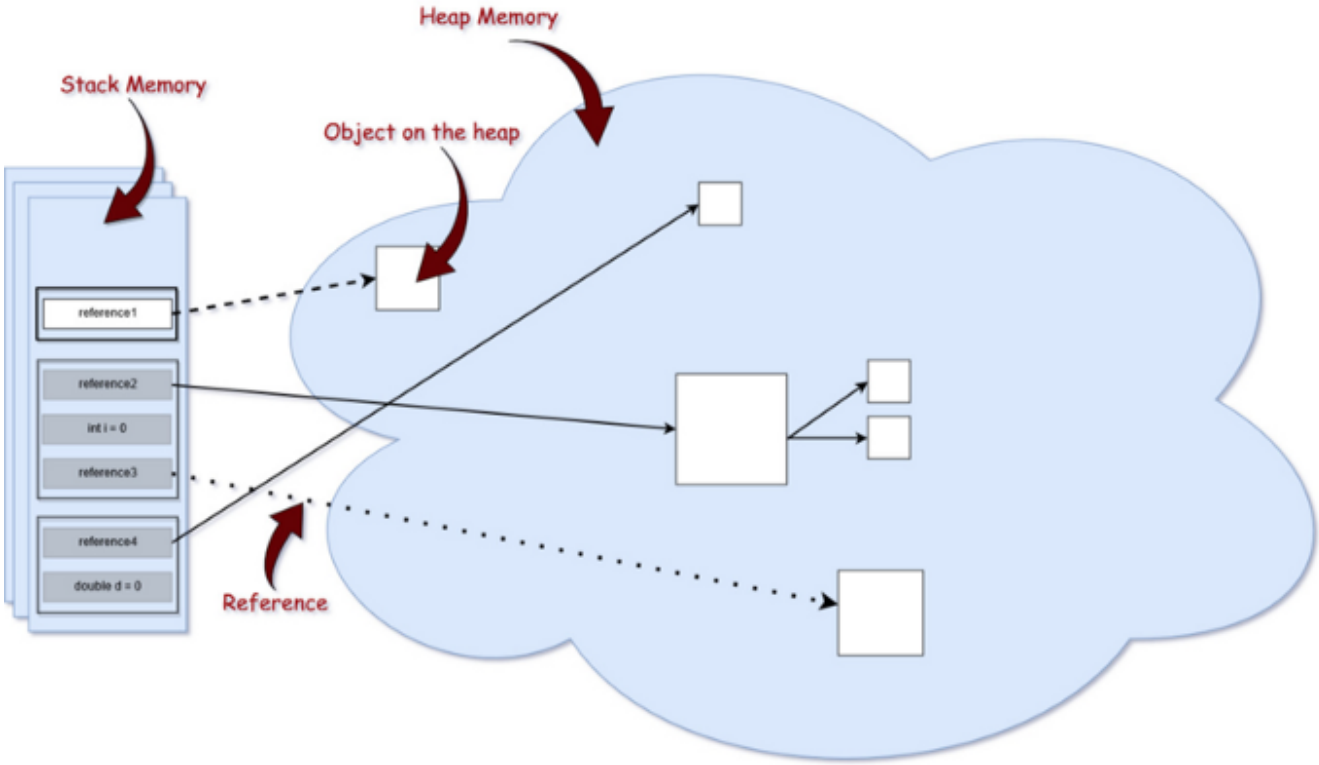
Java Memory Management

Alankar Gupta Follow
Aug 24, 2018 · 5 min read

While we often work on many java applications little do we know about the management JVM does on it's own to make stuff easy for us so that we can focus on things that we do best, develop awesome Applications.
Unlike C, C++ in Java we have Garbage collector to save us from cleaning up the table after we've dined. This section explains in brief about the memory management in Java.
There are two parts in memory management

1. How is memory allocated and referenced.
2. How is memory cleared and become available for allocation again.

Let's look at the first one..
Overall, we can say that memory is divided into three parts — Stack, Heap and non-Heap. Let's look at them in sequence.



Stack

- Variables on stack keeps a reference to objects on Heap and primitives.
- There is one single stack per Thread.

- Divided into parts for easy cleanup and the new allocation (we'll see later)
- Size can be defined via configuration at startup

Non Heap

- The memory required for internal processing of the JVM.
- Constant Pool, Static variables.
- Field and Method data.
- Code for Methods and Constructors.
- Classloaders.
- PermGen (Heap/non Heap) vs Metaspace (Java 8).
- May or may not be a part of Heap — implementation varies.

Reference Type

Now you might've wondering why there are some dashed lines from stack to heap. Let's see how the variables on Stack/Heap refer to objects on Heap. These are called reference and there are of four of them.

