**Java Memory Management**

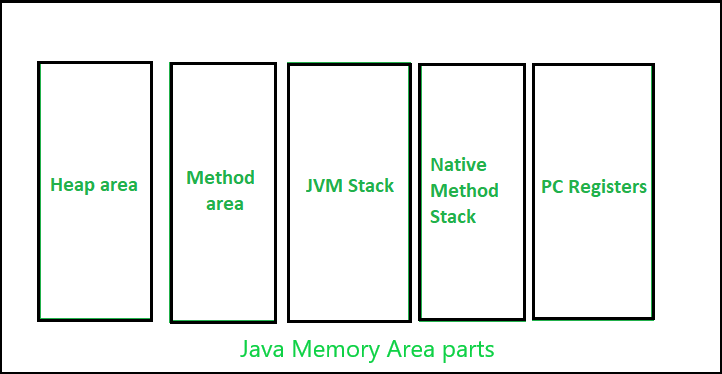
In every programming language, the memory is a vital resource and is also scarce in nature. Hence it’s essential that the memory is managed thoroughly without any leaks. Allocation and deallocation of memory is a critical task and requires a lot of care and consideration. However in Java, unlike other programming language, the JVM and to be specific Garbage Collector has the role of managing memory allocation so that the programmer needs not to. Whereas in other programming languages such as C the programmer has direct access to the memory who allocates memory in his code, thereby creating a lot of scope for leaks.

The major concepts in Java Memory Management :

* JVM Memory Structure
* Working of Garbage Collector

**Java Memory Structure:**

JVM defines various run time data area which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of thread are created during instantiation and destroyed when the thread exits.



JVM Memory area parts

Let’s study these parts of memory area in detail:

**Heap :**

* It is a shared runtime data area and stores the actual object in a memory. It is instantiated during the virtual machine startup.
* This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.
* There exists one and only one heap for a running JVM process.

Scanner sc = new Scanner(System.in);

The above statement creates the object of Scanner class which gets allocated to heap whereas the reference ‘sc’ gets pushed to the stack.

**Note:** Garbage collection in heap area is mandatory.

**Method Area:**

* It is a logical part of the heap area and is created on virtual machine startup.
* This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.

**Note:** Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.

**JVM Stacks:**

* A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
* Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created.
* The memory for stack needs not to be contiguous.

**Native method Stacks:**

Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature.

**Program counter (PC) registers:**

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

**Working of a Garbage Collector:**

* JVM triggers this process and as per the JVM garbage collection process is done or else withheld. It reduces the burden of programmer by automatically performing the allocation or deallocation of memory.
* Garbage collection process causes the rest of the processes or threads to be paused and thus is costly in nature. This problem is unacceptable for the client but can be eliminated by applying several garbage collector based algorithms. This process of applying algorithm is often termed as **Garbage Collector tuning** and is important for improving the performance of a program.
* Another solution is the generational garbage collectors that adds an age field to the objects that are assigned a memory. As more and more objects are created, the list of garbage grows thereby increasing the garbage collection time. On the basis of how many clock cycles the objects have survived, objects are grouped and are allocated an ‘age’ accordingly. This way the garbage collection work gets distributed.
* In the current scenario, all garbage collectors are generational, and hence, optimal.

**Note:** **System.gc()** and **Runtime.gc()** are the methods which requests for Garbage collection to JVM explicitly but it doesn’t ensures garbage collection as the final decision of garbage collection is of JVM only.

Knowing how the program and it’s data is stored or organized is essential as it helps when the programmer intends to write an optimized code in terms of resources and it’s consumption. Also it helps in finding the memory leaks or inconsistency, and helps in debugging memory related errors. However, the memory management concept is extremely vast and therefore one must put his best to study it as much as possible to improve the knowledge of the same.

**Q2. What Is Garbage Collection and What Are Its Advantages?**

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

**Q3. Are There Any Disadvantages of Garbage Collection?**

Yes. Whenever the garbage collector runs, it has an effect on the application's performance. This is because all other threads in the application have to be stopped to allow the garbage collector thread to effectively do its work.

**Q5. What Are Stack and Heap? What Is Stored in Each of These Memory Structures, and How Are They Interrelated?**

The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap defined in currently executing methods.

This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all local variables after exiting the method. Every thread has its own stack

The heap is a large bulk of memory intended for allocation of objects. When you create an object with the *new* keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

**Q6. What Is Generational Garbage Collection and What Makes It a Popular Garbage Collection Approach?**

Generational garbage collection can be loosely defined as the strategy used by the garbage collector where the heap is divided into a number of sections called generations, each of which will hold objects according to their “age” on the heap.

Whenever the garbage collector is running, the first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned.

As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived.

With generational garbage collection, objects are grouped according to their “age” in terms of how many garbage collection cycles they have survived. This way, the bulk of the work spread across various minor and major collection cycles.

Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

**Q7. Describe in Detail How Generational Garbage Collection Works**

To properly understand how generational garbage collection works, it is important to first **remember how Java heap is structured** to facilitate generational garbage collection.

The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.

The **young generation hosts most of the newly created objects**. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only “promoted” to the old generation space after they have attained a certain “age”.

The term **“age”** in generational garbage collection **refers to the number of collection cycles the object has survived**.

The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).

The **old generation hosts objects that** **have lived in memory longer than a certain “age”**. The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less frequently than in the young generation.

The **permanent generation** **or more commonly called, *PermGen,* contains metadata required by the JVM** to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.

First, **any new objects are allocated to the Eden space**. Both survivor spaces start out empty. When the Eden space fills up, a minor garbage collection is triggered. Referenced objects are moved to the first survivor space. Unreferenced objects are deleted.

This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space. For each major GC, there are several minor GCs.

**Q14. How Are Strings Represented in Memory?**

A *String* instance in Java is an object with two fields: a *char[] value* field and an *int hash* field. The *value* field is an array of chars representing the string itself, and the *hash* field contains the *hashCode* of a string which is initialized with zero, calculated during the first *hashCode()* call and cached ever since. As a curious edge case, if a *hashCode* of a string has a zero value, it has to be recalculated each time the *hashCode()* is called.

Important thing is that a *String* instance is immutable: you can't get or modify the underlying *char[]* array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical *String* objects in your source code, they are all represented by a single instance at runtime.