Interview Questions | MCQs

300+ [LATEST] Java Memory Management Interview Questions and Answers

Q1. 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.

Q2. 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.

Q3. 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.
- During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1).
- In addition, objects from the last minor GC in the first survivor space (S0) have their age incremented and are moved to S@Once all surviving objects have been moved to S1, both S0 and Eden space are cleared. At this point, S1 contains objects with different ages.
- At the next minor GC, the same process is repeated. However this time the survivor spaces switch. Referenced objects are moved to S0 from both Eden and S@Surviving objects are aged. Eden and S1 are cleared.
- After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.
- 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 cle up and compacts that space. For each major GC, there are several minor GCs.

Q4. What Is A Stringbuilder And What Are Its Use Cases? What Is The Difference Between Appending A String To A Stringbuilder And

Concatenating Two Strings With A + Operator? How Does Stringbuilder Differ

- StringBuilder allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the String class which is immutable.
- When concatenating two String instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. StringBuilder allows handling string manipulations much more efficiently.
- StringBuffer is different from StringBuilder in that it is thread-safe. If you need to manipulate a string in a single thread, use StringBuilder instead.

Q5. When Does An Object Become Eligible For Garbage Collection? Describe How The Gc Collects An Eligible Object?

- An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.
- The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.
- Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

Q6. 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.

- Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

Q7. Describe Strong, Weak, Soft And Phantom References And Their Role In Garbage Collection?

Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to minimize latency and maximize throughput, in critical applications. Much as it is impossible to explicitly control when garbage collection is triggered in the JVM, it is possible to influence how it occurs as regards the objects we have created.

Java provides us with reference objects to control the relationship between the objects we create and the garbage collector.

By default, every object we create in a Java program is strongly referenced by a variable:

StringBuilder sb = new StringBuilder();

In the above snippet, the new keyword creates a new StringBuilder object and stores it on the heap. The variable sb then stores a strong reference to this object. What this me for the garbage collector is that the particular StringBuilder object is not eligible for collection at all due to a strong reference held to it by sb. The story only changes when we nullify sb like this:

sb = null;

After calling the above line, the object will then be eligible for collection.

We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside java.lang.ref package.

A soft reference can be created to the above object like this:

StringBuilder sb = new StringBuilder();

SoftReference sbRef = new SoftReference<>(sb);

```
sb = null;
```

In the above snippet, we have created two references to the StringBuilder object. The first line creates a strong reference sb and the second line creates a soft reference sbRef. The third line should make the object eligible for collection but the garbage collector will postpone collecting it because of sbRef.

The story will only change when memory becomes tight and the JVM is on the brink of throwing an OutOfMemory error. In other words, objects with only soft references are collected as a last resort to recover memory.

A weak reference can be created in a similar manner using Weak Reference class. When sb is set to null and the StringBuilder object only has a weak reference, the JVM's garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

A phantom reference is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was collected.

Q8. 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.

Q9. What Happens When There Is Not Enough Heap Space To Accommodate Storage Of New Objects?

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws OutOfMemoryError or more specifically java.lang.OutOfMemoryError heap space.

Q10. Suppose We Have A Circular Reference (two Objects That Reference Each Other). Could Such Pair Of Objects Become Eligible For Garbage Collection And Why?

Yes, a pair of objects with a circular reference can become eligible for garbage collection. This is because of how Java's garbage collector handles circular references. It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static field). If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

Q11. How Do You Trigger Garbage Collection From Java Code?

You, as Java programmer, can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

Before removing an object from memory garbage collection thread invokes finalize()method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

Additionally, there are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it's not guaranteed that garbage collection will happen.

Q12. 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.
- An in-use object, or a referenced object, me that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.
- The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deal location from us so that we can focus on solving the problem at hand.

---- >> Related Posts Of Above Questions :::

------>>[MOST IMPORTANT]<<------

1. 300+ [LATEST] Java Garbage Collection Interview Questions and Answers
2. 250+ TOP MCQs on Memory Management and Answers
3. 300+ [LATEST] Java String Interview Questions and Answers
4. 300+ [LATEST] Java.lang Package Interview Questions and Answers
5. 250+ TOP MCQs on Performance Navigation and Memory and Answers
6. 300+ [LATEST] Ericsson Java Interview Questions and Answers
7. 300+ [LATEST] Aricent Java Interview Questions and Answers
8. 300+ [LATEST] Java Equals And Hashcode Interview Questions and Answers
9. 300+ [LATEST] Java Hadoop Developer Interview Questions and Answers
10. 300+ [MOSK ASKED] Birlasoft Java Interview Questions and Answers
11. 300+ [LATEST] Java J2ee Technical Support Engineer Interview Questions and Answers
12. 250+ TOP MCQs on Memory Allocation of Object and Answers
13. 300+ [MOSK ASKED] Deloitte Java Interview Questions and Answers
14. 250+ TOP MCQs on Heap and Garbage Collection and Answers
15. 300+ [LATEST] Java Developer Interview Questions and Answers
16. 300+[LATEST] Oracle Memory Management Interview Questions and Answers
17. 300+[LATEST] Python Automation Testing Interview Questions and Answers
18. 300+ [LATEST] Java Management Extensions (JME) Interview Questions and Answers

19. **300+ [LATEST] Java.util Interview Questions and Answers**
20. **300+ [LATEST] Java Collections Interview Questions and Answers**