- *Strong Reference* — The usual one we use in Java.

```
A a = new A();
```

- *Weak Reference* — A reference to an object which is susceptible for next Garbage collection (if not Strongly referenced).

```
contextWeakReference = new WeakReference<Context>(context);
//Later on.....
Context context = contextWeakReference.get();
if(context != null){
    //Context is not Gc'ed yet
}else{
    //Context is Gc'ed already
}
```

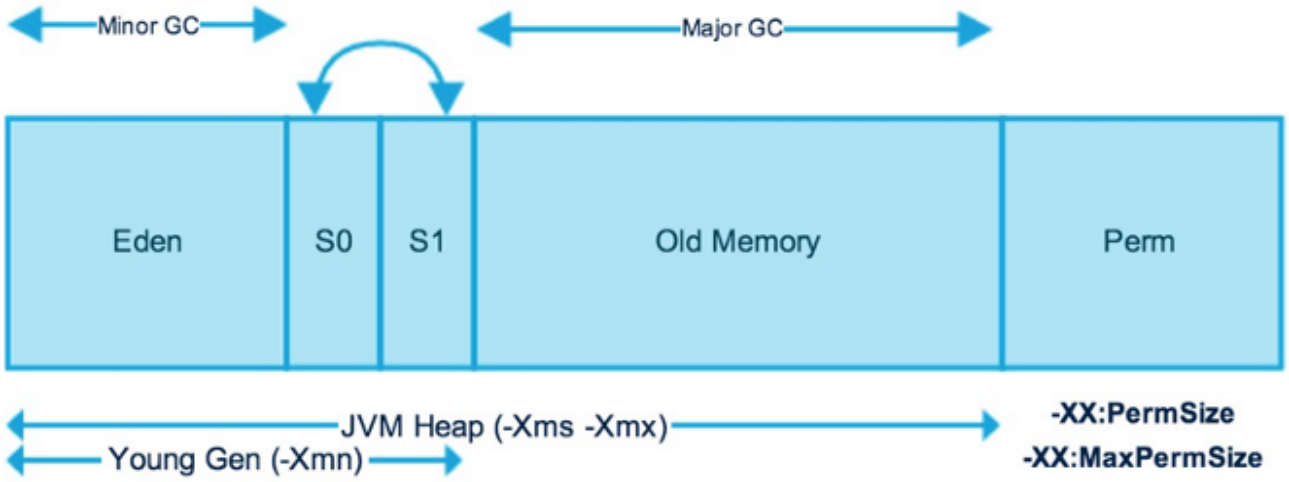
- *Soft Reference* — This is similar to WeakReference it's just that JVM will GC this only when there is a memory crunch. And it is absolutely sure that all the soft references are garbage collected before the JVM throws any OOM.

```
contextSoftReference = new SoftReference<Context>(context);
//Later on.....
Context context = contextSoftReference.get();
if(context != null){
    //Context is not Gc'ed yet
}else{
    //Context is Gc'ed already
}
```

- *Phantom Reference* — It is used for pre-mortem cleanup and is recommended to be used over `finalize()`. Finalize is unpredictable, nondeterministic and slows down the application. In the

```
            A a = new A();
            a.s ="hello";
            Reference r = new PhantomReference(a,rq);
            a =null;
            System.gc();
            Reference ref = (Reference) rq.poll();
            while(ref != null){
                System.out.println(ref.get());
            }
        }
    }
    class A{
        String s;
    }
```

