JVM – JRE Understanding

Terminologies...

JVM

Java Virtual Machine is an Abstract computing machine that enables the computer to run a Java program. It is a specification that provides runtime environment in which java bytecode can be executed.

JVM has an instruction set and manipulates various memory areas at run time

JRE

Java Runtime Environment is a software package that contains what is required to run a Java program. Contains Java class libraries, Java virtual machines, and other tools that the programs need to execute.

JRE is the implementation of JVM

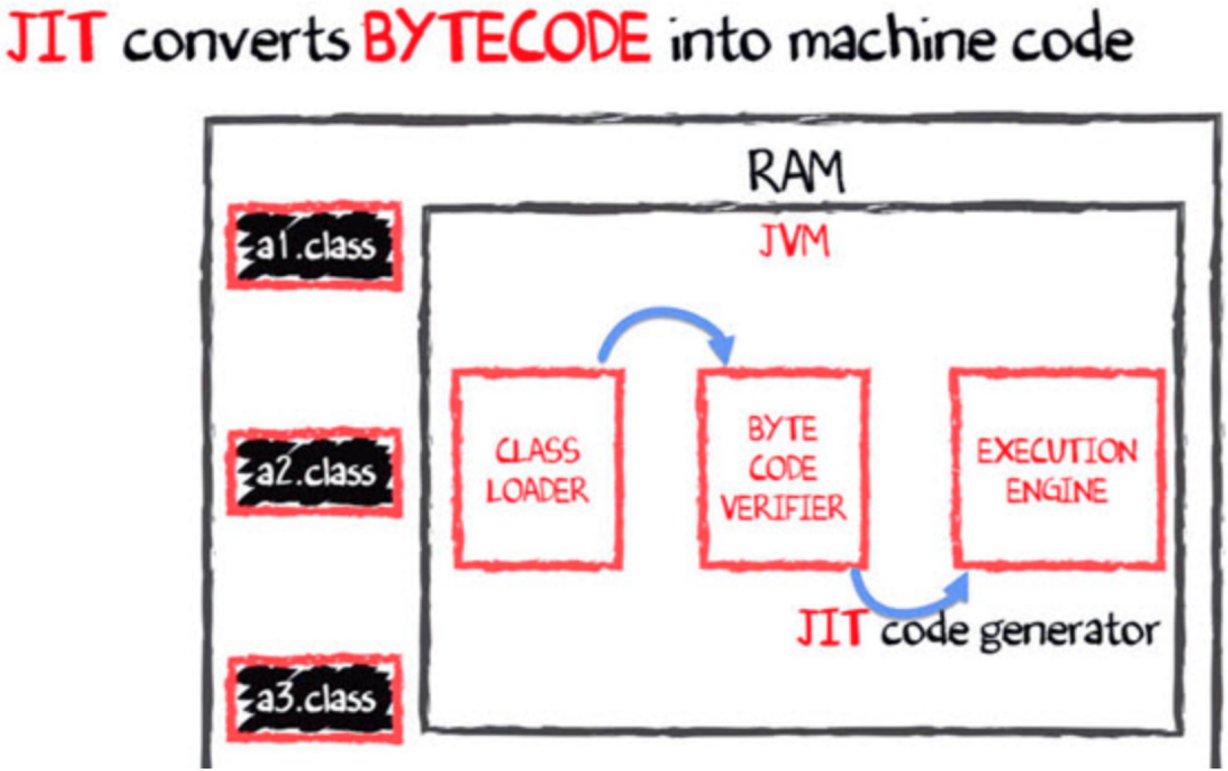
JDK

Java Development Kit is super set of JRE which also includes the tools for developing Java programmers ex: java compiler

