Useful links:

|  |  |
| --- | --- |
| PermGen space | http://www.integratingstuff.com/2011/07/24/understanding-and-avoiding-the-java-permgen-space-error/ |
| Garbage Collector Setup | <http://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1-and-whats-new-in-java-8/> |
| JEP | JDK Enhancement Proposal |
| JSR | Java Specification Requests |
| Java 1.8 features | <https://www.javacodegeeks.com/2014/03/8-new-features-for-java-8.html>  <http://www.journaldev.com/2389/java-8-features-for-developers-lambdas-functional-interface-stream-and-time-api>  <http://www.lambdafaq.org/why-are-lambda-expressions-being-added-to-java/>  https://www.javacodegeeks.com/2014/03/8-new-features-for-java-8.html |
| Java 1.7 feaures | https://dzone.com/articles/new-java-7-language-features |
| Lambdas also called as Closures |  |
| Java – Generics | http://www.tutorialspoint.com/java/java\_generics.htm |
| NODE .js | <http://www.tutorialspoint.com/nodejs/>  <https://www.codeschool.com/courses/real-time-web-with-node-js> |
| Java SE (Standard Edition) |  |
| Java EE(Enterprise Edition) |  |
| Java ME(Micro Edition) |  |
| Maven (Apache Maven 3.3.9) | <https://maven.apache.org/download.cgi> |
| Drop wizard learning | <http://www.dropwizard.io/0.9.3/docs/getting-started.html> |
| Collections | <https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html> |
| Collections questions | https://docs.oracle.com/javase/tutorial/collections/interfaces/QandE/questions.html |
| Modernize old code | <http://www.javapractices.com/topic/TopicAction.do?Id=225> |
|  |  |

Thread size in Java? Thread pool size? Java default stack size?

What is the size of the thread?

Lambdas Expressions:

# What is a lambda expression?

In conventional Java terms lambdas can be understood as a kind of anonymous method with a more compact syntax that also allows the omission of modifiers, return type, and in some cases parameter types as well.

#### Syntax

The basic syntax of a lambda is either

**(parameters) -> expression**

or

**(parameters) -> { statements; }**

#### Examples

**1. (int x, int y) -> x + y // takes two integers and returns their sum**

**2. (x, y) -> x - y // takes two numbers and returns their difference**

**3. () -> 42 // takes no values and returns 42**

**4. (String s) -> System.out.println(s) // takes a string, prints its value to the console, and returns nothing**

**5. x -> 2 \* x // takes a number and returns the result of doubling it**

**6. c -> { int s = c.size(); c.clear(); return s; } // takes a collection, clears it, and returns its previous size**

#### Syntax notes

* Parameter types may be explicitly declared (ex. 1,4) or implicitly inferred (ex. 2,5,6). Declared- and inferred-type parameters may not be mixed in a single lambda expression.
* The body may be a block (surrounded by braces, ex. 6) or an expression (ex. 1-5). A block body can return a value (value-compatible, ex. 6) or nothing (void-compatible). The rules for using or omitting the **return** keyword in a block body are the same as those for an ordinary method body.
* If the body is an expression, it may return a value (ex. 1,2,3,5) or nothing (ex. 4).
* Parentheses may be omitted for a single inferred-type parameter (ex. 5,6)
* The comment to example 6 should be taken to mean that the lambda could act on a collection. Equally, [**depending on the context in which it appears**](http://www.lambdafaq.org/what-is-the-type-of-a-lambda-expression), it could be intended to act on an object of some other type having methods **size** and **clear**, with appropriate parameters and return types.

**Java memory structure:**

To understand the error, we have to look into how the jvm memory is structured.

There are two memory regions in the JVM: the heap and the stack. Local variables and methods reside on the stack, everything else on the heap.

This Java heap memory is structured again into regions, called generations. The longer an object lives, the higher the chance it will be promoted to an older generation. Young generations(such as Eden on Sun JVM) are more garbage collected than older generations(survivor and tenured on Sun JVM). However, there is also some separate heap space called permanent generation. Since it is a separate region, it is not considered part of the Java Heap space. Objects in this space are relatively permanent. Class definitions are stored here, as are static instances.

Without getting into details, Classloaders deploy and undeploy classes all the time. For example, this happens when an application is deployed or undeployed on a webserver. On web servers, all applications have their own Classloader. When an application is deployed or undeployed, its class definitions and Classloader are respectively put into and removed from the permanent generation heap.