Heap memory Structure

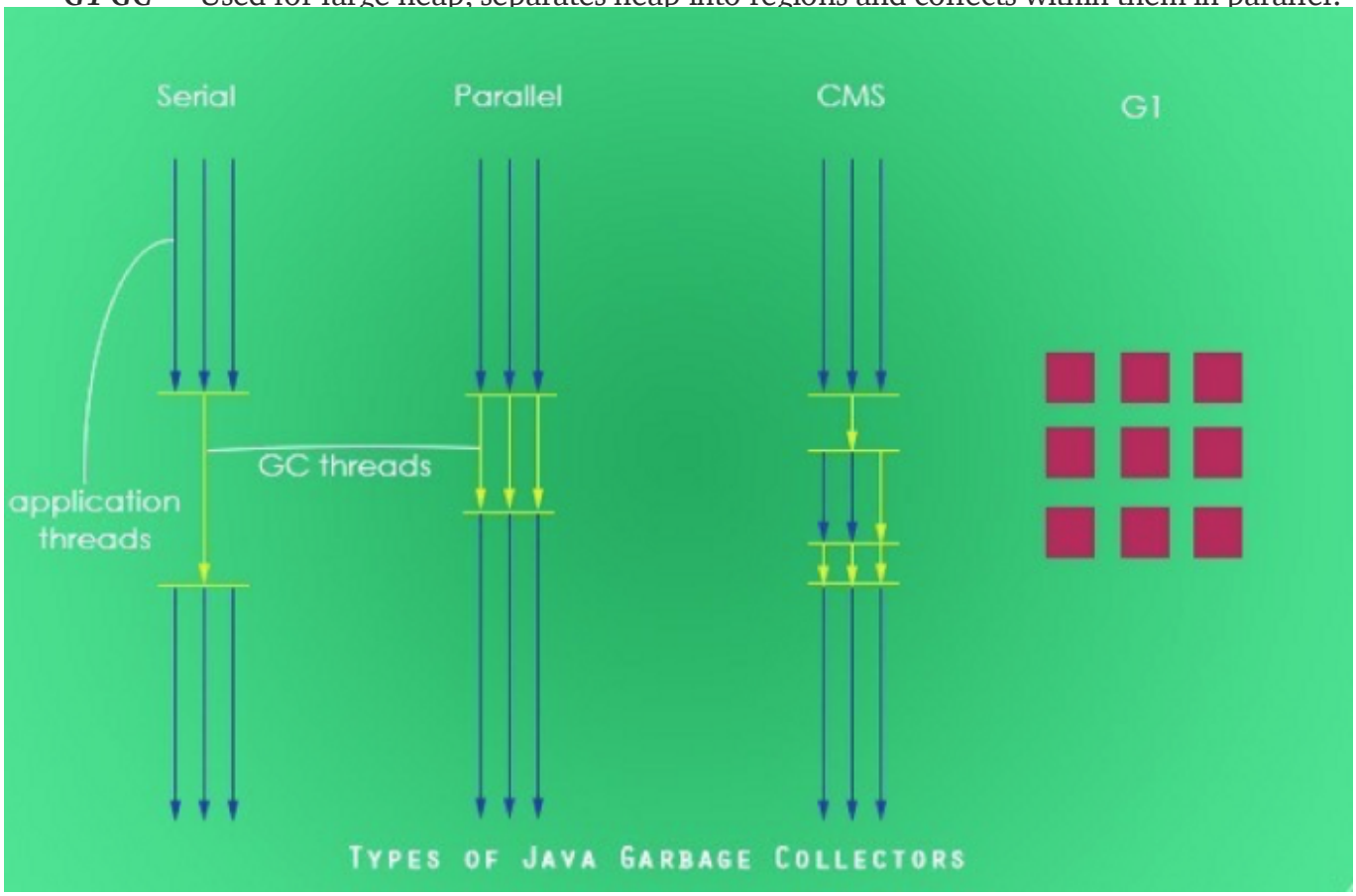


Heap is divided into sections which allows easy allocation and cleanup. **Eden** Space is the place where new objects are allocated. When Eden is about to reach its limit, there is a **Minor GC** which clears up objects available for GC from Eden space to one of the **Survivor spaces** (S0 and S1). Also the objects available for GC in one of the survivor spaces are cleared and remaining objects move to the other survivor space. So at a time there is always one survivor space which is empty. After n (depends on JVM implementation) such iteration objects which have survived are moved to the **old memory**. Whenever old memory is on the verge of being a full **Major GC** occurs and unused resources of old memory are cleared.

How Garbage collection actually happens

It's a *stop-the-world* event. The application thread is paused while GC is in progress. By doing `System.gc();` we can only ***request*** for a GC. JVM internally decides when to execute GC. There are

- ***Concurrent Mark Sweep*** (Parallel New GC) — similar to parallel GC, minimize application pause, does most work with application thread.
- ***G1 GC*** — Used for large heap, separates heap into regions and collects within them in parallel.



We can provide arguments at the start of JVM to specify which GC to use. Some examples are :

-XX:+UseSerialGC	Serial Garbage Collector
-XX:+UseParallelGC	Parallel Garbage Collector
-XX:+UseConcMarkSweepGC	CMS Garbage Collector
-XX:ParallelCMSThreads=	CMS Collector – number of threads to use
-XX:+UseG1GC	G1 Gargbage Collector

JVM args for GC

-XX:PermGen	For setting the initial size of the Permanent Generation Memory
-XX:MaxPermGen	For setting the maximum size of Perm Gen
-XX:SurvivorRatio	For providing ratio of Eden space, for example if young generation size is 10m and VM switch is –XX:SurvivorRatio=2 then 5m will be reserved for Eden space and 2.5m each for both the Survivor spaces. The default value is 8
-XX:NewRatio	For providing ratio of old/new generation sizes. The default value is 2

General JVM arguments

Key Takeaways

- Limit the scope of a variable
- Initialize when you actually need
- Avoid initialization in loop — takes up GC time
- Explicitly refer ***null***
- Avoid finalizers. Prefer Phantom references for cleanup
- Configure JVM based on the requirement by specifying arguments

Cheers !!!

Programming Java Memory Management Garbage Collection JVM

WRITTEN BY

Alankar Gupta

Tech, hack, FIFA and Gym

MindOrks

Our community publishes stories worth reading on software development and design. Android | Machine Learning | #MakeEveryoneCode