



/ DevOps Zone (/devops-tutorials-tools-news)

Over a million developers have joined DZone. Sign In / Join



JVM and Garbage Collection Interview

REFCARDZ (/REFCARDZ) GUIDES (/GUIDES) ZONES (/PORTALS) | AGILE (/AGILE-METHODOLOGY-TRAINING-TOOLS-NEWS)

Questions: The Beginners Guide



(/users/1338295/samberic.html) by Sam Atkinson (/users/1338295/samberic.html) MVB · Dec. 08, 14 · DevOps

Zone (/devops-tutorials-tools-news)

Like (0) Comment (0) Save Tweet

20.38k Views

The DevOps Zone is brought to you by DZone in partnership with Hewlett Packard Enterprise DevOps (/go?i=99831&u=http%3A%2F%2Fwww8.hp.com%2Fus%2Fen%2Fsoftware-solutions%2Fdevops-solutions%2Findex.html%3Foriginid%3D510023039%26jumpid%3Dba_7z1i3g6vre_AID-510023039). Learn where you are in your DevOps journey by taking the HPE/Forrester benchmarking tool (/go?i=99831&u=http%3A%2F%2Fwww8.hp.com%2Fus%2Fen%2Fsoftware-solutions%2Fdevops-solutions%2Findex.html%3Foriginid%3D510023039%26jumpid%3Dba_7z1i3g6vre_AID-510023039).

The Java Virtual Machine is the achilles heel of most developers and can cause even the most seasoned developers to be come unstuck. The simple fact is that unless something is going wrong, we don't normally care about it. Maybe we tune it a little when the application goes live but after that it remains untouched until something goes wrong. This makes it a very difficult subject to excel in during interviews. Even worse, interviewers love to ask questions about it. Everyone should have a basic knowledge of the JVM to be able to do their job but often people recruiting are looking for someone who knows how to fix a problem

» like a memory leak when it happens.

In this guide we take a ground up approach to the JVM and garbage collection so you can feel some level of confidence going into your big day.

Java Question: What is the JVM? Why is it a good thing? What is “write once, run anywhere”? Are there negatives?

JVM stands for Java Virtual Machine. Java code is compiled down into an intermediary language called byte code. The Java Virtual Machine is then responsible for executing this byte code. This is unlike languages such as C++ which are compiled directly to native code for a specific platform.

This is what gives Java its ‘Write once, run anywhere’ ability. In a language which compiles directly to platform you would have to compile and test the application separately on every platform on which you wish it to run. There would likely be several issues with libraries, ensuring they are available on all of the platforms for example. Every new platform would require new compilation and new testing. This is time consuming and expensive.

On the other hand a java program can be run on any system where a Java Virtual Machine is available. The JVM acts as the intermediary layer and handles the OS specific details which means that as developers we shouldn't need to worry about it. In reality there are still some kinks between different operating systems, but these are relatively small. This makes it quicker and easier to develop, and means developers can write software on windows laptops that may be destined for other platforms. I only need to write my software once and it is available on a huge variety of platforms, from android to Solaris.

In theory, this is at the cost of speed. The extra layer of the JVM means it is slower than direct-to-tin languages like C. However java has been making a lot of progress in recent years, and given the many

languages like C. However Java has been making a lot of progress in recent years, and given the many other benefits such as ease of use, it is being used more and more often for low latency applications.

The other benefit of the JVM is that any language that can compile down to byte code can run on it, not just Java. Languages like Groovy, Scala and Clojure are all JVM based languages. This also means the languages can easily use libraries written in other languages. As a Scala developer I can use Java libraries in my applications as it all runs on the same platform.

The separation from the real hardware also means the code is sandboxed, limiting the amount of damage it can do to a host computer. Security is a great benefit of the JVM.

There is another interesting facet to consider; not all JVMs are built equal. There are a number of different implementations beyond the standard JVM implementation from Oracle. JRockit (<http://www.oracle.com/technetwork/middleware/jrockit/overview/index-101826.html>) is renowned for being an exceptionally quick JVM. OpenJDK (<http://openjdk.java.net/>) is an open source equivalent. There are tons of JVM implementations available. Whilst this choice is ultimately a good thing, all of the JVMs may behave slightly differently. A number of areas of the Java specification are left intentionally vague with regards to their implementation and each VM may do things differently. This can result in a bug which only manifests in a certain VM in a certain platform. These can be some of the hardest bugs to figure out.

