JAVA GARBAGE COLLECTORS INTERVIEW QUESTIONS

Contents

What is the purpose of garbage collection?

The purpose of garbage collection is to reclaim memory that is no longer being used by a program. In a programming language like Java, which uses automatic memory management, objects are created and stored in memory as the program runs. When an object is no longer needed, the memory it occupies can be reclaimed and used for other purposes.

Garbage collection is an automated process that runs in the background, searching for objects that are no longer being used by the program and freeing the memory they occupy. This helps to prevent memory leaks, which can occur when an object is no longer needed but its memory is not released, and it helps to ensure that a program has sufficient memory available to run effectively.

Overall, the purpose of garbage collection is to make it easier for programmers to manage memory and to improve the performance and reliability of programs by ensuring that memory is used efficiently and released when it is no longer needed.

What does System.gc() and Runtime.gc() methods do?

The System.gc() method and Runtime.gc() method are both used to request that the Java virtual machine (JVM) perform garbage collection. When these methods are called, the JVM will attempt to reclaim memory that is no longer being used by the program by finding and deleting objects that are no longer needed.

The main difference between the two methods is that System.gc() is a static method that is called on the System class, while Runtime.gc() is an instance method that is called on a Runtime object. The Runtime class represents the JVM itself, and you can obtain an instance of it by calling the getRuntime() method.

Both of these methods are provided as a way for a program to request garbage collection, but they do not guarantee that garbage collection will actually be performed. The JVM is free to ignore the request if it determines that it is not necessary or if it would not be beneficial to perform garbage collection at that time.

It is generally not necessary to call these methods in your code, as the JVM will perform garbage collection automatically as needed. However, if you are experiencing memory issues in your program, you may want to consider calling one of these methods to see if it helps to resolve the problem.

Here is a sample Java program that demonstrates how to use the System.gc() and Runtime.gc() methods:

```
public class GarbageCollection {
  public static void main(String[] args) {
    // Create an array of large objects
    Object[] objects = new Object[100];
    for (int i = 0; i < objects.length; i++) {
      objects[i] = new byte[1024 * 1024]; // 1 MB
    }

    // Request garbage collection using System.gc()
    System.out.println("Requesting garbage collection using
System.gc()...");
    System.gc();

    // Request garbage collection using Runtime.gc()
    Runtime runtime = Runtime.getRuntime();
    System.out.println("Requesting garbage collection using
Runtime.gc()...");
    runtime.gc();
  }
}
```

In this example, we create an array of large objects and then request garbage collection using both System.gc() and Runtime.gc(). These methods are called after the large objects are no longer needed, and they will cause the JVM to attempt to reclaim the memory that they occupy.

Keep in mind that calling System.gc() or Runtime.gc() does not guarantee that garbage collection will actually be performed. The JVM is free to ignore the request if it determines that it is not necessary or if it would not be beneficial to perform garbage collection at that time.

When is the finalize() called?

The finalize() method is a special method in Java that is called by the garbage collector just before an object is garbage collected. It is defined in the Object class, which is the superclass of all classes in Java, and it can be overridden in a subclass to perform any necessary cleanup before the object is discarded.

The purpose of finalization is to give an object an opportunity to perform any necessary cleanup before it is garbage collected. This can be useful if an object has resources that need to be released, such as file handles or network connections. By overriding the finalize() method, you can specify code that will be executed just before the object is garbage collected, which can help to prevent resource leaks.

It is important to note that the finalize() method is not guaranteed to be called. The garbage collector is free to skip the finalization process if it determines that it is not necessary or if it would not be beneficial to perform finalization at that time. Therefore, you should not rely on the finalize() method for critical cleanup tasks. Instead, you should use other techniques, such as try-with-resources statements or explicit cleanup methods, to ensure that resources are released in a timely and reliable manner.

```
public class ReferenceObject
{
	public void finalize()
	{
		System.out.println("object is garbage collected");

	}
}
```

If an object reference is set to null, will the Garbage Collector immediately free the memory held by that object?

Setting an object reference to null does not necessarily cause the garbage collector to immediately reclaim the memory occupied by the object. The garbage collector operates on its own schedule and is free to reclaim memory as it sees fit, regardless of the state of object references.

However, setting an object reference to null does make it easier for the garbage collector to identify and reclaim objects that are no longer needed. When an object is no longer reachable from any live object references, it is considered to be eligible for garbage collection. By setting an object reference to null, you can make it more likely that the object will be considered eligible for garbage collection and that its memory will be reclaimed.

