Zoho Schools for Graduate Studies

Notes

Resources required to run a process ?

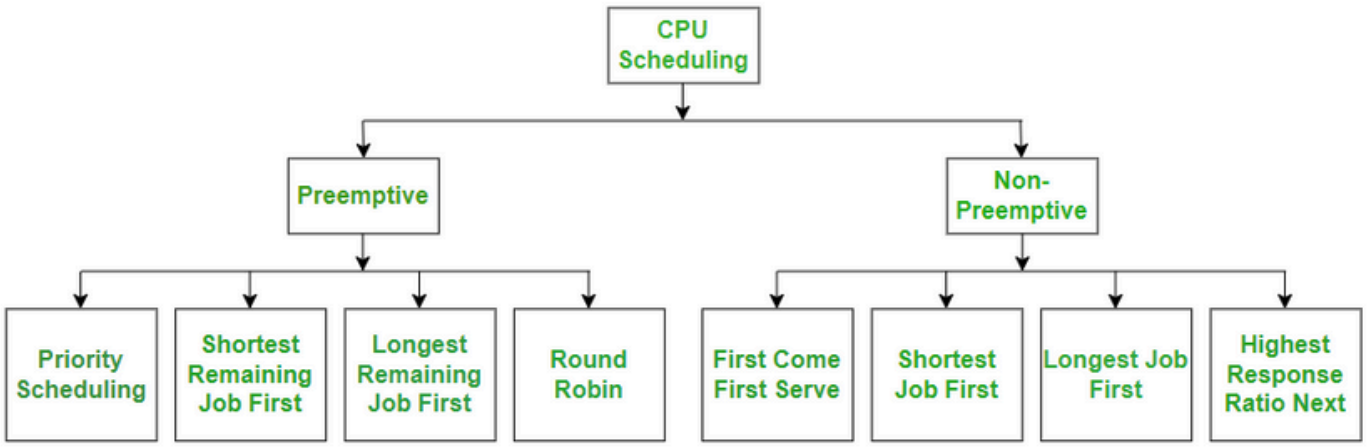
- CPU
- A Primary memory
- I/O processor

The **CPU** (Central Processing Unit) is the core component of a computer system responsible for **executing instructions** of a program by performing basic *arithmetic, logical, control*, and *input/output operations.*

CPU scheduling:

CPU scheduling is a process used by the operating system to decide which task or process gets to use the CPU at a particular time.
The main function of CPU scheduling is to ensure that whenever the CPU remains **idle**, the OS always keeps it **busy** by selecting one of the process in **Ready Queue .**

Types of CPU Scheduling:



Preemptive Scheduling:

Preemptive scheduling is used when a process switches from **running state to ready state.**
*(OS Process with high priority running at background which are called as **Daemon Process**)*

Non-Preemptive Scheduling:

Non-Preemptive scheduling is used when a process terminates , or when a process switches from **running state to waiting state.**
(*User Process with low priority will be terminated sometimes)*

CPU Scheduling Algorithm:

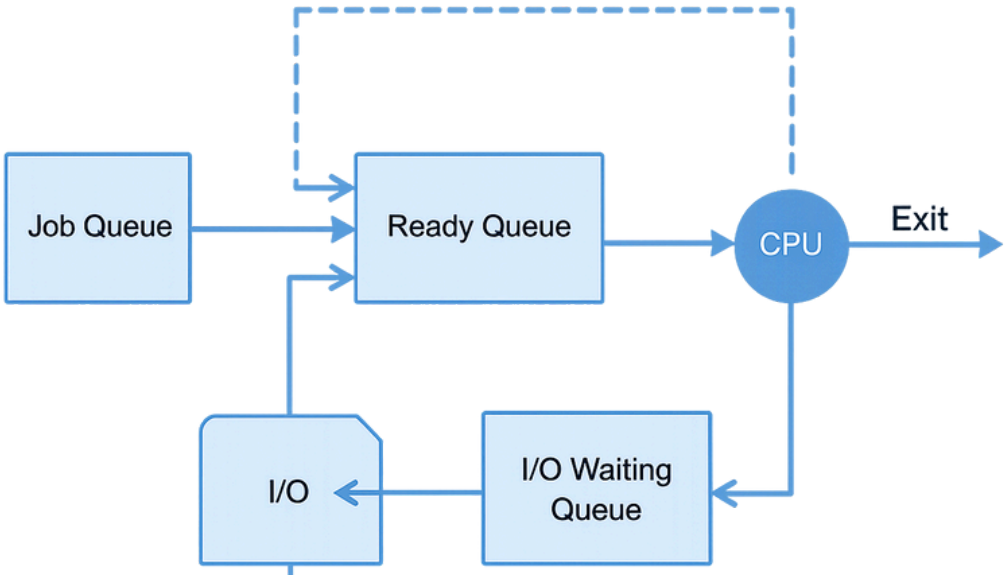
- **CPU Utilization:** The main purpose of any CPU algorithm is to keep the CPU as busy as possible.
- **Throughput:** The number of processes performed and completed during each unit from queue will be stored. This is called Throughput.
- **Turn Round Time:** The time elapsed from the time of process delivery to the time of completion.
- **Waiting Time :** The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time in the ready queue.
- **Response Time :** The time taken in the submission of the application process until the first response is issued. This measure is called response time.

