1. Introduction
2. The ways to make an object eligible for GC
3. The methods for requesting JVM to run GC
4. Finalization

* Introduction

→ In old languages like C++, programmer is responsible to create new object & to destroy useless object. usually programmer taking very much care while creating object and neglecting destroying of useless object. Bcz of his negligence, at certain point for creation of new object, sufficient memory may not be available bcz total memory filled with useless objects only. & total application will be down with the memory problem hence OutOfMemory ~~Pro~~ Error is very common problem in old languages like C++.

→ But in Java, programmer is responsible only for creation of object & programmer is not

responsible to destroy useless object. SUN people provided one assistant to destroy useless object, This assistant is always running in the background (Daemon Thread) & destroy useless object. Just bcz of this assistant, the chance of failing Java program with memory problem is very very low. This assistance is nothing but Garbage collector.

⇝ Hence the main objective of Garbage collector is to destroy useless objects.

⊛ The ways to make an object eligible for GC.

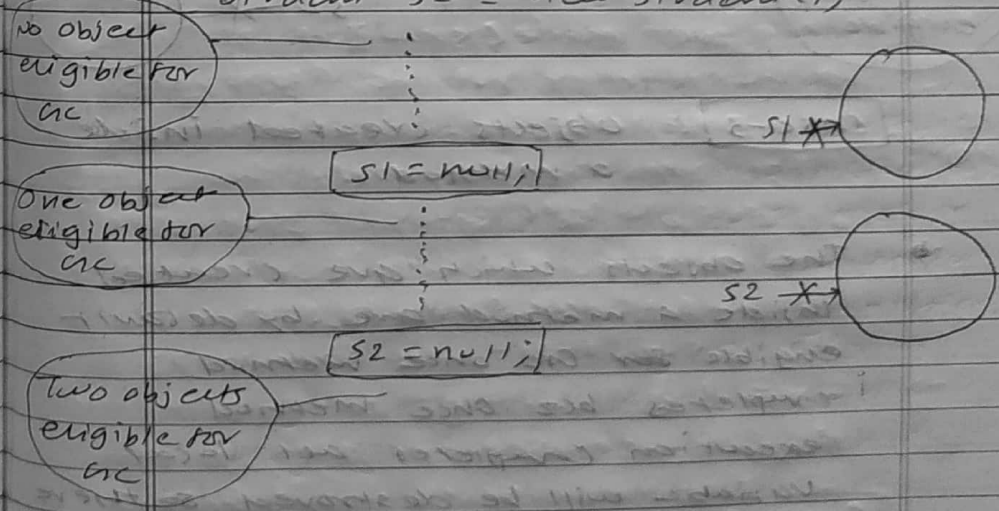
⇝ Even though programmer is not responsible to destroy useless objects, it is highly recommanded to make an object eligible for GC if it is not longer required.

⇝ An object is said to be eligible for GC if and only if it doesn't contain any reference variable.

