(/)

Memory Management in Java Interview Questions (+Answers)

Last modified: June 4, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) **+**

Interview (https://www.baeldung.com/tag/interview/)

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-start)

This article is part of a series:

1. Introduction

In this article, we'll explore some memory management questions that frequently pop up during Java developer interviews. Memory management is an area that not so many developers are familiar with.

In fact, developers don't generally have to deal with this concept directly – as the JVM takes care of the nitty-gritty details. Unless something is going seriously wrong, even seasoned developers may not have accurate information about memory management at their fingertips.

On the other hand, these concepts are actually quite prevalent in interviews – so let's jump right in.

2. Questions

Q1. What does the statement "memory is managed in Java" mean?

Memory is the key resource an application requires to run effectively and like any resource, it is scarce. As such, its allocation and deallocation to and from applications or different parts of an application require a lot of care and consideration.

However, in Java, a developer does not need to explicitly allocate and deallocate memory – the JVM and more specifically the Garbage Collector – has the duty of handling memory allocation so that the developer doesn't have to.

This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

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.

An in-use object, or a referenced object, means 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/deallocation from us so that we can focus on solving the problem at hand.

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.

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.

Q4. What is the meaning of the term "stop-the-world"?

When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete.

Depending on the needs of an application, "stop the world" garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

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.

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 S1. 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 S1. 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 cleans up and compacts that space. For each major GC, there are several minor GCs.

Q8. 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.

Q9. 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.

Q10. 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.**

Q11. Is it possible to «resurrect» an object that became eligible for garbage collection?

When an object becomes eligible for garbage collection, the GC has to run the *finalize* method on it. The *finalize* method is guaranteed to run only once, thus the GC flags the object as finalized and gives it a rest until the next cycle.

In the *finalize* method you can technically "resurrect" an object, for example, by assigning it to a *static* field. The object would become alive again and non-eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be marked as finalized, so when it would become eligible again, the finalize method would not be called. In essence, you can turn this "resurrection" trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

Q12. 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 means 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<StringBuilder> 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 *WeakReference* 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.

Q13. 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.

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.

Q15. 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 from *StringBuffer?*

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.

3. Conclusion

In this article, we have covered some of the most common questions that frequently appear in Java engineer interviews. Questions about memory management are mostly asked for Senior Java Developer candidates as the interviewer expects that you have built non-trivial applications which are, a lot of times, plagued by memory issues.

This should not be treated as an exhaustive list of questions, but rather a launch pad for further research. We, at Baeldung, wish you success in any upcoming interviews.

Next »
Java Generics Interview Questions (+Answers) (https://www.baeldung.com/java-generics-interview-questions)

« Previous
Java 8 Interview Questions(+ Answers) (https://www.baeldung.com/java-8-interview-questions)

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to "Build your API **with Spring**"?

Enter your email address

>> Get the eBook

newest oldest most voted

PRIYANKA SONAWANE

Guest

Indeed a great blog on Java memory.

5

1 year ago

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)