Terminologies Used in CPU Scheduling:

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.
  (Turn Around Time = Completion Time – Arrival Time)
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.
  (Waiting Time = Turn Around Time – Burst Time)

Process Control Blocks *(PCB)* :

The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state. When the state of a process is changed, its PCB is **unlinked** from its current queue and moved to its new state queue.

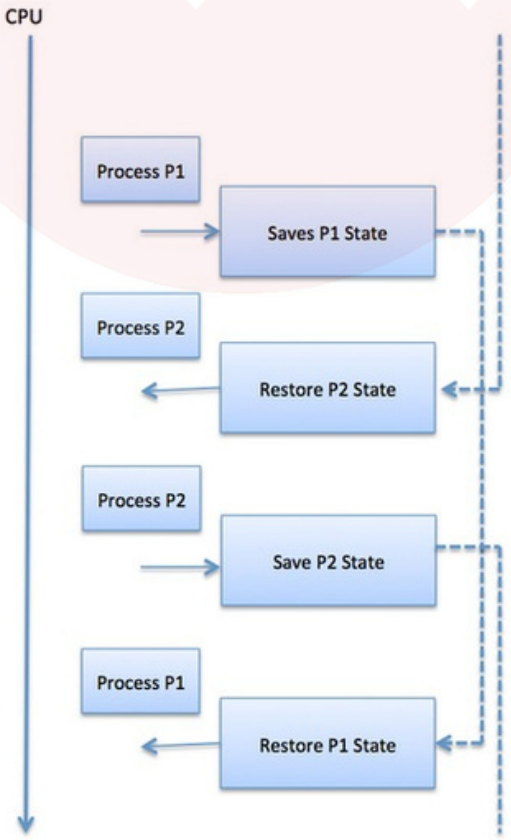


- **Job queue** - This queue keeps all the processes in the system.
- **Ready queue** - This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** - The processes which are blocked due to unavailability of an I/O device constitute this queue.

The OS can use different policies to manage each queue *(FIFO, Round Robin, Priority, etc.)*.

Context Switching :

A context switching is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be **resumed from the same point** at a later time.



(Context Switching Diagram)

Using this technique, a context switcher enables **multiple processes** to share a **single CPU.** When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block.

When the process is switched, the following information is stored for later use.

- **Program Counter -** stores the address of the next instruction to be executed**.**
- **Scheduling information -** Details the process's priority and other data the scheduler uses to determine the next process to run.
- **Base and limit register value -** Define the starting address and maximum allowed address of a process's memory space for *memory protection*
- **Currently used register -** Hold the current data and execution state of a process for *resuming* operations
- **I/O State information -** : Tracks the I/O devices assigned to a process
- **Accounting information -** Records resource usage, like CPU time and execution history, for monitoring and management.