⇝ The following are various ways to make an object eligible for GC:-

case-1: Nullifying the reference variable

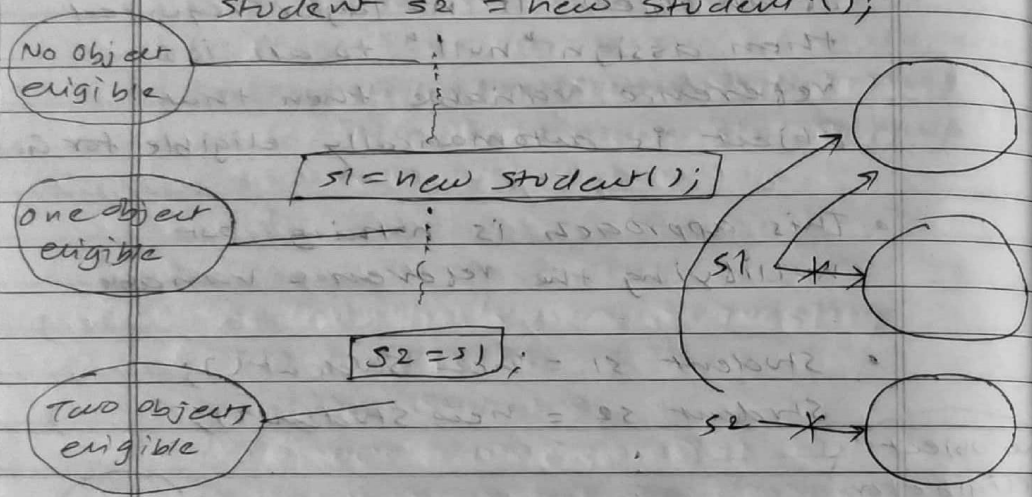
- If an object is no longer required then assign "null" to all its reference variable then that object is automatically eligible for GC.
- This approach is nothing but nullifying the reference variable.
- Student s1 = new Student();
  Student s2 = new Student();



case-2: Reassigning the reference variable

- If an object is no longer required then re-assign its reference variable to some other object then old object by default eligible for GC.

- Student s1 = new Student();
  Student s2 = new Student();

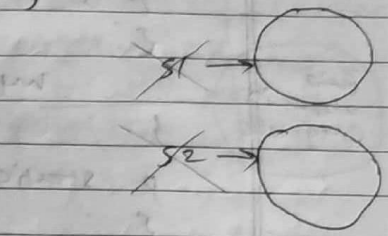


Case-3 : Objects created inside a method

- The objects which are created inside a method are by default eligible for GC once method completes bcz once method execution completes all local variable will be destroyed so there won't be any reference variable (local variable) pointing to objects.

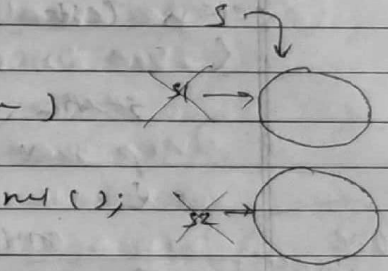
- class Test
  {
    p s v main(Str)
    {
      m1();
    }
    p s v m1()
    {
      Student s1 = new Student();
      Student s2 = new Student();
    }
  }

Two objects eligible for GC

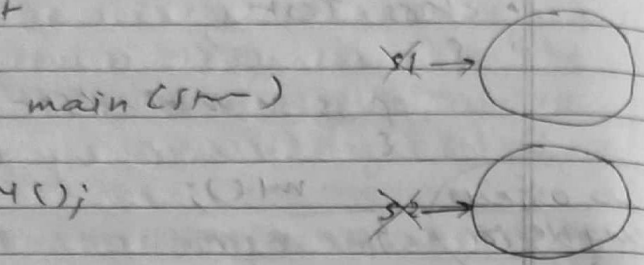


- class Test
  {
    p s v main(Str)
    {
      Student s = m1();
    }
    p static Student m1()
    {
      Student s1 = new Student();
      Student s2 = new Student();
      return s1;
    }
  }

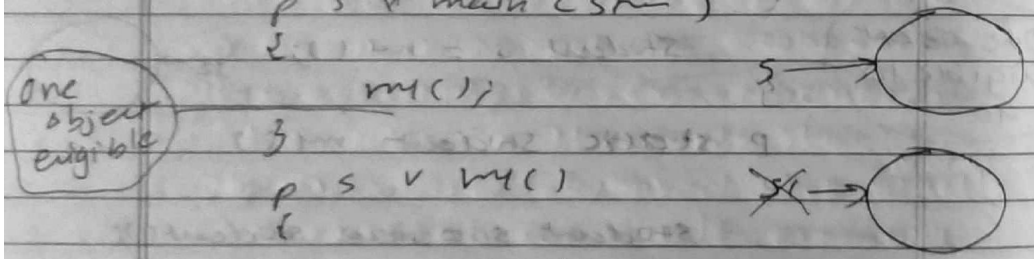
One object eligible for GC



- class Test
{
    p s v main(Str-)
    {
        m1();
    }
    p static Student m1()
    {
        Student s1 = new Student();
        Student s2 = new Student();
        return s1;
    }
}



