Memory Management in Java

In Java, memory management is the process of allocation and de-allocation of objects, called Memory management.

Java does memory management automatically.

Java uses an automatic memory management system called a **garbage collector**. Thus, we are not required to implement memory management logic in our application.

Java memory management divides into two major parts:

* **JVM Memory Structure**
* **Working of the Garbage Collector**

JVM creates various run time data areas in a heap.

These areas are used during the program execution.

The memory areas are destroyed when JVM exits, whereas the data areas are destroyed when the thread exits.



**Method Area aka Meta space**

Method Area is a part of the heap memory which is shared among all the threads. It creates when the JVM starts up. It is used to store class structure, superclass name, interface name, and constructors. The JVM stores the following kinds of information in the method area:

* A Fully qualified name of a type (ex: String)
* The type's modifiers
* Type's direct superclass name
* A structured list of the fully qualified names of super interfaces.

The Method Area, also known as the Metaspace in newer versions of Java (replacing the permanent generation **Permgen (Permanent Generation)**), is a part of the JVM memory that stores class-level information, including the bytecode of loaded classes, field and method data, constant pool , String pool , and other reflective information. It is shared among all threads and is typically allocated at JVM startup

**in Other word Metaspace** is a memory region in Java that stores static information pertaining to a Java application, including metadata associated with loaded classes. Unlike its predecessor, PermGen, Metaspace does not have an explicit size limit by default and expands dynamically. However, without an explicit boundary set, the Metaspace size is implicitly restricted by the available system memory on the hosting machine.

* The method area is another area of memory used by the JVM to store class data and other information related to the methods and fields of classes.
* It is also shared among all threads and is created at the start of the application.
* Examples of information stored in the method area include class and method objects, constant pool and runtime constant pool.

*The String pool in Java which is a special area of memory that stores String literals. It is a part of the JVM’s method area.*

**Heap Area**

Heap stores the actual objects. It creates when the JVM starts up. The user can control the heap if needed. It can be of fixed or dynamic size. When we use a new keyword, the JVM creates an instance for the object in a heap. While the reference of that object stores in the stack. There exists only one heap for each running JVM process. When heap becomes full, the garbage is collected. For example:

**StringBuilder sb= new StringBuilder();**

The above statement creates an object of the StringBuilder class.

The object is allocated to the heap, and the reference sb allocates to stack.

Heap is divided into the following parts:

* Young generation
* Survivor space
* Old generation
* Permanent generation & Code CacheA screenshot of a computer screen

  AI-generated content may be incorrect.

The heap is the memory region where objects are allocated. It is also a part of the JVM memory. As I mentioned earlier, the heap is divided into generations, such as the young generation and the old generation, to optimize garbage collection. The heap is where most of the dynamic memory allocation occurs in Java

* The heap is the main area of memory used by the JVM to **store objects and their associated data.**
* It is shared among all threads and is created at the start of the application.
* Examples of objects that are stored on the heap include instances of classes, arrays, and objects created using the “new” keyword.
* The heap is divided into two parts: Young Generation and Old Generation. Young generation is further divided into Eden Space and Survivor Space

The **Young Generation (Eden)**is where newly created objects are initially stored.

It is further divided into three parts: **the Eden space two Survivor spaces S1 and S2**

When the Eden space becomes full, a process called a minor GC (garbage collection) is performed.

During this process, the JVM moves all live objects from the Eden space to one of the Survivor spaces. The objects that are not moved to the Survivor spaces are garbage collected.

The Survivor spaces are used to store objects that have survived one GC but not yet moved to the Old Generation.

The **Old Generation**is where long-lived objects are eventually moved to. Objects that have survived multiple minor GCs are promoted to the Old Generation.

The Old Generation is subject to a full GC, which is a more expensive process than the minor GC.

**Stack:**

Each thread in a Java program has its own stack, which is used for tracking method invocations and managing local variables. The stack memory is used to store method frames, which contain information about method parameters, local variables, and intermediate results. The stack operates on a Last-In-First-Out (LIFO) principle.

* The stack is a memory area used to store the execution context of a thread.
* Each thread has its own stack, which contains information such as the current method call, local variables, and the address of the instruction currently being executed.
* When a method is called, a new frame is created on the stack to store the method’s parameters and local variables.
* When the method completes, its frame is removed from the stack.

**Native Method Stack:**

Native Method Stacks: The native method stacks are used to support native (non-Java) method calls. They are similar to the Java stacks but are used for executing native code.

* The native method stack stores the execution context for native methods, which are methods written in a language other than Java.
* The native method stack operates separately from the Java stack and has its own memory allocation.

**Code Cache:**

* The code cache is an area of memory used by the JVM to store compiled code for quick execution.
* It stores the native machine code version of the bytecode.

**PC Registers:**

* Each thread has a Program Counter (PC) register, which holds the address of the currently executing instruction. This register is part of the JVM memory.