From a developer perspective, the JVM offers a number of benefits, specifically around memory management and performance optimisation.

Java Interview Question: What is JIT?

JIT stands for “Just in Time”. As discussed, the JVM executes bytecode. However, if it determines a section of code is being run frequently it can optionally compile a section down to native code to increase the speed of execution. The smallest block that can be JIT compiled is a method. By default, a piece of code needs to be executed 1500 times for it to be JIT compiled although this is configurable. This leads to the concept of “warming up” the JVM. It will be at its most performant the longer it runs as these optimisations occur. On the downside, JIT compilation is not free; it takes time and resource when it occurs.

Java Garbage Collection Interview Questions

Java Interview Question: What do we mean when we say memory is managed in Java? What is the Garbage Collector?

In languages like C the developer has direct access to memory. The code literally references memory space addresses. This can be difficult and dangerous, and can result in damaging memory leaks. In Java on the other hand all memory is managed. As a programmer we deal exclusively in objects and primitives and have no concept of what is happening underneath with regards to memory and pointers. Most importantly, Java has the concept of a garbage collector. When objects are no longer needed the JVM will automatically identify and clear the memory space for us.

Java Interview Question: What are the benefits and negatives of the Garbage Collector?

On the positive side:

- The developer can worry much less about memory management and concentrate on actual problem solving. Although memory leaks are still technically possible they are much less common.
- The GC has a lot of smart algorithms for memory management which work automatically in the background. Contrary to popular belief, these can often be better at determining when best to perform GC than when collecting manually.

On the negative side

- When a garbage collection occurs it has an effect on the application performance, notably slowing it down or stopping it. In so called “Stop the world” garbage collections the rest of the application will freeze whilst this occurs. This can be unacceptable depending on the application requirements, although GC tuning can minimise or even remove the impact.
- Although it’s possible to do a lot of tuning with the garbage collector, you cannot specify when or how the application performs GC.

Java Interview Question: What is “Stop the World”?

When a GC happens it is necessary to completely pause the threads in an application whilst collection occurs. This is known as Stop The World. For most applications long pauses are not acceptable. As a result it is important to tune the garbage collector to minimise the impact of collections to be acceptable for the application.

Java Interview Question: How does Generational GC work? Why do we use generational GC? How is the Java Heap structured?

It is important to understand how the Java Heap works to be able to answer questions about GC. All objects are stored on the Heap (as opposed to the Stack, where variables and methods are stored along with references to objects in the heap). Garbage Collection is the process of removing Objects which are no longer needed from the Heap and returning the space for general consumption. Almost all GCs are “generational”, where the Heap is divided into a number of sections, or generations. This has proven significantly more optimal which is why almost all collectors use this pattern.

New Generation

Most applications have a high volume of short lived objects. Analyzing all objects in an application during a GC would be slow and time consuming, so it therefore makes sense to separate the shortlived objects so that they can be quickly collected. As a result all new Objects are placed into the new generation. New gen is split up further:

- Eden Space: all new objects are placed in here. When it becomes full, a minor GC occurs. all objects that are still referenced are then promoted to a **survivor space**
- Survivor spaces: The implementation of survivor spaces varies based on the JVM but the premise is the same. Each GC of the New Generation increments the age of objects in the survivor space. When an object has survived a sufficient number of minor GCs (Defaults vary but normally start at 15) it will then be promoted to the Old Generation. Some implementations use two survivor spaces, a From space and a To space. During each collection these will swap roles, with all promoted Eden objects and surviving objects move to the To space, leaving From empty.

A GC in the NewGen is known as a **minor GC**.

