How to create/avoid memory leak in Java and .NET?

Venkat Subramaniam
venkats@durasoftcorp.com
http://www.durasoftcorp.com

Abstract

Java and .NET provide run time environment for managed code, and Automatic Garbage Collection (AGC) is one of the features that every programmer is eager to utilize and benefit from. One of the myths with managed code is that you don't have to worry about memory leaks. However, the truth is that even though these managed code run time environments provide AGC, developers are still required to worry about a few details. In this article, we will first discuss the AGC process of Java and what it takes to clean-up the resources. Then we address how easy it is to create memory leaks while programming in Java and .Net. Finally, we will discuss the AGC process in .NET, and how the resource clean-up is addressed in .NET.

Automatic Garbage Collection

Generally, C++ developers would agree that garbage collection is a pain in the neck. Objects created on the stack are automatically garbage collected; however, objects created on the heap are not. You always need to remember to write a call to “delete” the object from the heap. Even for an object that is on the stack, if it aggregates objects on the heap, the proper cleanup of the aggregated objects happen only if you write a *destructor* for the class. In C++, if you loose a pointer on an object and if there are no other pointers on that object, you have caused a memory leak. The memory used by the object will not be reclaimed by the system until the program is terminated. This may result, eventually in out of memory exception if your program continues to create objects and cause memory leaks.

Like we mention before, Java and .NET provides AGC. You do not have to “delete” or “free” an object. Authors who are in favor of AGC tend to convey that you, as a developer, do not have to do any thing for the AGC to work. While it takes away substantial amount of tedium out of the code you write, unfortunately, it is not 100% worry free, especially when it comes to memory leaks and resource garbage collection.

Lazy Garbage Collection in Java

Logically, Java maintains reference counts on objects. When a reference goes out of scope, the reference count on the referenced object goes down. If an object’s reference count goes to zero, then the object is said to be *unreachable*[1]. This definition needs to be refined a bit. For instance, let’s say that we have a reference on the stack (a local variable within a method) to an object of class A, and this object has a reference to another object of class B. The reference count on both the objects is now 1. If the reference on the stack goes out of scope, then the reference count on the object of A goes down to 0 and is considered unreachable. The reference count on object of class B, however, is still at 1. But, since B is now unreachable from any reference on the stack or any static references, and is reachable only through another unreachable object, the object of B is also considered as unreachable.

When an object becomes unreachable, it is simply ready for garbage collection. I call these objects as *zombie* objects. These objects take up memory; however, they will not be accessed any more. Java does not provide instant garbage collection, i.e., memory or resources used by an object is not claimed as soon as the object is unreachable. On the other hand, a lazy garbage collection thread that runs in the background will periodically check to see if the garbage collection process needs to be activated. One may ask, what causes the garbage collection process to be activated? Generally speaking, the Java’s garbage collector is activated if it finds that the memory utilization of the program has reached a certain threshold level. Until that point, even if there are zombie objects, the garbage collector does not care to garbage collect. It is not unusual for these zombie objects to remain in memory for a long time. This process is some what like the garbage collection in some of our neighborhoods! You have no idea when it would get picked up.

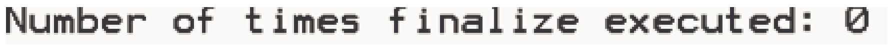
What about my resources?

The AGC process in Java (and in .NET) is focused on releasing memory that is no longer being referenced, i.e., the objects that are unreachable. In the case of C++, a destructor is used for two purposes: (1) to release the heap memory that is no longer needed (2) to manage other resources like open database connections, sockets, etc. It may also involve some book keeping operations. For example, if an account in a bank were deleted (closed), you may want to notify a manager object to take some actions at this time.

The AGC process attempts to take care of the memory cleanup issue. But what about the non-memory related resource management and book keeping. In order to address this, Java introduced the finalize method. Each class may provide a public void method called finalize(). In the finalize method, you could take care of the book keeping operations. Unfortunately, if there is enough memory available for a program to run, the finalize method may never get called on some objects. Try the following example code:

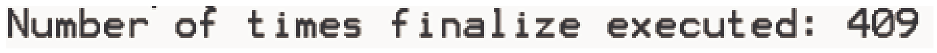
```
public class Test
{
     //int[] val = new int[100];
     private static int finalizeCount = 0;
     public void finalize()
     {
          super.finalize();
          finalizeCount++;
     }

     public static void main(String[] args)
     {
          for (int i = 0; i < 1000; i++)  { new Test(); }

          System.out.println(
               "Number of times finalize executed: "
                         + Test.finalizeCount);
     }
}
```

On my system, when I executed this program and it produced the following result:



```
Number of times finalize executed: 0
```

