# Topic: Java Garbage Collection

**Course :Object Oriented Proogramming**

**Paper Code: AIDS202/AIML202/IOT202**

**Faculty : Dr. Shivanka**

**Assistant Professor**

**VIPS**

## Introduction

In Java, allocation and de-allocation of memory space for objects are done by the garbage collection process in an automated way by the JVM. Unlike C language the developers need not write code for garbage collection in Java. This is one among the many features that made Java popular and helps programmers write better Java applications.

Java garbage collection is an automatic process to manage the runtime memory used by programs. By doing it automatic, JVM relieves the programmer of the overhead of assigning and freeing up memory resources in a program.

Garbage collection is the process of reclaiming the unused memory space and making it available for the future instances.

# How Java Garbage Collection Works?

## Java Garbage Collection GC Initiation

Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. `System.gc()` and `Runtime.gc()`are hooks to request the JVM to initiate the garbage collection process.

Though this request mechanism provides an opportunity for the programmer to initiate the process but the onus is on the JVM. It can choose to reject the request and so it is not guaranteed that these calls will do the garbage collection. This decision is taken by the JVM based on the "eden" space availability in heap memory. The JVM specification leaves this choice to the implementation and so these details are implementation specific.

# How Java Garbage Collection Works?

## Memory Generations

HotSpot VM's garbage collector uses generational garbage collection. It separates the JVM's memory into two parts and they are called young generation and old generation.

## Young Generation

Young generation memory consists of two parts, Eden space and survivor space. Shortlived objects will be available in Eden space. Every object starts its life from Eden space. When GC happens, if an object is still alive then it will be moved to survivor space and other dereferenced objects will be removed.

## Old Generation – Tenured and PermGen

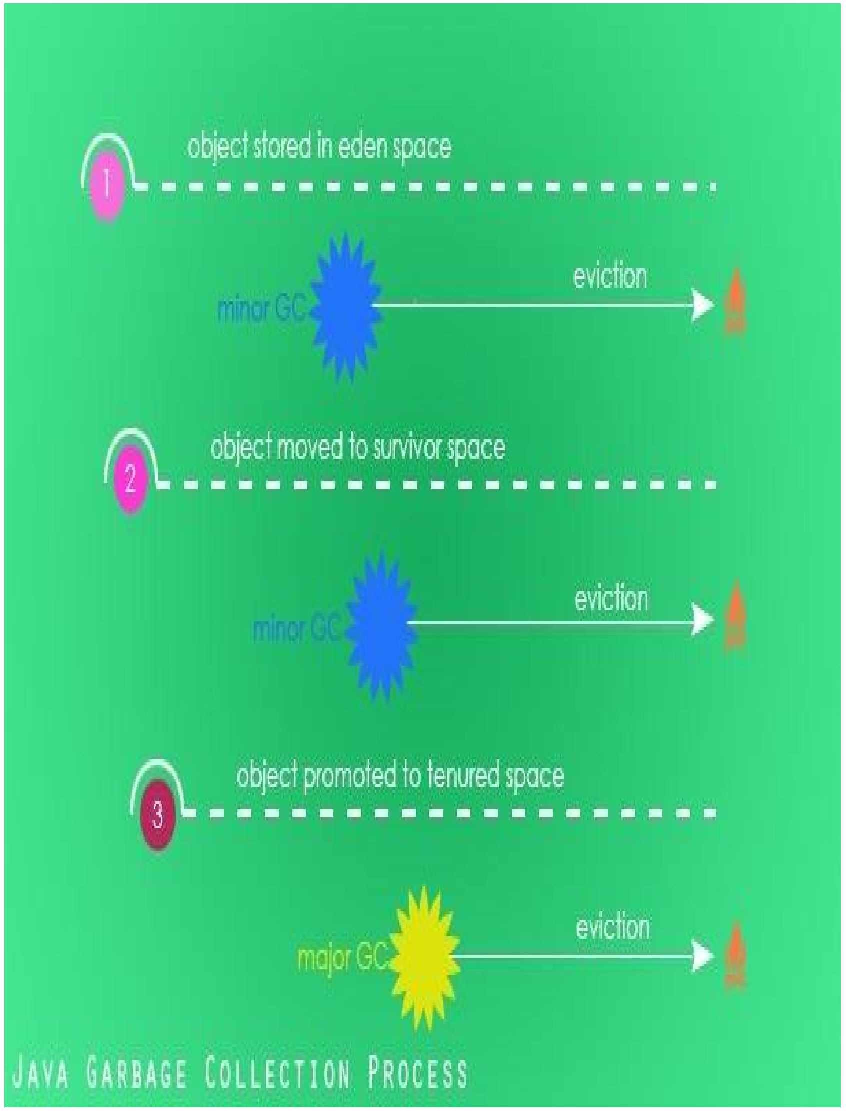
Old generation memory has two parts, tenured generation and permanent generation (PermGen). PermGen is a popular term. We used to get error like PermGen space not sufficient.