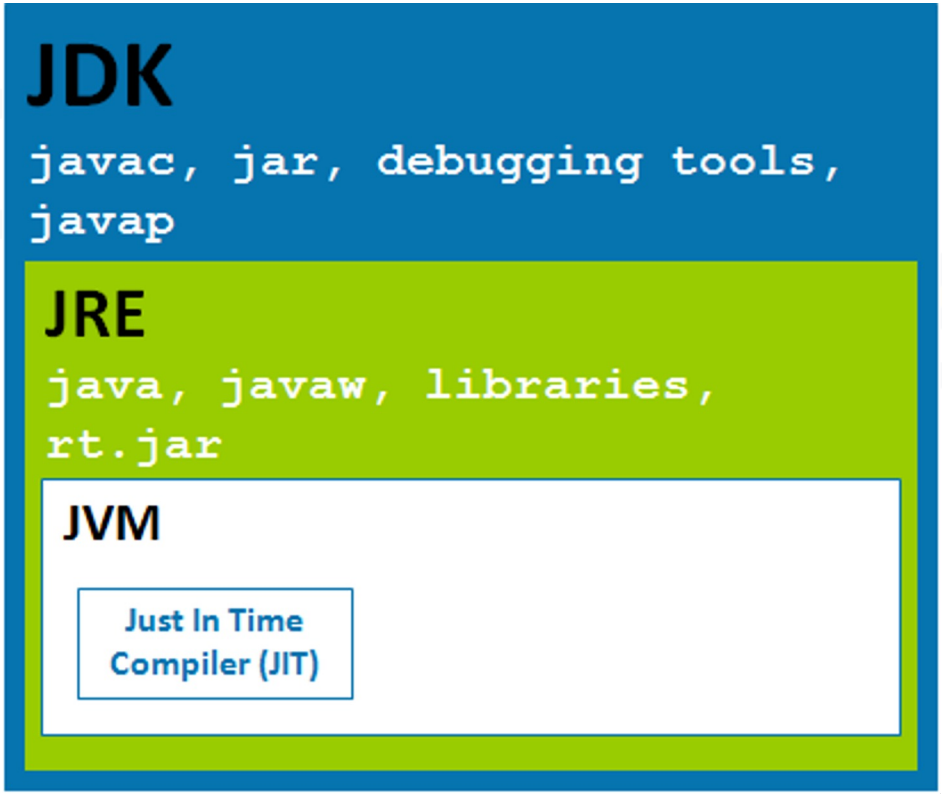
JIT

Just In Time Compiler translates the instruction set of JVM (byte codes) to instruction set of a specific CPU

Java Program Life Cycle



JVM/JRE/JDK relationship



Terminologies...

JVM

Java Virtual Machine is an Abstract computing machine that enables the computer to run a Java program. It is a specification that provides runtime environment in which java bytecode can be executed

JRE

Java Runtime Environment is a software package that contains what is required to run a Java program. Contains Java class libraries, Java virtual machines, and other tools that the programs need to execute.

JDK

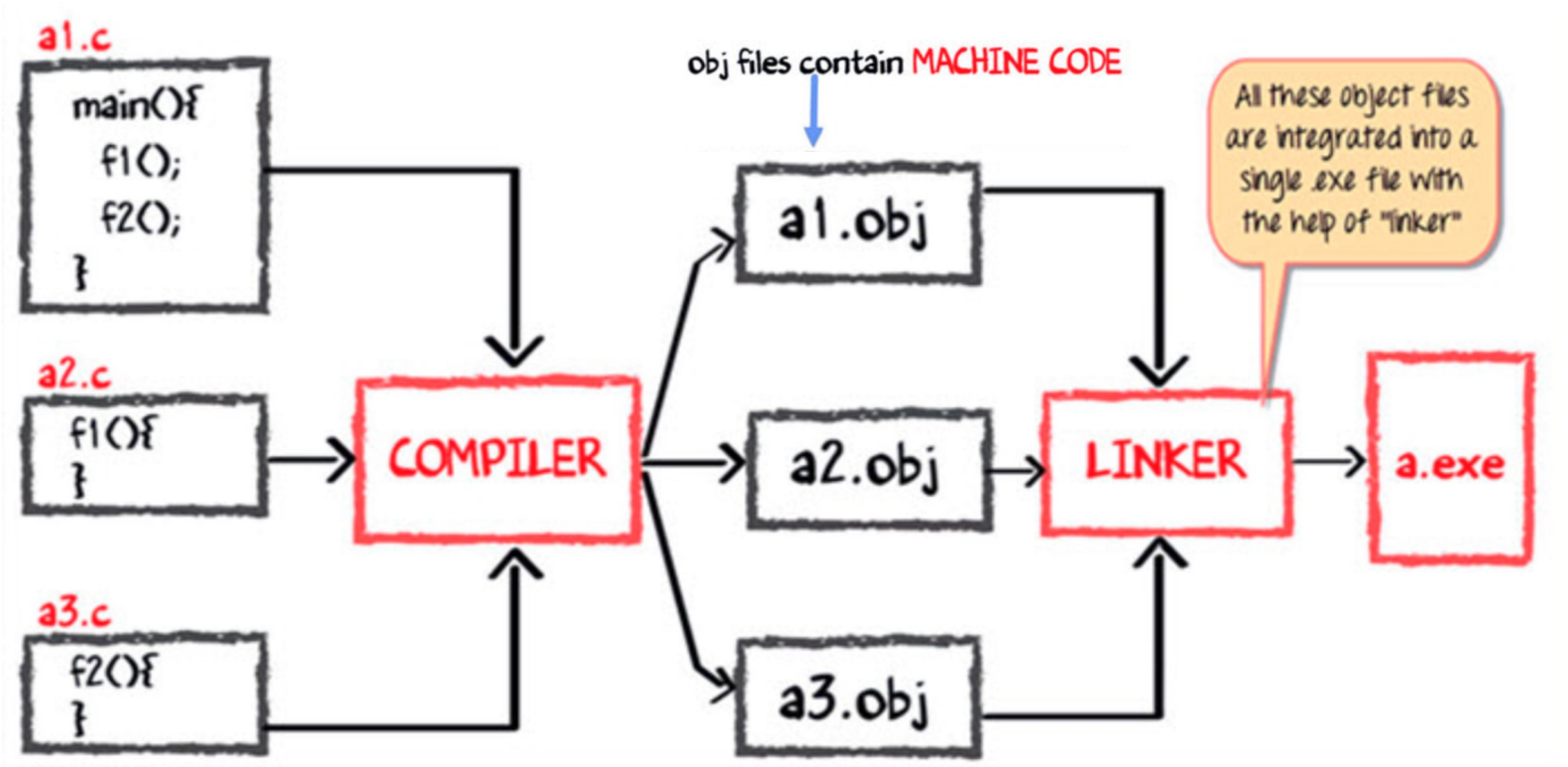
Java Development Kit is super set of JRE which also includes the tools for developing Java programmers ex: java compiler

