Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize() method on the object.

The finalize( ) method has this general form:

protected void finalize( )

```
{
          // finalization code here   }
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. It is important to understand that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means

that you cannot know when—or even if—finalize( ) will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize( ) for normal program operation.

```
class GrabageCollection
   {
     public void finalize( ) throws Throwable
        {     System.out.println("Garbage Collector is working");    }

    public static void main(String ar[])
        {
                 GrabageCollection obj1=new GrabageCollection();
                 GrabageCollection obj2=new GrabageCollection();
                         obj1=null;
                       obj2=null;
             System.gc();
             System.runFinalization();
        }
}
                 O/P: Garbage Collector is working
                        Garbage Collector is working
```

The java.lang.Runtime.runFinalization() method runs the finalization methods of any objects pending finalization. Calling this method suggests that the Java virtual machine expend effort toward running the finalize methods of objects that have been found to be discarded but whose finalize methods have not yet been run. When control returns from the method call, the virtual machine has made a best effort to complete all outstanding finalizations.

The virtual machine performs the finalization process automatically as needed, in a separate thread, if the runFinalization method is not invoked explicitly. The method System.runFinalization() is the conventional and convenient means of invoking this method.

class GarbageCollectionDemo

```
{
        public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        long mem1, mem2;
        Integer someints[] = new Integer[1000];
        System.out.println("Total memory is: " +
        r.totalMemory());
        mem1 = r.freeMemory();
        System.out.println("Initial free memory: " + mem1);
        r.gc();
        mem1 = r.freeMemory();
        System.out.println("Free memory after garbage collection: "
        + mem1);
        for(int i=0; i<1000; i++)
        someints[i] = new Integer(i); // allocate integers
        mem2 = r.freeMemory();
        System.out.println("Free memory after allocation: "
        + mem2);
        System.out.println("Memory used by allocation: "
        + (mem1-mem2));
        // discard Integers
        for(int i=0; i<1000; i++) someints[i] = null;
        r.gc(); // request garbage collection
        mem2 = r.freeMemory();
        System.out.println("Free memory after collecting" +
        " discarded Integers: " + mem2);
        }
        }
```

```
O/P:        Total memory is: 264241152
Initial free memory: 262856976
Free memory after garbage collection: 7734824
Free memory after allocation: 7692864
Memory used by allocation: 41960
Free memory after collecting discarded Integers: 7735128
```