# What are some Java memory management best practices? [closed]

Ask Question

I am taking over some applications from a previous developer. When I run the applications through Eclipse, I see the memory usage and the heap size increase a lot. Upon further investigation, I see that they were creating an object over-and-over in a loop as well as other things.

I started to go through and do some clean up. But the more I went through, the more questions I had like "will this actually do anything?"

For example, instead of declaring a variable outside the loop mentioned above and just setting its value in the loop... they created the object in the loop. What I mean is:

```
for(int i=0; i < arrayOfStuff.size(); i++) {
    String something = (String) arrayOfStuff.get(i);
    ...
}
```

versus

```
String something = null;
for(int i=0; i < arrayOfStuff.size(); i++) {
    something = (String) arrayOfStuff.get(i);
}
```

Am I incorrect to say that the bottom loop is better? Perhaps I am wrong.

Also, what about after the second loop above, I set "something" back to null? Would that clear out some memory?

In either case, what are some good memory management best practices I could follow that will help keep my memory usage low in my applications?

**Update:**

I appreciate everyones feedback so far. However, I was not really asking about the above loops (although by your advice I did go back to the first loop). I am trying to get some best practices that I can keep an eye out for. Something on the lines of "when you are done using a Collection, clear it out". I just really need to make sure not as much memory is being taken up by these applications.

java memory-management

edited Mar 26 '10 at 8:40
cherouvim
25.7k 11 86 132

asked Mar 9 '09 at 20:00
Ascalonian
7,820 12 56 91

**closed** as primarily opinion-based by Wai Ha Lee, Uwe Plonus, sdfx, ale, Rostyslav Dzinko Feb 16 at 19:00

Many good questions generate some degree of opinion based on expert experience, but answers to this question will tend to be almost entirely based on opinions, rather than facts, references, or specific expertise.

If this question can be reworded to fit the rules in the help center, please edit the question.

Re your edit: As three people have said so far, profile it! – Michael Myers ♦ Mar 9 '09 at 20:17

I am looking for some specific practices to use so I don't have to have someone profile it. I would rather develop it properly in the first place than to code, hope it works, profile it, and make fixes. – Ascalonian Mar 9 '09 at 20:21

2 The "specific practice" is to write well structured code without doing premature optimization, and then profiling it to find out what needs optimizing. – Esko Luontola Mar 9 '09 at 22:07

So what are the practices to avoid doing premature optimization? – Ascalonian Mar 10 '09 at 12:13

1 (Wikipedia) "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" — Donald Knuth – ahoffer Jan 11 '12 at 20:54

## 13 Answers

Don't try to outsmart the VM. The first loop is the suggested best practice, both for performance and maintainability. Setting the reference back to null after the loop will not guarantee immediate memory release. The GC will do its job best when you use the minimum scope possible.

Books which cover these things in detail (from the user's perspective) are Effective Java 2 and Implementation Patterns.

If you care to find out more about performance and the inners of the VM you need to see talks or read books from Brian Goetz.

edited Sep 2 '11 at 7:38

answered Mar 9 '09 at 20:08
cherouvim
**25.7k** 11 86 132

"use minimum scope possible" is it considered a best practice in general? – KillBill Jun 19 '13 at 9:14

I think that generally yes. But of course there should be exceptions to that generalization. – cherouvim Jun 19 '13 at 9:48

---

Those two loops are equivalent except for the scope of `something` ; see this question for details.

General best practices? Umm, let's see: don't store large amounts of data in static variables unless you have a good reason. Remove large objects from collections when you're done with them. And oh yes, "Measure, don't guess." Use a profiler to see where the memory is being allocated.

edited May 23 '17 at 12:18
Community ♦
**1** 1

answered Mar 9 '09 at 20:08
Michael Myers ♦
**146k** 35 250 274

3 +1 solely for "Measure, don't guess." – matt b Mar 10 '09 at 0:02

---

There are no objects created in both of your code samples. You merely set an object reference to a string that is already in the arrayOfStuff. So memorywise there is no difference.

answered Mar 9 '09 at 20:10
Kees de Kooter
**5,041** 2 29 33

---

The two loops will use basically the same amount of memory, any difference would be negligible. "String something" only creates a reference to an object, not a new object in itself and thus any additional memory used is small. Plus, compiler / combined with JVM will likely optimize the generated code anyway.

For memory management practices, you should really try to profile your memory better to understand where the bottlenecks actually are. Look especially for static references that point to a big chunk of memory, since that will never get collected.

You can also look at Weak References , and other specialized memory management classes.

Lastly, keep in mind, that if an application takes up memory, there might be a reason for it....

**Update** The key to memory management is data structures, as well as how much performance you need / when. The tradeoff is often between memory and CPU cycles.

For example, a lot of memory can be occupied by caching, which is specifically there to improve performance since you are trying to avoid an expensive operation.

So think through your data structures and make sure you don't keep things in memory for longer than you have to. If it's a web app, avoid storing a lot of data into the session variable, avoid having static references to huge pools of memory, etc.

edited Mar 9 '09 at 20:27

answered Mar 9 '09 at 20:09
Jean Barmash
**4,300** 26 40

---

The JVM is best at freeing short-lived objects. Try not to allocate objects you don't need. But you can't optimize the memory usage until you understand your workload, the object lifetime, and the object sizes. A profiler can tell you this.

Finally, the #1 thing you must avoid doing: never use Finalizers. Finalizers interfere with garbage collection, since the object can't be just freed but must be queued for finalization, which may or may not occur. It's best to never use finalizers.