JIT

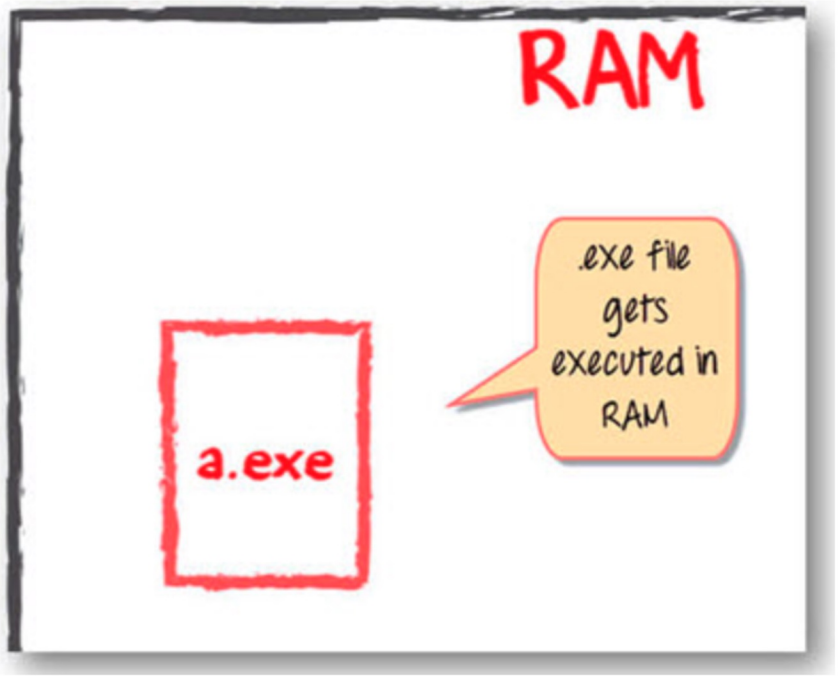
Just In Time Compiler translates the instruction set of JVM (byte codes) to instruction set of a specific CPU

