Types of Garbage Collectors in Java

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

Java Garbage Collectors

You now know the basics of garbage collection and have observed the garbage collector in action on a sample application. In this section, you will learn about the garbage collectors available for Java and the command line switches you need to select them.

**Common Heap Related Switches**

There are many different command line switches that can be used with Java. This section describes some of the more commonly used switches that are also used in this OBE.

| **Switch** | **Description** |
| --- | --- |
| -Xms | Sets the initial heap size for when the JVM starts. |
| -Xmx | Sets the maximum heap size. |
| -Xmn | Sets the size of the Young Generation. |
| -XX:PermSize | Sets the starting size of the Permanent Generation. |
| -XX:MaxPermSize | Sets the maximum size of the Permanent Generation |

**The Serial GC**

The serial collector is the default for client style machines in Java SE 5 and 6. With the serial collector, both minor and major garbage collections are done serially (using a single virtual CPU). In addition, it uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

**Usage Cases**

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work (therefore, its name). Still, on today's hardware, the Serial GC can efficiently manage a lot of non-trivial applications with a few hundred MBs of Java heap, with relatively short worst-case pauses (around a couple of seconds for full garbage collections).

Another popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer. And the Serial GC fits this trade-off nicely.

Finally, with the proliferation of embedded hardware with minimal memory and few cores, the Serial GC could make a comeback.

**Command Line Switches**

To enable the Serial Collector use:  
-XX:+UseSerialGC

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The Parallel GC

The parallel garbage collector uses multiple threads to perform the young genertion garbage collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection. The number of garbage collector threads can be controlled with command-line options:  
-XX:ParallelGCThreads=<desired number>

On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested. On a host with two CPUs the parallel garbage collector generally performs as well as the default garbage collector and a reduction in the young generationgarbage collector pause times can be expected on hosts with more than two CPUs. The Parallel GC comes in two flavors.

#### Usage Cases

The Parallel collector is also called a throughput collector. Since it can use multilple CPUs to speed up application throughput. This collector should be used when a lot of work need to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

#### -XX:+UseParallelGC

With this command line option you get a multi-thread young generation collector with a single-threaded old generation collector. The option also does single-threaded compaction of old generation.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

#### -XX:+UseParallelOldGC

With the -XX:+UseParallelOldGC option, the GC is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector. HotSpot does compaction only in the old generation. Young generation in HotSpot is considered a copy collector; therefore, there is no need for compaction.

Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. It is possible that a garbage collector be a non-compacting collector. Therefore, the difference between a parallel collector and a parallel compacting collector could be the latter compacts the space after a garbage collection sweep. The former would not.

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelOldGC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The Concurrent Mark Sweep (CMS) Collector

The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

**Note:** CMS collector on young generation uses the same algorithm as that of the parallel collector.

#### Usage Cases

The CMS collector should be used for applications that require low pause times and can share resources with the garbage collector. Examples include desktop UI application that respond to events, a webserver responding to a request or a database responding to queries.

#### Command Line Switches

To enable the CMS Collector use:  
-XX:+UseConcMarkSweepGC  
and to set the number of threads use:  
-XX:ParallelCMSThreads=<n>

Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2 -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### The G1 Garbage Collector

The Garbage First or G1 garbage collector is available in Java 7 and is designed to be the long term replacement for the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously. However, detailed discussion is beyond the scope of this OBE.

