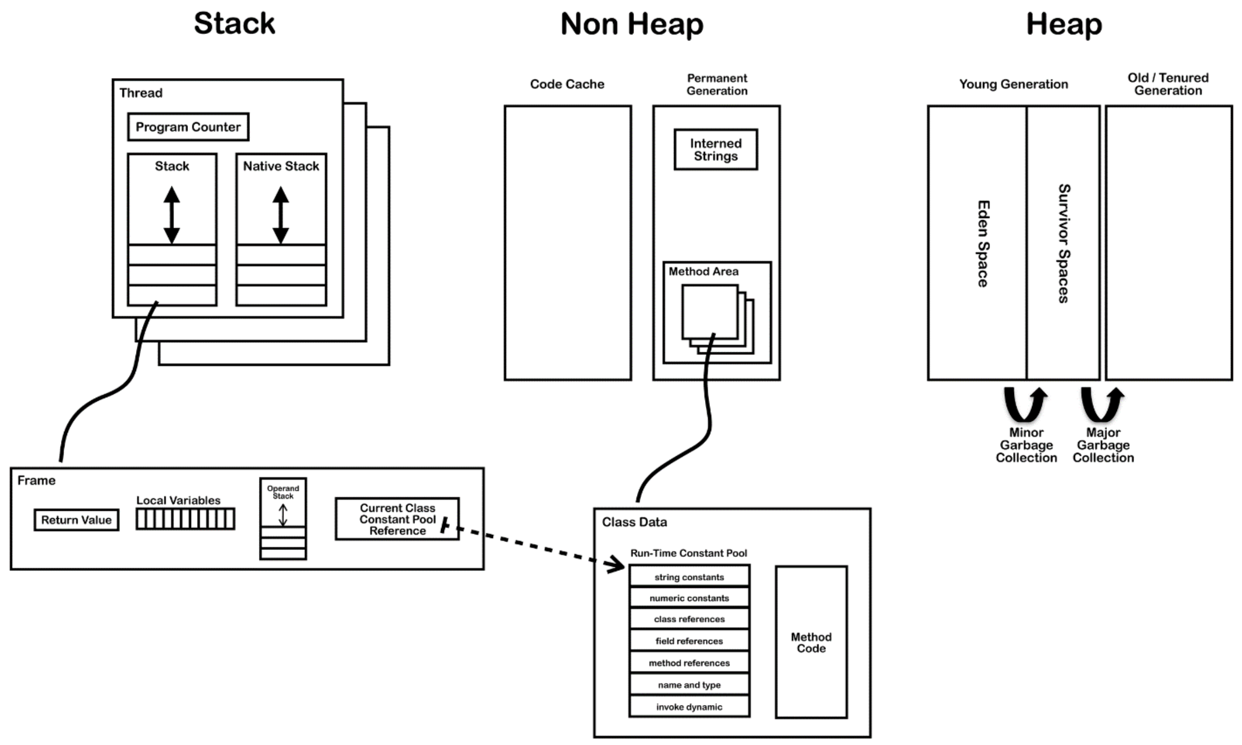
JVM Internals



The following are the major components of a JVM specification.

1. Classloader subsystem
2. Runtime Data Area
3. Execution Engine
4. Java Native Interface (JNI)

Classloader subsystem

The **classloader subsystem** is a primary component in the JVM specification responsible for loading the **bytecodes (.class files)** from the filesystem or network into the main memory and preparing those bytecodes for execution.

Each of the classloader subsystems extends the “**java.lang.ClassLoader”** parent class.

The classloader subsystem performs three major functionalities namely **Loading**, **Linking**, and **Initialization**.

Loading is the process of bringing the compiled bytecodes (.class files) into RAM storing type metadata into the Runtime Data Area for later execution (discussed below).

Loading is hierarchical and classes are lazily loaded when they are referenced for the first time. Loading generates corresponding binary data and is stored in the Method Area of the Runtime Data Area.

The classloading starts from the Main Class of the program and subsequent class loading attempts are done based on referencing under the following scenarios.

1. When the bytecode makes a static reference to a class (Ex: system.out.println)
2. When the bytecode creates an object of a particular class.
3. When the class is referenced using Reflection.

