Garbage Collection

8) State the behavior that is guaranteed by the garbage collection system and write code that explicitly makes objects eligible for collection.

- Automated memory management is one of Java's most valuable services. Many languages require programmers to specify the details of acquiring storage from the operating system and returning it when no longer needed. But, Java handles this for programmer. Since Java does automatic garbage collection, manual memory reclamation is not needed. The runtime system tracks each object you create, notices when the last reference to it has vanished, and frees the object for you. Of course, some objects utilize a resource other than memory, such as a file or a handle to another object that uses system resources. In this case it is important that the resource be reclaimed and recycled when it is no longer needed.
- It is gauranteed that only objects with no references will be garbage collected.
- You can turn off garbage collection in your application by,

```
java −noasyncgc .....
```

Note: If you do this, it is almost gauranteed to fail with memory exhaustion at some point. It can be used only to experiment the performance difference it makes.

- Alternatively, you can call the method

```
System.gc();
```

to run garbage collector at any point you choose. (doesn't mean you will get it)

- Java uses a "mark sweep garbage collection algorithm, which traverses all the object references, marking any objects that are referred to and then garbage collecting any objects that are unmarked.
- Java allows you to add a finalize() method to any class. The finalize() method will be called before the garbage collector sweeps away the object. In practice, do not rely on the finalize method for recycling any resources that are in short supply − you simply cannot know when this method will be called. Java1.1 will guarantee that finalizers are run, if the method '*System.runFinalizersOnExit()*' is called before exit.
- If a finalize() method exist for an object it is gauranteed to be called once (only once) for the object before the object is garbage collected.
- The gc will usually call an object's finalize() method just before the object is gc−ed
- Any class that includes a finalize method should invoke its superclass' finalize() method
- Assuming that there is no compiler optimization, the earliest point that a local object becomes available for garbage collection will be when the runtime environment knows that there are no more references to the object in question. Typically, for local objects, this is when the method ends, but there are circumstances when the developer might wish to give the runtime environment a hint that the object is no longer needed. for eg,

```
public void method() {
BigObject bo = new BigObject(2000);
//do something
//let the runtime environment know that "bo" is
no longer needed
```

```
bo = null;
}
//do some processing
```

The runtime environment could collect the "BigObject" anytime after "bo" is set to null. Setting the reference to null doesn't guarantee that the object will be garbage collected quickly, but it does help.

- When the runtime environment exits, there may be many objects that have not been collected, This is because garbage collection runs as a separate, low priority thread and there are objects that it may never collect.

In the exam point of view :

- You must be able to identify when an object is available for gc − you have either set it to null or you have "redirected" the variable that was originally referring to it, so that it now refers to a different object.
- if you have a reference to an object say, A and then you pass A as an argument to some constructor − new obj(A); − then even if you null your reference − A=null; − you can't say that A is available for gc. So just follow the references and when they drop to zero you know its eligible/available for gc, not that it will happen.

eg,

```
1. obj = new Jo();
2. obj.doSomething();
3. obj = new Jo(); //Same as obj=null;
4. obj.doSomething();
```

The object referred by obj in line 1 becomes eligible for gc at line 4 (anytime after line 3 has been executed).

- References are not garbage collected and objects are. So if you have 2 references to the same object.

```
1. Object a = new Object();
2. Object b=a;
```

you now have 2 references to one object. The object has no name (it has a unique internal identity). Only the references have a name.
So it is not correct to use the phrase. "a is available for gc". you should only say "the object referred to by "a" is available for gc"
And this can happen when there are no references to an object.
If the only reference to an object are instance (member) variables then when these variables are set to null, or set to other objects, then the objects those variables originally referenced will be available for gc.

```
Object a = new Object();
Object a=null; //Now the object created in 1st line is available for gc
Object a=new Object();
a = new Object(); //same.
// Now original object created in line 1 is available for gc and a new
object is now out there referenced by "a".
```

But, if you say,

```
1. Object a = new Object();
2. Object b=a;
3. a=null;
```

The object referred to by "a" is still reachable, because "b" has a reference to it. When you say "Object b=a; you are making a copy of the reference in "a" and assigning that new copy to b. So now you have obe object and two references to it.
In line 3 you redirect "a" to be null (so it is no longer pointing to any object), but since "b" is still holding its own reference to that same original object, the object is still now available for gc.

another eg,

```
Aclass a = new Aclass(); // Object 1
Aclass b= new Aclass(); // Object 2
Aclass c = new Aclass(); // Object 3
a=b; // now we have no valid object reference to object "a" and it will be
// garbage collected sometime after this statement. But when?.....
a=c;
c=null; // no garbage collection will be eligible since
// "a" still refers to Object 3
a=null; // now object "c" is eligible for gc since it always had a valid reference.
// Should "b" go out of scope; then we would possibly have eligibility for gc.
// there might still be other references to object "b" preventing the collection.
```

eg by Kathy,

```
public class Test extends Applet {

   int a=3;

   String b="jo";  // call this object1

   String c=b;  // two references to object1

   Object d=new Object();  // object2

   public void init() {

     methodA(b);  // pass a copy of a reference to Object1

     methodB(d);  // pass a copy of a reference to Object2

     a=5;

   }

   void methodA(String b) {  // a new local var b, with a ref to Object 1

     b=c;  // local b directed to Object1 (copy of c ref)

     c=null;  // c no longer ref object1

   }  // local b goes out of scope but instance var b still refers object1

   void methodB(Object o) {  // a new local var o with a ref to object2
         d=new Object();  // HERE IT IS! d no longer refer to Object2

   }  // local o goes out of scope so no more ref to object2

}
```

Remember: You can suggest Garbage Collection, but no gaurantee that it will ever happen.