One of the benefits of using a New Generation is the reduction of the impact of fragmentation. When an Object is garbage collected, it leaves a gap in the memory where it was. We can compact the remaining Objects (a stop-the-world scenario) or we can leave them and slot new Objects in. By having a Generational GC we limit the amount that this happens in the Old Generation as it is generally more stable which is good for improving latencies by reducing stop the world. However if we do not compact we may find Objects cannot just fit in the spaces inbetween, perhaps due to size concerns. If this is the case then you will see objects failing to be promoted from New Generation.

Old Generation

Any objects that survive from survivor spaces in the New Generation are promoted to the Old Generation.

The Old Generation is usually much larger than the New Generation. When a GC occurs in old gen it is known as a **full GC**. Full GCs are also stop-the-world and tend to take longer, which is why most JVM tuning occurs here. There are a number of different Algorithms available for Garbage Collection, and it is possible to use different algorithms for new and old gen.

Serial GC

Designed when computers only had one CPU and stops the entire application whilst GC occurs. It uses **mark-sweep-compact**. This means it goes through all of the objects and marks which objects are available for Garbage Collection, before clearing them out and then copying all of the objects into contiguous space (so therefore has no fragmentation).

Parallel GC

Similar to Serial, except that it uses multiple threads to perform the GC so should be faster.

Concurrent Mark Sweep

CMS GC minimises pauses by doing most of the GC related work concurrently with the processing of the application. This minimises the amount of time when the application has to completely pause and so lends itself much better to applications which are sensitive to this. CMS is a non compacting algorithm which can lead to fragmentation problems. The CMS collector actually uses Parallel GC for the young generation.

G1GC (garbage first garbage collector)

A concurrent parallel collector that is viewed as the long term replacement for CMS and does not suffer from the same fragmentation problems as CMS.

PermGen

The PermGen is where the JVM stores the metadata about classes. It no longer exists in Java 8, having been replaced with metaspace. Generally the PermGen doesn't require any tuning above ensuring it has enough space, although it is possible to have leaks if Classes are not being unloaded properly.

Java Interview Question: Which is better? Serial, Parallel or CMS?

It depends entirely on the application. Each one is tailored to the requirements of the application. Serial is better if you're on a single CPU, or in a scenario where there are more VMs running on the machine than CPUs. Parallel is a throughput collector and really good if you have a lot of work to do but you're ok with pauses. CMS is the best of the three if you need consistent responsiveness with minimal pauses.

From the code

Java Interview Question: Can you tell the system to perform a garbage collection?

This is an interesting question. The answer is both yes and no. We can use the call "System.gc()" to suggest to the JVM to perform a garbage collection. However, there is no guarantee this will do anything. As a java developer, we don't know for certain what JVM our code is being run in. The JVM spec makes no guarantees on what will happen when this method is called. There is even a startup flag, -XX:+DisableExplicitGC, which will stop this from doing anything.

It is considered bad practice to use System.gc().

Java Interview Question: What does finalize() do?

finalize() is a method on java.lang.Object so exists on all Objects. The default implementation does nothing. It is called by the garbage collector when it determines there are no more references to the object.

As a result there are no guarantees the code will ever be executed and so should not be used to execute actual functionality. Instead it is used for clean up, such as file references. It will never be called more than once on an object (by the JVM).

Tuning

Java Interview Question: What flags can I use to tune the JVM and GC?

There are textbooks available on tuning the JVM for optimal Garbage Collection. Nonetheless it's good to know a few for the purpose of interview.

-XX:-UseConcMarkSweepGC: Use the CMS collector for the old gen.

-XX:-UseParallelGC: Use Parallel GC for New Gen

-XX:-UseParallelOldGC: Use Parallel GC for Old and New Gen.

-XX:-HeapDumpOnOutOfMemoryError: Create a thread dump when the application runs out of memory. Very useful for diagnostics.

-XX:-PrintGCDetails: Log out details of Garbage Collection.

-Xms512m: Sets the initial heap size to 512m

-Xmx1024m: Sets the maximum heap size to 1024m

-XX:NewSize and -XX:MaxNewSize: Specifically set the default and max size of the New Generation

- XX:NewRatio=3: Set the size of the Young Generation as a ratio of the size of the Old Generation.

-XX:SurvivorRatio=10: Set the size of Eden space relative to the size of a survivor space.

