Garbage Collection

Topics to be covered in this chapter

- Introduction
- The ways to make an object eligible for Garbage Collector
- The methods for requesting JVM to run Garbage Collector
- Finalization

Introduction

- Garbage in programming means any unused objects. Garbage Collection is mainly responsible for removing of unused objects.
- In old programming language like C++, Programmer is responsible for Object creation as well as destruction of Useless Objects. Usually, programmer takes very care while creating object but while destroying objects they don't show much care. That's the reason memory gets filled with large number of unused objects and we get OutOfMemoryError.
- But in modern programming language like Java, programmers are responsible for Object creation only but for destruction of unused objects there is an assistant called Garbage Collector (GC) which runs in the background by the JVM. Due to this the chance of failing Java programs due to out of memory is very low.
- Garbage collector is a Daemon Thread which always run in the background.

The ways to make an object eligible for Garbage Collector

- Even though programmer is not responsible to perform garbage collector. But it is highly recommended to make the objects eligible for Garbage collection if that object is not in use.
- If an Object doesn't have any reference variable associated with it then that Object is considered as unused object and is eligible for Garbage Collection.

There are 4 ways of making any Object Eligible for Garbage Collection.

1.. Nullifying the Reference Variable

- If any Object is not in use then to make it eligible for Garbage Collection, we can assign null to all the reference variables associated to that Object. This process is called Nullifying the reference variable.

```java
import java.util.*;
class GarbageCollectionDemo {
    public static void main(String args[]){
        ArrayList l1 = new ArrayList(); //Object 1
        ArrayList l2 = new ArrayList(); //Object 2
        //At this Point 0 Objects are eligible for GC
        ....
        ....
        ....
        ....
        ....
        ....
        l1 = null; //At this point 1 Object is Eligible for GC
        ....
        ....
        ....
        ....
        ....
        ....
        l2=null; //At this point 2 Objects are Eligible for GC
    }
}
```

2.. Reassigning the reference variable

- If any Object is not in use then to make it eligible for Garbage Collection, then we can re-assign its reference variables to some other Object then automatically old object will be eligible for Garbage Collection. This approach is called re-assigning the reference variable.

Main.java

```java
import java.util.*;
class GarbageCollectionDemo {
    public static void main(String args[]){
        ArrayList l1 = new ArrayList(); //Object 1
        ArrayList l2 = new ArrayList(); //Object 2
        //At this Point 0 Objects are eligible for GC
        ....
        ....
        ....
        ....
        ....
        ....
        l1 = new ArrayList();; //At this point 1 Object is Eligible for GC
        ....
        ....
        ....
        ....
        ....
        ....
        l2=new ArrayList();; //At this point 2 Objects are Eligible for GC
    }
}
```

3.. Objects Created inside a Method

The Objects which are created inside a method are by default eligible for Garbage Collection. Because, inside the method those objects are treated as a Local Variable and the scope of Local Variable Ends once method completed its execution.

```
import java.util.*;
class GarbageCollectionDemo {
    public static void main(String args[]){
        m1()
    }
    public static void m1()
    {
        Student s1 = new Student();
        Student s2 = new Student();
        //Scope of s1 and s2 is inside this method only Once method COmpletes,
        //Both s1 and s2 are eligible for Garbage Collection
    }
}
```

If method m1 is Student Type and returning object s1 which we are holding in Student s then in that case we are having only One Object eligible for Garbage Collection.

```
1  import java.util.*;
2  class GarbageCollectionDemo {
3      public static void main(String args[]){
4          Student s = m1()
5      }
6      public static Student m1()
7      {
8          Student s1 = new Student();
9          Student s2 = new Student();
10         return s1;
11         /*
12         Scope of s1 and s2 is inside this method only Once method COmpletes,
13         Both s1 and s2 are eligible for Garbage Collection but before that
14         We are returning s1 so Student s in main method is referencing to that Object.
15         So only 1 object is eligible for Garbage Collection.
16         */
17     }
18 }
```

Note: In above program if we don't hold the return object s1 to Student s then instead of 1, 2 Objects are eligible for Garbage Collection.

If we have static Object for that Scope of Object is not limited to Method.

```java
import java.util.*;
class GarbageCollectionDemo {
    static Student s;
    public static void main(String args[]){
       m1()
    }
    public static void m1()
    {
        s = new Student();
        Student s1 = new Student();
        /*
        Scope of s1 is inside this method only Once method COmpletes, because s1 is a local variable.
        So, only s1 is eligible for Garbage Collection and s is a static variable which is still
        pointing to Object. So only 1 object is eligible for Garbage Collection.
        */
    }
}
```