- class Test
{
    static Student s;
    p s v main(Str-)
    {
        m1();
    }
    p s v m1()
    {
        s = new Student();
        Student s1 = new Student();
    }
}



Case-4 :- Island of Isolation

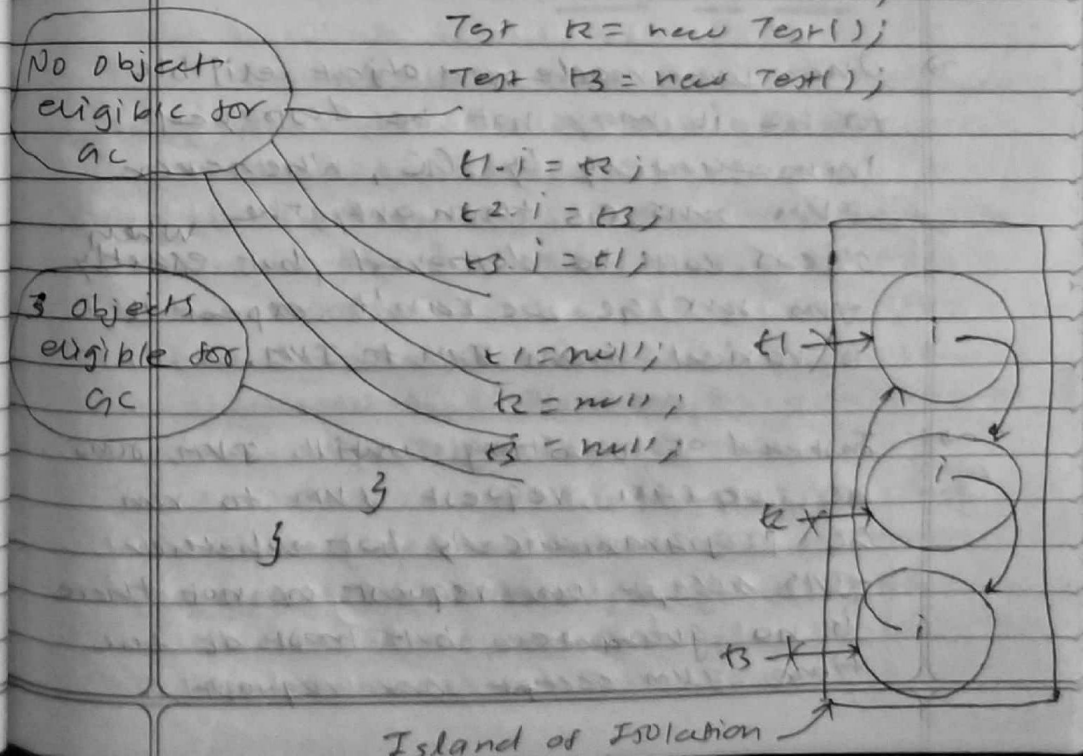
- Object is said to be eligible for GC if it has not any external references means though it has any no. of internal references it will be eligible for GC.

- class Test
{
    Test i;

    p s v main(Str-)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test();

        t1.i = t2;
        t2.i = t3;
        t3.i = t1;

        t1 = null;
        t2 = null;
        t3 = null;
    }
}



Note :- 1) If an object doesn't contain any ref variable then it is eligible for GC always.

2) Even though object having references sometimes it is eligible for GC (If all references are internal references e.g. Island of Isolation.