Types of ClassLoaders

There are three different types of Classloaders.

1. **Bootstrap/Primordial ClassLoader: Bootstrap ClassLoader** is the root of all class loaders which loads the JDK classes from **rt.jar** in the bootstrap classpath. The jar includes classes to support the runtime such as java.lang, applets, java.util, java.io, java.net, and the likes. These are written in native platform-dependent code.
2. **Extension ClassLoader: Extension ClassLoader** is responsible for loading core Java extensions from the JDK extension library present in the "jre/lib/ext" folder such as locales, security providers, and other libraries. These can also include libraries present in the directories specified by the environment variable "java.ext.dirs".
3. **System / Application ClassLoader: System / Application ClassLoader** is responsible for loading the application-specific classes mentioned in the system classpath or the jars mentioned in -cp while running the application.

Linking

Linking is the process of verifying already loaded classes, interfaces its parent classes, and interfaces post the **Loading** phase.

The process of **Linking** goes through three phases namely **Verify**, **Prepare** and **Resolve**.

Verify

Verifier checks if the bytecode is generated by a valid compiler and follows compatibility rules. Some of the checks performed by Verifier include

1. Ensure variables are initialized before usage.
2. Ensure final classes are not sub-classed and final methods are not overridden.
3. Ensure methods and classes respect access rules and methods are invoked with the correct number and type of parameters and return values.
4. Variables and classes are assigned and initialized with values of the correct type.

If any of these checks are violated, the Linker throws a **VerifyError** exception and the entire process comes to a halt.

Prepare

Prepare phase is executed post the **Verify** phase that allocates memory and initializes static fields of the class to their default values.

For example, int is initialized to 0, objects are initialized to null, and so on. **Prepare** phase doesn't execute static blocks which is the role of the **Initialization** phase.

Resolve

The **Resolve** phase replaces all symbolic references present in the bytecode with actual references by doing a lookup into the **Runtime Constant Pool** (described below) into the **Method Area**.

Initialization

This is the phase where the initialization logic for each class will be executed that initializes all static variables to the respective assigned values and executes the static block for each class.

Runtime Data Area

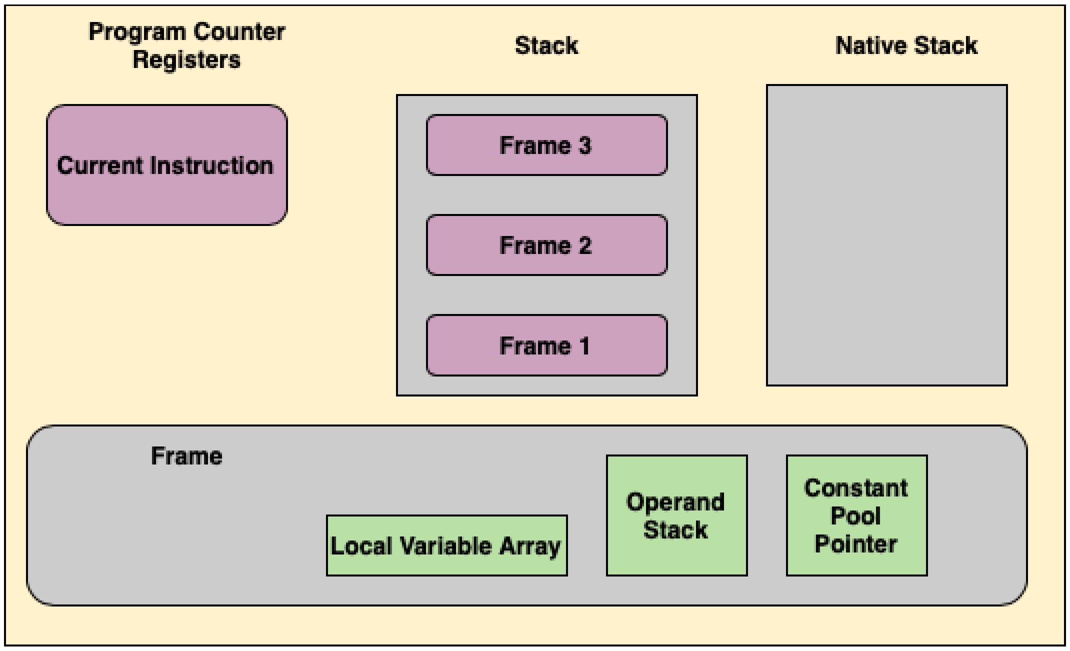
Runtime Data Area is essentially the memory region of JVM that acts as a storage area for class and class instance data.