WORA

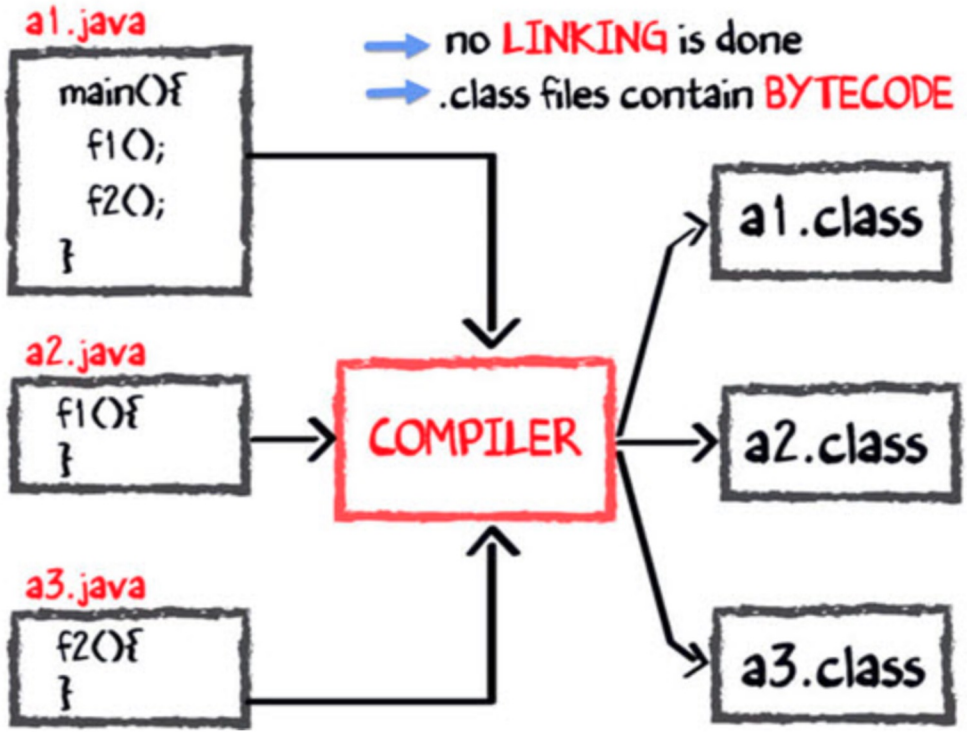
C program Life Cycle



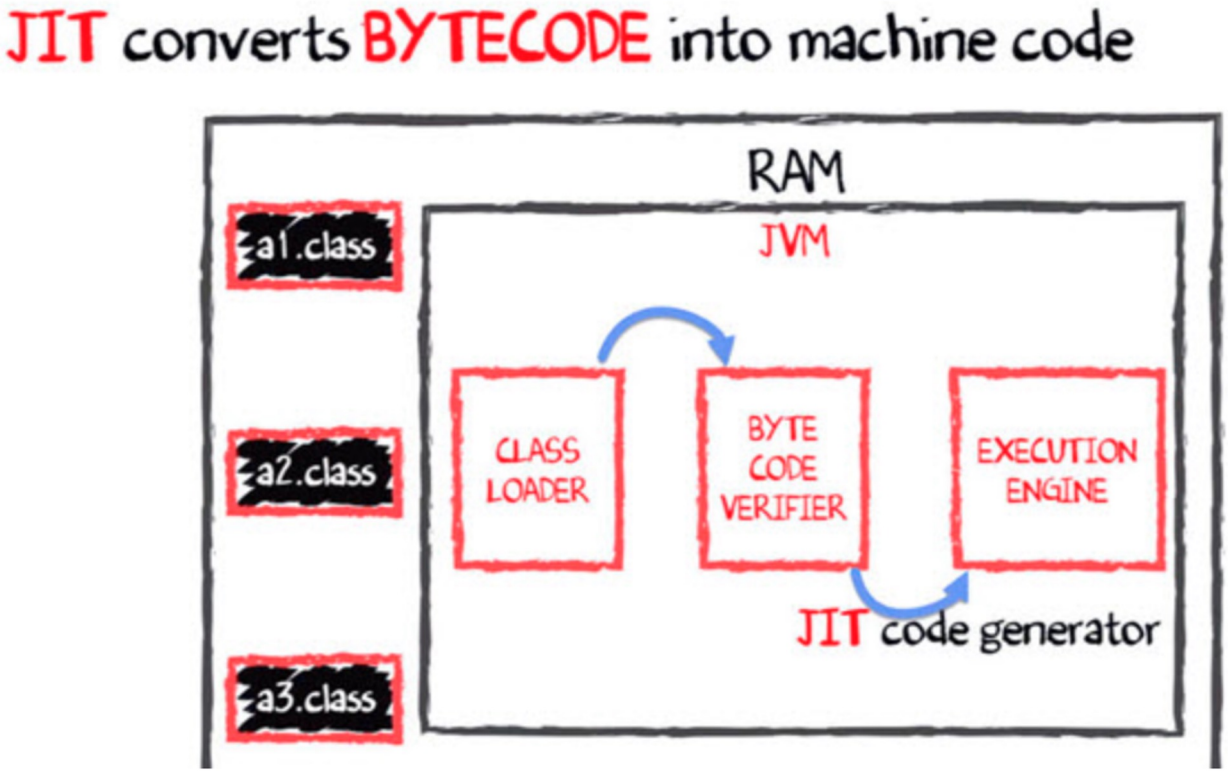
C program Life Cycle



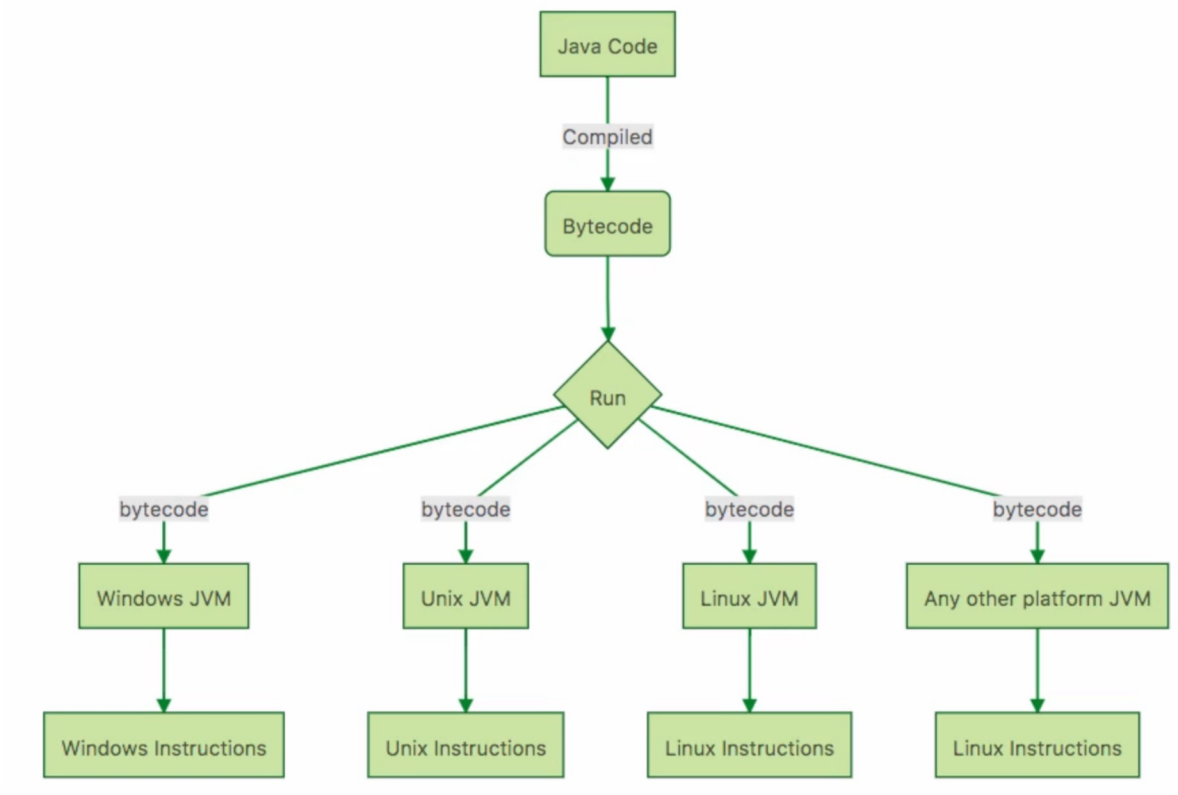
Java Program Life Cycle

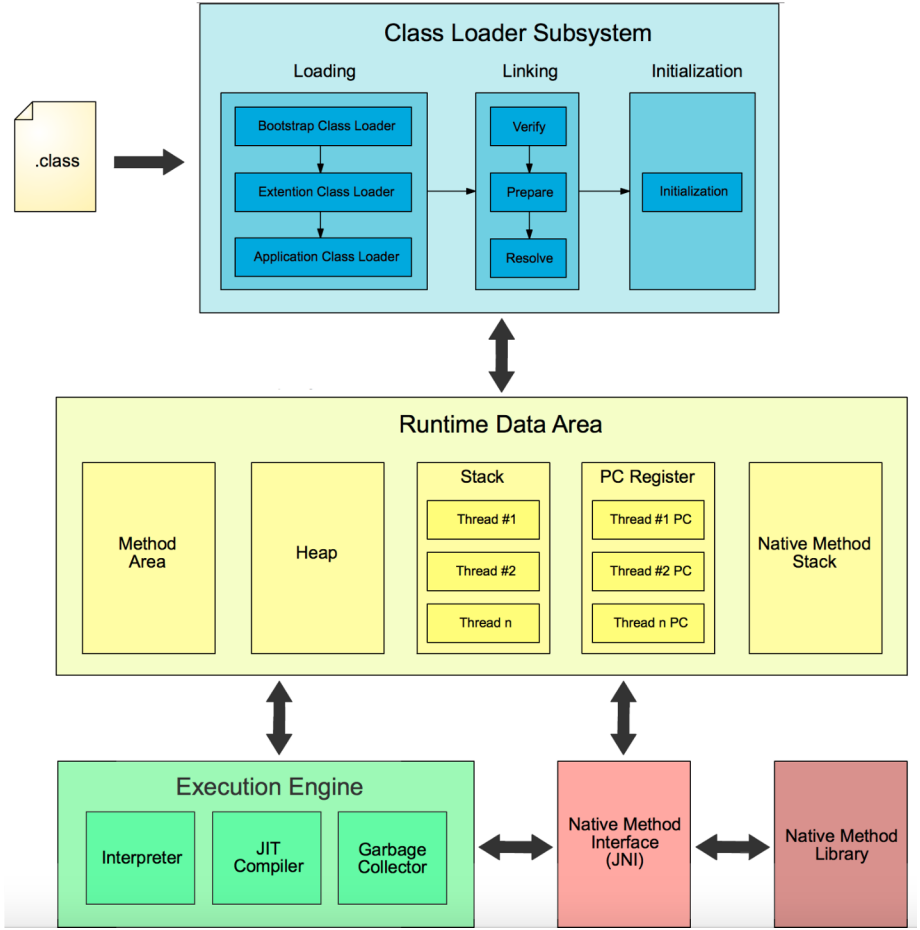


Java Program Life Cycle



WORA summary




Class Loader Subsystem
Loading
Linking
Initialization
.class
Bootstrap Class Loader
Extention Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Runtime Data Area
Method Area
Heap
Stack
Thread #1
Thread #2
Thread n
PC Register
Thread #1 PC
Thread #2 PC
Thread n PC
Native Method Stack
Execution Engine
Interpreter
JIT Compiler
Garbage Collector
Native Method Interface (JNI)
Native Method Library

Class Loader subsystems - **Load**

- **Bootstrap Class Loader** – It loads JDK internal classes, typically loads rt.jar and other core classes for example java.lang.* package classes
- **Extensions Class Loader** – It loads classes from the JDK extensions directory, usually $JAVA_HOME/lib/ext directory.
- **System or Application Class Loader** – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.
- Java Class Loader are hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way uniqueness is maintained in the runtime environment. If the parent class loader doesn't find the class then the class loader itself tries to load the class.

Class Loader subsystems - **Link**

- **Verify:** ensuring the correctness of the imported type
  - It is a process of ensuring that Binary representation of a class is a structurally correct or not.
  - JVM will check whether the .class file is generated by valid compiler or not.
  - .class file is properly formatted or not.
- **Prepare:** allocating memory for class variables and initializing the memory to default values
- **Resolve:** transforming symbolic references from the type into direct references.
- **Initialization:** invoking Java code that initializes class variables to their proper starting values.

Runtime Data Area

Method Area

- All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource
- The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine
- virtual machine extracts information about the class from the binary data and stores the information in the method area
- The virtual machine will search through and use the class information stored in the method area as it executes the application it is hosting