(*) The methods for requesting JVM to run GC

→ Once we made an object eligible for GC it may not be destroyed immediately by GC, whenever JVM runs GC then only the objects will be destroyed, but when exactly JVM runs GC we can't expect it is varied from JVM to JVM.

→ Instead of waiting until JVM runs GC, we can request JVM to run GC programmatically but whether JVM accept our request or not there is no guarantee, but most of the time JVM accept our request.

→ The following are two ways for requesting JVM to run GC

(1) By Using System class

- System class contains a static method "gc()" for this purpose
- `System.gc();`

(2) By Using Runtime class

- Java application can communicate with JVM by using Runtime object
- Runtime class present in "java.lang" package & it is a singleton class.
- we can create Runtime object by using "Runtime.getRuntime()" method.

```
Runtime r = Runtime.getRuntime();
```

- Once we got Runtime object we can call the following methods on that object

(i) totalMemory() :- It returns no. of bytes of total memory present in heap i.e heap size.

(ii) freeMemory() :- It returns no. of bytes of free memory present in the heap.

(iii) gc() :- For requesting JVM to run GC.

• Ex.

```
import java.util.Date;
class RuntimeDemo
{
    p s v main(Strn)
    {
        Runtime r = Runtime.
                    getRuntime();
        s.o.p(r.totalMemory());
        s.o.p(r.freeMemory());
        for(int i=0; i<10000; i++)
        {
            Date d = new Date();
            d = null;
        }
        s.o.p(r.freeMemory());
        r.gc();
        s.o.p(r.freeMemory());
    }
}
```

O/P :-

```
javac RuntimeDemo.java
java RuntimeDemo

5177344
4945200
4714464
5059352
```

Note :- • gc() method present in System class is a static method.
Whereas
• gc() method present in Runtime class is a instance method.

Q: Which of the following is valid way for requesting JVM to run GC

Ans :-
(a) System.gc(); ✓
(b) Runtime.gc(); ✗
(c) (new Runtime()).gc(); ✗
(d) Runtime.getRuntime().gc(); ✓

~ Note :- It is convinient to use System class gc() method when compared with Runtime class gc().

- with respect to performance it is highly recommanded to use Runtime class gc() method when compared with System class gc() method, bcz System class gc() method internally called Runtime class gc() method.

```
class System
{
    public static void gc()
    {
        Runtime.getRuntime.gc();
    }
}
```

Finalization

~ Just before destroying an object GC calls finalize() method to perform cleanup activities.

~ Once finalize() method completes automatically GC destroys that object.

~ finalize() method present in Object class with following declaration :

```
protected void finalize()
                throws Throwable
```

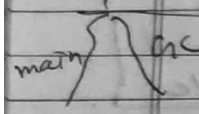
~ we can override finalize() in our class to override clean up activities.

~ **Case-1** :- Just before destroying an Object GC calls finalize() method which is eligible for GC, then the corresponding class finalize() method will be executed.

Eg If String Object eligible for GC then String class finalize() method will be executed but not Test class finalize() method.

Ex.

```
class Test
{
    p s v main(Str-)
    {
        String s = new String("durga");
        s=null;
        System.gc();
        s.o.p("End of main");
    }
    public void finalize()
    {
        s.o.p("finalize method of Test class called");
    }
}
```



- In the above example String object eligible for GC & hence String class finalize() method got executed which has empty implementation & hence the o/p is :-

```
End of main
```

- If we replace String object with Test object then Test class finalize() method will be executed. In this case the o/p is :-

(or)

```
End of main
finalize method of Test class called
```

```
finalize method of Test class called
End of main
```

⇝ **Case-2** :- Based on our requirement we can call finalize() method explicitly, then it will be executed just like a normal method call & object won't be destroyed.