#### Command Line Switches

To enable the G1 Collector use:  
-XX:+UseG1GC

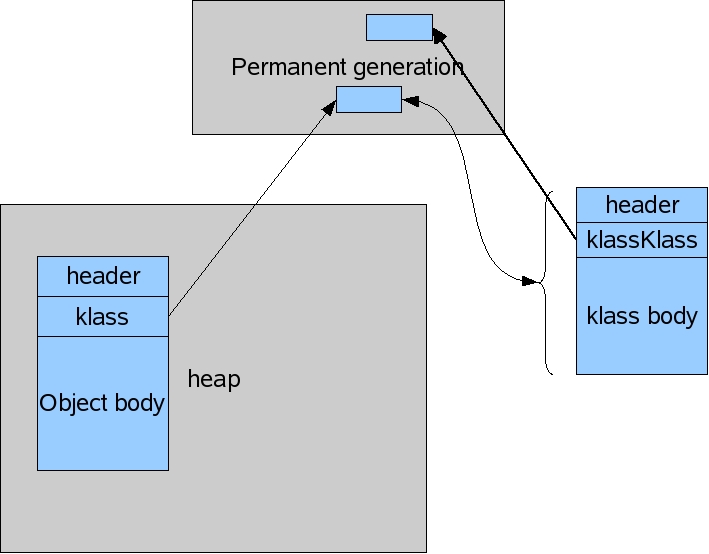
Here is a sample command line for starting the Java2Demo:  
java -Xmx12m -Xms3m -XX:+UseG1GC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

### Presenting the Permanent Generation

<https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation>

Have you ever wondered how the permanent generation fits into our generational system? Ever been curious about what's in the permanent generation. Are objects ever promoted into it? Ever promoted out? We'll you're not alone. Here are some of the answers.

Java objects are instantiations of Java classes. Our JVM has an internal representation of those Java objects and those internal representations are stored in the heap (in the young generation or the tenured generation). Our JVM also has an internal representation of the Java classes and those are stored in the permanent generation. That relationship is shown in the figure below.



The internal representation of a Java object and an internal representation of a Java class are very similar. From this point on let me just call them Java objects and Java classes and you'll understand that I'm referring to their internal representation. The Java objects and Java classes are similar to the extent that during a garbage collection both are viewed just as objects and are collected in exactly the same way. So why store the Java objects in a separate permanent generation? Why not just store the Java classes in the heap along with the Java objects?

Well, there is a philosophical reason and a technical reason. The philosophical reason is that the classes are part of our JVM implementation and we should not fill up the Java heap with our data structures. The application writer has a hard enough time understanding the amount of live data the application needs and we shouldn't confuse the issue with the JVM's needs.

The technical reason comes in parts. Firstly the origins of the permanent generation predate my joining the team so I had to do some code archaeology to get the story straight (thanks Steffen for the history lesson).

Originally there was no permanent generation. Objects and classes were just stored together.

Back in those days classes were mostly static. Custom class loaders were not widely used and so it was observed that not much class unloading occurred. As a performance optimization the permanent generation was created and classes were put into it. The performance improvement was significant back then. With the amount of class unloading that occur with some applications, it's not clear that it's always a win today.

It might be a nice simplification to not have a permanent generation, but the recent implementation of the parallel collector for the tenured generation (aka parallel old collector) has made a separate permanent generation again desirable. The issue with the parallel old collector has to do with the order in which objects and classes are moved. If you're interested, I describe this at the end.

So the Java classes are stored in the permanent generation. What all does that entail? Besides the basic fields of a Java class there are

* Methods of a class (including the bytecodes)
* Names of the classes (in the form of an object that points to a string also in the permanent generation)
* Constant pool information (data read from the class file, see chapter 4 of the JVM specification for all the details).
* Object arrays and type arrays associated with a class (e.g., an object array containing references to methods).
* Internal objects created by the JVM (java/lang/Object or java/lang/exception for instance)
* Information used for optimization by the compilers (JITs)

That's it for the most part. There are a few other bits of information that end up in the permanent generation but nothing of consequence in terms of size. All these are allocated in the permanent generation and stay in the permanent generation. So now you know.

This last part is really, really extra credit. During a collection the garbage collector needs to have a description of a Java object (i.e., how big is it and what does it contain). Say I have an object X and X has a class K. I get to X in the collection and I need K to tell me what X looks like. Where's K? Has it been moved already? With a permanent generation during a collection we move the permanent generation first so we know that all the K's are in their new location by the time we're looking at any X's.