Round - Robin Scheduling :

Round Robin Scheduling is a method used by operating systems to manage the execution time of multiple processes that are competing for CPU attention.
The primary goal of this scheduling method is to ensure that all processes are given an **equal opportunity** to execute,

Example:

01 Step

Time quantum = 2ms

at t = 0

Process	Arrival Time	Burst Time
P_1	0 ms	4 ms
P_2	0 ms	5 ms
P_3	0 ms	3 ms

Ready Queue : | P_1 | P_2 | P_3 | | |

02 Step

at t = 2

Gantt chart

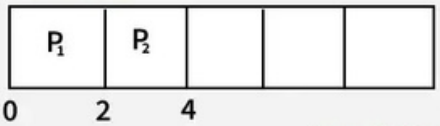
Remaining time left(P_1) : 2 ms

Ready Queue : | ~~P_1~~ | P_2 | P_3 | P_1 |

03 Step

at t = 4

Gantt chart



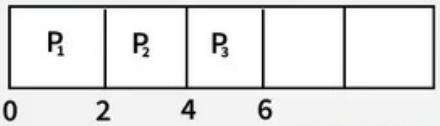
remaining time left(P_2) : 3 ms

Ready Queue : | ~~P_1~~ | ~~P_2~~ | P_3 | P_1 | P_2 |

04 Step

at t = 6

Gantt chart

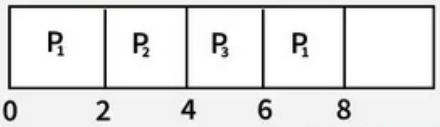


Remaining time left(P_3) : 1 ms

Ready Queue : | ~~P_1~~ | ~~P_2~~ | ~~P_3~~ | P_1 | P_2 | P_3 |

05 Step — at t = 8

Gantt chart

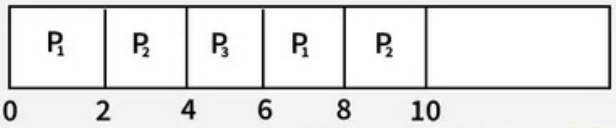


Remaining time left(P_1) : 0 ms

Ready Queue :

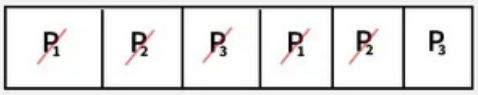
06 Step — at t = 10

Gantt chart



Remaining time left(P_2) : 1 ms

Ready Queue :

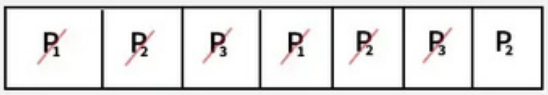


07 Step — at t = 11

Gantt chart

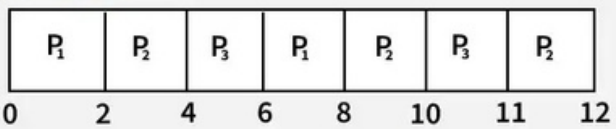
Remaining time left(P_3) : 0 ms

Ready Queue :



