# Understanding Java Memory Management

Ask Question

Java programmers know that JVM runs a Garbage Collector, and System.gc() would just be a suggestion to JVM to run a Garbage Collector. It is not necessarily that if we use System.gc(), it would immediately run the GC.

Please correct me if I misunderstand Java's Garbage Collector.

Is/are there any other way/s doing memory management other than relying on Java's Garbage Collector?
If you intend to answer the question by some sort of programming practice that would help managing the memory, please do so.

java memory-management memory-leaks garbage-collection jvm

asked Dec 13 '12 at 7:28

Michael Ardan
**2,261** 8 23 50

For System.gc() haters, please explain why would you hate calling GC. – Michael Ardan Dec 13 '12 at 7:29

1 Have you read this question? – Andrew Logvinov Dec 13 '12 at 7:32

You should never rely on System.gc() as in some environments it may never be called. You can prevent software from outside of the vm to do that. – sorencito Dec 13 '12 at 7:32

1 I think it boils down to the System.gc() call not being reliable, and not (always) freeing the resources that need to be freed when called. So if you have a tool you can't really rely on there aren't too many reasons for you to actually use it. – omu_negru Dec 13 '12 at 7:33

1 JVM has malloc()'d a certain amount of space that is the heap. The operating system is obligated to have that much memory available to JVM. GC does not by rule always reduce that amount. Depending on how JVM is configured that number will not go below a certain amount no matter how few actual objects are in memory. Modern OS is smart enough to allow other processes to use the real physical memory and include swap space in the calculation of total reserved for JVM, but if you're in a position where this stuff is a concern in the first place, it may matter. – Affe Dec 13 '12 at 7:57

## 10 Answers

The most important thing to remember about Java memory management is "nullify" your reference.

Only objects that are not referenced are to be garbage collected.

For example, objects in the following code is never get collected and your memory will be full just to do nothing.

```
List objs = new ArrayList();
for (int i = 0; i  < Integer.MAX_VALUE; i++) objs.add(new Object());
```

But if you don't reference those object ... you can loop as much as you like without memory problem.

```
List objs = new ArrayList();
for (int i = 0; i  < Integer.MAX_VALUE; i++) new Object();
```

So what ever you do, make sure you remove reference to object to no longer used (set reference to `null` or clear collection).

When the garbage collector will run is best left to JVM to decide. Well unless your program is about to start doing things that use a lot of memory and is speed critical so you may suggest JVM to run GC before going in as you may likely get the garbaged collected and extra memory to go on. Other wise, I personally see no reason to run `System.gc()` .

Hope this help.

answered Dec 13 '12 at 7:38

NawaMan
**16.8k** 7 41 70

Well, not everything. Local variables (as well as method parameters) will be cleared when the method call ends so you don't need to do that. Be aware of something sticky like static members (caches are the worst), on going threads or UI related objects and everything tide to all of them. – NawaMan Dec 13 '12 at 16:11

Below is little summary I wrote back in the days (I stole it from some blog, but I can't remember where

from - so no reference, sorry)

1. **There is no manual way of doing garbage collection in Java.**
2. Java Heap is divided into three generation for the sake of garbage collection. These are the young generation, tenured or old generation, and Perm area.
3. New objects are created in the young generation and subsequently moved to the old generation.
4. String pool is created in Perm area of Heap, Garbage collection can occur in perm space but depends on upon JVM to JVM.
5. Minor garbage collection is used to move an object from Eden space to Survivor 1 and Survivor 2 space, and Major collection is used to move an object from young to tenured generation.
6. Whenever Major garbage collection occurs application, threads stops during that period which will reduce application's performance and throughput.
7. There are few performance improvements has been applied in garbage collection in Java 6 and we usually use JRE 1.6.20 for running our application.
8. JVM command line options `-Xms` and `-Xmx` is used to setup starting and max size for Java Heap. The ideal ratio of this parameter is either 1:1 or 1:1.5 based on my experience, for example, you can have either both `-Xmx` and `-Xms` as 1GB or –Xms 1.2 GB and 1.8 GB.

Command line options: `-Xms:<min size> -Xmx:<max size>`

edited Sep 16 '16 at 8:15

px06
**1,128** 1 12 28

answered Dec 13 '12 at 7:36

aviad
**6,297** 5 31 79

However, as much as possible, I want to use the minimal possible amount of heap size. – Michael Ardan Dec 13 '12 at 8:42

Just to add to the discussion: **Garbage Collection is not the only form of Memory Management in Java**.

In the past, there have been efforts to avoid the GC in Java when implementing the memory management (see Real-time Specification for Java (RTSJ)). These efforts were mainly dedicated to real-time and embedded programming in Java for which GC was not suitable - due to performance overhead or GC-introduced latency.

The RTSJ characteristics

- Immortal and Scoped Memory Management - see below for examples.
- GC and Immortal/Scoped Memory can coexist withing one application
- RTSJ requires a specially modified JVM.