OutOfMemoryError: PermGen Space

The OutOfMemoryError: PermGen Space error occurs when the permanent generation heap is full. Although this error can occur in normal circumstances, usually, this error is caused by a memory leak.

In short, such a memory leak means that a classloader and its classes cannot be garbage collected after they have been undeployed/discarded.

To give an example on how this can happen, let’s say we have a Payment class, which is part of a jar in a web application that is deployed on some webserver. In the lib folder of the web server, there is some logging framework present, which has a Log class with the method register(Class clazz) with which classes can be registered for logging. Let’s say that the Payment class gets registered by this method and the Log class starts keeping a reference to the clazz object. When the Payment class gets undeployed, it is still registered with the Log class. The Log class will still have a reference to it and hence, it will never be garbage collected. Moreover, since the Payment Class has a reference to its ClassLoader in turn, the ClassLoader itself will never be garbage collected either, and so will none of the classes it loaded.

An even more typical example is with the use of proxy objects. Spring and Hibernate often make proxies of certain classes. Such proxy classes are loaded by a classloader as well, and often, the generated class definitions – which are loaded like classes and stored in permanent generation heap space – are never discarded, which causes the permanent generation heap space to fill up.

Avoiding the error

1. Increasing the maximum size of the permgen heap

The first thing one can do is to make the size of the permanent generation heap space bigger.  
This cannot be done with the usual –Xms(set initial heap size) and –Xmx(set maximum heap size) JVM arguments, since as mentioned, the permanent generation heap space is entirely separate from the regular Java Heap space, and these arguments set the space for this regular Java heap space. However, there are similar arguments which can be used(at least with the Sun/OpenJDK jvms) to make the size of the permanent generation heap bigger:

-XX:MaxPermSize=256m

would set its maximum size to 256m, which is 4 times bigger than the default size.

2. Use common sense when using static fields on classes.

Make sure you do not write classes that have static variables keeping references to class definitions and the like.

Using JDK dynamic proxies instead of cglib proxies

[Some third party frameworks, such as cglib(although it might be better for newer versions of the library?), appear to be permgen monsters.](http://loweringexpectations.nfshost.com/?tag=permgen)

So using jdk dynamic proxies instead of cglib might be a good idea when getting the error.

Also, newer versions of Hibernate appear to not use cglib as a bytecode provider anymore, so upgrading your version of Hibernate, might drastically lower your chances on getting the error.

Summary

In general, when getting the error, one needs to determine why certain class definitions are not garbage collected. Once that is known, it should be possible to battle the error.

When we speak about garbage collection, the vast majority of us know the concept and employ it in our everyday programming. Even so, there’s much about it we don’t understand, and that’s when things get painful. One of the biggest misconceptions about the JVM is that it has one garbage collector, where in fact it provides**four different ones**, each with its own unique advantages and disadvantages. The choice of which one to use isn’t automatic and lies on your shoulders and the differences in throughput and application pauses can be dramatic.

What’s common about these four garbage collection algorithms is that they are generational, which means they split the managed heap into different segments, using the age-old assumptions that most objects in the heap are short lived and should be recycled quickly. As this too is a well-covered area, I’m going to jump directly into the different algorithms, along with their pros and their cons.

## 1. The Serial Collector

The serial collector is the simplest one, and the one you probably won’t be using, as it’s mainly designed for single-threaded environments (e.g. 32 bit or Windows) and for small heaps. This collector freezes all application threads whenever it’s working, which disqualifies it for all intents and purposes from being used in a server environment.

How to use it: You can use it by turning on the *-XX:+UseSerialGC* JVM argument,

## 2. The Parallel / Throughput collector

Next off is the Parallel collector. This is the JVM’s default collector. Much like its name, its biggest advantage is that is uses multiple threads to scan through and compact the heap. The downside to the parallel collector is that it will stop application threads when performing either a minor or full GC collection. The parallel collector is best suited for apps that can tolerate application pauses and are trying to optimize for lower CPU overhead caused by the collector.

## 3. The CMS Collector

Following up on the parallel collector is the CMS collector (“*concurrent-mark-sweep*”). This algorithm uses multiple threads (“concurrent”) to scan through the heap (“mark”) for unused objects that can be recycled (“sweep”). This algorithm will enter “stop the world” (STW) mode in two cases: when initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables) and when the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked.