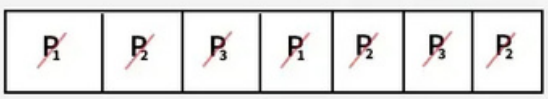
08 Step — at t = 12

Gantt chart



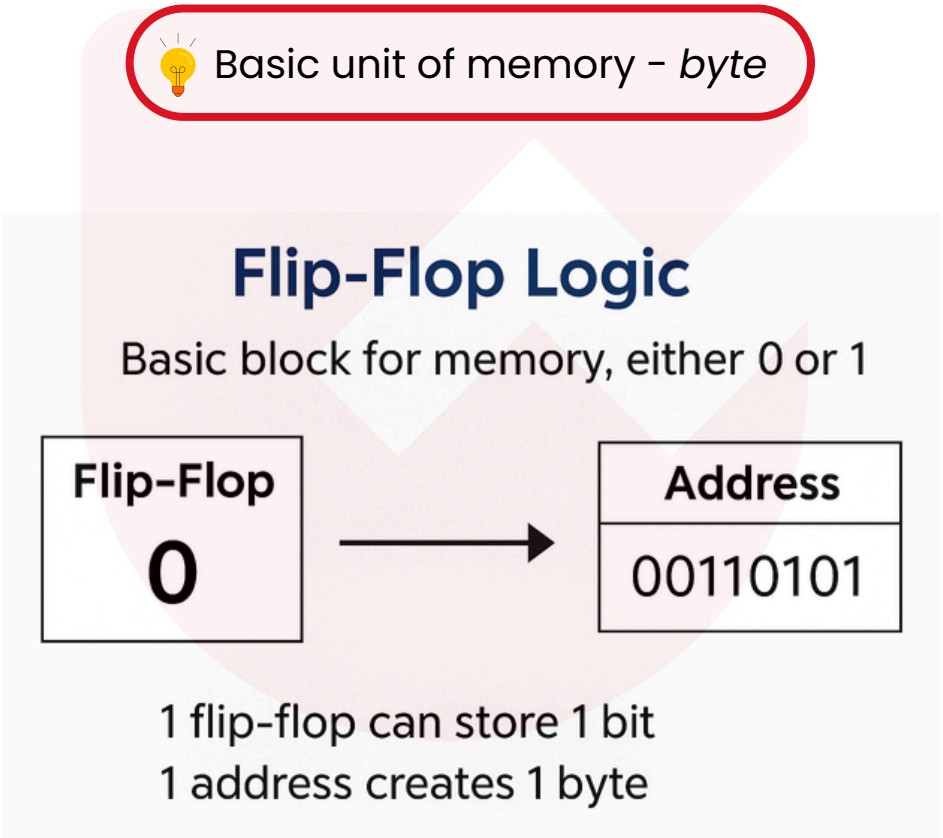
Remaining time left(P_2) : 0 ms

Ready Queue :



Flip - Flop Logic: The basic block for memory

Single flip-flop stores only **one bit**, **groupings** of flip-flops can store **larger** units of data. For instance, a register, which is a small, fast storage unit within the CPU, is made up of multiple flip-flops. To store 8 bits of data, a register would require 8 flip-flops.



Introduction to Java

Java was developed by **SUN MICROSYSTEMS** in **1991** later acquired by **Oracle Corp**. Developed by **James Gosling** and invented in **1995** writing, compiling and debugging.

Java is ,

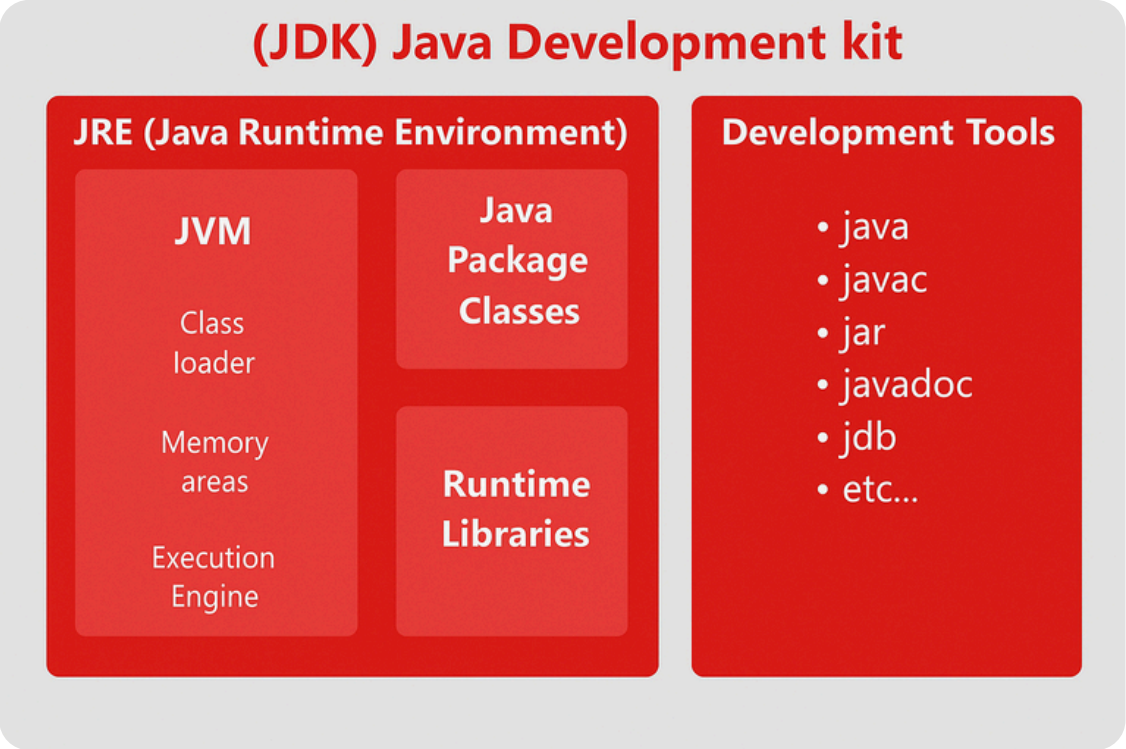
- **Object Oriented -** Organizes code around real-world objects combining data and behavior.
- **Platform Independent -** Runs compiled code (bytecode) on any system with a Java Virtual Machine (JVM).
- **Robust -** Handles errors and unexpected conditions gracefully
- **Portable -** Java's bytecode runs on diverse platforms without modification.
- **Mulithreated -** Executes multiple instructions(piece of code) concurrently for efficiency.
- **Distributed -** Builds applications that run across multiple network-connected computers.
- **Secured -** Java doesn't provide explicit pointers for direct memory manipulation