Diagnosis

Whilst all of the questions above are very good to know to show you have a basic understanding of how the JVM works, one of the most standard questions during an interview is this: **“Have you ever experience a memory leak? How did you diagnose it?”**. This is a difficult question to answer for most people as although they may have done it, chances are it was a long time ago and isn't something you've done recently. The best way to prepare is to actually try and write an application with a memory leak and attempt to diagnosis it. Below I have created a ridiculous example of a memory leak which will allow us to go step by step through the process of identifying the problem. **I strongly advise you download the code and follow through this process.** It is much more likely to be committed to your memory if you actually do this process.

```
1
public class Main {

2
    public static void main(String[] args) {

3
        TaskList taskList = new TaskList();

4
        final TaskCreator taskCreator = new TaskCreator(taskList);

5
```

6

```
new Thread(new Runnable() {
```

7

```
@Override
```

8

```
public void run() {
```

9

```
for (int i = 0; i < 100000; i++) {
```

10

```
taskCreator.createTask();
```

11

```
}
```

12

```
}
```

13

```
}).start();
```

14

```
}
```

15

```
}
```

16

```
public class TaskList {
```

17

```
private static Deque<Task> tasks = new ArrayDeque<Task>();
```

18

19

```
public void addTask(Task task){
```

20

```
tasks.add(task);
```

21

```
tasks.peek().execute();//Memory leak!
```

22

```
}
```

23

```
}
```

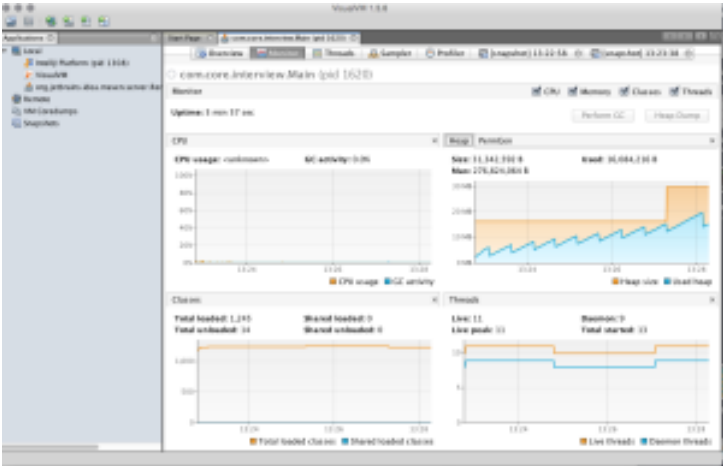

In the above very contrived example, the application executes tasks put onto a Deque. However when we run this we get an out of memory! What could it possibly be?

```
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
at com.core.interview.Task.<init>(Task.java:8)
at com.core.interview.TaskCreator.createTask(TaskCreator.java:13)
at com.core.interview.Main$.run(Main.java:13)
at java.lang.Thread.run(Thread.java:722)
```

(<http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/Memory-Leak.png>)

To find out we need to use a **profiler**. A profiler allows us to look at exactly what is going on the VM. There are a number of options available. VisualVM (<https://visualvm.java.net/download.html>) is free and allows basic profiling. For a more complete tool suite there are a number of options but my personal favourite is Yourkit (<http://www.yourkit.com/overview/>). It has an amazing array of tools to help you with diagnosis and analysis. However the principles used are generally the same.

I started running my application locally, then fired up VisualVM and selected the process. You can then watch exactly what’s going on in the heap, permgen etc.



(<http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/visual-vm.png>)

You can see on the heap (top right) the tell tail signs of a memory leak. The application sawtooths, which is not a problem per se, but the memory is consistently going up and not returning to a base level. This smells like a memory leak. But how can we tell what’s going on? If we head over to the Sampler tab we can get a clear indication of what is sitting on our heap.