The entire memory region of JVM is divided into three main areas namely **Stack Area**, **Heap Area,** and **Non-Heap Area**.

Stack

The Stack component of the Runtime Data Area includes three data structures namely Program Counter Register, Stack, and Native Stack.

Each thread of execution has its own Program Counter Register, a Stack, and an optional Native Stack making it thread-safe.



Program Counter (PC) Register

PC Register holds the address of the currently executing instruction **(opcode)** of a non-native method for the given Thread. It is updated with a new address after the instruction is executed.

These instructions include symbolic references that will be pointing to the **Runtime Constant Pool** of the **Method Area**.

Stack

Each Thread has its Stack with **LIFO (Last in First Out)** data structure that holds one **Frame** for each method executed by that thread.

A new **Frame** is pushed to the top of the Stack for a method invocation and the **Frame** is popped out of the Stack when the method returns. This way the current execution method will be the **Frame** at the top of the Stack.

The **Frames** themselves are allocated on Heap and are essentially pointers to the Heap.

Native Stack

A Native Stack is created for each thread when the native methods are invoked. It stores information about the native methods.

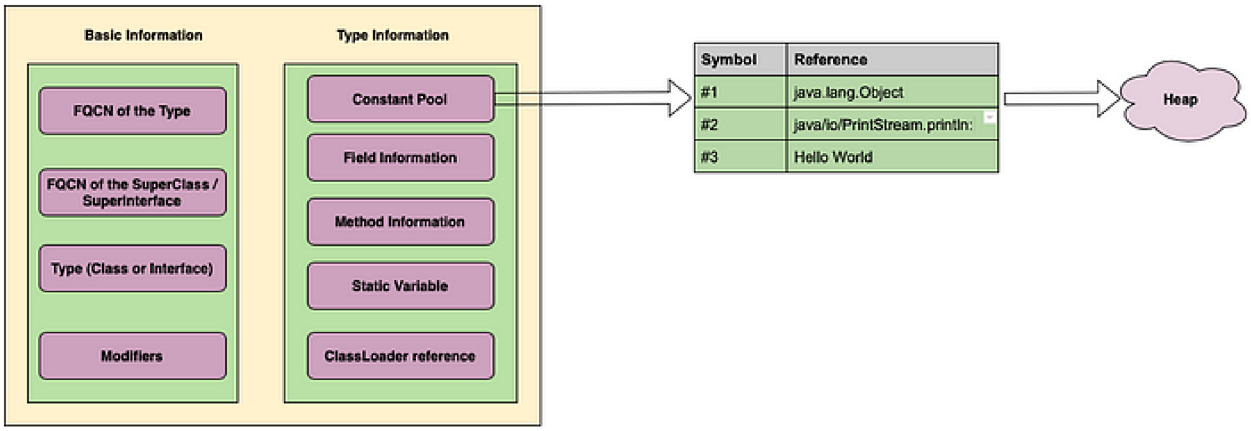
If a JVM has been implemented using a C-linkage model for Java Native Invocation (JNI) then the native stack will be a C stack and the functionality of push and pop are similar to C-Stack push and pop.

Non-Heap

Code Cache

Code Cache Region of the Non-Heap is used for storing the binary code which was compiled from the bytecode by the JIT Compiler. This is frequently referred to as JIT code cache.

Method Area



Symbol	Reference
#1	java.lang.Object
#2	java/io/PrintStream.println:
#3	Hello World

Method Area of the Non-Heap store class-level metadata and the code for methods and constructors.

When the JRE loads the class, it uses ClassLoader Subsystem to locate the appropriate class file, reads the bytecode, and passes it to JRE which in turn extracts class type metadata and stores them in the Method Area.

All the threads have access to a shared Method Area and hence it is not thread-safe.