 Write once , Run anywhere.

Java **D**evelopment **K**it (***JDK***):

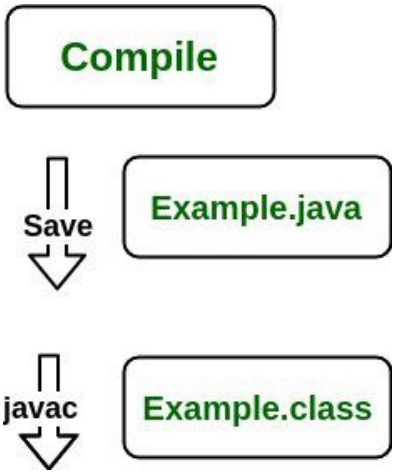
The **JDK** is a software development kit that provides tools to develop and run Java applications. It includes two main components:

- Development Tools (to provide an environment to develop your java programs)
- JRE (to execute your java program)



- JDK is only for development (it is not needed for running Java programs)
- JDK is platform-dependent (different version for windows, Linux, macOS)

Working of JDK:



Java **R**untime **E**nvironment *(**JRE**)*:

The **JRE** is an installation package that provides an environment to **only run**(not develop) the Java program onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.
It includes components like **JVM , Java Packages & Classes, Runtime libraries.**

Working of JRE:

When you run a Java program, the following steps occur:

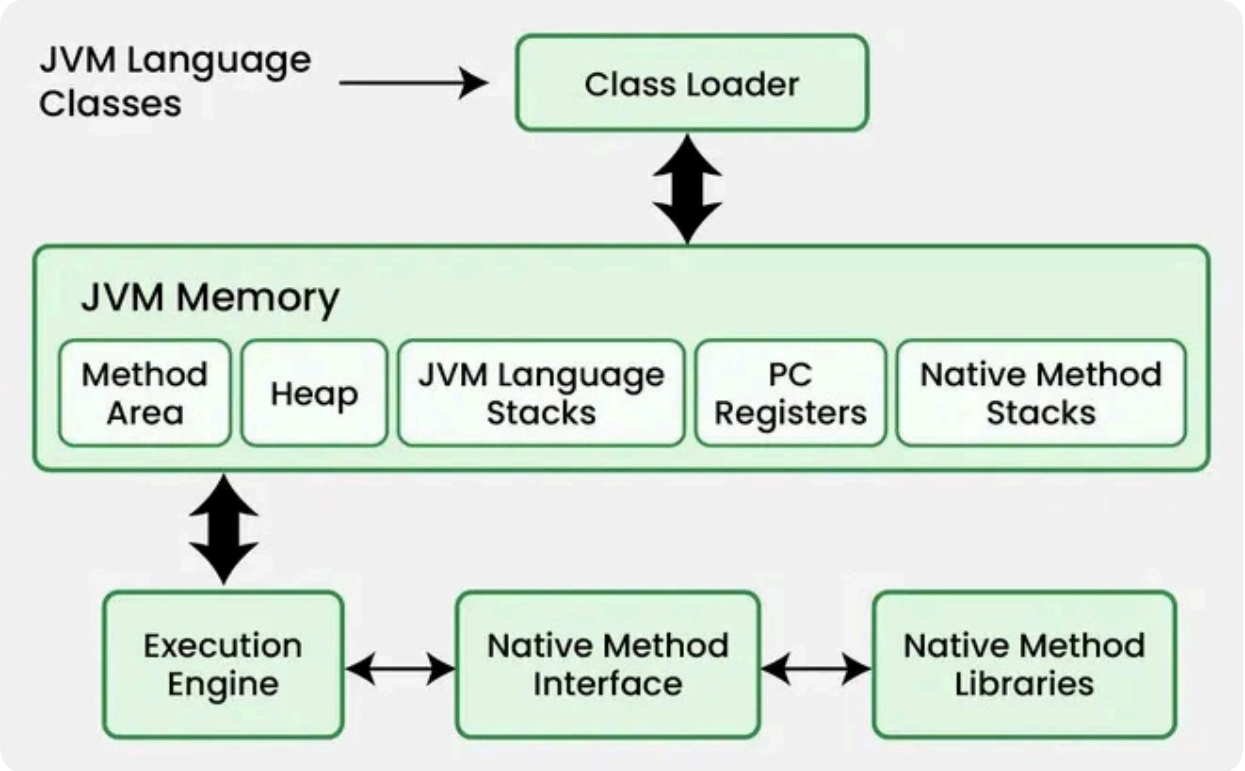
- **Class Loader:** The JRE's class loader loads the .class file containing the bytecode into memory.
- **Bytecode Verifier:** JRE includes a bytecode verifier to ensure security before execution
- **Interpreter:** JVM uses an interpreter + JIT compiler to execute bytecode for optimal performance
- **Execution:** The program executes, making calls to the underlying hardware and system resources as needed.

