Garbage Collection

Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

Unreachable Object: An object is said to be unreachable if it doesn't contain any reference to it. Also note that objects which are part of island of isolation are also unreachable.

Ways to make an object eligible for GC

1. **Island of Isolation:**
   a. Object 1 references Object 2 and Object 2 references Object 1. Neither Object 1 nor Object 2 is referenced by any other object. That's an island of isolation.

   b. Basically, an island of isolation is a group of objects that reference each other but they are not referenced by any active object in the application. Strictly speaking, even a single unreferenced object is an island of isolation too.

2. **Object created inside a method:** When a method is called it goes inside the stack frame. When the method is popped from the stack, all its members dies and if some objects were created inside it then these objects becomes unreachable or anonymous after method execution and thus becomes eligible for garbage collection **Note :** If a method returns the object created inside it and we store this object reference by using a reference-type variable than it is no longer eligible for garbage collection.
3. **Reassigning the reference variable:**

```
Test t1 = new Test("t1");

    Test t2 = new Test("t2");


    //t1 now referred to t2

    t1 = t2;
```

```
output : t1 successfully garbage collected
```

4. **Nullifying all References:** When all the reference variables of an object are changed to NULL, it becomes unreachable and thus becomes eligible for garbage collection

5. **Anonymous object :** The reference id of an anonymous object is not stored anywhere.
   new Test("ok"); ///**anonymous**

Ways for requesting JVM to run Garbage Collector

1. Once we made object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed.

2. We can also request JVM to run Garbage Collector. There are two ways to do it :
   a. **Using *System.gc()***
   b. **Using *Runtime.getRuntime().gc()***

   Note :
   i. There is no guarantee that any one of above two methods will definitely run Garbage Collector.

   ii. The call ***System.gc()*** is effectively equivalent to the call ***Runtime.getRuntime().gc()***

3. You can force object finalization to occur by calling **System.runFinalization() . This** method calls the finalize methods on all objects that are waiting to be garbage collected.

gc() suggests that the garbage collector should dispose of unused objects
runFinalization() suggests that it should run the finalize() methods of objects that are already queued for disposal.

Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.

- *finalize()* method is present in Object class with following prototype.

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

Note:

1. The *finalize()* method called by Garbage Collector not JVM. Although Garbage Collector is one of the module of JVM.

2. Object class *finalize()* method has empty implementation, thus it is recommended to override *finalize()* method to dispose of system resources or to perform other cleanup.

3. The *finalize()* method is never invoked more than once for any given object.

4. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored and finalization of that object terminates.

Ref : https://www.geeksforgeeks.org/garbage-collection-java/

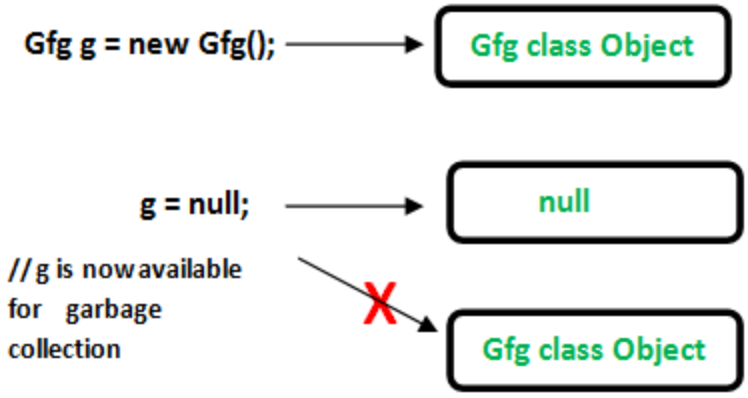
Types of References in Java

https://www.geeksforgeeks.org/types-references-java/

https://javarevisited.blogspot.com/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html

Reason: we want to keep a reference to an object, but we don't want to prevent GC from freeing it if there is no other reference.

1. **Strong References:** This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.



2. **Weak References:** *Any way collected by GC.*

```
Counter counter = new Counter(); // strong reference - line 1

java.lang.ref.WeakReference<Counter> weakCounter = new java.lang.ref.WeakReference
<Counter>(counter); //weak reference

 counter = null; // now Counter object is eligible for garbage collection

counter  = weakCounter .get();//get the reference back
```

Now as soon as you make strong reference `counter = null`, counter object created on line 1 becomes eligible for garbage collection; because it doesn't have any more Strong reference and Weak reference by reference variable `weakCounter` can not prevent `Counter` object from being garbage collected.

One convenient example of **`WeakReference`** is **`WeakHashMap`**, which is another implementation of Map interface like HashMap or TreeMap but with one unique feature.

`WeakHashMap` wraps keys as `WeakReference` which means once strong reference to actual object removed, `WeakReference` present internally on `WeakHashMap` doesn't prevent them from being Garbage collected.

3. **Soft References:** In Soft reference, even if the object is free for garbage collection then also its not garbage collected, until JVM is in need of memory badly. The objects gets cleared from the memory when JVM runs **out of memory**.

```
Counter prime = new Counter();  // prime holds a strong reference - line 2

SoftReference<Counter> soft = new SoftReference<Counter>(prime); //soft
reference variable has SoftReference to Counter Object created at line 2

prime = null;

prime = soft.get();//get the reference back
```

SoftReference are more suitable for **caches** and WeakReference are more suitable for storing **meta data**.

4. **Phantom References:** The objects which are being referenced by phantom references are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called 'reference queue' . They are put in a reference queue after calling finalize() method on them.To create such references java.lang.ref.PhantomReference class is used.

```
DigitalCounter digit = new DigitalCounter(); // digit reference
variable has strong reference - line 3

ReferenceQueue< DigitalCounter  > refQueue = new ReferenceQueue<
DigitalCounter >();

PhantomReference<DigitalCounter>
phantom = new PhantomReference<DigitalCounter>(digit); // phantom
reference to object created at line 3

digit = null;

digit = phantom .get();//get the reference back

digit is still null
```

Phantom doesn't have a link to the actual object.

Summary:

- **Soft references try to keep the reference.**
- **Weak references don't try to keep the reference.**
- **Phantom references don't free the reference until cleared.**