GC moves live objects from survivor space to tenured generation. The permanent generation contains meta data of the virtual machine, class and method objects.

# Java Garbage Collection Process

**Eden Space:** When an instance is created, it is first stored in the "eden" space in young generation of heap memory area.

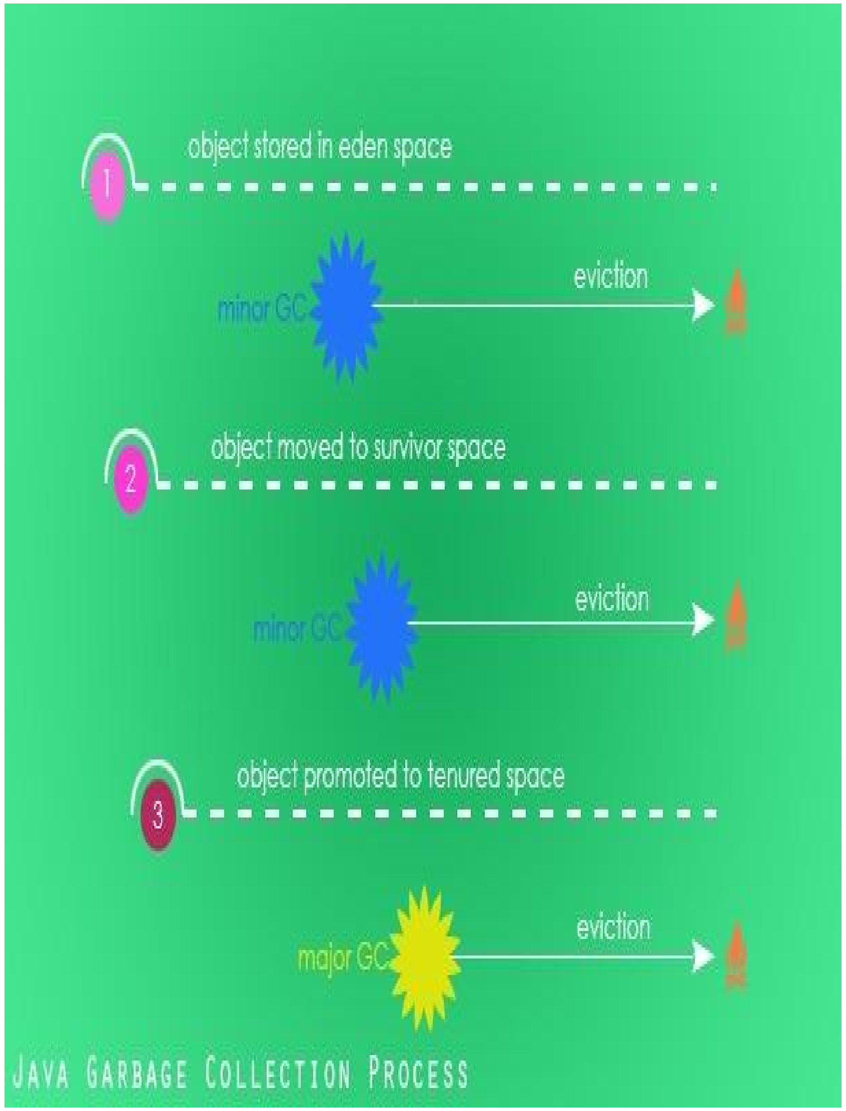
**Survivor Space (S0 and S1):** As part of the minor garbage collection cycle, objects that are live (which is still referenced) are moved to survivor space S0 from eden space. Similarly the garbage collector scans S0 and moves the live instances to S1.
Instances that are not live (dereferenced) are marked for garbage collection. Depending on the garbage collector (there are four types of garbage collectors available) chosen either the marked instances will be removed from memory on the go or the eviction process will be done in a separate process.


