



This heap area is divided into two parts

1. Young generation
2. Old generation

#### Young generation

In this image you can see the first three segments are called as young generation. SO when ever a new object is created it is first saved in this young generation. And in this young generation first it is saved in EDEN space.

When the eden space is full a process called as minor GC is triggered and all the usable objects will be moved to S0 and eden space is freed up. And new objects will be created.

Now again when eden space is filled up, the objects from S0 will be moved to S1 and only the objects which are useful are moved and others are deleted. And objects from edenspace are moved to S0 again.

And this process continues.

* This is reserved for containing newly-allocated objects
* Young Gen includes three parts — Eden Memory and two Survivor Memory spaces (S0, S1)
* Most of the newly-created objects goes Eden space.
* When Eden space is filled with objects, Minor GC (a.k.a. Young Collection) is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation

#### Old generation

Now the objects are which are old and are staying in young generation from long time will be moved to old generation. SO from S1 the objects will be moved to old generation.

As we said minor gc will trigeered in young generation. So all the objects which are survived in GCs will be moved to old generation.

Now when this old generation is full then major GC is triggered and will try to free up some space.

* This is reserved for containing long lived objects that could survive after many rounds of Minor GC
* When Old Gen space is full, Major GC (a.k.a. Old Collection) is performed (usually takes longer time)

Another area called as Permgen is available in heap space but from java 8 this permgen removed from heap and moved as a separate method area or non heap.

## JVM Stacks:

Java Stack memory is used for the execution of a thread. They contain method-specific values that are short-lived and references to other objects in the heap that is getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as the method ends, the block becomes unused and becomes available for the next method. Stack memory size is very less compared to Heap memory.

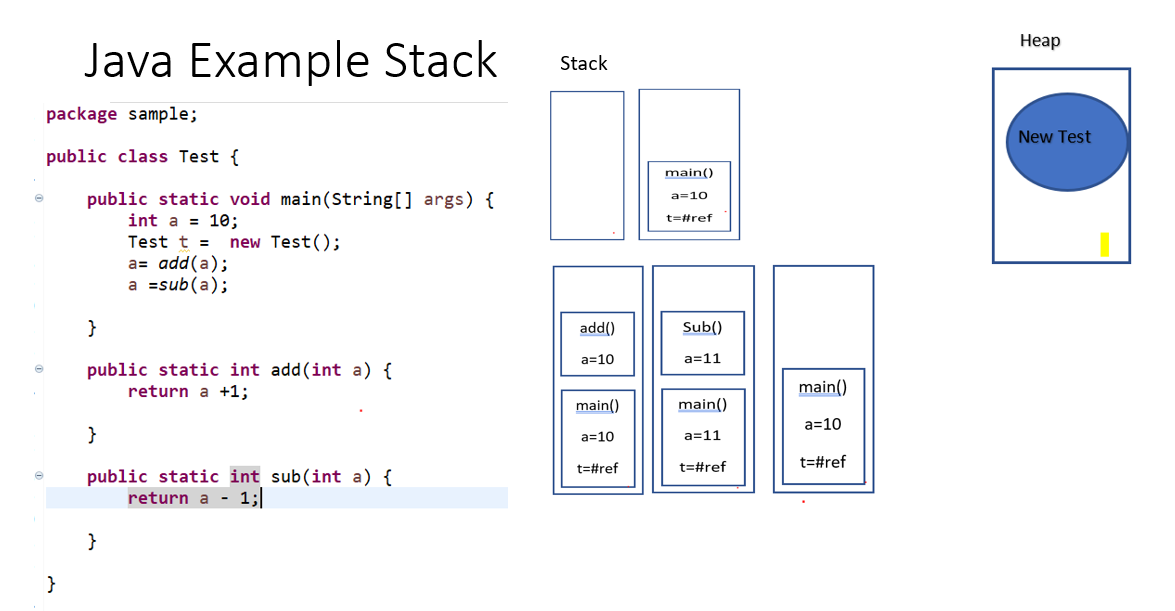
* A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
* Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created.
* The memory for stack needs not to be contiguous.
* If we don't specify a size for the stacks, the JVM will create one with a default size
* Stack memory only contains local primitive variables and reference variables to objects in heap space

For windows, the default stack size is 320k for 32bit and 1024k for 64bit,

As mentioned earlier, stack memory is limited in size and cannot be enlarged or shrunk once created. Therefore, if we use all of the stack memory, there will be no space left for upcoming method calls, and we will get the StackOverflowError.