Runtime Data Area

PC Registers

- Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction

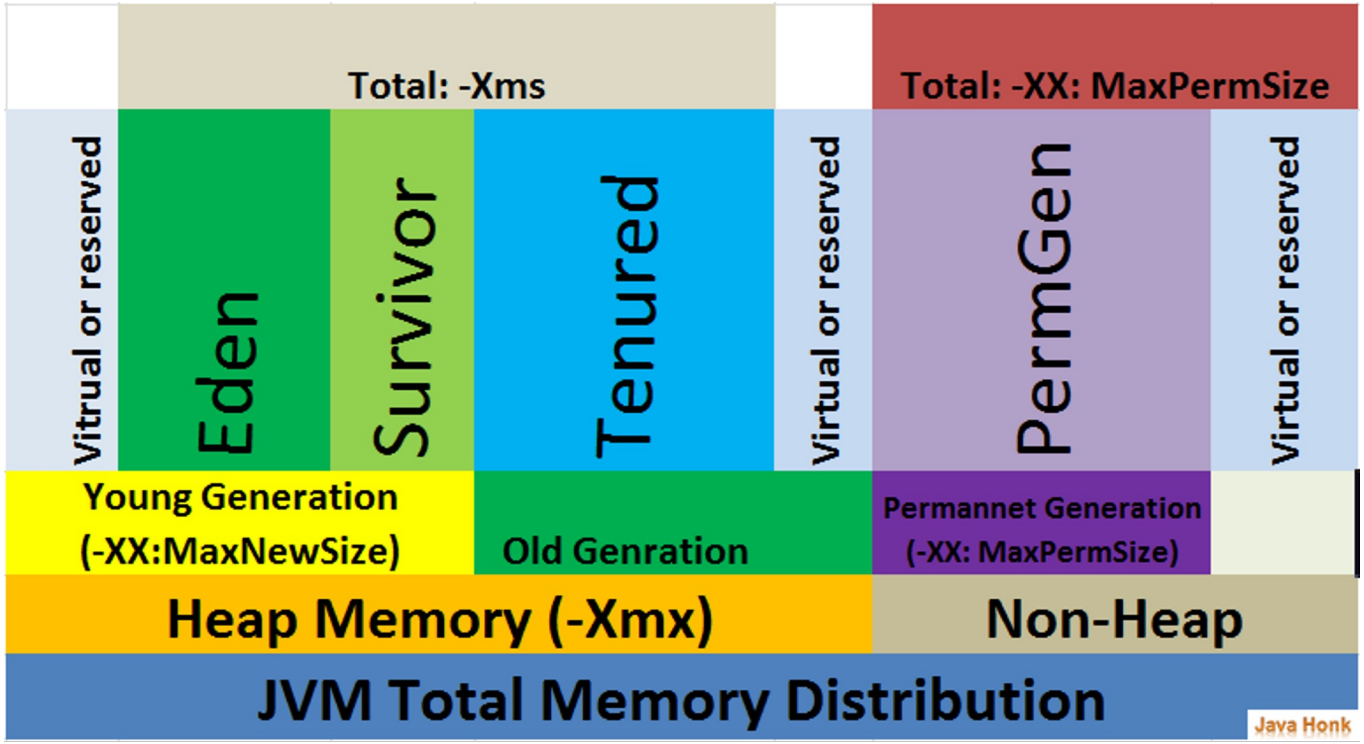
Native Method Stacks

- Native Method Stack holds native method information. For every thread, a separate native method stack will be created

Execution Engine

- **Just-in-time Compiler (JIT)** - JIT is the part of the Java Virtual Machine (JVM) that is used to speed up the execution time.
- JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term “compiler” refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.
- **Interpreter** – Converts the byte code into machine instructions

Runtime Data Area - Heap



Runtime Data Area - **Heap**

- All Objects and their corresponding instance variables data will be stored here
- One heap memory per JVM and hence a shared resource between threads (not thread safe)
- Heap memory is created during the instantiation of JVM
- Heap memory is further divided into three generations
  - Young Generation (Eden space)
    - Eden memory and two Survivor memory spaces
  - Old (Tenured space)
- When object is created then it first go to Young generation(especially Eden space) when objects get old then it moves to Old/tenured Generation.

