Welcome

Garbage Collector in Java

Prakash Badhe

prakash.badhe@vishwasoft.in

Copyright Notice

Memory Management

- Memory management plays an important role in determining the performance of an application.
- It involves recognizing when allocated objects are no longer needed, de-allocating the memory used by such objects, and eventually making it available for further use in allocating objects.
- Managing memory manually is a complex task as it can cause unexpected or erroneous program behavior and crashes.

Issues in Memory Management

- **Memory leaks**
- **Dangling references**
- **Heavy allocations**
- **Process in Memory even after termination**
- **Heavy swapping of memory with Hard Disk**

Garbage Collector in Java

- The garbage collector in java is the background daemon thread monitoring and managing the memory of the JVM automatically.
- With this GC, we don’t have to explicitly manage the lifecycle of objects, i.e. objects are created when needed, and when the object is no longer in use, the JVM automatically frees the object with the help from GC.

Object Eligibility for Collection

- An object is eligible for garbage collection when no live thread can access it.
- The garbage collection revolves around ensuring that heap has enough memory available for allocating objects

The GC Process

- **Marking: In** this step, the garbage collector identifies which objects are used and which objects are not being used .
- **Normal Deletion: The** garbage collector removes the unused objects and reclaims the free space which can be used for further allocating other objects.

Deletion with Compaction

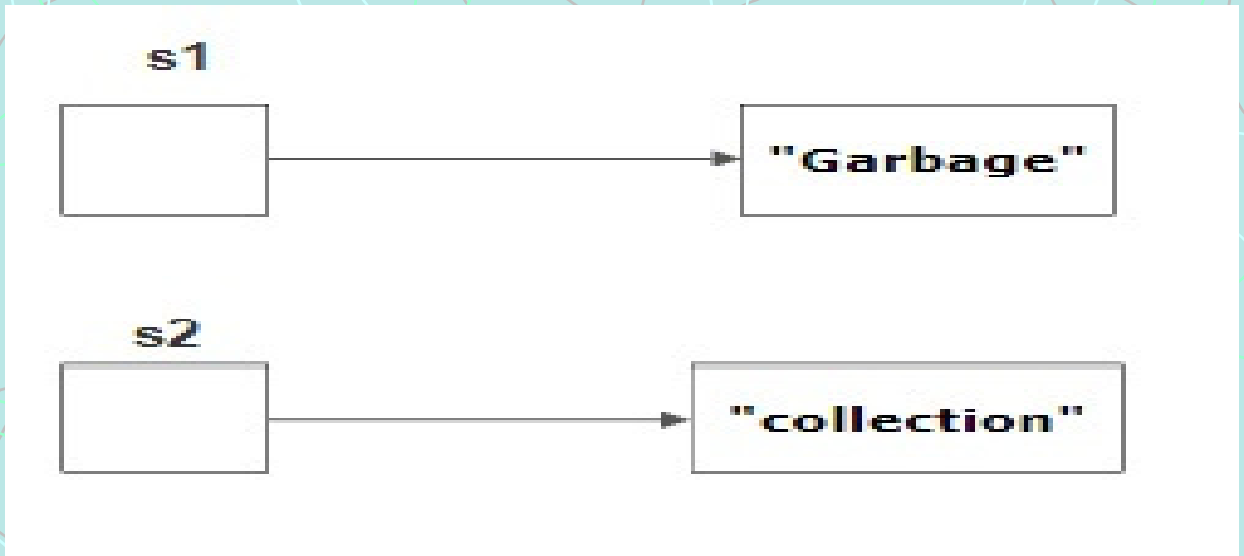
- Since there can be free spaces left in small chunks at the different location of the heap when the objects are freed, there might be a chance that enough memory is not available in any one contiguous area for allocating objects that require large memory.
- To avoid such a memory fragmentation scenario, the garbage collector does the process of compaction of memory.

