Garbage Collection

Garbage Collection (GC) is an automatic memory management feature in Java that automatically reclaims memory occupied by objects that are no longer referenced or in use by the program. It frees developers from explicitly deallocating memory, making memory management more convenient and reducing the chances of memory leaks.

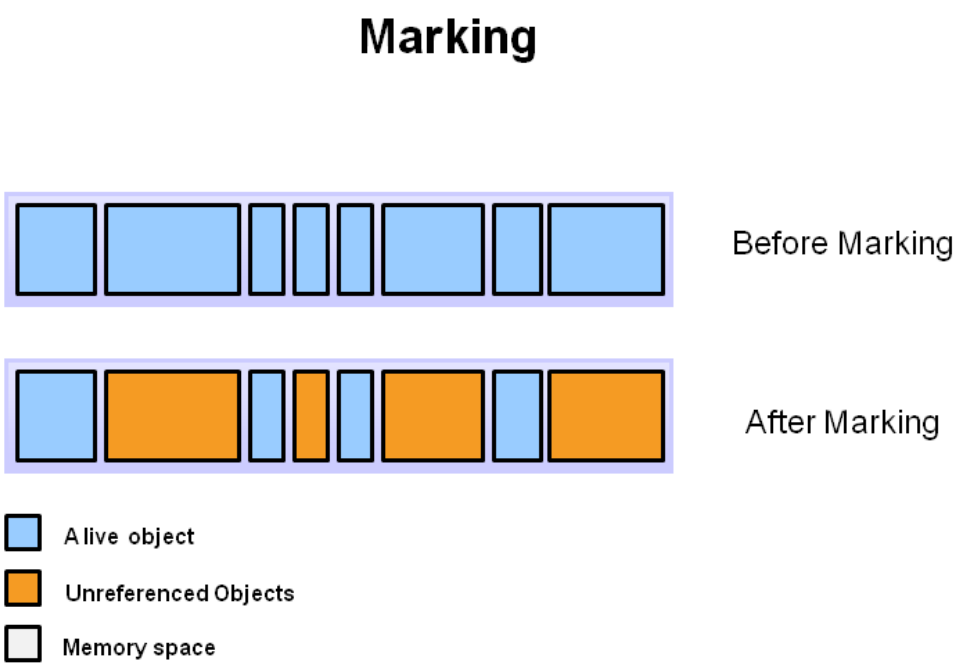
Here are some key points about garbage collection in Java:

Heap Memory:

In Java, objects are dynamically allocated on the heap memory. The heap is divided into different regions, including the young generation, old generation, and permanent generation (until Java 7) or metaspace (starting from Java 8).

Mark-and-Sweep Algorithm:

The garbage collection process typically involves a mark-and-sweep algorithm. It works in two phases:



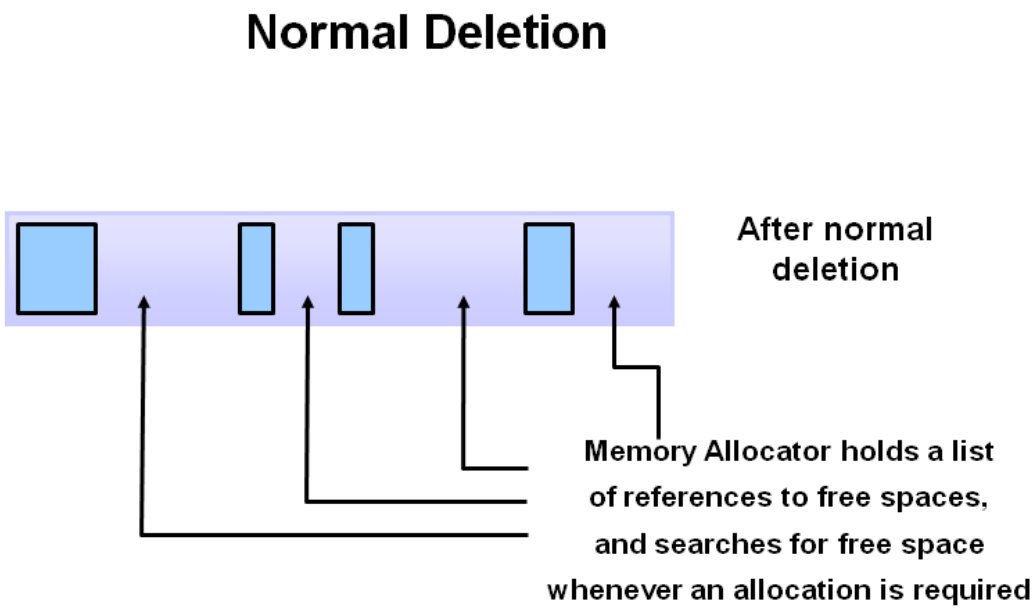
Marking phase:

The garbage collector identifies which objects are still reachable by traversing the object graph starting from the root objects (such as static variables, local variables, and method parameters).