How do the classes in the permanent generation get collected while the classes are moving? Classes also have classes that describe their content. To distinguish these classes from those classes we spell the former klasses. The classes of klasses we spell klassKlasses. Yes, conversations around the office can be confusing. Klasses are instantiation of klassKlasses so the klassKlass KZ of klass Z has already been allocated before Z can be allocated. Garbage collections in the permanent generation visit objects in allocation order and that allocation order is always maintained during the collection. That is, if A is allocated before B then A always comes before B in the generation. Therefore if a Z is being moved it's always the case that KZ has already been moved.

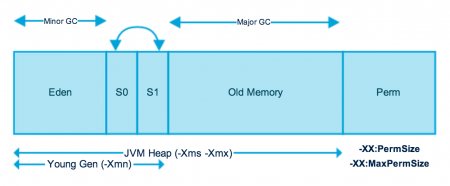
And why not use the same knowledge about allocation order to eliminate the permanent generations even in the parallel old collector case? The parallel old collector does maintain allocation order of objects, but objects are moved in parallel. When the collection gets to X, we no longer know if K has been moved. It might be in its new location (which is known) or it might be in its old location (which is also known) or part of it might have been moved (but not all of it). It is possible to keep track of where K is exactly, but it would complicate the collector and the extra work of keeping track of K might make it a performance loser. So we take advantage of the fact that classes are kept in the permanent generation by collecting the permanent generation before collecting the tenured generation. And the permanent generation is currently collected serially.

# **Java (JVM) Memory Model and Garbage Collection**

<http://www.journaldev.com/2856/java-jvm-memory-model-and-garbage-collection-monitoring-tuning>

Understanding **JVM Memory Model** is very important if you want to understand the working of **Java Garbage Collection**. Today we will look into different parts of JVM memory and how to monitor and perform garbage collection tuning.

### Java (JVM) Memory Model

[](http://www.journaldev.com/wp-content/uploads/2014/05/Java-Memory-Model.png)

As you can see in the above image, JVM memory is divided into separate parts. At broad level, JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

### Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

### Old Generation

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it’s full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.

### Stop the World Event

All the Garbage Collections are “Stop the World” events because all application threads are stopped until the operation completes.

Since Young generation keeps short-lived objects, Minor GC is very fast and the application doesn’t get affected by this.

However Major GC takes longer time because it checks all the live objects. Major GC should be minimized because it will make your application unresponsive for the garbage collection duration. So if you have a responsive application and there are a lot of Major Garbage Collection happening, you will notice timeout errors.

The duration taken by garbage collector depends on the strategy used for garbage collection. That’s why it’s necessary to monitor and tune the garbage collector to avoid timeouts in the highly responsive applications.

### Permanent Generation

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

### Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

### Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

### Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

### Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should read [Difference between Stack and Heap Memory](http://www.journaldev.com/4098/java-heap-memory-vs-stack-memory-difference).

### Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

|  |  |
| --- | --- |
| VM Switch | VM Switch Description |
| -Xms | For setting the initial heap size when JVM starts |
| -Xmx | For setting the maximum heap size. |
| -Xmn | For setting the size of the Young Generation, rest of the space goes for Old Generation. |
| -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 and Survivor 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. |

Most of the times, above options are sufficient, but if you want to check out other options too then please check [JVM Options Official Page](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

### Java Garbage Collection

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

**Garbage Collector** is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

There are two problems with simple mark and delete approach.

1. First one is that it’s not efficient because most of the newly created objects will become unused
2. Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.

The above shortcomings with the simple approach is the reason that **Java Garbage Collection is Generational** and we have **Young Generation** and **Old Generation** spaces in the heap memory. I have already explained above how objects are scanned and moved from one generational space to another based on the Minor GC and Major GC.

### Java Garbage Collection Types

There are five types of garbage collection types that we can use in our applications. We just need to use JVM switch to enable the garbage collection strategy for the application. Let’s look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.

Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

1. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using -XX:ParallelGCThreads=n JVM option.

Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

1. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
2. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can’t afford longer pause times. We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it’s long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at [Garbage-First Collector Oracle Documentation](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

### Java Garbage Collection Monitoring

We can use Java command line as well as UI tools for monitoring garbage collection activities of an application. For my example, I am using one of the demo application provided by Java SE downloads.