RTSJ advantages:

- low latency, no GC pauses
- delivers predictable performance that is able to meet real-time system requirements

Why RTSJ failed/Did not make a big impact:

- Scoped Memory concept is hard to program with, error-prone and difficult to learn.
- Advance in Real-time GC algoritms reduced the GC pause-time in such way that Real-time GCs replaced the RTSJ in most of the real-time apps. However, Scoped Memories are still used in places where no latencies are tolerated.

Scoped Memory Code Example (take from An Example of Scoped Memory Usage):

```
import javax.realtime.*;
public class ScopedMemoryExample{

    private LTMemory myMem;

    public ScopedMemoryExample(int Size) {

       // initialize memory
       myMem = new LTMemory(1000, 5000);
    }

public void periodicTask() {

  while (true)) {
    myMem.enter(new Runnable() {
        public void run() {
          // do some work in the SCOPED MEMORY
          new Object();
          ...
          // end of the enter() method, the scoped Memory is emptied.
```

```
        }
    });
  }


}
}
```

Here, a ScopedMemory implementation called `LTMemory` is preallocated. Then a thread enters the scoped memory, allocates the temporary data that are needed only during the time of the computation. After the end of the computation, the thread leaves the scoped memory which immediately makes the whole content of the specific ScopedMemory to be emptied. No latency introduced, done in constant time e.g. predictable time, no GC is triggered.

edited Dec 14 '12 at 21:00

answered Dec 13 '12 at 21:41

Ales
**6,444** 4 43 87

This comment may be out of topic. But I just want to share this. I have seen a question regarding why Java is not widely used for commercial game development ( Yes, I'm aware of Runescape and MineCraft ). Some says that GC might pause the game for about a few seconds. I think this might be a solution if their only reason is GC being slow. – Michael Ardan Dec 17 '12 at 1:45

1 These concepts are known since 1999 so there was enough time for their adoption. The reason why Java is not used in Game industry is probably different - perhaps that platform compilers and Game engines are still developed in C/C++. – Ales Dec 19 '12 at 1:01

I see. Thank you very much Ales! – Michael Ardan Dec 19 '12 at 1:05

---

You cannot avoid garbage collection if you use Java. Maybe there are some obscure JVM implementations that do, but I don't know of any.

A properly tuned JVM shouldn't require any System.gc() hints to operate smoothly. The exact tuning you would need depends heavily on what your application does, but in my experience, I always turn on the concurrent-mark-and-sweep option with the following flag: `-XX:+UseConcMarkSweepGC` . This flag allows the JVM to take advantage of the extra cores in your CPU to clean up dead memory on a background thread. It helps to drastically reduce the amount of time your program is forcefully paused when doing garbage collections.

answered Dec 13 '12 at 7:50

cambecc
**2,769** 16 23

Hello, this is very useful. Please explain further. – Michael Ardan Dec 13 '12 at 9:20

1 Found this after a bit of googling: javarevisited.blogspot.jp/2011/04/…. It's a pretty good overview of how GC works in Java, included a description of concurrent-mark-and-sweep. The GC is one of the best things about the language. No more (well, much reduced) manual memory management compared to other languages like C++. – cambecc Dec 13 '12 at 9:52

---

From my experience, in java you should rely on the memory management that is provided by JVM itself.

The point I'd focus on in this topic is to configure it in a way acceptable for your use case. Maybe checking/understanding JVM tuning options would be useful:
http://docs.oracle.com/cd/E15523_01/web.1111/e13814/jvm_tuning.htm

answered Dec 13 '12 at 7:35

Peter Butkovic
**4,173** 3 29 57

because at the end, that might be one of the few significant things you can do there. Of course, you should behave and keep referenced only those objects you really need, but generally from my experience, that's probably the only other thing you should keep in mind. – Peter Butkovic Dec 13 '12 at 8:58

---

You are correct in saying that `System.gc()` is a request to the compiler and not a command. But using below program you can make sure it happens.

```
import java.lang.ref.WeakReference;

public class GCRun {

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