Ex.

```
class Test
{
    p s v main(Str-)
    {
        Test t = new Test();
        t.finalize();
        t.finalize();
        t=null;
        System.gc();
        s.o.p("End of main");
    }
    public void finalize()
    {
        s.o.p("Finalize called");
    }
}
```

→ In the above program finalize() method got executed 3 times in that 2 times explicitly by the programmer & 1 time by the GC. In this case o/p is:-

```
finalize called
finalize called
End of main
finalize called.
```

→ **Note**:- If we are calling finalize() method explicitly then it will be executed just like a normal method & object won't be destroyed.

- If GC called finalize() method then object will be destroyed.

→ **Note**:- init(), service() & destroy() method are considered as life cycle methods of servlets.

- Just before destroying servlets object web container calls destroy() method to perform clean up activities

- But based on our requirement we can call destroy() method from init() & service() method then destroy() method will be called just like normal method & servlet object won't be destroyed.

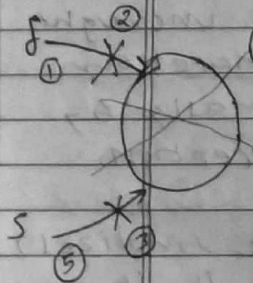
→ **Case-3**:- If we are calling finalize() method explicitly & if any exception raised while execution of finalize() method which is uncaught then JVM terminates our program abnormally by raising that exception.

→ But Garbage collector calls finalize() method & while executing it if any exception raised which is uncaught then JVM ignores that exception & rest of the program will be continued normally.

⇝ **Case 4** :- Even though object eligible for gc multiple times but gc calls finalize() method only once.

Ex:

```
class FinalizeDemo
{
    static FinalizeDemo s;

    p s v main(Str ) throws IE
    {
        FinalizeDemo f = new
            FinalizeDemo(); —(1)

        s.o.p(f.hashcode());

        f = null; —(2)

        System.gc(); —(6)

        Thread.sleep(5000);
        s.o.p(s.hashcode());
        s=null; —(3)

        System.gc(); —(4)
        Thread.sleep(10000);
        s.o.p("End of main");
    }
```




```
    public void finalize()
    {
        s.o.p("finalize method");
        s=this; —(5)
    }
}
```

O/P :-
```
25724761
finalize method
25724761
End of main
```

⇝ finalize() called only once.

⇝ In the above program, even though object eligible for gc 2 times but gc calls finalize() method only once.

⇝ **Case-5** :- We can't expect exact behaviour of garbage collector, it is varied from JVM to JVM, hence for the following questions we can't answers exactly.

1) When exactly JVM runs gc?
2) In which order gc identifies eligible objects?
3) ~~the~~ In which order gc destroys eligible objects?

4) whether GC destroys all eligible objects or not ?

5) What is algorithm followed by GC ?

!

etc.

Note :- We can give guarantee upto certain level like,

- Whenever program runs with low memory then JVM runs GC, but we can't expect exactly at what time.
- Most of the GC follows standard algorithm : Mark & sweep algorithm It doesn't mean every GC follows the same algorithm.

↝ **Ex.**

```
class Test
{
    static int count=0;
    p s v main(str[])
    {
        for(int i=0; i<10; i++)
        {
            Test t = new Test();
            t=null;
        }
    }
    public void finalize()
    {
        S.o.p("finalize method called :"+count);
    }
}
```

↝ In the above program, if we keep increasing the no. in box means more objects will be created & more objects will be eligible for GC, so when JVM runs with low memory it will start to destroy objects until memory became enough, so for those objects finalize method will be called.

↝ **Case-6 :-** Memory Leaks :- The objects which are not using in our program which are not eligible for GC, such type of useless objects are called "Memory Leaks".

⇝ In our program, If memory leaks are present then the program will be terminated by raising "OutOfMemoryError".

⇝ If an object no longer require, it is highly recommanded to make that object eligible for GC.

⇝ The following are various third party Memory Management Tools to identify "Memory Leaks". like

- HP OVO
- HP J Meter
- JProbe
- Patrol
- IBM Tivoli

End of Garbage Collection

Start of Enum