If you want to use the same application, go to [Java SE Downloads](http://www.oracle.com/technetwork/java/javase/downloads/index.html) page and download **JDK 7 and JavaFX Demos and Samples**. The sample application I am using is **Java2Demo.jar** and it’s present injdk1.7.0\_55/demo/jfc/Java2D directory. However this is an optional step and you can run the GC monitoring commands for any java application.

# **FINALIZE METHOD IN JAVA - WHY NOT TO USE, LIMITED USE CASES AND ALTERNATIVES**

<http://javajee.com/finalize-method-in-java-why-not-to-use-limited-use-cases-and-alternatives>

#### ****Why using finalize is not a good idea****

Finalizers are bad for functional reasons as well as for performance reasons. We will first see functional reasons and then the performance reasons.

**Functional reasons not to use finalize**

You should never depend on a finalizer to update critical data or do anything time-critical in a finalizer, or assume that finalize will work the same on another JVM, because:

1. You can resurrect your object from a finalize method (creating a new strong reference to the referent object), but Java won’t call finalize again when the object is freed again, and the expected cleanup of the referent won’t happen.
2. Java language specification does not guarantee that finalize will always be executed; specification only guarantees that finalize() will not be called twice. Even System.gc and System.runFinalization (though may increase the chances of finalizers getting executed,) don’t guarantee it either.
3. There is also no guarantee finalizers will be executed promptly; even after an object becomes eligible for garbage collection, it can take any time before finalizer is executed.
4. When to execute finalizer may also dependent on the GC algorithm in use in a particular JVM, thus also affecting portability.
5. If a runtime exception is thrown within the finalize method, finalization of that object terminates and the exception is ignored. Therefore there is a change for some activities to be missed due to an exception.

**Performance impact of finalize**

Finalizers also has great impact on performance. The JVM uses a special private reference object class java.lang.ref.Finalizer to keep track of objects that have defined a finalize() method. The java.lang.ref.Finalizer in turn is a java.lang.ref.FinalReference. This is a special reference similar to other public reference object classes available in the java.lang.ref package. You can learn more about reference objects @ <http://javajee.com/soft-and-week-reference-object-classes-in-java> and for further reference you can refer to Oracle documentation for java.lang.ref package @  <http://docs.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html>.

When an object that has a finalize() method is allocated, the JVM allocates two objects: the object itself, and a Finalizer reference that also refer to this object. As with other less strong references, it takes at least two GC cycles before this less strong reference object can be freed. However, the performance penalty here is greater than with other less strong reference types. When the referent of a soft or weak reference is eligible for GC, the referent object is freed first, and the soft or weak reference is placed on the reference queue and is freed in next GC cycle; the two-cycle penalty for GC applies only to the reference object itself (and not the referent). However in case of finalizers, referent object is not freed immediately when the finalizer reference is placed on its reference queue, as the implementation of the Finalizer class must have access to the referent in order to call the referent’s finalize() method. Only when the reference queue processes the finalizer, the Finalizer object will be removed from the queue and then eligible for collection. Unlike other less strong references, here the memory of the referent object is also retained, which can be usually much bigger than the reference object.

#### ****Unavoidable cases****

However in some cases, finalize might be unavoidable. For instance, if you have some native resource to be freed, then you can have an explicit method to free that resource, but also can have finalize as a backup plan in case the developer forget to call that explicit method. For instance, classes that handle zip files in JDK uses this technique as opening a ZIP file uses some native code that allocates native memory and it needs to be closed even if developer forget to call close. Other JDK classes such as FileInputStream, FileOutputStream, Timer, Connection etc. also have finalizers. The book Effective Java suggests that the finalizer should log a warning in these cases if it finds that the resource has not been terminated, so that the client code can be fixed (however JDK classes don’t do that).  You should also make sure that the memory accessed by the object is kept to a minimum in such cases when it is unavoidable to use finalizers.