1
object stored in eden space
minor GC
eviction
2
object moved to survivor space
minor GC
eviction
3
object promoted to tenured space
major GC
eviction
JAVA GARBAGE COLLECTION PROCESS

# Java Garbage Collection Process

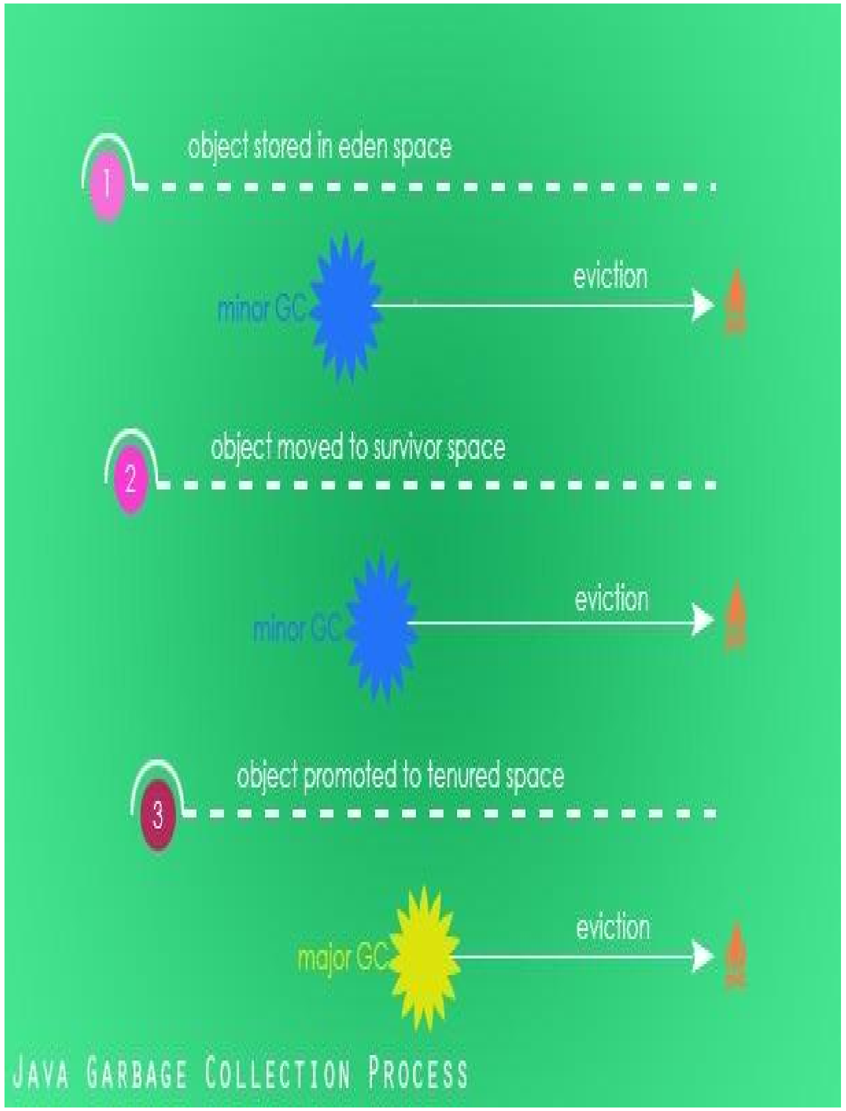
**Old Generation:** Old or tenured generation is the second logical part of the heap memory. When the garbage collector does the minor GC cycle, instances that are still live in the S1 survivor space will be promoted to the old generation. Objects that are dereferenced in the S1 space is marked for eviction.

**Major GC:** Old generation is the last phase in the instance life cycle with respect to the Java garbage collection process. Major GC is the garbage collection process that scans the old generation part of the heap memory. If instances are dereferenced, then they are marked for eviction and if not they just continue to stay in the old generation.


1
object stored in eden space
minor GC
eviction
2
object moved to survivor space
minor GC
eviction
3
object promoted to tenured space
major GC
eviction
JAVA GARBAGE COLLECTION PROCESS

# Java Garbage Collection Process

**Memory Fragmentation:** Once the instances are deleted from the heap memory the location becomes empty and becomes available for future allocation of live instances. These empty spaces will be fragmented across the memory area. For quicker allocation of the instance it should be defragmented. Based on the choice of the garbage collector, the reclaimed memory area will either be compacted on the go or will be done in a separate pass of the GC.