Sweeping phase:

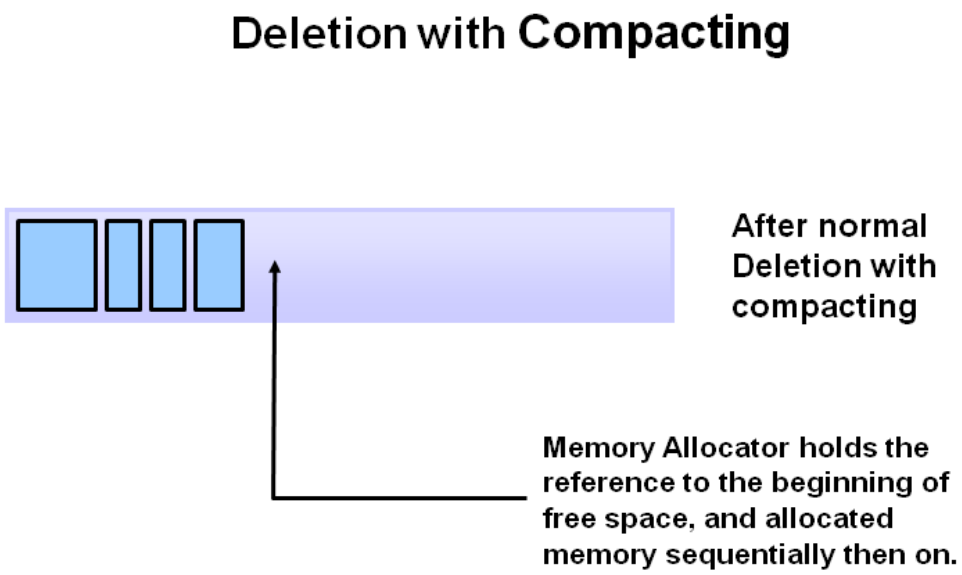
The garbage collector reclaims memory occupied by objects that were not marked as reachable. The memory is then made available for future object allocations.

Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.



The memory allocator holds references to blocks of free space where new object can be allocated.

Deletion with Compacting *To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.*



Garbage Collector Types:

Java provides different garbage collectors to accommodate various application requirements. Some commonly used garbage collectors are:

Serial:

A single-threaded collector suitable for small applications or environments with limited resources.

Parallel:

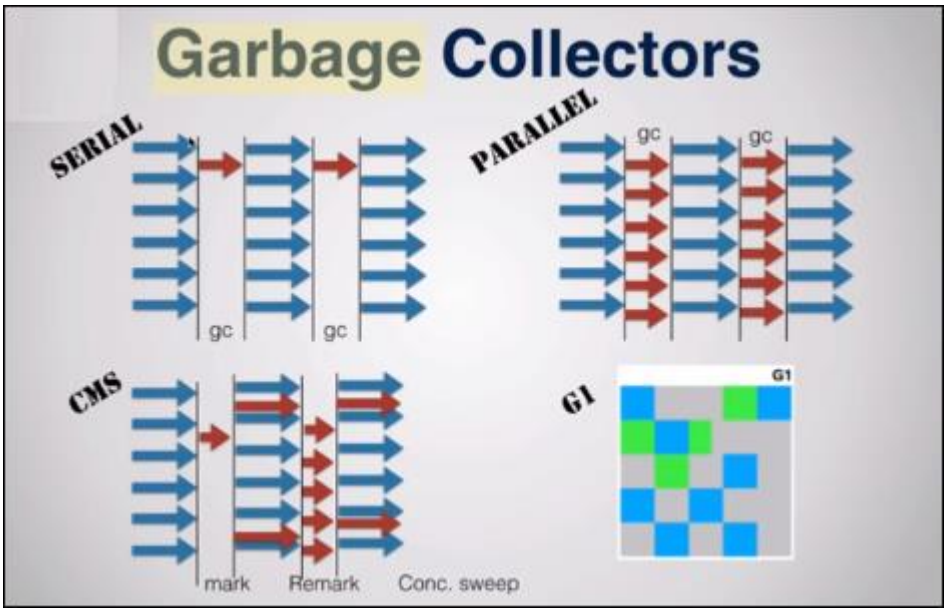
Uses multiple threads for garbage collection, suitable for applications that can benefit from parallelism.

CMS (Concurrent Mark Sweep):

Designed for applications that require reduced pause times and have large heaps.

G1 (Garbage-First):

Introduced in Java 7, it's a low-pause, region-based garbage collector that divides the heap into regions to perform concurrent and incremental garbage collection.



System.gc() and Finalization:

The System.gc() method can be called to suggest the JVM to run the garbage collector. However, there is no guarantee that it will execute immediately. Additionally, Java provides a finalize() method that can be overridden in a class to perform cleanup actions before an object is garbage collected. However, the usage of finalize() is discouraged due to its unpredictable nature and potential performance impact.