#### ****Finalizer Guardian idiom****

Unlike constructors, finalize won’t automatically call its parent. So you need to call the super finalize explicitly (like any other method), and there is a change that you may miss that. The section ‘Item 7: Avoid finalizers’ in Effective Java talks about a solution to this problem: Put the finalizer on an anonymous inner class (called a finalizer guardian), and a single instance of the anonymous class is stored in a private instance field so the finalizer guardian becomes eligible for finalization at the same time as the enclosing instance. When the guardian is finalized, it performs the finalization activity desired for the enclosing instance, just as if its finalizer were a method on the enclosing class.

#### ****Alternatives to finalize****

You can overcome some of the disadvantages of finalizers by using PhantomReference class (along with ReferenceQueue) rather than implicitly using a Finalizer reference. This can overcomes two limitations of the traditional finalizer: the memory associated with the referent object is released as soon as the referent is collected (rather than doing that in the finalizer() method and giving a chance to resurrect). There is also no way for the referent object to be resurrected in the cleanup code, since it has already been collected. Other limitations like no guarantee of timing, no guarantee that it will be actually cleared etc. can happen here also.

I will not explain the approach here, but just wanted to tell that there is one such approach possible. You can search online or refer to a book that explains the approach. I recommend you look at the book ‘Java Performance: The Definitive Guide’ as that is my primary reference book for learning java performance.

#### ****Clearing and monitoring the Finalizer Queue****

The finalizer queue is the reference queue used to process the Finalizer references when the referent is eligible for GC.

You can cause the finalizer queue to be processed by executing GC.run\_finalization of jcmd command:

jcmd <process\_id> GC.run\_finalization

You can monitor the finalizer queue using jmap as:

jmap -finalizerinfo <process\_id>

You can also monitor the finalizer queue using jconsole.

## **Replacing Finalizers With Phantom References**

<http://resources.ej-technologies.com/jprofiler/help/doc/index.html>

### Why finalizers are bad

Sometimes one must perform pre-garbage collection actions such as freeing resources. In a JDBC driver, for example, a database connection may be held by a connection object. Before the connection object is garbage collected, the actual database connection must be closed. In such a case, one typically cannot rely on the close() method being called by the user application code.

Most often, **finalizers** are used to solve this problem. A finalizer is created by overriding the finalize() method of java.lang.Object. In that case, before the object is garbage collected, this finalize method will be called. Unfortunately, there are severe problems with the design of this finalizer mechanism. Using finalizers has a negative impact on the performance of the garbage collector and can break data integrity of your application if you're not very careful since the "finalizer" is invoked in a random thread, at a random time. If you use a lot of finalizers, the finalizer system may be completely overwhelmed which can lead to OutOfMemoryErrors. In addition, you have no control about when a finalizer will be run, so it can create problems with locking, the shutdown of the JVM and other exceptional circumstances.

Because the random execution of the finalizers break the call tree, JProfiler eliminates them from the profiling results.

The solution for all these problems is to **eliminate finalizers** where they are not strictly required and **replace the necessary ones with phantom references**.

### What are phantom references?

Phantom references can be used to perform actions before an object is garbage collected in a safe way. In the constructor of a java.lang.ref.PhantomReference, you specify a java.lang.ref.ReferenceQueue where the phantom reference will be enqueued once the referenced object becomes "phantom reachable". Phantom reachable means unreachable other than through the phantom reference. The initially confusing thing is that although the phantom reference continues to hold the referenced object in a private field (unlike soft or weak references), its getReference() method always returns null. This is so that you cannot make the object strongly reachable again.

From time to time, you can poll the reference queue and check if there are any new phantom references whose referenced objects have become phantom reachable. In order to be able to do anything useful, one can for example derive a class fromjava.lang.ref.PhantomReference that references resources that should be freed before garbage collection. The referenced object is only garbage collected once the phantom reference becomes unreachable itself.

### How to replace finalizers with phantom references

Let's continue with the example of the JDBC driver above: Before a connection object is garbage collected, the actual database connection must be closed. The following steps are necessary to achieve this with phantom references:

* **Add data structure that holds phantom references**   
  The JDBC driver class gets a data structure that holds phantom references to the connection objects. A private field

private LinkedList phantomReferences = new LinkedList();

would be appropriate. This is necessary to ensure that phantom references are not garbage collected as long as they have not been handled by the reference queue.

