Performing Garbage Collection in Java

In programming, garbage collection is a form of automatic memory management, whereby the memory used by an object is freed or released by destroying the object when it becomes redundant in the program. Garbage collection is considered one of the best practices to avoid any untoward results when a program is executed.

In object-oriented programming languages (like C++ and Java), there could be unused objects that are still a part of the program. These will create undesirable results like an out of memory error, for instance, during runtime, if left unattended. The memory occupied by such objects is known as 'garbage' and the mechanism to eliminate this from the program is called 'garbage collection'.

In languages like C++ (which is one of the older OOP languages), the programmer is responsible for creating and destroying the objects. If the purpose for which the object was created is fulfilled but the programmer neglects to delete the now-useless objects, the entire memory will be filled with old and unwanted objects. This will prevent the creation of new objects, thereby bringing down the entire application.

Garbage Collection (GC) in Java

Unlike C++, Java provides the programmer with an assistant to delete useless objects, which is called the garbage collector. In Java, there is no *OutOfMemoryError* thanks to the garbage collector, which destroys unused objects. Java is also called a robust programming language because of such mechanisms. Because of this background mechanism, the stress on the programmer is significantly less.

The various ways to make an object eligible for GC

In Java, even though the programmer is not responsible for destroying the objects, it is still highly recommended that you make an object eligible for garbage collection if it is no longer required. Thereby, the garbage collector will take care of this object as a part of the routine clean-up. For example, if you put a used tea-cup in a garbage bin, the local cleaning authority will naturally take care of it. In Java, when the object does not have any reference pointing to it, then it becomes eligible for garbage collection. The subsequent sections of this article list various ways to make an object eligible for GC.

Illustration 1- Nullifying the reference variable

Let us look at the following code snippet.

```
Student s1 = new Student();
Student s2 = new Student();
s1=null; // One object eligible for garbage collection
```

In the above code, we can see two objects of Student (*s1* and *s2*). When we create a second object of the same class, it is obvious that you don't need the first object or the previous object; in which case, it is highly recommended that the previous object is made eligible for garbage collection. So we use *s1=null* to make the previous object eligible for garbage collection.

Illustration 2- Re-assigning the reference variable

Let us take a look at another snippet shown below.

```
Student s1 = new Student(); // first object
Student s2 = new Student(); //second object
/*Creating third object*/
s1 = new Student(); // first object becomes eligible for
garbage collection
s2=s1; // second object becomes eligible for garbage
collection
```

In the above code, we have created three objects. As our latest object is the third object, we don't need the first and the second objects. So, we will make them eligible for garbage collection. The third object is now pointed by 's1' so there is no longer any reference pointing to the first object, making it eligible for garbage collection. Similarly, there is no need for the second object as well, so we assign the 's1' reference variable of the third object to 's2' which was earlier pointing to the second object. Now both 's1' and 's2' point to the third object, so the first and second objects become eligible for garbage collection.

Illustration 3-Objects created inside a method

Now let us move on to an illustration of objects inside a method, using three examples.

Example 1

```
Class Student{ }
Class Test{
public static void main(String[] args){
m1();
}
public static void m1(){
Student s1 = new Student();
Student s2 = new Student();
 }
}
```

In the above code, we see that two Student objects are created inside a method; so the moment method *m1()* finishes its execution, the local variables *s1* and *s2* are no longer available. Hence, there is no reference variable at all pointing to either of the two Student objects, in which case, the above two objects become eligible for garbage collection.

Example 2

```
Class Student { }

Class Test {

     public static void main(String[] args){

       Student s =m1();
     }

     public static Student m1(){
       Student s1 = new Student();
       Student s2 = new Student();
       return s1;
     }
}
```

In the above code, we see that method *m1()* returns the *s1* object; so even though *s1* is a local variable, it exists even after the *m1()* method finishes its execution. But *s2* is not returned by the *m1()* method; so once the *m1()* method finishes its execution, *s2* does not exist any more. Hence, in the above code only one object becomes eligible for garbage collection. Let's make a small change in the above code in the *main()* function, as follows:

```
public static void main(String[] args){
m1();
}
```

We know that the *m1()* method return type is Student, but we are not holding the return type while calling the *m1()* method in the *main()* function, as there is no hard and fast rule to compulsorily hold the return type of the method. In such a case, when we don't hold the return type *m1()* method in the *main()* function, then both objects referred by *s1* and *s2* become eligible for garbage collection. In general, the object created inside a method is by default eligible for garbage collection, except in the above exceptional case.