Java Object for Garbage Collection

- **Assign null to the Reference Variable.**
- By assigning null to a reference of an object will make it no longer accessible.
- String s1 = new String(“Iset”);
- The s11 is not eligible for collection.
- Here s1=null; will make it eligible for garbage collection and accumulated to the heap once the garbage collector runs for the cleanup process.
-

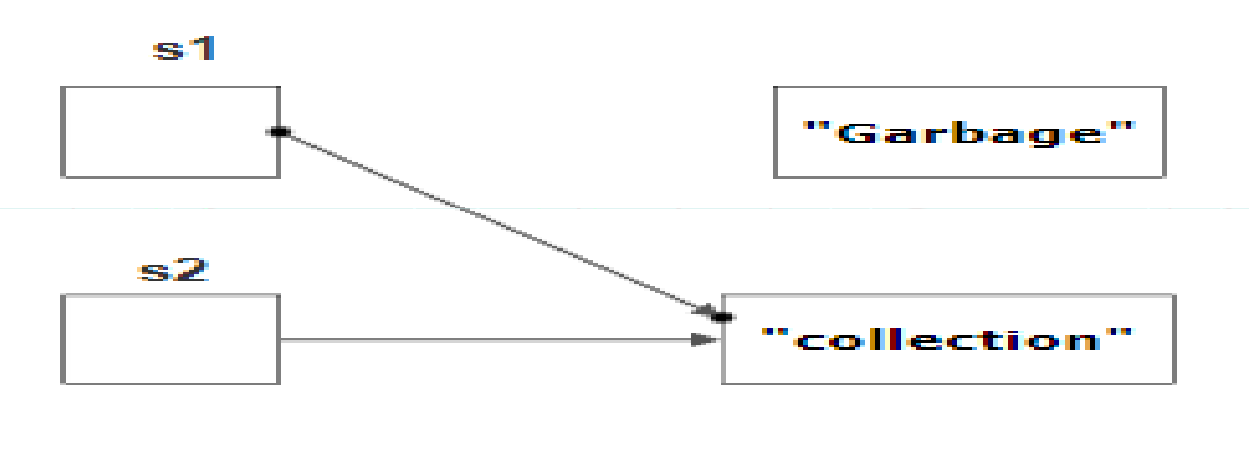
Reassigning a Reference Variable

- StringBuffer s1 = new StringBuffer("Garbage");
  StringBuffer s2 = new StringBuffer("collection");



Change the reference

- s1 = s2; // Redirects reference variable s1 to refer to the s2 object



The object referred earlier by s1 with value “Garbage” eligible for garbage collection as it cannot be accessed by any thread.

Life of Local Variable

- The lifetime of a local variable is up until the method is executing.
- Thus, an object created in the method will be eligible for garbage collection once the same method has completed its execution.
- The only exception is when an object is returned from the method, Its reference may be assigned to some reference variable in the method that called it, then the returned object will not be eligible for garbage collection.

```
class Student{
    int id;
    String name;
    public Student(int id, StringBuffer name) {     this.id = id;
        this.name = name;    }
    public Student makeStudentObject() {
        StringBuffer name = new StringBuffer("Suyash");
        Student s1 = new Student(1, name);
        return s1;    }
    public static void main(String [] args) {
        Student s = makeStudentObject();
        //do other stuff
    }
}
```

Local references

- The name and s1 are two local variable present in makeStudentObject() method.
- Student object, s1, is returned by the method and it being inside main() method so it will not be eligible for garbage collection.
- But the StringBuffer object, name, will be eligible for garbage collection as it not returned by a method. In other words, *s1* is still alive and the *name* did not survive.

Anonymous Objects

- Since the reference to an anonymous object is not stored anywhere, it can't be accessed by any thread.
- Hence it can be regarded as objects which will be eligible for garbage collection.

Anonymous under the GC's sword..

