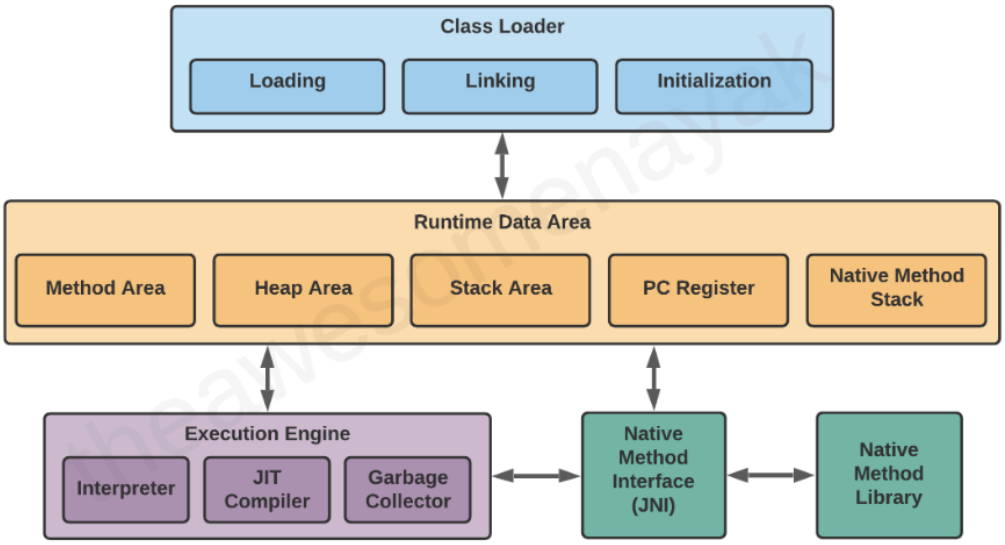
Memory Management
Amo Samuel

JVM Internals and Garbage Collector Behavior.

The JVM consists of three distinct components:

1. Class Loader
2. Runtime Memory/Data Area
3. Execution Engine



1. Class Loader Subsystem

The **Class Loader** subsystem in the JVM is responsible for dynamically loading Java classes into memory as needed during runtime. There are three types of class loaders:

- **Bootstrap Class Loader**: Loads core Java libraries, such as java.lang and java.util.
- **Extension Class Loader**: Loads classes from the ext directory, which contain additional libraries and extensions.
- **Application Class Loader**: Loads application-specific classes from the classpath (typically .jar files or compiled .class files).

The class loading mechanism follows a delegation model, where each class loader delegates the loading request to its parent class loader before attempting to load the class itself. This approach ensures that classes are loaded efficiently and securely.

2. Memory Management

Memory management is one of the most important aspects of the JVM, as it directly impacts performance and resource usage. The JVM's memory is divided into several areas, each serving specific purposes:

Main divisions

Heap Area: Allocates memory for objects created by Java programs.

Stack Area: Stores method calls and local variables. Each thread gets its own stack.

PC Register: Holds the address of the current instruction being executed for each thread.

Method Area: Stores class-level information like method data, field information, and the runtime constant pool.

Native Method Stack: Used when Java programs invoke native methods (written in C/C++).

The memory heap is a generational garbage collector divided into spaces.

- **Young Space** - The Young Region, as the name suggests, is the heap region that contains newly allocated objects. The Young Region is itself further divided into more regions.
    - ***Eden Space*** - On allocation, an object is stored in the Eden region of the heap until its first garbage collection.
    - ***Survivor Spaces*** - Objects that have survived a GC cycle are copied to a survivor region.
- **Old Region** - If an object gains enough "age" by surviving GC cycles, it will be copied to the old region. The garbage collector rarely scans the old region for no longer reachable objects.
- **Permanent/Metaspace Region** - The final region is the permanent or metaspace region. Objects stored here are typically JVM metadata, core system classes, and other data that typically exist for nearly the entire duration of the JVM life.

3 Execution Engine

The **Execution Engine** is a core component of the Java Virtual Machine (JVM) responsible for executing the bytecode generated by the Java compiler. It plays a crucial role in converting Java bytecode into machine code, which is understandable by the underlying operating system and processor.

Components

- *Interpreter:* The interpreter reads and executes the bytecode **line by line**.
- *JIT Compiler:* To overcome the inefficiency of the interpreter, the **JIT compiler** is used.
- Gabbage Collector: Part of memory management within the JVM, the garbage collector automatically identifies and reclaims memory occupied by objects that are no longer in use

Gabbage Collection process

At a high level, garbage collectors have three phases; mark, sweep, and compaction. Each of these steps have distinct responsibilities. Depending on the garbage collector implementation, there might be additional sub-phases within each phase.

Mark: On object creation, every object is given, by the VM, a 1 bit marking value, initially set to false (0). The garbage collector uses this value to mark if an object is reachable. At the start of a garbage collection, the garbage collector traverses the object graph and marks any object it can reach as true (1).

Sweep: During the sweep phase all objects that are unreachable, those whose marking bit is currently false (0), are removed.

Compaction: The final phase of a garbage collection is the compaction phase. Live objects in the eden region or an occupied survivor region are moved and/or copied to an empty survivor region. If an object in a survivor region has gained enough tenureship, it is moved or copied to an old regions

Garbage Collection Pause

During a garbage collection, there might be periods where some, or even all, processing within the JVM is paused, these are called Stop-the-World Events. During a garbage collection, part, or all, of the JVM must be paused for a period while the garbage collector works to prevent errors from occuring as objects are checked for usage, deleted, and moved or copied.

A comparison of Different Garbage Collector Performance

Garbage Collector	Latency	Throughput	Best for
Serial GC	High	Low	Small, single-threaded apps
Parallel GC	Moderate	High	High-throughput, multi-threaded apps
CMS GC	Low	Moderate	Low-latency, interactive apps
G1 GC	Low-Moderate	Moderate-High	Large, heap-sensitive applications
ZGC	Extremely Low	Moderate	Large, latency-sensitive apps
Shenandoah GC	Very Low	Moderate-High	Real-time, low-latency apps

Summarizing Memory Management Best Practices.

These are some few memory management best practices that can help improve the performance of applications

- **Avoiding Creating Unnecessary Objects**: Reuse objects and minimize temporary ones to reduce garbage collection frequency and overhead.
- **The use StringBuilder for String Concatenation**: Prefer StringBuilder over + for concatenating strings in loops to avoid creating multiple temporary String objects.
- **Choose the Right Garbage Collector**: Select the appropriate GC, like Serial for small apps or G1/ZGC for large-scale or latency-sensitive apps, to balance performance and pause times.
- **Monitoring and Tuning the Heap Size**: Adjust JVM options like -Xms and -Xmx to set optimal heap sizes based on your application's memory needs.
- **Avoiding Memory Leaks**: Prevent memory leaks by avoiding unnecessary long-lived object references and regularly monitoring memory usage with tools.