The biggest concern when using this collector is encountering **promotion failures** which are instances where a race condition occurs between collecting the young and old generations. If the collector needs to promote young objects to the old generation, but hasn’t had enough time to make space clear it,  it will have to do so first which will result in a full STW collection – the very thing this CMS collector was meant to prevent. To make sure this doesn’t happen you would either increase the size of the old generation (or the entire heap for that matter) or allocate more background threads to the collector for him to compete with the rate of object allocation.

Another downside to this algorithm in comparison to the parallel collector is that it uses more CPU in order to provide the application with higher levels of continuous throughput, by using multiple threads to perform scanning and collection. For most long-running server applications which are adverse to application freezes, that’s usually a good trade off to make. Even so, this algorithm is **not on by default**. You have to specify *XX:+USeParNewGC* to actually enable it. If you’re willing to allocate more CPU resources to avoid application pauses this is the collector you’ll probably want to use, assuming that your heap is less than 4Gb in size.  However, if it’s greater than 4GB, you’ll probably want to use the last algorithm – the G1 Collector.

**Java developers?  
Get new posts directly to your inbox**

Subscribe

## ****4. The G1 Collector****

The Garbage first collector (G1) introduced in JDK 7 update 4 was designed to better support heaps larger than 4GB. The G1 collector utilizes multiple background threads to scan through the heap that it divides into regions, spanning from 1MB to 32MB (depending on the size of your heap). G1 collector is geared towards scanning those regions that contain the most garbage objects first, giving it its name (Garbage first). This collector is turned on using the *–XX:+UseG1GC*flag.

This strategy the chance of the heap being depleted before background threads have finished scanning for unused objects, in which case the collector will have to stop the application which will result in a STW collection. The G1 also has another advantage that is that it compacts the heap on-the-go, something the CMS collector only does during full STW collections.

Large heaps have been a fairly contentious area over the past few years with many developers moving away from the single JVM per machine model to more micro-service, componentized architectures with multiple JVMs per machine. This has been driven by many factors including the desire to isolate different application parts, simplifying deployment and avoiding the cost which would usually come with reloading application classes into memory (something which has actually been improved in Java 8).

Even so, one of the biggest drivers to do this when it comes to the JVM stems from the desire to avoid those long “stop the world” pauses (which can take many seconds in a large collection) that occur with large heaps. This has also been accelerated by container technologies like Docker that enable you to deploy multiple apps on the same physical machine with relative ease.

### Java 8 and the G1 Collector

Another beautiful optimization which was just out with Java 8 update 20 for is the G1 Collector **String deduplication**. Since strings (and their internal char[] arrays) takes much of our heap, a new optimization has been made that enables the G1 collector to identify strings which are duplicated more than once across your heap and correct them to point into the same internal char[] array, to avoid multiple copies of the same string from residing inefficiently within the heap. You can use the *-XX:+UseStringDeduplication*JVM argument to try this out.

### Java 8 and PermGen

One of the biggest changes made in Java 8 was [removing](http://java.dzone.com/articles/java-8-permgen-metaspace) the permgen part of the heap that was traditionally allocated for class meta-data, interned strings and static variables. This would traditionally require developers with applications that would load significant amount of classes (something common with apps using enterprise containers) to optimize and tune for this portion of the heap specifically. This has over the years become the source of many OutOfMemory exceptions, so having the JVM (mostly) take care if it is a very nice addition. Even so, that in itself will probably not reduce the tide of developers decoupling their apps into multiple JVMs.

Each of these collectors is configured and tuned differently with a slew of toggles and switches, each with the potential to increase or decrease throughput, all based on the specific behavior of your app. We’ll delve into the key strategies of configuring each of these in our next posts.

In the meanwhile, what are the things you’re most interested in learning about regarding the differences between the different collectors? Hit me up in the comments section 

Additional reading –

1. A really great in-depth review of the G1 Collector on [InfoQ](http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All).

2. Java performance – The definitive guide. My favorite [book](http://www.amazon.com/Java-Performance-The-Definitive-Guide/dp/1449358454) on Java performance.

3. More about String deduplication on the CodeCentric [blog](https://blog.codecentric.de/en/2014/08/string-deduplication-new-feature-java-8-update-20-2/).

Takipi shows you when and why your code breaks in production. It detects caught and uncaught exceptions, HTTP and log errors, and gives you the code and variable state when they happened. Get actionable information, solve complex bugs in minutes. Installs in 5-min. Built for production.