Tuning and Monitoring:

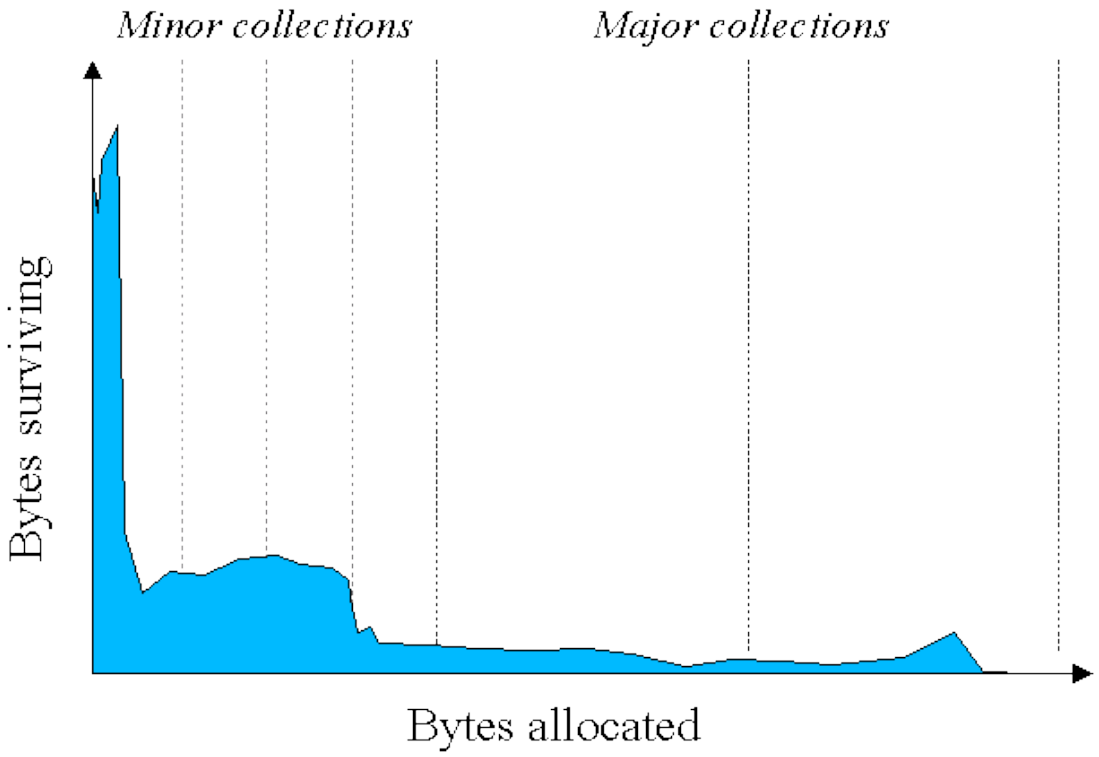
The garbage collector can be tuned and monitored to optimize memory management for specific application needs. JVM options like -Xms (initial heap size) and -Xmx (maximum heap size) can be used to control the heap size. JVM tools like JVisualVM, Java Mission Control, and GC logs can be employed to analyze garbage collection behavior and performance.

It's important to note that while garbage collection automates memory management, improper object lifecycle management or excessive object creation can still lead to performance issues. It's recommended to design applications with efficient memory usage and avoid unnecessary object allocation and retention.

Why Generational Garbage Collection?

As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

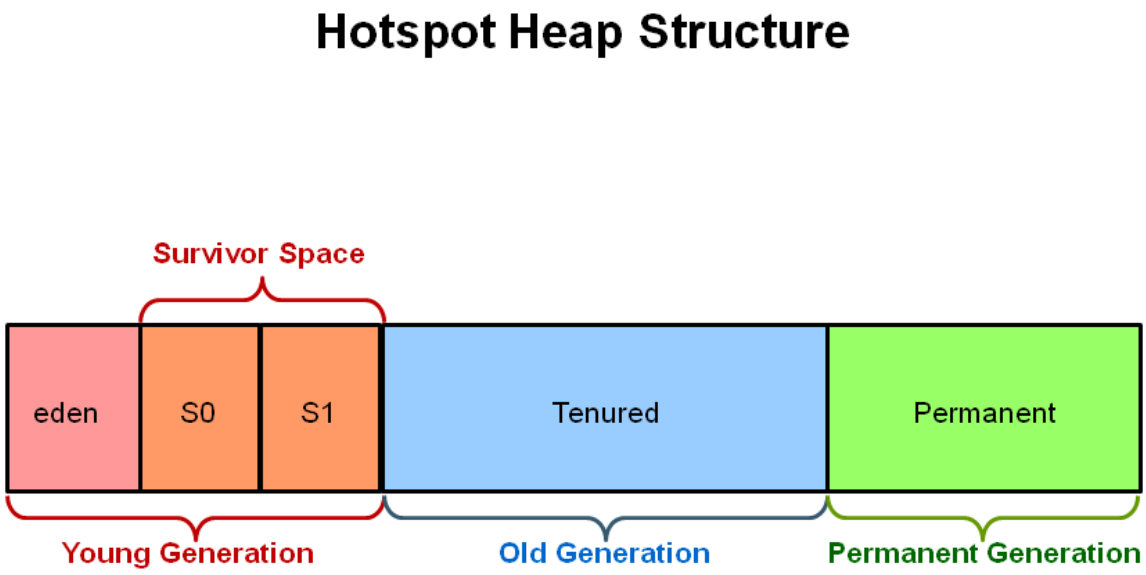
Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.