Java **V**irtual **M**achine *(**JVM**)*:

The **JVM** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an **interpreter.**

- JVM is platform -dependent
- While JVM includes an interpreter, modern implementations primarily use **JIT** compilation for faster execution

JVM Architecture:



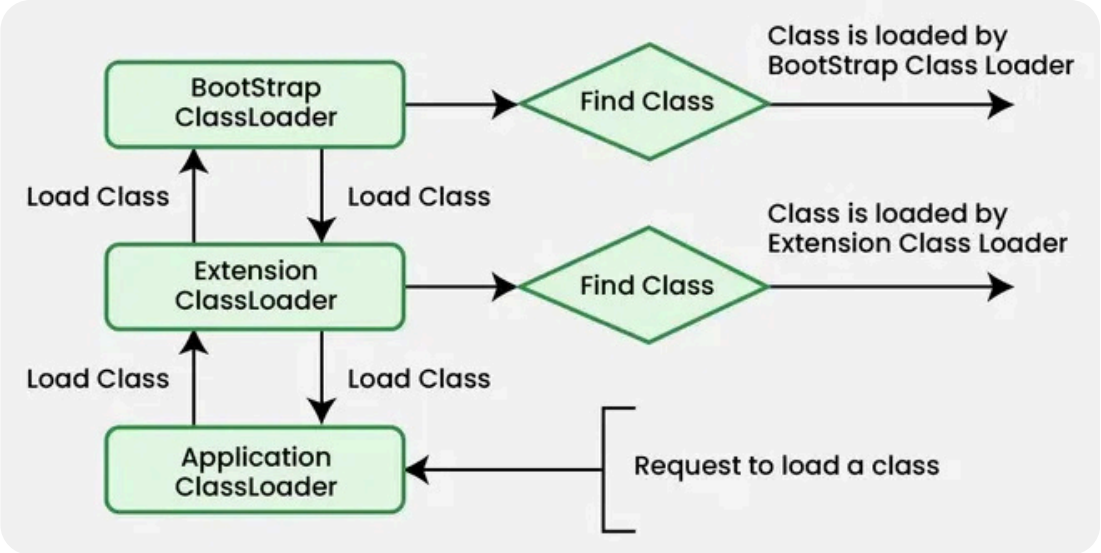
Core components of JVM:

1. Class Loader Subsystem:

- **Loading:** The Class loader reads the "*.class*" file, generate the corresponding binary data and save it in the method area. After loading the ".class" file, JVM creates an **object** of type Class to represent this file in the *heap memory.* To get this object reference we can use **getClass()** method of Object class.
- **Linking:** Performs verification, preparation, and (optionally) resolution.
  1. **Verification:** It checks whether this file is properly formatted and generated by a valid compiler or not.
  2. **Preparation:** JVM allocates memory for class static variables
  3. **Resolution:** It is the process of replacing symbolic references from the type with direct references
- **Initialization:** This is executed from top to bottom in a class and from parent to child in the class hierarchy.