For each type, JRE stores the following basic information in the Method Area.

1. The fully qualified name of the class and its superclass and/or super interfaces.
2. Whether the type is a class or interface
3. Modifiers of that particular type.

A fully qualified name is nothing but the **package name** followed by a **dot** ending with a **class** or **interface name**. (Ex: java.util.List)

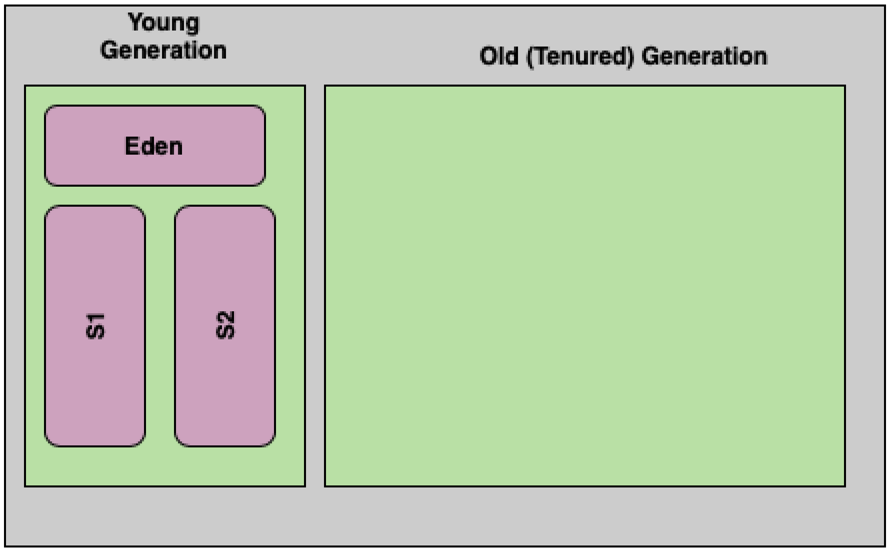
In addition to the basic information, each type in Method Area also includes

1. The Constant Pool
2. Field information
3. Method information
4. All static variables declared in the class
5. A pointer reference to classLoader.

Heap

Heap is a chunk of memory in JRE where all the class instances and arrays are allocated. Unlike Stack**,** objects allocated on Heap are never reclaimed when the method returns but is delegated to a daemon **Garbage Collector** to reclaim the space occupied by class instances.

All Threads share the same Heap region and hence are not thread-safe.



A Heap is divided into two regions namely **Young Generation** and the **Old (Tenured) Generation**.

Young Generation includes an **Eden Space** and two **Survivor Spaces (S1 and S2)**.

Object allocation starts in the **Eden Space** of the **Young Generation** and is subsequently moved to **Survivor Spaces** of the **Young Generation** and finally to the **Old (Tenured)**

Generation if the objects are alive for a longer tenure post multiple Garbage Collection cycles.

The size of the **Old(Tenured) Generation** is generally larger than the **Young Generation**.

Execution Engine

The **Execution Engine** is where the bytecodes are executed using an Interpreter or JIT Compiler using the data stored in **Runtime Data Area**.

The **Execution Engine** is composed of three main components: **Interpreter**, **JIT Compiler** and **Garbage Collector**.

Interpreter

The **Interpreter** reads the bytecode generated by Java Compiler and executes the instruction set (opcode) line by line.

Before executing the instructions, a sequence of steps are carried out as mentioned below.

1. Types are loaded on-demand using **ClassLoader Subsystem** storing metadata in **Method Area** and binary data on **Heap**.
2. Types are dynamically linked using the **Linking** module verifying the bytecodes and resolving symbolic references.
3. Classes are initialized with static variables and static blocks.

Once these steps are executed, the opcode is ready to be executed line by line. The major shortcoming of interpreting every instruction is, that when a method is called multiple times, interpretation is required every time resulting in slower execution.

(Just in Time) JIT Compiler

JIT Compiler solves the shortcomings of the Interpreter by compiling the frequently executed instructions from bytecode into native code and executing the native code thereby improving the performance.

JIT Compiler performs various optimizations to improve the performance of the execution.