Runtime Data Area - **Non-Heap**

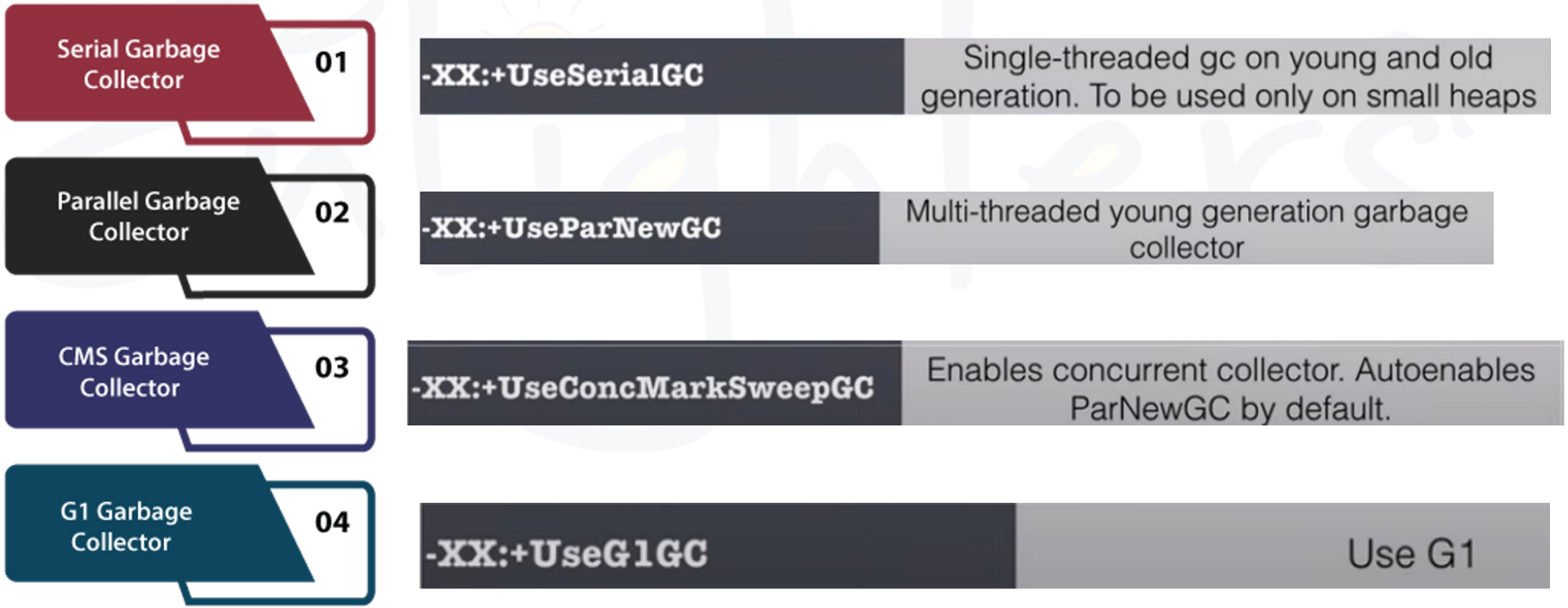
- In PermGen space all static & instance variables name-value pairs(name-references for object) are stored
- Contains application metadata required by the JVM to describe the classes and methods used in the application. PermGen is not part of Java Heap Memory
- Populated by JVM at runtime based on the classes used by the application.
- PermGen also contains Java library classes and methods
- From Java 8, PermGen is replaced with Metaspace which is very similar with a difference that Metaspace re-sizes dynamically

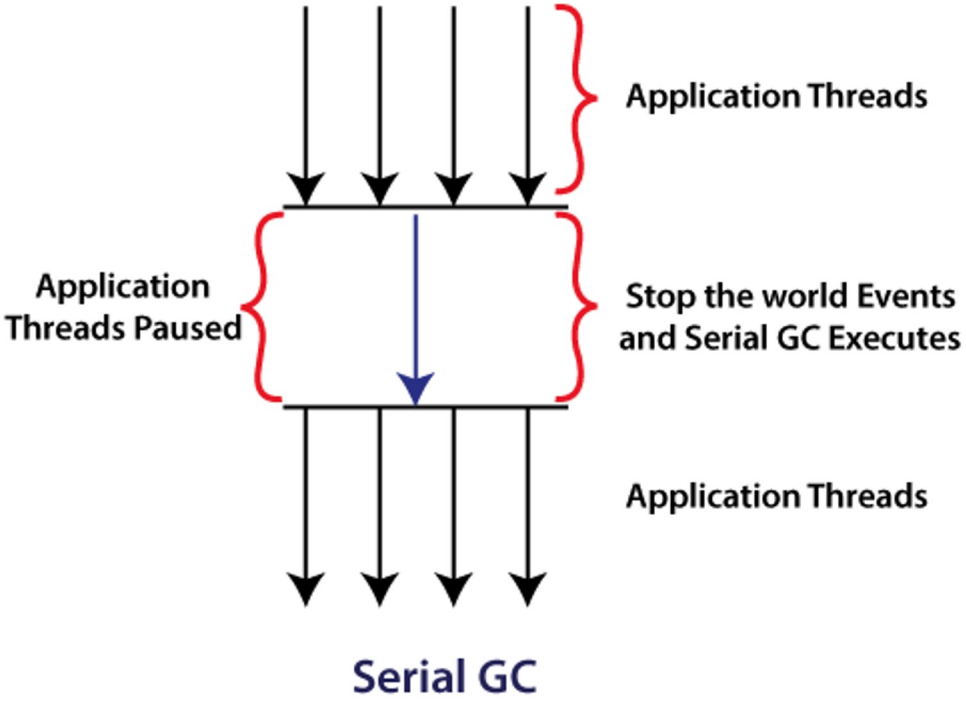
Runtime Data Area - Stack

- All local variables & function calls of each thread are stored in separate stack
- A Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java Stacks: it pushes and pops frames
- When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.
- The stack frame has three parts: local variables, operand stack, and frame data
- It throws StackOverFlow error when stack get full