* **Create reference queue**   
  Before a connection object will be garbage collected, its phantom reference will be enqueued into the associated reference queue. The JDBC driver thus gets an additional private field

private ReferenceQueue queue = new ReferenceQueue();

* **Derive a class from PhantomReference that references resources**   
  You will not be able to access the original object from a phantom reference. Therefore, you have to add the resources that must be freed to the phantom reference itself. In our example JDBC driver this could be a class named DatabaseConnection. The phantom reference class will thus look like:
* public class ConnectionPhantomReference extends PhantomReference {
* private DatabaseConnection databaseConnection;
* public MyPhantomReference(ConnectionImpl connection, ReferenceQueue queue) {
* super(connection, queue);
* databaseConnection = connection.getDatabaseConnection();
* }
* public void cleanup() {
* databaseConnection.close();
* }

}

The custom phantom reference extracts the resource object from the implementation class of the connection and saves it in a private field. It additionally provides a cleanup() method that can be invoked once after the phantom reference is taken out of the reference queue.

* **Create and remember phantom references when objects are created**   
  When a connection object is created, a corresponding ConnectionPhantomReference must be created as well and added to the phantomReferences list:

phantomReferences.add(new ConnectionPhantomReference(connection, queue));

* **Create reference queue handler thread**   
  When a phantom reference is added to the queue by the garbage collector, no further action is taken. You have to handle and empty the reference queue yourself. It's best to create a separate daemon thread that removes phantom references from the queue and invokes the cleanup method:

Thread referenceThread = new Thread() {

public void run() {

while (true) {

try {

ConnectionPhantomReference ref = (ConnectionPhantomReference)queue.remove();

ref.close();

phantomReferences.remove(ref);

} catch (Exception ex) {

// log exception, continue

}

}

}

};

referenceThread.setDaemon(true);

referenceThread.start();

The phantom reference is removed from the phantomReferences list. Now the phantom reference is unreferenced itself and the referenced object can be garbage collected.

**10 Garbage Collection Interview Questions and Answers in Java Programming**  
<http://javarevisited.blogspot.in/2012/10/10-garbage-collection-interview-question-answer.html>

**GC Interview Questions Answer in Java**

Garbage collection interview questions are very popular in both core Java and advanced Java Interviews. Apart from  Java Collection and Thread  many [tricky Java questions](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) stems Garbage collections which are tough to answer. In this Java Interview article I will share some questions from GC which is asked in various core Java interviews.These questions are based upon concept of [How Garbage collection works](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html), Different kinds of Garbage collector and [JVM parameters](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) used for garbage collection monitoring and tuning. As I said GC is an important part of any Java interview so make sure you have good command in GC. One more thing which is getting very important is ability to comprehend and **understand Garbage collection Output**, more and more interviewer are checking whether candidate can understand GC output or not. During [Java interview](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) they may provide a snippet of GC output and ask various questions based on that e.g. Which Garbage collector is used, whether output is from major collection or minor collection, How much memory is free from GC, What is size of new generation and old generation after GC etc. I have included few Garbage collection interview questions and answers from GC output to help with that. It’s recommended to prepare [questions from Java collection](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html),  [multithreading](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) and [programming](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html)along with Garbage collection to do well in Java interviews at any stage.

## Interview questions on Java Garbage collection

[Java Garbage collection Interview Question Answer for 4+ experience](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Here is some Garbage collection Interview questions from my personal collection, which I have created from my experience and with the help of various friends and colleagues which has shared *GC interview questions* with me. Actually there are lot many questions than What I am sharing here but to keep this post small I thought to only share some questions, I can think of second part of GC interview question if you guys find this useful.

**Question 1 - What is structure of Java Heap ? What is Perm Gen space in Heap ?**

Answer : In order to better perform in Garbage collection questions in any Java interview, It’s important to have basic understanding of  Java Heap space. To learn more about heap, see my post [10 points on Java heap space](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html). By the way Heap is divided into different generation e.g. new generation, old generation and PermGen space.PermGen space is used to store class’s metadata and filling of PermGen space can cause[java.lang.OutOfMemory:PermGen space](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html). Its also worth noting to remember [JVM option to configure PermGen](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html) space in Java.