1. **Inlining** is the process by which smaller methods are merged and inlined in the caller method thereby reducing the call graph, branching of methods, and stack allocation.

2. Various **local optimizations** are performed such as tail recursion elimination into the iterative process, simplification of statements, replacing variables with literals to reduce lookups, etc…
3. **Control flow optimizations** by rearranging code paths to improve efficiencies such as Loop reduction and inversion, Loop unwrapping, branching reduction, etc…

JIT Compiler analyzes the **hotspots** ie., the instructions which are frequently executed, and adds them to the eligible instructions to be compiled by JIT Compiler.

JIT Compiler includes the following components

1. **Intermediate Code Generator** producing intermediate code
2. **Code Optimizer** for optimizing the intermediate code generated.
3. **Profiler** for analyzing hotspots based on frequently executing instructions with their thresholds. This data is stored in **Code Cache** for subsequent analysis.
4. **Target Code Generator** for generating the target binary code and storing them in the **Code Cache**.

There are also advancements in **Ahead of Time Compiler (AOT)** that compiles the code into native code partially or completely beforehand during the javac compilation phase instead of converting it into bytecode and can be used in conjunction with **JIT Compiler**.

Java Native Interface (JNI)

JNI is used to connect with Native Method Libraries (typically written in other programming languages such as C/C++) necessary for the execution and to provide the features of such Native Libraries.

This allows JVM to call C/C++ libraries and be called by C/C++ libraries that are hardware-specific.

Garbage Collection(GC)

Local primitive variables are allocated on **Stack** whereas all objects and arrays whether it is localized to method or class are allocated on **Heap**.

The primitive variables allocated on **Stack** are freed up when the method returns.

The objects and arrays allocated on **Heap** are not freed up when the method is freed up. This is where **Garbage collectors** play a prominent role.

The primary responsibility of the **Garbage Collector** is to free up Heap memory for objects and arrays that are not referenced. An object is considered alive as long as it is referenced. If the reference count of the object is 0, it is considered eligible for Garbage collection.

There are two types of GC: **Minor GC** and **Major GC**

New objects are allocated in the Eden Space of the Young Generation. Once the Eden Space gets full, **Minor GC** is triggered. Unreachable objects in **Eden Space** are freed, live objects are moved from Eden Space to one of the empty Survivor Spaces (S1 and S2) and Eden Space is cleared.

This process continues every time the Eden Space is full. When the **Minor GC** is triggered again, unreachable objects in Eden Space and non-empty Survivor Space are freed. Live objects from Eden Space and non-empty Survivor Space are moved to empty Survivor Space**.**

Eden Space and previously occupied Survivor Space are cleared.

Minor GC alternates between Survivor Space S1 and S2 on every trigger and each trigger increments the reference count on live objects indicating the age of the object in the Young Generation.

Minor GC usually takes less time as the size of the Young Generation is generally small.

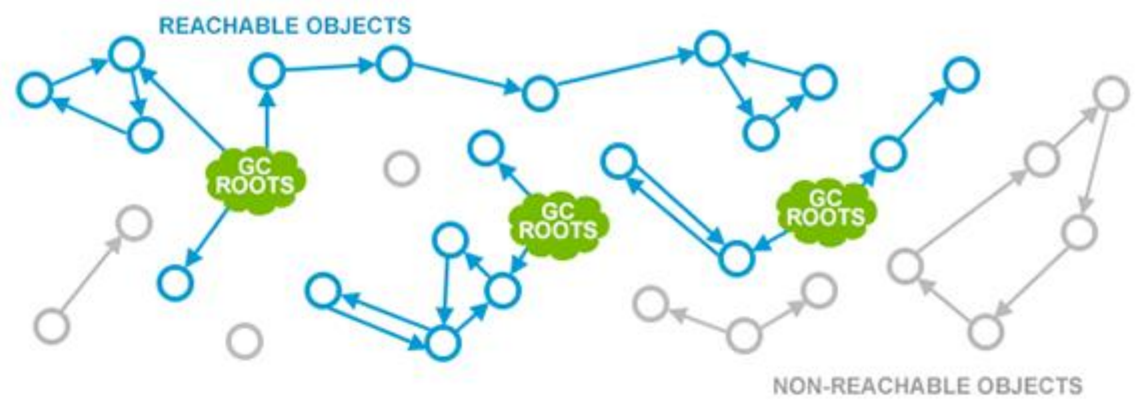
If the age of the objects exceeds a certain threshold in the Young Generation, they are moved to the Old (Tenured) Generation during the **Minor GC** phase.