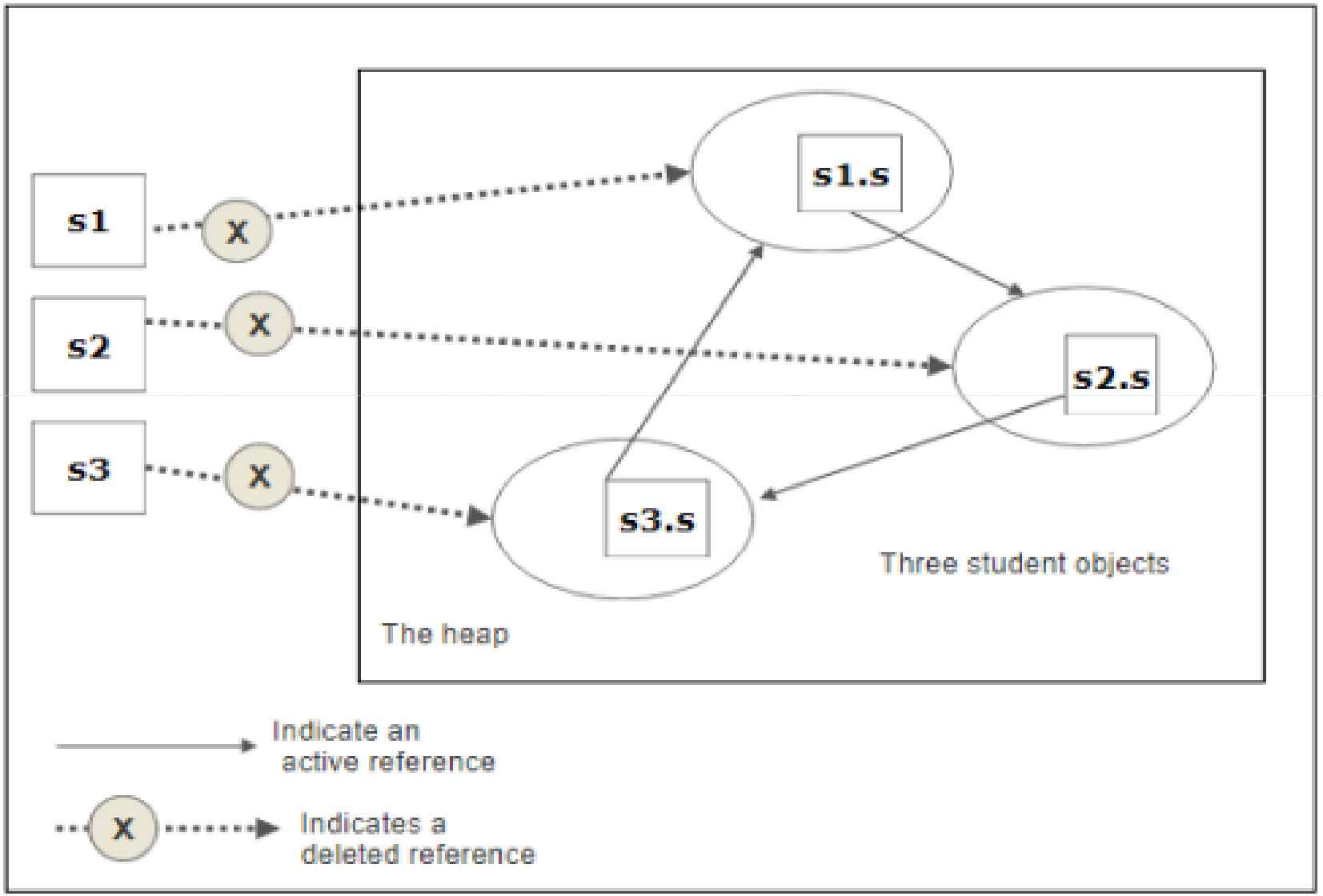
```
class Employee {
        String empName;
        public Employee(String name) {
            this.empName = name;        }
        public static void main(String[] args) {
                new Employee("Avantika");
// anonymous object without any reference
                // do other stuff       } }
```

Isolated Objects

- Objects can become eligible for garbage collection, even if they have correct references.
- For example, a class has an instance variable that is a reference variable to another instance of the same class. Suppose there are three objects and they refer to each other.
- If the original references to the objects are removed, then even if they have a valid reference, they won't be accessible .
- This condition is known as island of isolation.

```
class Student {
  Student s;
   public static void main(String [] args) {
      Student s1 = new Student();
      Student s2 = new Student();
      Student s3 = new Student();
      s1.s = s2; // s1 refers to s2
      s2.s = s3; // s2 refers to s3
      s3.s = s1; // s3 refers to s1
      s1 = null;
      s2 = null;
      s3 = null;
      // do other stuff    }
```

Isolated References



Invoke the GC..?