**Question 2 - How do you identify minor and major garbage collection in Java?**

Answer: Minor collection prints “GC” if garbage collection [logging](http://javarevisited.blogspot.sg/2011/05/top-10-tips-on-logging-in-java.html) is enable using –verbose:gc or -XX:PrintGCDetails, while Major collection prints “Full GC”. This Garbage collection interview question is based on understanding of Garbage collection output. As more and more Interviewer are asking question to check candidate’s ability to understand GC output, this topic become even more important.

**Question 3 - What is difference between ParNew and DefNew Young Generation Garbage collector?**

Answer : This *Garbage Collection interview questions* is recently asked to one of my friend. It require more than average knowledge on GC to answer this question. By the way ParNew and DefNew is two young generation garbage collector. ParNew is a multi-threaded GC used along with concurrent Mark Sweep while DefNew is single threaded GC used along with Serial Garbage Collector.

**Question 4 - How do you find GC resulted due to calling System.gc()?**

Answer : Another GC interview question which is based on GC output. Similar to major and minor collection, there will be a word “System” included in Garbage collection output.

**Question 5 - What is difference between Serial and Throughput Garbage collector?**

Answer : Serial Garbage collector is a stop the world GC which stops application thread from running during both [minor and major collection](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). Serial Garbage collector can be enabled using JVM option -XX:UseSerialGC and it's designed for Java application which doesn't have pause time requirement and have client configuration. **Serial Garbage collector** was also default GC in JDK 1.4 before ergonomics was introduced in JDK 1.5. Serial GC is most suited for small application with less number of [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) while throughput GG is more suited for large applications. On the other hand Throughput garbage collector is parallel collector where minor and major collection happens in parallel taking full advantage of all the system resources available like multiple processor. Though both major and minor collection runs on stop-the-world fashion and introduced pause in application. Throughput Garbage collector can be enable using -XX:UseParallelGC or -XX:UseOldParallelGC. It increases overall throughput of application my minimizing time spent in Garbage collection but still has long pauses during full GC.This is a kind of *Garbage collection interview questions* which gives you an opportunity to show your knowledge in detail while answering. I always suggest to answer these kind of questions in detail.

**Question 6 – When does an Object becomes eligible for Garbage collection in Java ?**

Answer : An object becomes [eligible for garbage collection](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html) when there is no live reference for that object or it can not be reached by any live thread. Cyclic reference doesn’t count as live reference and if two objects are pointing to each other and there is no live reference for any of them, than both are eligible for GC. Also Garbage collection thread is a [daemon thread](http://javarevisited.blogspot.sg/2012/03/what-is-daemon-thread-in-java-and.html) which will run by JVM based upon GC algorithm and when runs it collects all objects which are eligible for GC.

**Question 7 - What is finalize method in Java ? When does Garbage collector calls finalize method in Java ?**

Answer : Finalize method in Java also called finalizer is a method defined in java.lang.Object and called by Garbage collector before collecting any object which is eligible for GC. Finalize() method provides last chance to object to do cleanup and free any remaining resource, to learn more about finalizers, read [What is finalize method in Java](http://javarevisited.blogspot.sg/2012/03/finalize-method-in-java-tutorial.html).

**Question 8 - If Object A has reference to Object B and Object B refer to Object A, apart from that there is no live reference to either object A or B, Does they are eligible to Garbage collection ?**

This Garbage collection interview questions is related question 5 “When object become eligible for Garbage collection”. An object becomes eligible for Garbage collection if there is no live reference for it. It can not be accessible from any Thread and cyclic dependency doesn’t prevent Object from being Garbage collected. Which means in this case both Object A and Object B are eligible of Garbage collection. See [How Garbage collection works in Java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) for more details.

**Question 9 -Can we force Garbage collector to run at any time ?**

Answer : No, you can not force Garbage collection in Java. Though you can request it by calling Sytem.gc() or its cousin Runtime.getRuntime().gc(). It’s not guaranteed that GC will run immediately as result of calling these method.