As below we can define the stack size.

java -Xss=512M "Class Name".



## Native method Stacks:

Methods written in languages other than Java programming languages such as C, C++ are called native methods.

When a thread calls a Java method, a new frame is created and is being pushed onto the Java stack. When a thread calls a native method, the thread switches from Java stack to native method stack. The parameters(if any) are pushed on native method stack. If a native method calls back a Java method, the thread leaves the native method stack and enters in Java stack again.

## Program counter (PC) registers:

Program Counter or PC register holds the address of JVM instruction currently being executed for any particular thread

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

## Method area:

Method Area stores the information related to classes such as class name, static variables in the class, final variables, fields in the class, method information, etc. For example, when we use getters, setters, or other methods related to business logic, they reside in Method Area. Method Area is shared among threads. In the Hotspot VM, Method Area corresponds to Permanent Generation.

The JVM extracts the information from the binary data that was given by the class loader and stores it in the method area.

JVM stores the following information.

— The fully qualified name of the type

— The fully qualified name of the type’s immediate superclass

— Whether type is a class or an interface

— The type’s modifiers

Inside the java class file and Java virtual machine, type names are always stored as fully qualified names.

— Runtime constant pool

— Field Information (field’s name, field’s type, field’s modifiers)

— Method Information (method’s name, method’s return type, the number and types of method’s parameters, method’s modifier, method’s bytecode)

— Class Variables — Class variables are shared among all instances of a class. These variables are associated with the class, not with the instances of the class. So they are logically part of the class data in the method area.

— A reference to Class Loader

— A reference to Class class — An instance of the class java.lang.Class is created by the JVM for every type it loads. Basically, it is a corresponding representation of the type loaded in the method area. You can use this java.lang.Class instance to access the metadata of the type.

The method area is shared among all JVM threads. It is created on JVM startup. The method area is just the JVM specification. The one who follows JVM specification must provide the implementation of the method area.

**permgenspace**

Permgenspace memory is same like the method area. Before java 8 this is available in heap memory. And from java 8 this memory is separated from heap memory.

A java.lang.OutOfMemoryError: PermGen Space is a runtime error in Java which occurs when the permanent generation (PermGen) area in memory is exhausted. The PermGen area of the Java heap is used to store metadata such as class declarations, methods and object arrays. Therefore, the PermGen size requirements depend on the number of classes and methods as well as their size.

The default maximum memory size for 32-bit JVM is 64 MB and 82 MB for the 64-bit version.

However, we can change the default size with the JVM options:

* -XX:PermSize=[size] is the initial or minimum size of the PermGen space
* -XX:MaxPermSize=[size] is the maximum size

In order to overcome out of memory error for PermGen space, the PermGen space was replaced by

Metaspace or method area both are same.

The metaspace is part of non-heap memory and it auto increases its size (up to what the underlying OS provides). The garbage collector now automatically triggers the cleaning of the dead classes once the metadata usage reaches its maximum size.

Its size can be changed using -XX:PermSize and -XX:MaxPermSize.

## Cache Memory

<https://poonamparhar.github.io/codecache_is_full/>

JVM Code Cache is an area where JVM stores its bytecode compiled into native code

The just-in-time (JIT) compiler is the biggest consumer of the code cache area

The JVM JIT generates compiled code and stores that in a memory area called CodeCache

 The default maximum size of the CodeCache on most of the platforms is 48M. If any application needs to compile large number of methods resulting in huge amount of compiled code then the CodeCache may become full.

When it becomes full, the compiler is disabled to stop any further compilations of methods, and a message like the following gets logged:

Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full. Compiler has been disabled.

Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the code cache size using -XX:ReservedCodeCacheSize=

Code Cache [0xffffffff77400000, 0xffffffff7a390000, 0xffffffff7a400000) total\_blobs=11659 nmethods=10690 adapters=882 free\_code\_cache=909Kb largest\_free\_block=502656

When this situation occurs, the JVM may invoke sweeping and flushing of this space to make some room available in the CodeCache. There is a JVM option UseCodeCacheFlushing that can be used to control the flushing of the Codecache. With this option enabled JVM invokes an emergency flushing that discards older half of the compiled code(nmethods) to make space available in the CodeCache. In addition, it disables the compiler until the available free space becomes more than the configured CodeCacheMinimumFreeSpace. The default value of CodeCacheMinimumFreeSpace option is 500K.

* This includes Code Cache
* Stores compiled code (i.e. native code) generated by JIT compiler, JVM internal structures, loaded profiler agent code and data, etc.
* When Code Cache exceeds a threshold, it gets flushed (and objects are not relocated by the GC)

The Java Virtual Machine (JVM) generates native code and stores it in a memory area called the codecache. The JVM generates native code for a variety of reasons, including for the dynamically generated interpreter loop, Java Native Interface (JNI) stubs, and for Java methods that are compiled into native code by the just-in-time (JIT) compiler. The JIT is by far the biggest user of the codecache. This appendix describes techniques for reducing the JIT compiler's codecache usage while still maintaining good performance.

## Static variables:

Before java 8, static variables are stored in the Permanent Generation area of heap.

When we create a static variable or method it is stored in the special area on heap: PermGen(Permanent Generation). Starting from Java 8 the PermGen became - Metaspace. After java 8 static variables are stored in heap.

# Heap and stack

package com.journaldev.test;

public class Memory {

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

} // Line 9

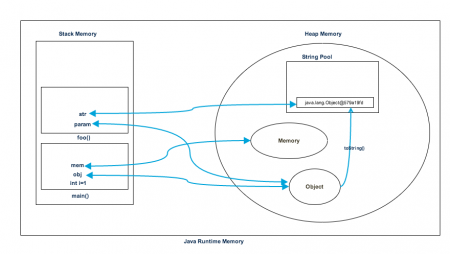
private void foo(Object param) { // Line 6

String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8

}

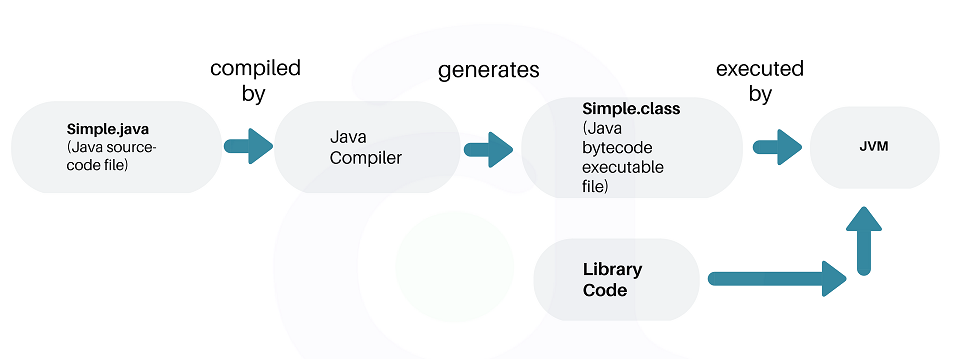


1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at [Java Garbage Collection](https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-management-in-java).
5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
6. We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.
7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

# Compiling and running a java program is very easy after JDK installation.

Following are the steps −

* Open a command prompt window and go to the directory where you saved the java program (MyFirstJavaProgram.java). Assume it's C:\.
* 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).
* Now, type ' java MyFirstJavaProgram ' to run your program.
* You will be able to see the result printed on the window.



In short, if you are a Java application developer who writes code, you will need JDK installed in your machine. But, if you only want to run applications built in Java, you only need JRE installed into your machine