As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.

JVM Generations

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a ***minor garbage collection***. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

Stop the World Event - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are *always* Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a ***major garbage collection***.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

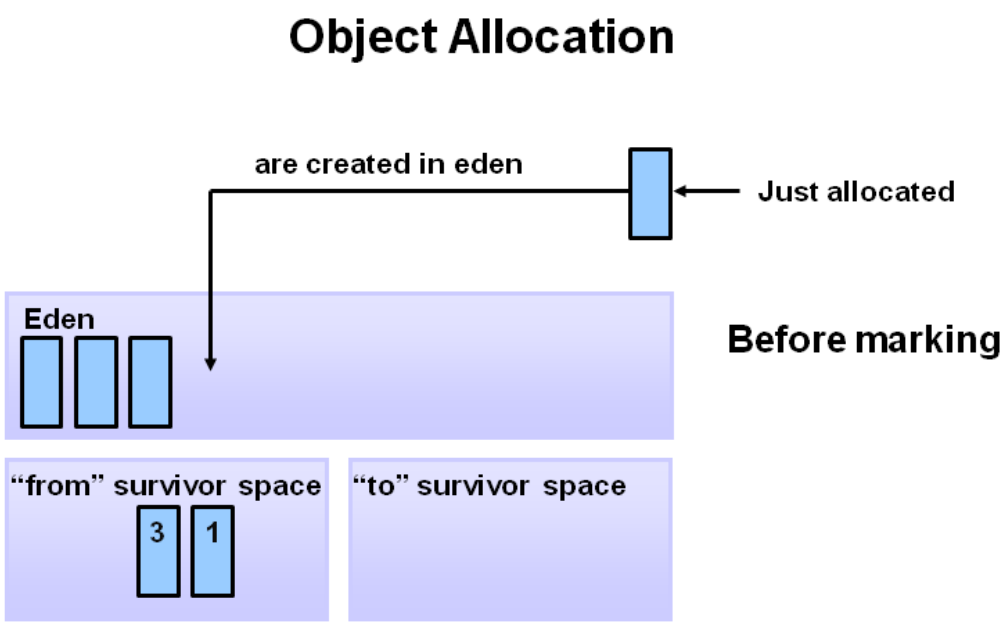
(2) Tuning GC with JVM 5 - Section 3 Generations

The Generational Garbage Collection Process

Now that you understand why the heap is separted into different generations, it is time to look at how exactly these spaces interact. The pictures that follow walks through the object allocation and aging process in the JVM.

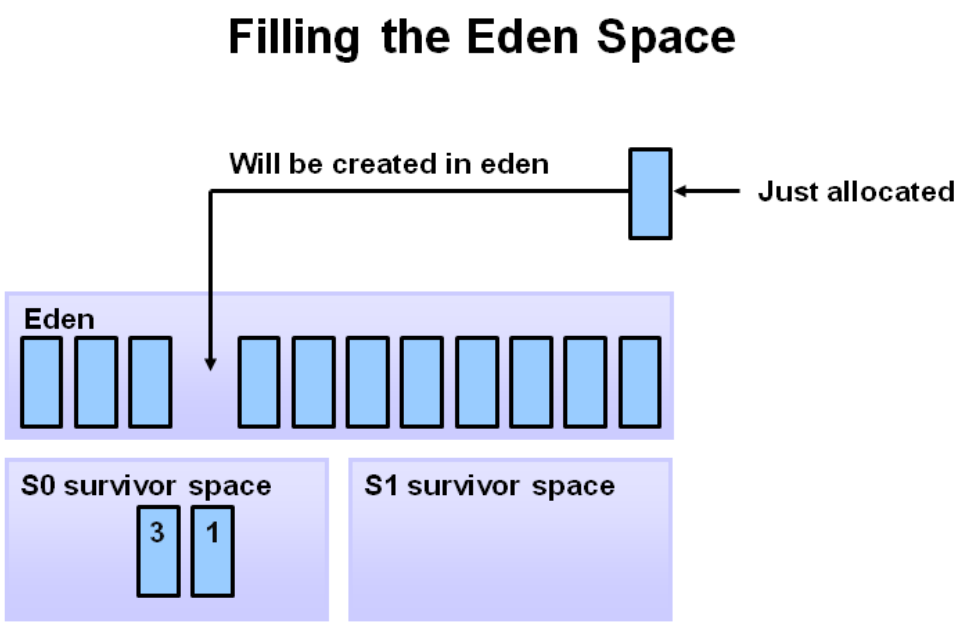
First, any new objects are allocated to the eden space. Both survivor spaces start out empty.

 Contact: 8087883669



When the eden space fills up, a minor garbage collection is triggered.