4.. Island of Isolation

In this scenario, consider a group of Objects which are interrelated with each other through internal references. These Objects are eligible for Garbage Collection once these group of Objects gets isolated from external world. i.e., None of the external reference is pointing towards any of these objects.

```java
import java.util.*;
class Test {
    Test i;
    public static void main(String args[]){
       Test t1 = new Test();
       Test t2 = new Test();
       Test t3 = new Test();
       //Till Here we have three Objects t1, t2 and t3 of Test Type and each of these objects holding a
       //instance variable i

       t1.i = t2;
       t2.i = t3;
       t3.i = t1;
       //All Objects Got Internally Connected

       t1 = null; //0 Objects eligible for Garbage Collection as they are internally connected with t2
           and t3
       t2 = null; //0 Objects eligible for Garbage Collection as they are internally connected with t3
       t3 = null; //All external connections are removed so all 3 are eligible for GC
    }
}
```

The methods for requesting JVM to run Garbage Collector

- Once we make Objects eligible for Garbage Collection. It is not compulsory that it will be destroyed immediately by Garbage Collector. Whenever JVM runs Garbage Collector then only Objects will be destroyed. But we can't expect when the JVM will run Garbage Collector as it varies from JVM to JVM.
- Instead of waiting for JVM to run Garbage Collector automatically, we can request JVM to run Garbage Collector programmatically. But there is no guarantee that weather JVM will accept our request to run Garbage Collector or not. But most of the times JVM will accept our request.

There are two ways by which we can request JVM to run Garbage Collector:

1.. By using System Class

- **System** class contains a static method **gc()** which can be called which will request JVM to run Garbage Collector.

  System.gc();

2.. By using Runtime Class

- Java application can communicate with JVM by using **Runtime** Class Object. **Runtime** is a singleton class present in java.lang package.
- We can create Runtime Object as: Runtime r = Runtime.getRuntime();
- Once we get Runtime Object r, we can call the following methods on that object:

totalMemory()	It returns the number of bytes of total memory present in a heap. i.e., Heap Size.
freeMemory()	It returns the number of bytes of free memory present in a heap.
gc()	This is an instance method in Runtime class For requesting JVM to run Garbage Collector.

Main.java

Run

Output

```
import java.util.*;
class RunTimeDemo {
    public static void main(String args[]){
        Runtime r = Runtime.getRuntime();
        System.out.println(r.totalMemory());
        System.out.println(r.freeMemory());
        for(int i = 1; i <= 10000; i++)
        {
            Date d = new Date();
            d = null;
        }
        System.out.println(r.freeMemory());
        r.gc();
        System.out.println(r.freeMemory());
    }
}
```

```
java -cp /tmp/oz6Lrq9ErA RunTimeDemo
16252928
15347272
15166360
15773784
```

Note: It is convenient to use System class gc() method when compared with Runtime class gc() method. But, internally System class gc() method calls Runtime.getRuntime().gc(). So it is recommended to use Runtime.getRuntime().gc() directly.

Finalization

- Once programmer makes an unused objects eligible for Garbage Collection and request JVM to run Garbage Collector through **System.gc()** method or **Runtime.getRuntime().gc()** method, the process of **finalization** takes place.
- Garbage Collector will call the **finalize()** method present in **Object** class of **java.lang** package. finalize() method is available for all the Objects as it is present in Object class and is responsible for performing various cleanup activities like reallocating the resources etc. before destroying an unused objects. We can override this finalize() method in our class to define our own cleanup activities.
- The complete prototype of finalize() method is:
  protected void finalized() throws Throwable;

Case 1

Just before destroying object, Garbage Collector calls finalize method on Object which is eligible for Garbage Collection. Then, the corresponding class finalize() method will be called.

```
1  class GarbageCollectionDemo
2  {
3      public static void main(String[] args)
4      {
5          String s = new String("Vikash");
6          s=null;
7          System.gc();
8          System.out.println("End of Main");
9      }
10     public void finalize()
11     {
12         System.out.println("Calling Finalize for Cleanup");
13     }
14 }
```

```
java -cp /tmp/X9fvp9yGbE GarbageCollectionDemo
End of Main
```

In this above program String object s is eligible for Garbage Collection. So, finalize() method of String class gets called which is having empty implementation so on the console we are getting only "End of Main".

But if we wants to call finalize() method of GarbageCollectionDemo class then we have to make GarbageCollectionDemo class Object eligible for Garbage Collection. See below example:

Main.java Run Output

```
1  class GarbageCollectionDemo
2  {
3      public static void main(String[] args)
4      {
5          GarbageCollectionDemo t = new GarbageCollectionDemo();
6          t=null;
7          System.gc();
8          System.out.println("End of Main");
9      }
10     public void finalize()
11     {
12         System.out.println("Calling Finalize for Cleanup");
13     }
14 }
```

```
java -cp /tmp/X9fvp9yGbE GarbageCollectionDemo
Calling Finalize for Cleanup
End of Main
```

Note: Garbage collector is a Daemon thread so order of Print statements on the console might change from run to run.