```
        String str = new String("TEMP");
        WeakReference<String> wr = new WeakReference<String>(str);
        str = null;
        String temp = wr.get();
        System.out.println("temp -- " + temp);
        while(wr.get() != null) {
            System.gc();
        }
    }
}
```

edited Sep 23 '17 at 19:27
Saad
595 12 21

answered Sep 23 '17 at 18:56
Ishan Aggarwal
61 3

This is actually interesting Ishan. I'll check this out! Thank you! – Michael Ardan Sep 25 '17 at 7:39

Basically the idea in Java is that you should not deal with memory except using "new" to allocate new objects and ensure that there is no references left to objects when you are done with them.

All the rest is deliberately left to the Java Runtime and is - also deliberately - defined as vaguely as possible to allow the JVM designers the most freedom in doing so efficiently.

To use an analogy: Your operating system manages named areas of harddisk space (called "files") for you. Including deleting and reusing areas you do not want to use any more. You do not circumvent that mechanism but leave it to the operating system

You should focus on writing clear, simple code and ensure that your objects are properly done with. This will give the JVM the best possible working conditions.

answered Dec 13 '12 at 7:35
Thorbjørn Ravn Andersen
54.8k 21 133 277

Well, the GC is always there -- you can't create objects that are outside its grasp (unless you use native calls or allocate a direct byte buffer, but in the latter case you don't really have an object, just a bunch of bytes). That said, it's definitely possible to circumvent the GC by reusing objects. For instance, if you need a bunch of `ArrayList` objects, you could just create each one as you need it and let the GC handle memory management; or you could call `list.clear()` on each one after you finish with it, and put it onto some queue where somebody else can use it.

Standard best practices are to *not* do that sort of reuse unless you have good reason to (ie, you've profiled and seen that the allocations + GC are a problem, and that reusing objects fixes that problem). It leads to more complicated code, and if you get it wrong it can actually make the GC's job harder (because of how the GC tracks objects).

answered Dec 13 '12 at 7:35
yshavit
32.9k 6 56 94

With modern JVM's the amount of work needed to pool and reuse objects is much larger than the work needed just to allocate new, short-lived ones. – Thorbjørn Ravn Andersen Dec 13 '12 at 7:36

@ThorbjørnRavnAndersen In most cases, yes. An application I'm working on (java-based RDBMS) has a case in which reusing objects helps significantly -- we reuse rows as we advance the cursor, and over a couple million rows it ends up being a big win. But again, we know this because we did the profiling to prove it. – yshavit Dec 13 '12 at 7:39

Then you have additional work besides mere allocation for an object to be reusable. – Thorbjørn Ravn Andersen Dec 13 '12 at 7:40

@ThorbjørnRavnAndersen Yes, we do. But I consider that part of the same answer, because in practice an allocated chunk of zero'ed out bytes isn't very useful. In order to save on the other costs, we also have to take the hit of "manually" managing the memory management. – yshavit Dec 13 '12 at 7:43

1 @MichaelArdan No, I mean rather than having each row be created as `new Row(backingBytes)` and then simply losing track of that row when we don't need it anymore (at which point it'll be eligible for GC), we create it once and then keep calling `row.init(backingBytes)` to go to the next row. But the point I tried to bring up in my answer, and which Thorbjørn emphasized, is that this is a rare case. In most all cases, you basically just need to make sure that a GC root (basically, any Thread or Runnable) doesn't hold onto objects longer that it needs to; the GC will take care of the rest. – yshavit Dec 13 '12 at 8:19

|

I would suggest to take a look at the following tutorials and its contents

This is a four part tutorial series to know about the basics of garbage collection in `Java` :

1. Java Garbage Collection Introduction
2. How Java Garbage Collection Works?
3. Types of Java Garbage Collectors
4. Monitoring and Analyzing Java Garbage Collection

I found This tutorial very helpful.

edited Dec 8 '16 at 9:25

DimaSan
**5,303** 8 29 46

answered Dec 8 '16 at 7:15

Tasawar Hussain
**58** 11

---

"Nullify"ing the reference when not required is the best way to make an object eligible for Garbage collection.

There are 4 ways in which an object can be Garbage collected. 1. point the reference to null, once it is no longer required.

```
String s = new String("Java");
```

Once this String is not required, you can point it to null.

```
s = null;
```

Hence, s will be eligible for Garbage collection.

2. point one object to another, so that both reference points to same object and one of the object is eligible for GC.

   String s1 = new String("Java");

   String s2 = new String("C++");

In future if s2 also needs to pointed to s1 then;

```
s1 = s2;
```

Then the object having "C++" will be eligible for GC.

3. All the objects created within a method are eligible for GC once the method is completed. Hence, once the method is destroyed from the stack of the thread then the corresponding objects in that method will be destroyed.
4. Island of Isolation is another concept where the objects with internal links and no extrinsic link to reference is eligible for Garbage collection. "Island of isolation" of Garbage Collection

Examples: Below is a method of Camera class in android. See how the developer has pointed mCameraSource to null once it is not required. This is expert level code.

```
public void release() {
        if (mCameraSource != null) {
            mCameraSource.release();
            mCameraSource = null;
        }
    }
```

How Garbage Collector works?
Garbage collection is performed by the daemon thread called Garbage Collector. When there is sufficient memory available that time this demon thread has low priority and it runs in background. But when JVM finds that the heap is full and JVM wants to reclaim some memory then it increases the priority of Garbage collector thread and calls Runtime.getRuntime.gc() method which searches for all the objects which are not having reference or null reference and destroys those objects.

edited Dec 18 '17 at 10:29

answered Feb 22 '17 at 10:51

Utsav
**624** 1 7 13

---