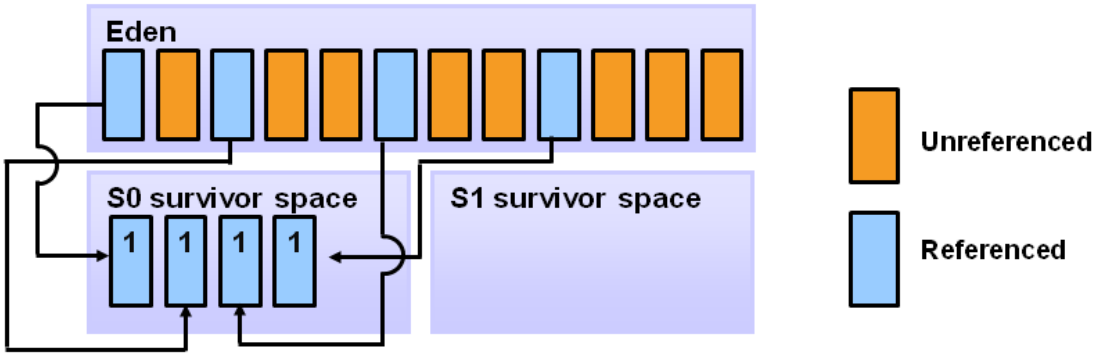
 Contact: 8087883669



Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.

 Contact: 8087883669

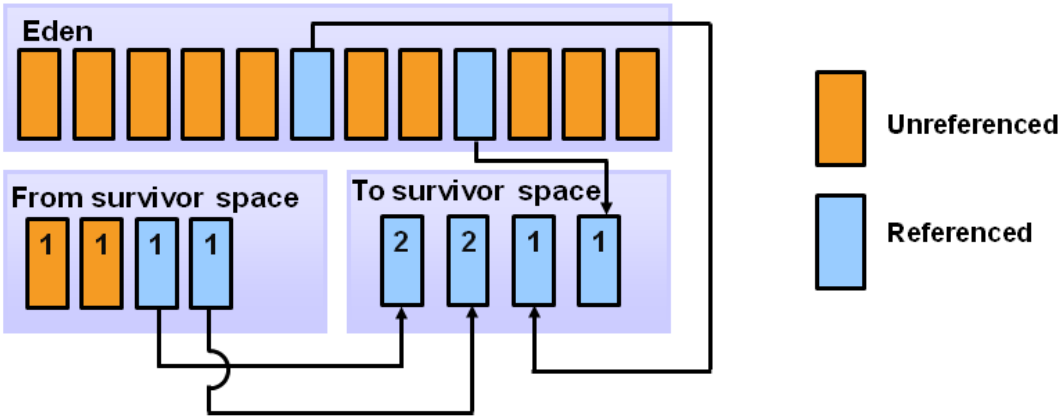
Copying Referenced Objects



At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.