Case 2

Based on our requirement we can call finalize() method explicitly, but this explicit call will be executed as a normal method and no objects gets destroyed. Garbage collector will again call finalize() method and that call will destroy objects.

```
class GarbageCollectionDemo
{
    public static void main(String[] args)
    {
        GarbageCollectionDemo t = new GarbageCollectionDemo();
        t.finalize();
        t.finalize();
        t=null;
        System.gc();
        System.out.println("End of Main");
    }
    public void finalize()
    {
        System.out.println("Calling Finalize for Cleanup");
    }
}
```

```
java -cp /tmp/X9fvp9yGbE GarbageCollectionDemo
Calling Finalize for Cleanup
Calling Finalize for Cleanup
End of Main
Calling Finalize for Cleanup
```

Here we are calling finalize() method twice explicitly. But we are seeing finalized() methods gets executed thrice. Two times as a normal explicit method and one time gets executed by Garbage Collector before destroying objects.

Case 3

For any unused Object, Garbage Collector will call finalize() method only once. Second time, it will directly destroy Object without calling finalize() method.

Main.java Run Output

```
class GarbageCollectionDemo
{
    static GarbageCollectionDemo s;
    public static void main(String[] args) throws InterruptedException
    {
        GarbageCollectionDemo t = new GarbageCollectionDemo();
        System.out.println(t.hashCode());
        t=null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(s.hashCode());
        s=null;
        System.gc();
        Thread.sleep(10000);
        System.out.println("End of Main");
    }
    public void finalize()
    {
        System.out.println("Calling Finalize for Cleanup");
        s=this;
    }
}
```

```
java -cp /tmp/X9fvp9yGbE GarbageCollectionDemo
212628335
Calling Finalize for Cleanup
212628335
End of Main
```

In this program we are seeing We have made GarbageCollectionDemo class Object t as eligible for Garbage Collection and call Garbage Collector which calls finalize() method before Object destruction. In finalize method we are assigning new reference variable s to this object. Now after some point of code we have nullifying s as well and calls again garbage collector. This time Garbage collector won't call finalize() method and will directly destroy object. In Output console we are seeing print statement of finalize() method is called only once.

Case 4

JVM automatically calls Garbage Collector when we have out of memory issue. Let's demonstrate this with small code.

Here for smaller value of int i like 10, 100, 1000, out of memory problem will be not there. So put value as 1000000 (1 Million).

Main.java

Run

```
class GarbageCollectionDemo
{
    static int count=0;
    public static void main(String[] args) throws InterruptedException
    {
        for(int i=0; i<1000000; i++)
        {
            GarbageCollectionDemo t = new GarbageCollectionDemo();
            t=null;
        }
    }
    public void finalize()
    {
        System.out.println("Calling Finalize for Cleanup " + ++count);
    }
}
```

Output

```
java -cp /tmp/X9fvp9yGbE GarbageCollectionDemo
Calling Finalize for Cleanup 1Calling Finalize for Cleanup 2
Calling Finalize for Cleanup 3
Calling Finalize for Cleanup 4
Calling Finalize for Cleanup 5
Calling Finalize for Cleanup 6
Calling Finalize for Cleanup 7
Calling Finalize for Cleanup 8
Calling Finalize for Cleanup 9
Calling Finalize for Cleanup 10
Calling Finalize for Cleanup 11
Calling Finalize for Cleanup 12Calling Finalize for Cleanup 13
Calling Finalize for Cleanup 14
Calling Finalize for Cleanup 15
Calling Finalize for Cleanup 16
Calling Finalize for Cleanup 17
Calling Finalize for Cleanup 18
Calling Finalize for Cleanup 19
Calling Finalize for Cleanup 20
Calling Finalize for Cleanup 21
Calling Finalize for Cleanup 22
Calling Finalize for Cleanup 23
Calling Finalize for Cleanup 24
Calling Finalize for Cleanup 25
Calling Finalize for Cleanup 26
Calling Finalize for Cleanup 27
Calling Finalize for Cleanup 28
Calling Finalize for Cleanup 29
Calling Finalize for Cleanup 30
Calling Finalize for Cleanup 31
Calling Finalize for Cleanup 32
Calling Finalize for Cleanup 33Calling Finalize for Cleanup 34
Calling Finalize for Cleanup 35
Calling Finalize for Cleanup 36
Calling Finalize for Cleanup 37
```

Case 5

The Objects which are not using in our program and which are not eligible for garbage collector. Such type of useless objects are called **memory leaks**. In our program, if memory leaks present then program will be terminated by OutOfMemoryError. Hence, if an Object is not required, then it is highly recommended to make it eligible for Garbage Collection.

The following are various third-party tools which can identify memory leaks in a Java Program:

- HP OVO
- HP Jmeter
- JProbe
- Patrol
- IBM Tivoli