Class Name – Allocated Objects	Bytes Allocated [%] ▼	Bytes Allocated	Objects Allocated
java.lang.Object[]		1,586,944 B (26.8%)	7,584 (9.6%)
char[]		950,568 B (16%)	11,509 (14.6%)
byte[]		847,208 B (14.3%)	5,311 (6.7%)
int[]		592,904 B (10%)	3,218 (4.1%)
java.lang.String		195,600 B (3.3%)	8,150 (10.3%)
java.lang.Class		166,704 B (2.8%)	1,383 (1.8%)
java.io.ObjectStreamClass\$WeakClassKey		146,624 B (2.5%)	4,582 (5.8%)
java.util.TreeMap\$Entry		132,560 B (2.2%)	3,314 (4.2%)

(<http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/java-heap-snapshot.png>)

Those Object arrays look a bit odd. But how do we know if that’s the problem? Visual VM allows us to take **snapshots**, like a photograph of the memory at that time. The above screenshot is a snapshot from after the application had only been running for a little bit. The next snapshot a couple of minutes later confirms this:

Class Name – Allocated Objects	Bytes Allocated [%]	Bytes Allocated	Objects Allocated
java.lang.Object[]		2,910,952 B (59.2%)	1,944 (5%)
char[]		385,184 B (7.8%)	5,519 (14.2%)
byte[]		286,208 B (5.8%)	1,577 (4%)
java.lang.Class		167,600 B (3.4%)	1,391 (3.6%)
int[]		166,504 B (3.4%)	2,621 (6.7%)
java.lang.String		130,008 B (2.6%)	5,417 (13.9%)
short[]		109,104 B (2.2%)	1,858 (4.8%)
int[][]		108,080 B (2.2%)	2,097 (5.4%)
java.lang.reflect.Method		90,800 B (1.8%)	1,135 (2.9%)
java.util.HashMap\$Entry[]		41,360 B (0.8%)	465 (1.2%)

(http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/java-heap-snapshot-2.png)

We can actually compare these directly by selecting both in the menu and selecting compare.

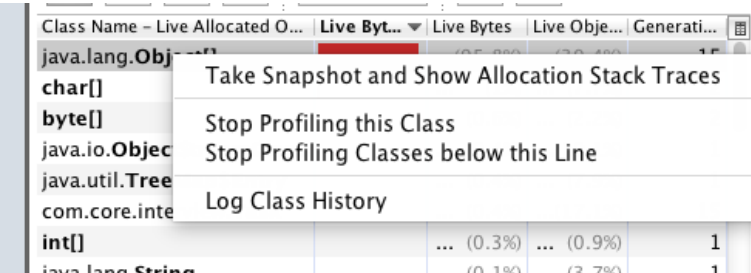


(http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/compare-heap-snapshots.png)

(http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/compare-items-from-heap-snapshots.png)

java.lang.Object[]	-967,880 B	-179	-2
--------------------	------------	------	----

There’s definitely something funky going on with the array of objects. How can we figure out the leak though? By using the profile tab. If I go to profile, and in settings enable “record allocations stack traces” we can then find out where the leak has come from.



Profile object allocations and GC

Track every 10 object allocations

☒ Record allocations stack traces

(http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/visual-vm-profile.png)

By now taking snapshot and showing allocation traces we can see where the object arrays are being instantiated.

Method Name – Allocation Call Tree	Live Bytes	Live Objects	Allocated Obj...	Avg. Age	Generations
java.lang.Object[]	11,000 (100%)	7,900 (100%)	20,558	2.3	15
com.core.interview.Task.<init> ()	11,000 (98.5%)	2,900 (36.6%)	2,926	6.4	15
java.io.ObjectOutputStream.defaultWriteObject ()	65,000 (0.5%)	3,500 (44.7%)	12,442	0.0	1
java.io.ObjectOutputStream\$Handle.writeObject ()	57,000 (0.5%)	66 (0.8%)	195	0.0	1
com.sun.jmx.mbeanserver.DefaultMXBean\$Stub.writeObject ()	17,000 (0.1%)	325 (4.1%)	1,132	0.0	1
java.io.ObjectStreamClass.invokeWriteObject ()	9,400 (0.1%)	392 (4.9%)	1,435	0.0	1
java.io.ObjectOutputStream\$Handle.writeObject ()	7,600 (0.1%)	136 (1.7%)	443	0.0	1
java.io.ObjectInputStream\$Handle.readObject ()	4,400 (0%)	80 (1%)	261	0.0	1
java.io.ObjectOutputStream\$Replaceable.writeObject ()	3,600 (0%)	66 (0.8%)	223	0.0	1
java.util.ArrayList.<init> (int)	3,600 (0%)	68 (0.9%)	238	0.0	1

(http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/memory-leak-source.png)

Looks like there are thousands of Task objects holding references to Object arrays! But what is holding

onto these Task items?

