Object's eligibility factors for Garbage Collection in Java

Author: Manivannan. P <manivannan.p (at) gmail.com>

Performance of any program depends on memory usage and smart coding. The Java programmer should always attempt to minimize the amount of heap usage by releasing the unused object spaces. All the objects used in Java are stored in heap memory. Their addresses are stored in corresponding references present in the stack.

Garbage collector (a low priority and daemon thread) frees up unused object spaces periodically. Since, Garbage collector is a low priority thread; it runs rarely and makes the GC (Garbage Collection) process become slow. System.gc() and Runtime.getRuntime().gc() are the methods can be assisted explicitly to force the GC. However, these methods may not succeed all the times.

As per the common terms, an Object becomes eligible for GC, when it holds no references (i.e., reference count = 0). Also, all the eligible objects are not immediately flushed. But it is always good practice to design code such that it improves the degree of Object's GC eligibility.

This article covers some common cases to improve the GC process.

Case 1: Delaying the Object creation and Making null assignment

Delaying object creation:

When a program needs to create bulk amount of objects, it can delay the object creation statements until they are used.

Ex:

```
class A {...}
A a1;
A a2;
A a3;

// some statements
a1 = new A();
// some statements
a2 = new A();
// some statements
a3 = new A();
```

This program creates 3 references a1,a2,a3 and does not initialize them at first. Later it initializes the references in the same order as they are put in use. This delay in object creation can help to keep track a moderate use of heap memory throughout the program's lifecycle.

Making null assignment:

The objects finished their service in the program can be null assigned.

Ex:

```
class A {}
A a1 = new A();  // 1    {a1} --> Object (A)
// statements that use a1
a1 = null ;          // 10     {a1} --> null
                     //     &  { (none)}  --> Object (A)
// statements that don't use a2
```

Now the object referred by a1 may become eligible for GC at line 10. The null assignment does not make an object become eligible for GC immediately. The purpose of null assignment is to reduce the reference count of an object. Also it depends on other references that the object hold.

Case 2 depicts this scenario clearly.

Case 2: Detaching all the unused object reference

```
class  A {}
A a1 = new A();    //1
A a2 = a1 ;      //2
   a1 = null;
// a1,a2 are not used in rest!
//.....
// program exits here //10
```

At which line, the object created at line 1 becomes eligible for GC?

Ans: When the program exits (line 10)

In most of the cases, we mistake that the object becomes eligible for GC at line 3. But the object is still held by the reference 'a2' at line 3. Hence, it becomes eligible for GC, only when the program exits.

The following notational explanation may clarify this case.

```
Line 1:  A a1 = new A();        => {a1} --> Object (A)
Line 2:  A a2 = a1;      => {a2,a1} --> Object (A)
Line 3: a1 = null;                  => {a1} --> null
                    But still, {a2} --> Object (A)
```

The object (created at line 1) is still held by the reference a2 (at line 3).
To make the object become eligible for GC, make all the references null (i.e., replace the line 3 by).

like,
a1 = a2 = null; {a1,a2} --> null
and hence { (none) } --> Object (A)

Now all the references to the objects are detached and hence the object become eligible for GC.

Case 3: Restricting method arguments

```
class A {}
A a1 = new A();        //1
doSomething(a1); //2
a1 = null;
```

At which line, the object created at line1 becomes eligible for GC?
Ans: It depends on the implementation of the method doSomething().

The reason is body of doSomething() method may assign some more new references to the object referred by a1. So it may increment the reference count of the object. Hence, the reference count of the object depends on internal design of the method doSomething().

It is often better practice to restrict the arguments to methods unless they are necessary.

Case 4: Override finalize method()

All the classes inherit the method 'protected void finalize()' from java.lang.Object class implicitly. The finalize() method is invoked before the object is garbage collected. Programmer can place some codes in this finalize() method to release other resources reserved by the object.
Ex:

```
protected void  finalize()
{
s.close();  // close the socket
in.close(); out.close();  // close IO streams
}
```

But, care should be taken when designing this finalize() method. Sometime it may lead to assign some new references to this object, which may cancel the GC process.
Ex:

```
A a2;  // class variable a2;
protected void finalize()
{
a2 = this; //a2  as a new reference to this object
}
```

In the above example the finalize() method assigns a new reference a2 to ‘this’ object that cancels the current GC process. Also the finalize() method is invoked once per GC.

Conclusion

- Concerning the 4 cases discussed above, the following points derived:
- Delay in object creation will result in moderate use of heap.
- ‘null’ object assignment will decrement the object’s reference count and improve the degree of object’s eligibility for GC.
- Minimal use of method arguments will reduce the number object references.
- Careful design of finalize() method may help in optimal resource dispersion.