**Question 10 - Does Garbage collection occur in permanent generation space in JVM?**

Answer : This  is a tricky Garbage collection interview question as many programmers are not sure whether PermGen space is part of [Java heap space](http://javarevisited.blogspot.sg/2011/08/increase-heap-size-maven-ant.html) or not and since it maintains class Meta data and String pool, whether its eligible for garbage collection or not. By the way Garbage Collection does occur in PermGen space and if PermGen space is full or cross a threshold, it can trigger Full GC. If you look at output of GC you will find that PermGen space is also garbage collected. This is why correct sizing of PermGen space is important to avoid frequent full GC. You can control size of PermGen space by [JVM options](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html) -XX:PermGenSize and -XX:MaxPermGenSize.

**Question 11 : How to you monitor garbage collection activities?**

Answer : One of my favorite interview questions on [Garbage collection](http://www.blogger.com/javarevisited.blogspot.in/2011/04/garbage-collection-in-java.html), just to check whether candidate has ever monitored GC activities or not. You can monitor garbage collection activities either offline or real-time. You can use tools like **JConsole** and **VisualVM** VM with its Visual GC plug-in to monitor real time garbage collection activities and memory status of JVM or you can redirect Garbage collection output to a log file for offline analysis by using -XlogGC=&lt;PATH&gt; JVM parameter. Anyway you should always enable GC options like -XX:PrintGCDetails -X:verboseGC and -XX:PrintGCTimeStamps as it doesn't impact [application performance](http://javarevisited.blogspot.sg/2012/01/improve-performance-java-database.html) much but provide useful states for performance monitoring.

**Question 12: Look at below Garbage collection output and answer following question :**

[GC

       [ParNew: 1512K->64K(1512K), 0.0635032 secs]

       15604K->13569K(600345K), 0.0636056 secs]

       [Times: user=0.03 sys=0.00, real=0.06 secs]

 1. Is this output of Major Collection or Minor Collection ?

 2. Which young Generation Garbage collector is used ?

 3. What is size of Young Generation, Old Generation and total Heap Size?

 4. How much memory is freed from Garbage collection ?

 5. How much time is taken for Garbage collection ?

 6. What is current Occupancy of Young Generation ?

This Garbage collection Interview questions is completely based on GC output. Following are answers of above GC questions which will not only help you to answer these question but also help you to understand and interpret GC output.

**Answer 1**:  It's Minor collection because of "GC" word, In case of Major collection, you would see "Full GC".

**Answer 2**: This output is of multi-threaded Young Generation Garbage collector "ParNew", which is used along with CMS concurrent Garbage collector.

**Answer 3**: [1512K] which is written in bracket is total size of Young Generation, which include Eden and two survivor space. 1512K on left of arrow is occupancy of Yong Generation before GC and 64K is occupancy after GC. On the next line value if bracket is total heap size which is (600345K). If we subtract size of young generation to total heap size we can calculate size of Old Generation. This line also shows occupancy of heap before and after Garbage collection.

**Answer 4**: As answered in previous garbage collection interview question, second line shows heap occupancy before and after Garbage collection. If we subtract value of right side 13569K, to value on left side 15604K, we can get total memory freed by GC.

**Answer 5**: 0.0636056 secs on second line denotes total time it took to collect dead objects during Garbage collection. It also include time taken to GC young generation which is shown in first line (0635032 secs).

**Answer 6**: 64K

Here are few more interesting *Garbage collection Interview question* for your practice, I haven’t provided answers of all garbage collection interview questions. If you know the answer than you can  post via comments.

Question -  What is difference between -XX:ParallelGC and -XX:ParallelOldGC?

Question - When do you ConcurrentMarkSweep Garbage collector and Throughput GC?

Question -  What is difference between ConcurrentMarkSweep and G1 garbage collector?

Question -  Have you done any garbage collection tuning? What was your approach**?**

These were some Garbage collection interview questions and answers, may help on your Java Interview preparation. If you have got any interesting interview questions related to GC than don’t forget to share with us.