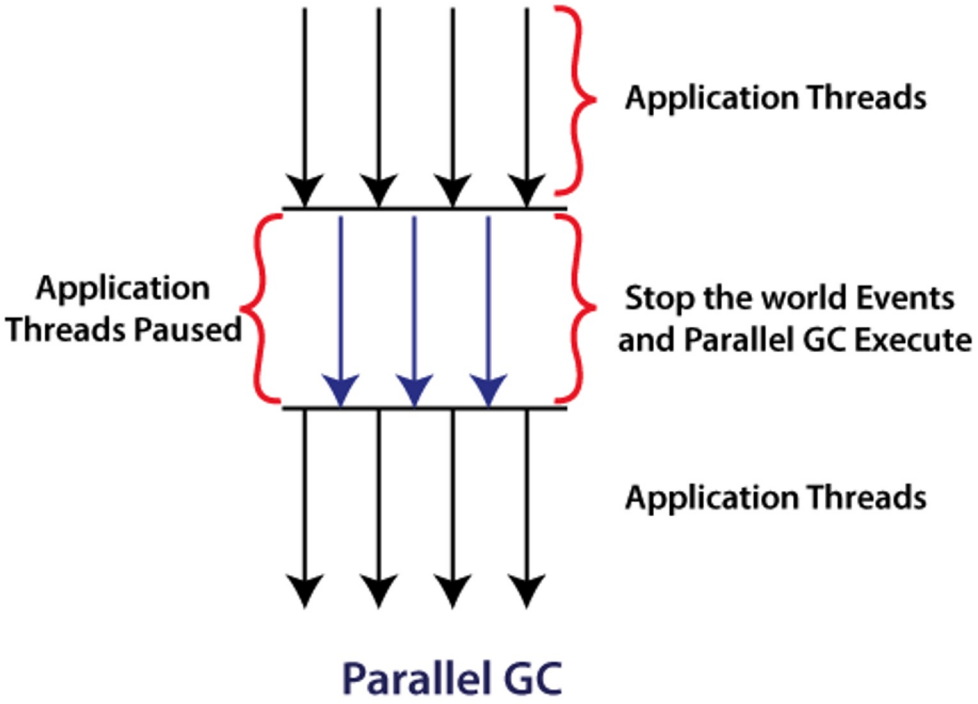
Garbage Collector

Types of Garbage Collector in Java





- Best suited for single-processor machines

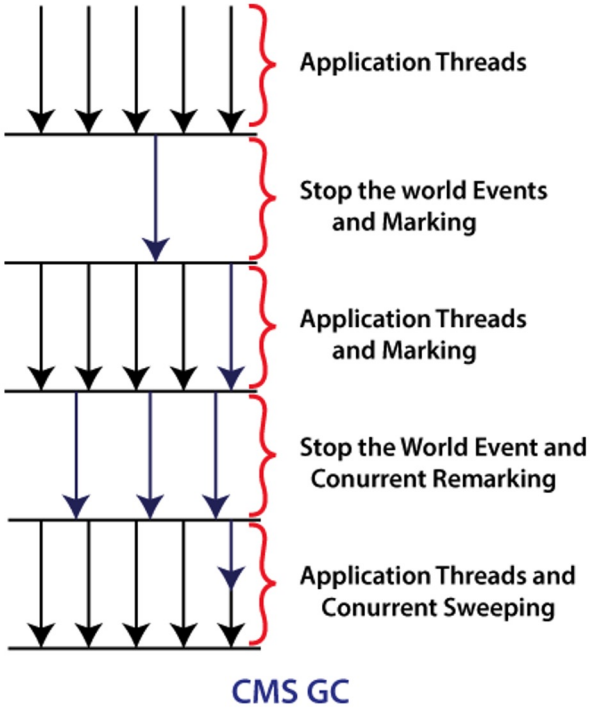


- Known as throughput collector as often best choice when throughput is important from latency

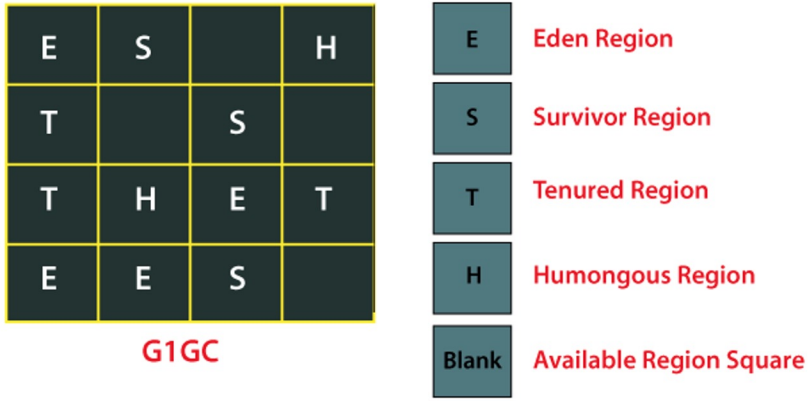
-XX:ParallelGCThreads - No of garbage collector threads

-XX:MaxGCPauseMillis - Max pause time in milliseconds

-XX:GCTimeRatio - Helps achieve application throughput goal. sets the amount of time devoted to garbage collection



- Deprecated as of JDK 9 and recommendation to use G1 collector
- Collector is more beneficial when long-lived tenured gen is high and application running on a machine with two or more available process



Garbage first collector

Designed for multiprocessor machine with large amount of memory

Heap is divided into a number of equally sized regions

Achieve high throughput and low latency for applications with large heap size (more than 6GB)

Garbage Collector notes

When Eden is full, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces

Minor GC also moved the object from one survivor to another. At a time, one of the survivor space is always empty

Obj that survived many cycles of GC, are moved to the old gen memory space. usually done by setting a threshold for the age of the young gen objects eligible to promote to old gen

All GC;s are “Stop the World” events because all application threads are stopped until the operation completes

Since Young Gen keeps short-lived objects, Minor GC is very fast and application doesn't get affected by this

Major GC takes long time because it checks all the live objects. Major GC should be minimized because application unresponsive for GC duration.

The duration of GC depends on the strategy used for garbage collection.