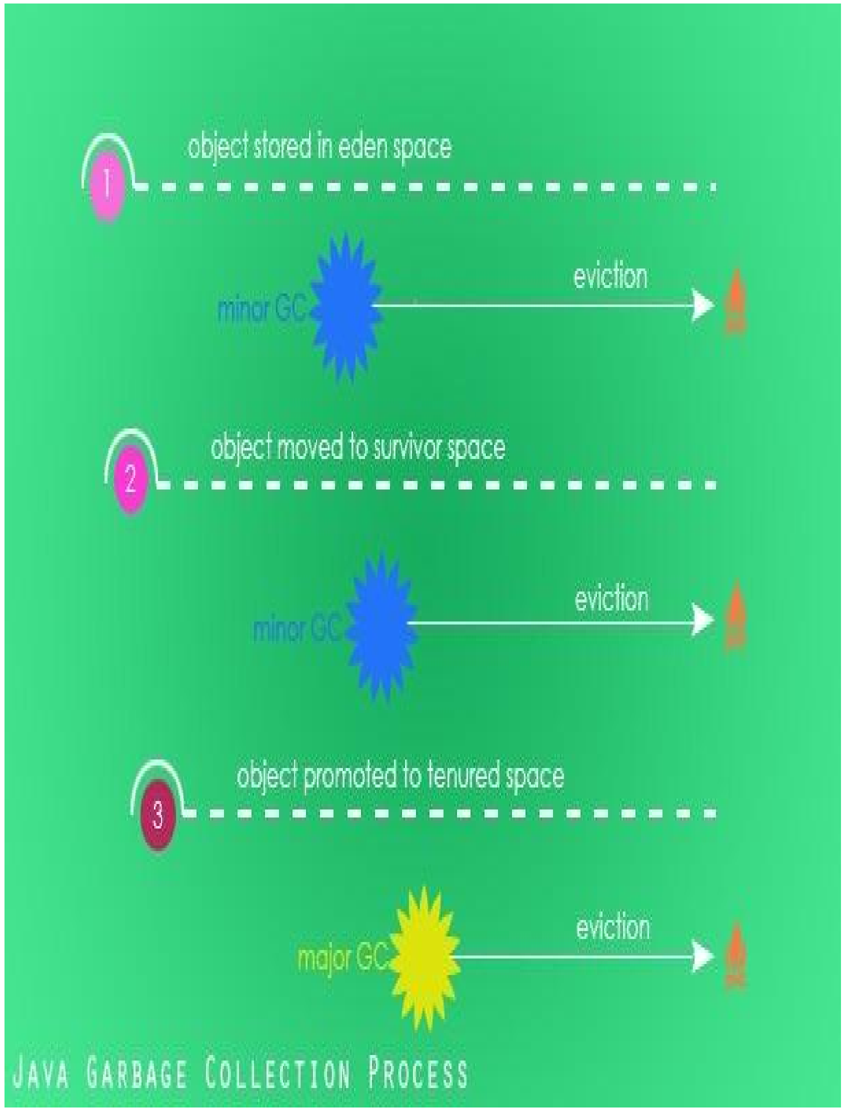

1
object stored in eden space
minor GC
eviction
2
object moved to survivor space
minor GC
eviction
3
object promoted to tenured space
major GC
eviction
JAVA GARBAGE COLLECTION PROCESS

# Java Garbage Collection Process

## Finalization of Instances in Garbage Collection

Just before evicting an instance and reclaiming the memory space, the Java garbage collector invokes the `finalize()` method of the respective instance so that the instance will get a chance to free up any resources held by it.

Though there is a guarantee that the `finalize()` will be invoked before reclaiming the memory space, there is no order or time specified. The order between multiple instances cannot be predetermined, they can even happen in parallel. Programs should not pre-mediate an order between instances and reclaim resources using the `finalize()` method.


1
object stored in eden space
minor GC
eviction
2
object moved to survivor space
minor GC
eviction
3
object promoted to tenured space
major GC
eviction
JAVA GARBAGE COLLECTION PROCESS

## Example Program for GC OutOfMemoryError

Garbage collection does not guarantee safety from out of memory issues. Mindless code will lead us to `OutOfMemoryError`.

```
import java.util.LinkedList;
import java.util.List;
public class GC { public static void main(String[] main)
{
List l = new LinkedList(); // Enter infinite loop which will add a
String to the list: l on each // iteration.
do { l.add(new String("Hello, World")); }
while (true); } }
```

Output:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.LinkedList.linkLast(LinkedList.java:142) at
java.util.LinkedList.add(LinkedList.java:338) at
com.javapapers.java.GCScope.main(GCScope.java:12)
```