As for the memory usage you're seeing in Eclipse, it's not necessarily relevant. The GC will do its job based on how much free memory there is. If you have lots of free memory you might not see a single

GC before the app is shut down. If you find your app running out of memory then only a real profiler can tell you where the leaks or inefficiencies are.

answered Mar 9 '09 at 20:46
Mr. Shiny and New 安宇
**12.4k** 5 33 63

In object-oriented, garbage collected languages, poor performance is often caused by large numbers of long-lived objects, especially if there are many layers of composition. For example, if I create a Spreadsheet object, with a 2D array of Cell objects, where each object contains a Color object, Value object and Formatting object, and each Formatting object contains..... Well, you get the idea. It will take the GC a long time to deal with the object graph. Also the layers of memory indirection cause lag (it does in Smalltalk, anyway. I'm not as sure about Java). – ahoffer Jan 11 '12 at 21:02

The first loop is better. Because

- the variable something will be clear faster (theoretical)
- the program is better to read.

But from point of memory this is irrelevant.

If you have memory problems then you should profile where it is consumed.

answered Mar 9 '09 at 20:10
Horcrux7
**14.2k** 16 67 108

Sorry, please correct me if I'm wrong, but the first loop allocates memory for a new variable on each loop. If the loop runs 100 times, how come it's better creating 100 strings instead of just overriding the same one and and allocating memory for only 1 string (like in the second loop) ? – Mapisto May 9 '17 at 13:04

The memory on the stack for all local variables of a method is allocate on method entry and not on declaring point in code. The size of such variable are 4 bytes. (possible 8 bytes on 64 bit VM). The size for the String is allocated on the heap and not on the stack. This allocation occur on the "new" keyword. There is no difference in the allocation of the strings. – Horcrux7 May 12 '17 at 8:16

In my opinion, you should avoid micro-optimizations like these. They cost a lot of brain cycles, but most of the time have little impact.

Your application probably has a few central data structures. Those are the ones you should be worried about. For example, if you fill them preallocate them with a good estimate of the size, to avoid repeated resizing of the underlying structure. This especially applies to `StringBuffer` , `ArrayList` , `HashMap` and the like. Design your access to those structures well, so you don't have to copy a lot.

Use the proper algorithms to access the data structures. At the lowest level, like the loop you mentioned, use `Iterator` s, or at least avoid calling `.size()` all the time. (Yes, you're asking the list every time around for it's size, which most of the time doesn't change.) BTW, I've often seen a similar mistake with `Map` s. People iterate over the `keySet()` and `get` each value, instead of just iterating over the `entrySet()` in the first place. The memory manager will thank you for the extra CPU cycles.

answered Mar 9 '09 at 22:01
Ronald Blaschke
**3,125** 2 14 15

As one poster above suggested, use profiler to measure the memory (and/or cpu) usage of certain parts of your program rather than trying to guess it. You may be surprised at what you find!

There is an added benefit to that as well. You'll understand about your programming language and your application more.

I use VisualVM for profiling and recommend it greatly. It comes with jdk/jre distribution.

answered Jul 5 '10 at 15:16
riz
**175** 2 8

The first example is fine. There isn't any memory allocation going on there, other than a stack variable allocation and deallocation each time through the loop (very cheap and quick).

The reason is that all that is being 'allocated' is a reference, which is a 4 byte stack variable (on most 32 bit systems anyway). A stack variable is 'allocated' by adding to a memory address representing the top of the stack, and so is very quick and cheap.

What you need to be careful of is for loops like:

```
for (int i = 0; i < some_large_num; i++)
{
   String something = new String();
   //do stuff with something
}
```

as that is actually doing memory allocatiton.

answered Mar 9 '09 at 20:11
workmad3
**19.6k** 3 30 49

There is not even the allocation on the stack for each iteration. These are allocated in the method local variable table (one time) yet the variable automatically goes out-of-scope when the loop ends. – Kevin Brock Mar 25 '10 at 22:00

yeah, but a new String object is created each iteration which is more expensive than a stack allocation. – workmad3 Mar 26 '10 at 15:08

---

If you haven't already, I suggest installing the Eclipse Test & Performance Tools Platform (TPTP). If you want to dump and inspect the heap, check out the SDK jmap and jhat tools. Also see Monitoring and Managing Java SE 6 Platform Applications.

answered Mar 9 '09 at 21:52
McDowell
**90.8k** 22 161 240

1 As TPTP seems well and truly dead by now (5 years later), go for jvisualvm which ships with JDK - and analyze gc logs (tagtraum.com/gcviewer.html or github.com/chewiebug/GCViewer). – Jan Chimiak May 5 '15 at 10:56

---

Well, the first loop is actually better, because the scope of something is smaller. Regarding memory management - it makes not a big difference.

Most Java memory problems come when you store objects in a collection, but forget to remove them. Otherwise the GC makes his job quite good.

answered Mar 9 '09 at 20:10 community wiki
siddhadev

---

From what I understand you are looking at, the bottom loop is no better. The reason being even if you are trying to reuse a single reference (Ex- something), the fact is the object (Ex - arrayOfStuff.get(i)) is still being referenced from the List (arrayOfStuff). To make the objects eligible for collection, they should not be referenced from any place. If you are sure about the life of the List after this point, you may decide to remove/free the objects from it, within a separate loop.

The optimization that can be done from a static perspective (ie no modification is happening to this List from any other thread), it's better to avoid the size() invocation repeatedly. That is if you are not expecting the size to change, then why calculate it again and again; after all it is not an array.length, it is list.size().

answered Jan 3 '13 at 12:21
Narita
**93** 1 12

---

"Am I incorrect to say that the bottom loop is better?", the answer is NO, not only is better, in same case is necessary... The variable definition (not the content), is made in Memory heap, and is limited, in the first example, each loop create an instance in this memory, and if the size of "arrayOfStuff" is big, can occur "Out of memory error:java heap space"....

answered Jan 24 '11 at 15:54
Eduard
**9**