Long-lived objects are eventually moved to Old Generation and remain there for a significant period. When the size of the **Old Generation** exceeds a certain threshold dictated by Heap Occupancy Percentage, **Major GC** is triggered which frees up Old Generation.

Major GC usually takes a lot of time as the size of the Old Generation is generally large and usually referred to as **Stop the World** as it pauses the application threads.

Minor GC is also Stop the World, but the pause time is so minimal in the Young Generation that it is not noticeable.

Most of the GC algorithms follow the following three steps to reclaim space.



Source: Internet

Mark Phase

Mark Phase is responsible for identifying all live(reachable objects) from the **Garbage Collection Roots (GC roots).** Examples of GC Roots include Active Threads, JNI References, Static fields of the loaded class, etc... The algorithm traverses the entire object graph starting from the GC roots to every object in the graph. Once the Mark Phase is complete, every live object is identified and marked.

Sweep Phase

In the **Sweep Phase**, all the space occupied by the unvisited object in the **Mark Phase** is considered unreachable and can be used for allocating new objects. This approach provides provision for allocating new objects but the memory chunks need not be contiguous. Hence the request to allocate a large object can fail if the memory chunks are not contiguous.

Compact Phase

In the **Compact Phase**, All the live objects are moved to the beginning of the memory region rewriting the memory region, its pointers, and GC Roots thus making the memory region contiguous, solving the shortcomings of the Sweep Phase.

The goal of every GC algorithm is to find an optimal tradeoff to reduce the pause time of the application and achieve maximum Garbage collection.

Types of Gabbage Collectors

1. Serial Garbage Collector

You can **turn on** the **Serial** garbage collector by adding the **-XX:+UseSerialGC** flag to your JVM application startup parameters. The Serial Garbage Collector is the simplest and oldest garbage collector in Java. It uses a single thread to perform garbage collection, making it suitable for single-threaded applications or small-scale applications with limited memory. It uses a "stop-the-world" approach, meaning that the application's execution is paused during garbage collection. This can lead to noticeable pauses in larger applications.

2. Parallel Garbage Collector

You can turn on the Parallel garbage collector by adding the **-XX:+UseParallelGC** flag to your JVM application startup parameters. The Parallel Garbage Collector, also known as the throughput collector, improves upon the Serial Garbage Collector by using multiple threads for garbage collection. It divides the heap into smaller regions and uses multiple threads to perform garbage collection concurrently, reducing the duration of stop-the-world pauses. It is well-suited for multi-core systems and applications that prioritize throughput.

3. CMS (Concurrent Mark-Sweep) Garbage Collector

You can turn it on by adding the **-XX:+UseConcMarkSweepGC** flag to your JVM application startup parameters. The CMS Garbage Collector aims to further reduce pause times by performing most of its work concurrently with the application threads. It divides the collection process into stages: marking, concurrent marking, and sweeping. While it significantly reduces pause times, it may not perform as well on very large heaps and can sometimes lead to fragmentation issues.

4. G1 (Garbage-First) Garbage Collector

You can turn it on by adding the **-XX:+G1GC** flag to your JVM application startup parameters. The G1 Garbage Collector is designed to provide high throughput and low-latency garbage collection. It divides the heap into regions and uses a mix of generational and concurrent collection strategies. It dynamically adjusts its behavior based on the application's memory requirements and aims to meet specified pause time goals. G1 is well-suited for applications that require low-latency performance and can handle larger heaps.

5. ZGC (Z Garbage Collector)

You can turn it on by adding **-XX:+UseZGC** flag to your application startup parameters. The Z Garbage Collector, or ZGC, is designed to provide consistent and low-latency performance. It focuses on keeping pause times predictable even for very large heaps. ZGC uses a combination of techniques, including a compacting collector and a read-barrier approach, to minimize pause times. It is suitable for applications where low-latency responsiveness is critical.