Example 3

```
Class Test {

     static Student s;

     public static void main(String[] args){
       m1();
     }

     public static void m1(){

       s = new Student();
       Student s1 = new Student();
     }
}
```

In the above code, 's' is a static variable, so although the two Student objects are created inside a method, only one object, which is referred by 's1', is eligible for garbage collection. The object that is referred by 's' is not eligible for garbage collection, because 's' is a static variable and not a local one.

Illustration 4-Island of isolation

In the real world, an island is a piece of land surrounded by water. In the case of Java, an island of isolation refers to an

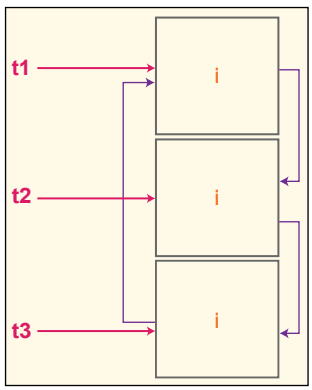

Figure 1: When objects have reference

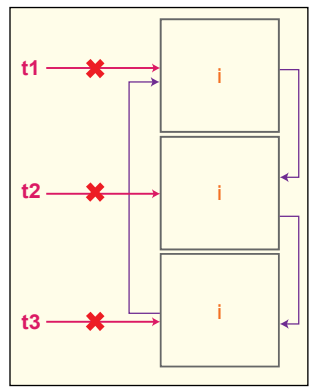

Figure 2: When objects do not have reference

object that has no reference variable pointing to it from outside, but all the objects pointing to each other internally. Let us consider the following code:

```
Class Test {

     Test i;
      public static void main (String[] args){

/*Creating three objects*/
     Test t1 = new Test();
     Test t2= new Test();
     Test t3 = new Test();

/*Assigning the reference of the all the three objects to
each object's instance variable.*/
     t1.i = t2;
     t2.i = t3;
     t3.i = t1;

/*Assigning null to the reference variable*/
     t1 = null;// no object eligible for garbage
collection(GC)
     t2= null;//no object eligible for GC
     t3 = null;// all three objects are now eligible for GC

     }
}
```

In Figure 1 there are three objects, referred to as *t1*, *t2* and *t3*, respectively, and each object has the 'i' instance variable associated with it. These instance variables point to each other's objects and, hence, the bonding between these objects becomes more powerful.

Figure 2 is a bit different. Here, all the three objects are still associated by the internal bonding of the instance variable but, unfortunately, the external bonding is lost by *t1*, *t2* and *t3;* so these three objects become isolated and consequently eligible for garbage collection.

Requesting JVM to run the garbage collector

The method to request JVM to run garbage collection is called the *gc()* method. It is used to give a call to the garbage collector explicitly. The call to the garbage collector by the *gc()* method doesn't mean that the garbage collector is ready to perform garbage collection. The *gc()* method is present inside *java.lang.System* class and *java.lang.Runtime* class.

```
Class Test {
Public static void main(String[] args){
Test t = new Test();
T=null;
System.gc();
}
/*Overriding the finalize() method. This method is explained
in the below section*/
public void finalize(){
System.out.println("Garbage collection is performed");
}
}
```

Finalisation

Finalisation takes place when the *finalize()* method is invoked. The *finalize()* method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. The method is called by the garbage collector thread each time, before the object is collected as garbage. This is the last option for any object to perform a clean-up task. The *finalize* method is declared in the *java.lang.Object* class. Inside the *finalize* method, you are supposed to write the code which has to be performed before an object is destroyed. It is a best practice to call the *super.finalize()* method once your class frees the resources it was holding. If you don't call *super.finalize()* then any resources held by the super class may never be free. The method is helpful when an object needs to perform some special task before it gets caught by the garbage collector, such as closing the open database connection, open socket or any files. However, if the database connections or files are programmed to be released within a very short period, you will rarely need *finalize*.

```
protected void finalize() {
//-------method code-------
super.finalize();
 }
```

By: Vikas Kumar Gautam

The author is a mentor at Emertxe Information Technology (P) Ltd. His main areas of expertise include application development using Java/J2EE and Android for both Web and mobile devices. A Sun Certified Java Professional (SCJP), his interests include acquiring greater expertise in the application space by learning from the latest happenings in the industry. He can be reached at *vikash_kumar@emertxe.com*