Ultimately, the garbage collector is responsible for deciding when to reclaim the memory occupied by an object. You cannot directly control when an object is garbage collected, and you should not rely on the garbage collector to reclaim memory in a specific time frame. Instead, you should focus on using object references appropriately and releasing resources when they are no longer needed to ensure that your program uses memory efficiently.

What is structure of Java Heap?

The JVM has a heap that is the runtime data area from which memory for all class instances and arrays is allocated. It is created at the JVM start-up. Heap memory for objects is reclaimed by an automatic memory management system which is known as a garbage collector. Heap memory consists of live and dead objects. Live objects are accessible by the application and will not be a subject of garbage collection. Dead objects are those which will never be accessible by the application, but have not been collected by the garbage collector yet. Such objects occupy the heap memory space until they are eventually collected by the garbage collector.

The Java heap is the portion of memory that is used by the Java virtual machine (JVM) to store objects during the execution of a Java program. The heap is divided into two main regions: the young generation and the old generation.

The young generation is further divided into three subregions: the Eden space and two survivor spaces. The Eden space is where new objects are initially allocated, and the survivor spaces are used to store objects that have survived at least one garbage collection.

The old generation is used to store long-lived objects that have survived multiple garbage collections in the young generation.

The structure of the Java heap is designed to support efficient garbage collection. Objects in the young generation are subject to frequent garbage collection, while objects in the old generation are collected less frequently. This helps to ensure that the garbage collector can run efficiently and that the heap is used efficiently.

The size of the Java heap and the ratio of the young generation to the old generation can be configured using command-line options when the JVM is started. The specific options and their meanings depend on the particular JVM implementation that you are using.

What is the difference between Serial and Throughput Garbage collector?

The serial garbage collector and the throughput garbage collector are two different garbage collection algorithms that are provided by the Java virtual machine (JVM).

The serial garbage collector is designed for use in single-threaded environments, where it can run in the same thread as the application. It is a simple, stop-the-world garbage collector that performs a full garbage collection of the entire heap on each collection cycle. It is a good choice for small heaps and applications with low memory allocation rates.

The throughput garbage collector, on the other hand, is designed for use in multi-threaded environments, where it can run in parallel with the application. It is a more complex garbage collector that divides the heap into multiple regions and uses multiple threads to perform garbage collection concurrently with the application. It is a good choice for larger heaps and applications with high memory allocation rates.

Overall, the main difference between the serial garbage collector and the throughput garbage collector is the way they are designed to run and the types of environments they are best suited for. The serial garbage collector is simple and efficient for small, single-threaded applications, while the throughput garbage collector is more complex and efficient for large, multi-threaded applications.

When does an Object becomes eligible for Garbage collection in Java?

In Java, an object becomes eligible for garbage collection when it is no longer reachable from any live object references. When an object is no longer reachable, it means that there is no way for the program to access the object or interact with it in any way.

For example, consider the following code:

```
MyObject obj = new MyObject();
obj = null;
```

In this code, the MyObject object is created and assigned to a reference obj. Then, the obj reference is set to null. At this point, the MyObject object is no longer reachable from any live object references, and it becomes eligible for garbage collection.

It is important to note that just because an object becomes eligible for garbage collection does not necessarily mean that it will be garbage collected immediately. The garbage collector operates on its own schedule and is free to reclaim memory as it sees fit.

Overall, the key to making objects eligible for garbage collection is to ensure that they are no longer needed by the program and to release any object references to them when they are no longer needed. This will help to ensure that the garbage collector can reclaim the memory occupied by these objects and that the heap is used efficiently.

Does Garbage collection occur in permanent generation space in JVM?

In the Java virtual machine (JVM), the permanent generation (also known as the permgen) is a separate area of memory that is used to store class and method objects. It is separate from the Java heap, which is used to store objects created by the application.

Garbage collection does not occur in the permanent generation. Instead, class and method objects are permanently stored in the permgen until the JVM is shut down. This is because these objects are essential to the operation of the JVM and are needed throughout the lifetime of the application.

The size of the permgen can be configured using command-line options when the JVM is started. If the permgen is not large enough to store all of the class and method objects that are needed by the application, an OutOfMemoryError will be thrown.

Overall, the permanent generation is an important part of the JVM, but it is not subject to garbage collection like the Java heap is. Class and method objects are stored in the permgen for the lifetime of the JVM and are not reclaimed until the JVM is shut down.