- System.gc()
- Runtime.gc()
- Invoking either of them does not guarantee all the unused objects will be removed.
- It will only make a request to JVM for some garbage collection.

Object Clean UP

- Java provides a finalize() method present in Object class which you can override for objects to do some task before it gets deleted by the garbage collector.
- It does not guarantee that it will run on every object.

Terms of finalize methods

- For each object, finalize() method will be invoked at most ONCE by the garbage collector.
- The finalize() can severely affect the performance of the application.
- Any Exception thrown by finalize() method is ignored by GC thread and it will not be propagated further.

Types of GC

- Serial Collector
- Parallel Collector
- The Mostly Concurrent Collectors
- The main operations which the garbage collector is responsible for:
  - Finding objects which are no longer used.
  - Freeing up the memory after those objects.
  - Compacting the heap.

Concurrent Mark and Sweep (CMS)

- Minor GC is performed with multiple threads using the **parallel mark-copy** algorithm.
- All application threads are stopped then.
- The Old Generation is mostly collected concurrently - application threads are paused for very short periods of time when the background GC thread scans the Old Generation.
- The algorithm used during Major GC is concurrent mark-sweep.

Garbage First Policy

- The Garbage First (G1) is a new low-pause garbage collector designed to process large heaps with minimal pauses.
- The heap is broken down into several regions of fixed size (while still maintaining the generational nature of the heap).

Generational Garbage Collectors

- JVM heap is divided into two different Generations.
- One is called Young and the second one is the Old (sometimes referred to as Tenured).
- The Young Generation is further separated into two main logical sections: Eden and Survivor spaces.
- There are also Virtual spaces for both Young and the Old Generations which are used by Garbage Collectors to resize other regions – mainly to meet different GC goals.

Memory Leaks

- Memory leaks are a silent killer.
- They start small and live during an initial time without anybody realizing it.
- With time, they keep growing, piling up, and accumulating.
- When you realize they're there, it's already too late: the entire code base is scattered with memory leaks, and finding a solution takes an enormous amount of effort.

What is Memory Leak..?

- A memory leak happens when an object that is no longer used is still referenced in-memory by another object.
- Eating up the available memory is the most direct result, but the effect of running low on memory makes the Garbage Collector (GC) start triggering more frequently.
- When the GC triggers, the world stops.

Causes of memory Leaks

- Not closing an open stream
- static fields for holding references: too many static fields for holding references to objects, will create memory leaks.
- HashSet that is using an incorrect hashCode() : the HashSet will start increasing in size, and objects will be inserted as duplicates.

Avoid Memory Leaks..

- Avoid using non-static inner classes .
- Avoid unnecessary local variables.
- Instead of returning from methods, assign local variables to global variables.
- Make unused objects equal to null.
- Avoid dangling references.
- Use Reference Objects to Avoid Memory Leaks.
- Avoid using string concatenation

Reference Objects

- Reference Objects to Avoid Memory Leaks
- PhantomReference, SoftReference, and WeakReference from java.lang.ref package allow to refer to objects indirectly.
- These reference objects are wrapper around actual objects.

Referent from Wrapper

- A referent, or an object that is referenced by these subclasses, is accessed using that reference object's get method.
- The advantage of using this method is that you can clear a reference easily by setting it to null and that the reference is pretty much immutable, or it cannot be changed.

GC with References..?

- SoftReference object: Garbage collector is required to clear all SoftReference objects when memory runs low.
- WeakReference object: When garbage collector senses a weakly referenced object, all references to it are cleared and ultimately taken out of memory.
- PhantomReference object: Garbage collector would not be able to automatically clean up PhantomReference objects, you would need to clean it up manually by clearing all references to it.
- Using reference objects, you can work with the garbage collector to automate the task of removing listeners that are weakly reachable.

Java Flight Recorder

- The Java Flight Recorder is a diagnostic and profiling tool that gives more information about a running application.
- It gives you better data than those provided by other tools, allows APIs created by third party services, and lowers your total cost of ownership.

Eclipse Memory Analyzer

- Memory Analyzer (MAT) from Eclipse .
- To analyze Java heap to search for memory look and lower memory use.
- Easily analyze heap dumps even when there are millions of objects living in them, and see the sizes of each object and why Garbage Collector is not deleting it from memory.