If we go back to the “Monitor” tab we can create a heap dump. If we double click on the Object[] in the heap dump it will show us **all instances** in the application, and in the bottom right panel we can identify where the reference is.

The screenshot shows the Eclipse IDE's Monitor tab with a heap dump analysis for `java.lang.Object[]`. The top bar indicates 41,929 instances and a total size of 330,685,104 bytes. The left pane shows a list of instances, with the first instance (#33558) selected, showing a size of 524,312 bytes. The right pane shows the fields of the selected instance, including an array of `TaskList` objects. The bottom right pane shows the references, highlighting the `tasks` field of type `TaskList`.

Instance	Size
<500 instances>	
#33558 65,536 items	524,312
#30 1,024 items	8,216
#56 1,000 items	8,024
#57 1,000 items	8,024
#58 1,000 items	8,024
#59 1,000 items	8,024
#60 1,000 items	8,024
#61 1,000 items	8,024
#62 1,000 items	8,024
#63 1,000 items	8,024
#64 1,000 items	8,024
#65 1,000 items	8,024

Field	Type	Value
this	Object[]	#33558 65,536 items
<items 0-499>	Object	(500 items)
<items 500-999>	Object	(500 items)
<items 1,000-1,499>	Object	(500 items)
<items 1,500-1,999>	Object	(500 items)
<items 2,000-2,499>	Object	(500 items)
<items 2,500-2,999>	Object	(500 items)
<items 3,000-3,499>	Object	(500 items)
<items 3,500-3,999>	Object	(500 items)

Field	Type	Value
this	Object[]	#33558 65,536 items
elements	ArrayDeque	#33
tasks	TaskList	class TaskList

(<http://www.corejavainterviewquestions.com/wp-content/uploads/2014/11/memory-leak-object-references.png>)

It looks like TaskList is the culprit! If we take a look at the code we can see what the problem is.

1

```
tasks.peek().execute();
```

We’re never clearing the reference after we’ve finished with it! If we change this to use `poll()` then the memory leak is fixed.

Whilst clearly this is a very contrived example, going through the steps will refresh your memory for if you are asked to explain how you would identify memory leaks in an application. Look for memory continuing to increase despite GCs happening, take memory snapshot and compare them to see which Objects may be candidates for not being released, and use a heap dump to analyze what is holding references to them.

The DevOps Zone is brought to you in partnership with Hewlett Packard Enterprise (/go?

i=99832&u=http%3A%2F%2Fwww8.hp.com%2Fus%2Fen%2Fsoftware-solutions%2Fasset%2Fsoftware-asset-viewer.html%3Foriginid%3D510023039%26asset%3D2002072%26module%3D1822258%26docname%3D4AA5-8744ENW%26page%3D1822263%26jumpid%3Dba_gzs7umvtq1). Download Through the DevOps Looking Glass: Learnings from HP's Own Transformation Initiative (/go?

i=99832&u=http%3A%2F%2Fwww8.hp.com%2Fus%2Fen%2Fsoftware-solutions%2Fasset%2Fsoftware-asset-viewer.html%3Foriginid%3D510023039%26asset%3D2002072%26module%3D1822258%26docname%3D4AA5-8744ENW%26page%3D1822263%26jumpid%3Dba_gzs7umvtq1) to get valuable best practices and “lessons learned” straight from the HP IT team to help you start or progress along your own DevOps Journey.

Topics: JAVA, DEVOPS, JVM

Published at DZone with permission of Sam Atkinson , DZone MVB . <http://www.corejavainterviewquestions.com/java-garbage-collection-interview-questions/>

Opinions expressed by DZone contributors are their own.