 Contact: 8087883669

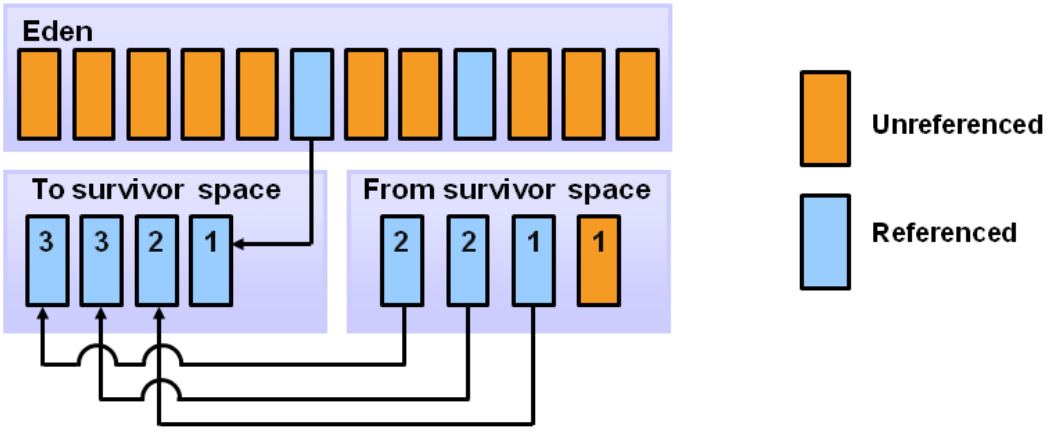
Object Aging



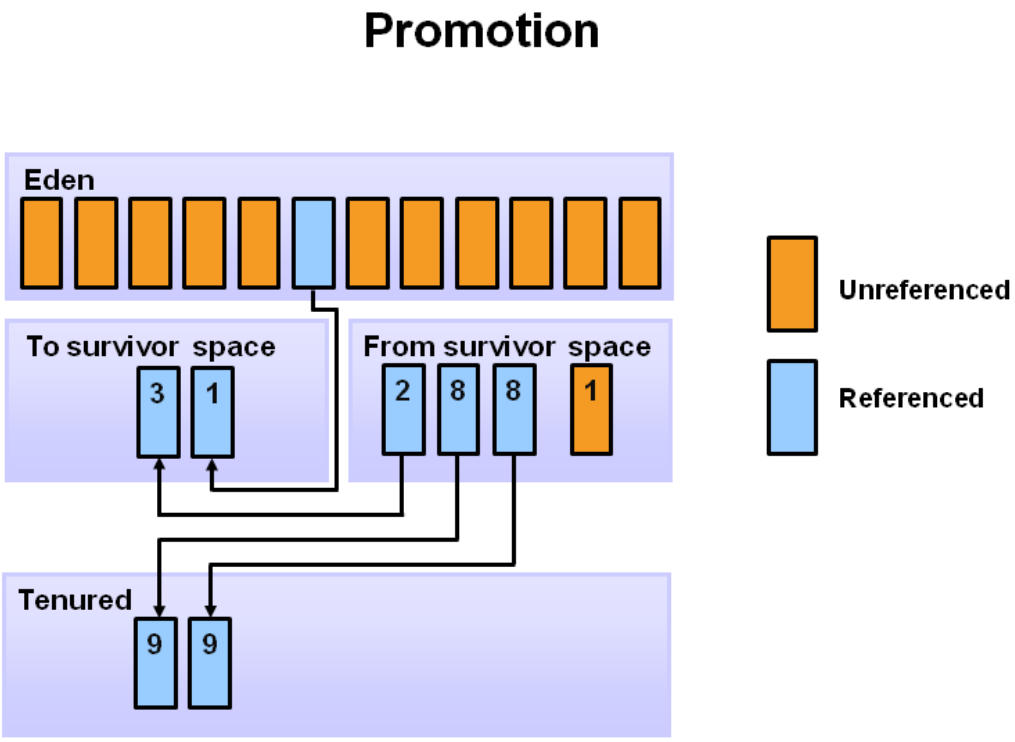
At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.

 Contact: 8087883669

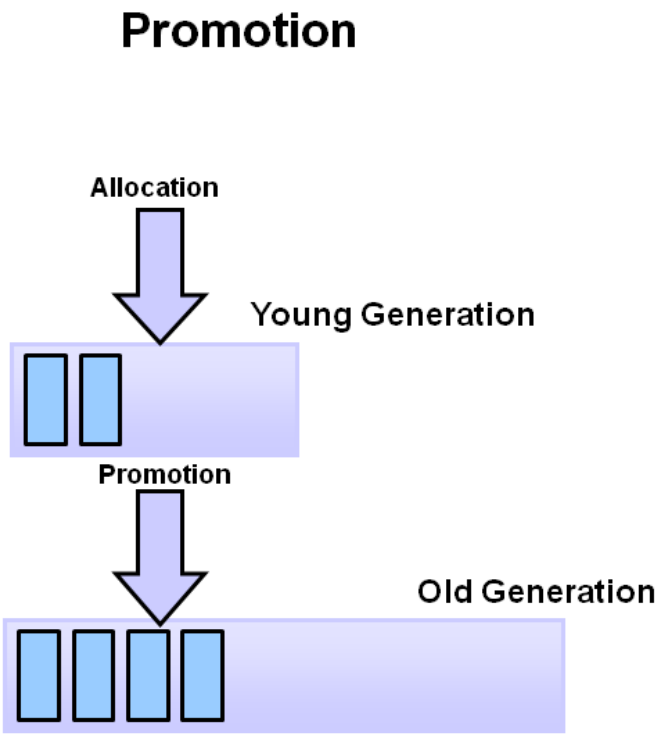
Additional Aging



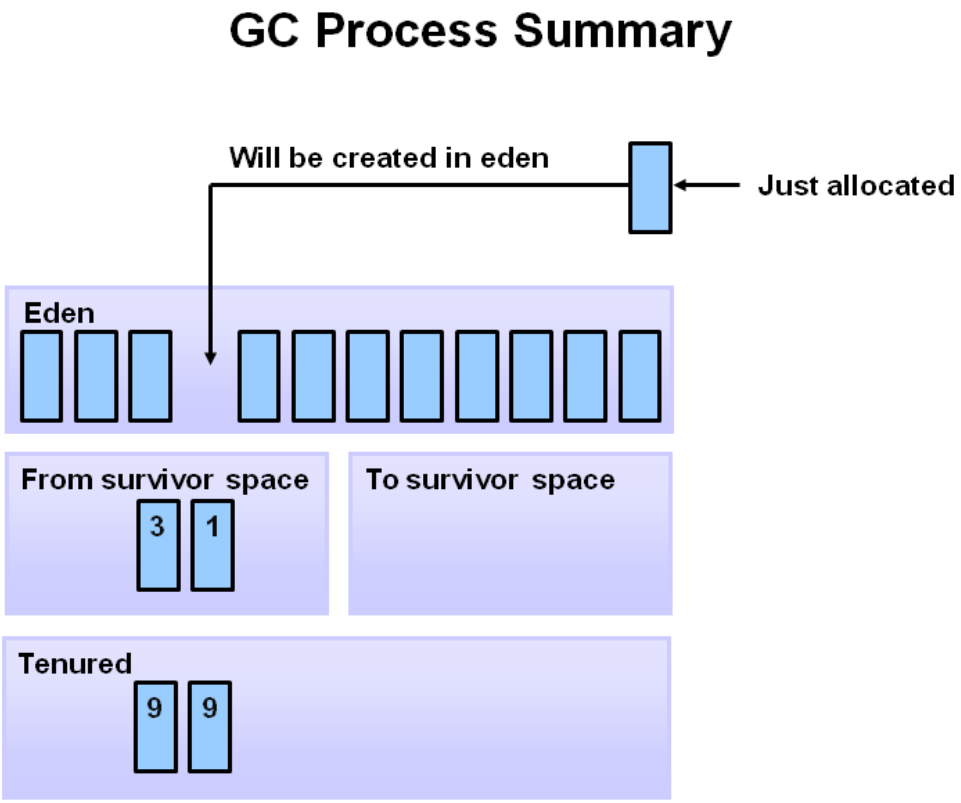
This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