2.Class Loaders:

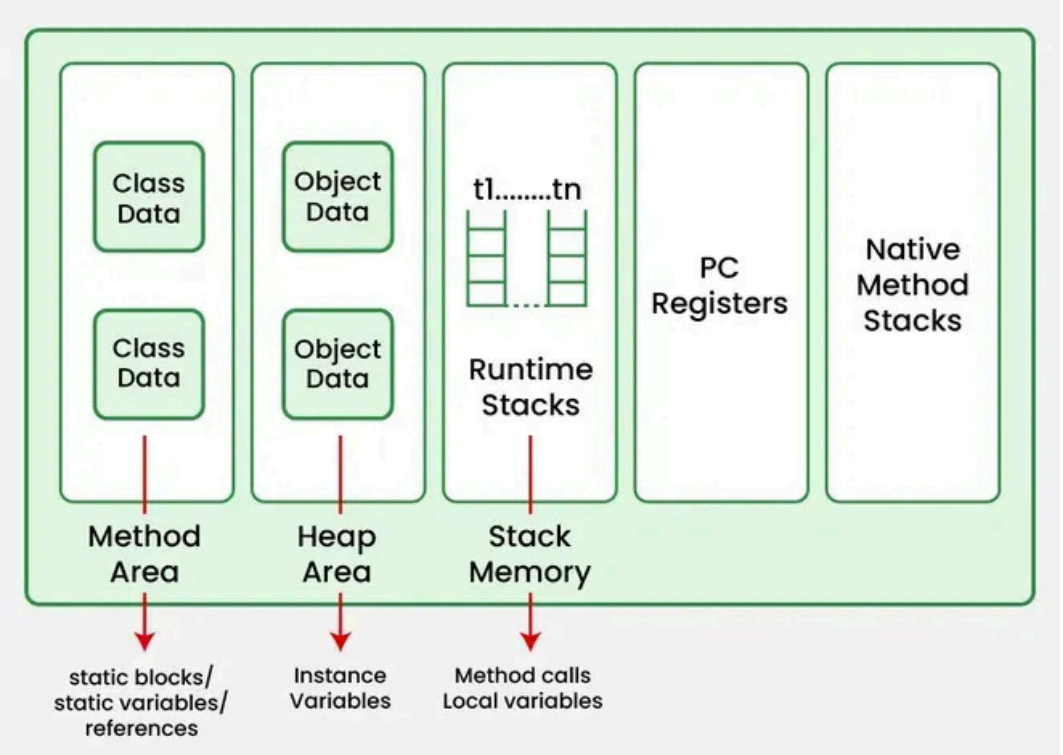
- **Bootstrap class loader:** Loads core java API classes present in the "JAVA_HOME/lib" directory.
- **Extension class loader:** Loads classes from the JAVA_HOME/jre/lib/ext directory or any directory specified by the java.ext.dirs system property.
- **System/Application class loader:** Loads classes from the application classpath, which is specified by the java.class.path environment variable.



(Class Loaders Diagram)

3. JVM Memory Areas

- **Method area:** In the method area, all **class level information** like class name, immediate parent class name, methods and variables information etc. are stored, including static variables.
- **Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource *(But it will grow)*
- **Stack area:** JVM creates one run-time stack which is stored here. Every block of this stack is called ***activation record/stack frame*** which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM.
- **PC Registers:** Store ***address*** of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
- **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



(JVM Memory Areas Diagram)

4. Execution Engine :

Execution engine executes the **".class"** (***bytecode***). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

- **Interpreter:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- **Just-In-Time Compiler(JIT):** It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required.
- **Garbage Collector:** It destroys un-referenced objects.

5. Java Native Interface (JNI) :

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

6. Native Method Libraries :

These are collections of native libraries required for executing native methods.