As minor GCs continue to occure objects will continue to be promoted to the old generation space.



So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.



Garbage collection:

Useless objects are garbage, Garbage collector is responsible to collect garbage collection.

1. Introduction
    - **C / C++** - programmer is responsible to create and neglecting destroy of the object.
- new keyword used to create object.
- OutOfMemeryObject
- At certain point, creation of new object, sufficient memory not available b'z total memory filled with useless object.
- An total application will be down with the memory problems. Hence outofmemory error is very common problem in old languages like c++

Java - Programmer responsible to create object.
- Assistance, (GC) running in the background, to delete useless object
- less chance of memory related problems

- Java is robust language.
- But in java programmer is responsible only for creation of objects and programmer is not responsible to destroy useless objects
- SUN people provided one assistant to destroy useless objects this Assistance is always running in the background (daemon thread) and destroy useless objects.
- Just because of this assistance, the chance of failing java program with memory problems is very very low.
- This assistance is nothing but Garbage Collector
- Hence the main objective GC is to destroy useless objects.

2. The way to make an Object eligible for GC - 4 ways

When there is no reference to an object, it will be eligible for GC.
An object is said to be eligible to GC if an only if it does not contains any reference variable.
The following are various ways to make an object eligible for GC

1. Nullifying the reference variable

```
Student s=new Student();
    s=null; // will be eligible for GC
```

2. Reassigning the reference variable

```
Student s1= new Student();
s1=new Student(); // one object will be  eligible for GC
```

3. Object created inside a method.

i. The objects created inside an method will be eligible for GC once the method execution is complete.

case1 :

```
Class Test{

    Main(){
        m(); // once method complete , two objects will be eligible for GC
        }

        m(){
        Student s= new Student();
        Student s2=new Student();
        }
    }
```

Case 2:

```
Class Test{
```

```
Main(){

    Student s=m(); // one object eligible to GC

    }

    Pubilc static Student m(){
        Student s1=new Student();
        Student s2 = new Student(); // this will be gc
        Return s1;
        }
    }
```

Case 3:

```
Class Test{

    Main(){

    m(); //  two objects will be eligible for GC as no reference found here.
    }

    pubilc static Student m(){
        Student s1=new Student();
        Student s2 = new Student(); // this will be gc
        Return s1;
        }
    }
```

Case 3:

```
Class Test{

    Static Student s;

            Main(){
                m(); // here only one object will be eligible for GC
                }
            P s v m(){
                s=new Student();
                Student s2=new Student();
                }
        }
```

4. Island of isolation

```
Class Test{
    Test i;

    Public static void main(String[] r){
            Test t1=new Test();
```

```
Test t2=new Test();

Test t3=new Test();
// no one eligible for GC

t1.i= t2;
t2.i=t3;
t3.i=t1;
// no one eligible for GC


T1=null;
// no one eligible for GC


T2=null;
// no one eligible for GC

T3=null;
// 3 objects eligible for GC



}
```

If an object doesn't contains any reference it is always eligible for GC
Even though it has reference it will eligible for GC (all references are internal) there is no outside of reference variable.

3. The methods/ways for requesting JVM to run GC

1. once we made an object eligible for GC, it may not be destroyed immediately , when GC run then only object will be GC.

When exactly GC run we can't expect, it may vary from JVMto JVM

Until waiting for gC, we can request JvM to run GC explicitly, JVM will accept our request it is not guaranteed.
But most of the time, JVM accept our request. The following are two ways to requesting JVM to run GC.

a. By Using System class
   i. System.gc(); // requsting JVM to run GC
b. By Using Runtime Class
   i. Runtime. Java program can communicate with JVM
   ii. Runtime r=Runtime.getRuntime();
   iii. r.totolMemory();
   iv. r.freeMemory();

v. r.gc();

Java application can communicate with JVM, runtime present in java.lang pacakge it is a singleton class
We can create runtime object using runtime.getruntime() method.

System.gc() // static method
Runtime.getRuntime().gc(); // gc() is instance method not static.

- System.gc();
  Runtime.gc();//invalid
  (new Runtime).gc(); //invalid
- Runtime.getRuntime().gc();

- Runtime.getRuntime().gc(); -- is recommonded.
  System.gc() internally it is calling Runtime.getRuntime().gc();

It is convinient to use System.gc() method
With respect to performace , it is highly recomonded to use runtime.getruntime().gc(); because system internally calls runtime.gc() method.

4. Finalization

Just before destroying an object, GC is calling finalize method to perform cleanup activities.
Once finalize method completes , GC destroyed that object.

Finalize method present in object with the following declaration.

```
@Override
protected void finalize() throws Throwable{
}
```

We can override finalize method in our class to define our own cleanup activities.

Case 1:

```
Class Test{

    Psvm(){
        String s=new String("Ravi");
        s=null;

        System.gc(); // new thread will be created
```

```
        Syso("end");

        }

Public void finalize(){
        Sop("clean");
        }
```

Output:end
: there is no clean method found in the output:

Reason : Finalize method call on the object who is going to destroy.
Here string object is eligible for garbage collection.
So GC all finalize method of String and String class finalize method has no implementation.

Case 2:

```
Class Test{

    Psvm(){
        Tests=new Test("Ravi");
        s=null;

        System.gc(); // new thread will be created
        Syso("end");

        }

    Public void finalize(){
        Sop("clean");
        }
```

Here Test object is eligible for gc() so Test class finalize method will be executed.

Just before destroying an object. GC calls finalize method on the object which is eligible for GC
Then correspond class finalize method will be executed.

If String object eligible for gc then String class finalize method will be executed.
But not the test class finalize method.

Case 1- string object eligible for GC, hence finalize method of String class has executed. Hence output is end.

Case 2- Test object is eligible for GC , hence test class finalize method will be executed.

End
Clean

Or

Clean
End

Both case may be possible in the output.

Case 3:

```
Class Test{

    Psvm(){

        Test t=new Test();
        finalize() // 1 time call, we can call finalize method explicitly, but object will not
        be eligible for GC here.

        finalize(); //2 time,  like a normal method call, object still not destroyed.
        t=null;
        System.gc(); // 3 gc class finalize
        Sop("end");


        }

        Public void finalize(){
            Sop(clean);
        }
```

Above program, finalize method call 3 times, when GC class finalize then only object will be destroyed.

Servlet - life cycle method of servlet.

```
Init()
Service()
Destory(); // like a normal method call, servlet object will not  be .destroyed
// this will be called by GC
//Webcontainer calls clean up activities,
```

```
package com.pune.it;

public class TestFinalizeDemo {

	static TestFinalizeDemo r;
	public static void main(String[] args) throws InterruptedException {

		TestFinalizeDemo f=new TestFinalizeDemo();
		System.out.println(f.hashCode());
		f=null;
		System.gc();//finalize will be called.

		Thread.sleep(5000);
		System.out.println(r.hashCode());
		r=null;
		System.gc();// finalize wont call here
		Thread.sleep(10000);
		System.out.println("END");

	}


	@Override
	protected void finalize() throws Throwable {
		// TODO Auto-generated method stub
		super.finalize();

		System.out.println("finalize");
		r=this;
	}
}
```

Even though object eligible for GC multiple times, but GC called finalize method only once.

In above program object eligible for GC two times, but GC call Finalize method only once.

```
package com.pune.it;

public class TestGC {

	static int count=0;
	public static void main(String[] args) {


		for (int i = 0; i < 10000000; i++) {// keep on increasing this number, at certain point memory
		problem will raised, then JVM runs GC, GC calls finalize method on every object and destroys
		that object.
```

```
                TestGC t=new TestGC();
                t=null;
            }
        }


        @Override
        protected void finalize() throws Throwable {
            super.finalize();
            System.out.println("finalize method called = "+count++);
        }
}
```

We can't expect exact behavirour of GC , it is varied from JVM to JVM.
Hence for the following question, we can't provide exact answers

1. We exact JVM runs GC
2. In which order GC identifies eligible objects,
3. In which order GC destroyed eligible objects,
4. Whether GC destroyed all eligible objects or not
5. What is algorithm followed by GC

Mark and Sweep algorithm - 90/100 GC follow this algorithm

Note

1. Whenever program runs with low memory, then JVM runs GC, but we can't expect what exact at what time.
2. Most of the GC follows, standard algorithm (Mark and sweep Algorithm), it doesn't mean every GC to follow the same Algorithm

Memory Leaks

Student s= new Student();
Student s1= new Student();
….

Student s1000000000000= new Student();

---------- Memory Leaks

OutOfMemoeryError come.

The object which are not in use program and also not eligible for GC, such type of useless objects are called Memory Leaks.

In our programs, if Memory leaks present then the program will be terminated by out of memory error.

Hence if an object no longer required, it is highly recommended to make that object eligible for GC.

There are some third party tools that will identify objects for memory leaks.

The following are third party to identify memory leaks,

1. HP OVO
2. HP Jmeter
3. Jprobe
4. Patrol
5. IBM Trivoli
6. JProfile

GC tuning

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:+UseP-
arallelGC -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:+UseP-
arallelOldGC -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:\ja-
vademos\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:\ja-
vademos\demo\jfc\Java2D\Java2demo.jar
```