Now, uncomment the statement *private int[] val = new int[100];*, compile and try. Again, on my system, it gave the following output:

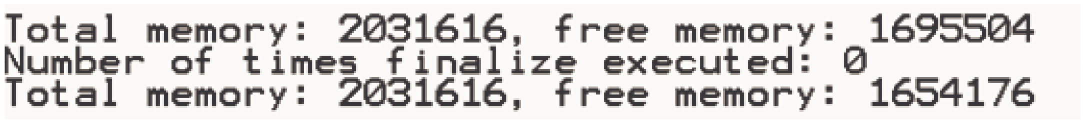


```
Number of times finalize executed: 409
```

As you can see, in the first case, the object being created was pretty small. There was enough memory to accommodate all the 1000 objects. In the second case, each object utilized at least 404 bytes. That is a request for a total of about 400K of memory. So, it decided to garbage collect less than half the created objects. Let's modify the main as follows to get a better understanding:

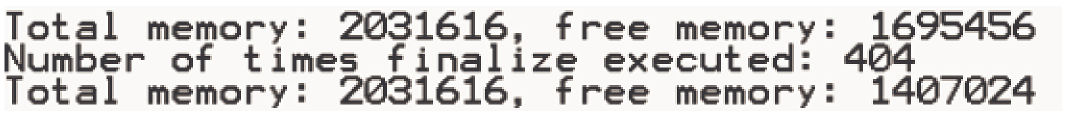
```
public static void main(String[] args)
{
    Runtime runtime = Runtime.getRuntime();
    System.out.println(
    "Total memory: " + runtime.totalMemory() + ", "
        + "free memory: " + runtime.freeMemory());
    for (int i = 0; i < 1000; i++)
    {
        new Test();
    }

    System.out.println(
        "Number of times finalize executed: "
                + Test.finalizeCount);

    System.out.println(
    "Total memory: " + runtime.totalMemory() + ", "
            + "free memory: " +
            runtime.freeMemory());
}
```

Now, the output produced without the *private int[] val;* is shown below:



```
Total memory: 2031616, free memory: 1695504
Number of times finalize executed: 0
Total memory: 2031616, free memory: 1654176
```

and the output with the *private int[] val;* is shown below:



```
Total memory: 2031616, free memory: 1695456
Number of times finalize executed: 404
Total memory: 2031616, free memory: 1407024
```

From this observation, we realize that the Java garbage collection kicks in only when the memory usage has reached a certain level. Further, the finalize method may not be called on an object, i.e., there is no guarantee that this will ever be executed. Also, you may hve noticed that there is a method called gc() in the System class. Invoking this call will result in the execution of the garbage collector. However, again, there is no guarantee that the garbage collector will collect all the zombies that may be around at that instant. Further, the call to the gc can substantially slow the program execution. Finally, there is a method called runFinalizersOnExit() in the System class. This method simply tells the Java Virtual Machine to garbage collect all objects when the program is about to exit. However, it has been deprecated with the following message "*This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.*" This is kind of like the aggressive garbage collectors in a neighborhood where I used to live – half my garage was gone the day after I complained!

What's the solution to release resources in Java?

If you want to release your resources, do not depend on the garbage collector to do it for you! A programmer with experience using JDBC may tell you that if you rely on the AGC, your program will fail with an error that you have too many resources open. The reason is the delayed AGC. As far as your code is concerned, you have released your objects. However, since the finalize method has not been called, the database layer still thinks that the resource is being used. One way to address this issue of resource cleanup is to follow a similar step like what has been done in the JDBC API. If your class uses a resource and you want it to be freed immediately, then provide a method that will free up the resources when called. You may name this method as close, cleanup, destroy, release, etc. The user of your object will have to call this method when the object is no longer needed. Unfortunately, this brings back the hard memories of calling the delete in C++. **In Java, the AGC takes care of memory cleanup, but not resource cleanup or book keeping. It is still your responsibility**.

At least we are not leaking memory, or can we?

It is indeed possible to cause memory leak in Java. To avoid memory leaks, it requires some caution and discipline in writing code. Here is an example from Effective Java[2] (Item# 5: Eliminate obsolete object references):

```
import java.util.*;

public class Stack
{
     private Object[] elements;
     private int size = 0;

     public Stack(int initialCapacity)
     {
          elements = new Object[initialCapacity];
     }
     public void push(Object e)
     {
          ensureCapacity();
          elements[size++] = e;
     }
     public Object pop()
     {
          if (size == 0) throw new EmptyStackException();
          return elements[--size];
     }

     private void ensureCapacity()
     {
          if (elements.length == size)
          {
               Object[] oldElements = elements;
               elements =
                    new Object[2 * elements.length + 1];
               System.arraycopy(oldElements, 0,
                                   elements, 0, size);
          }
     }
}
```

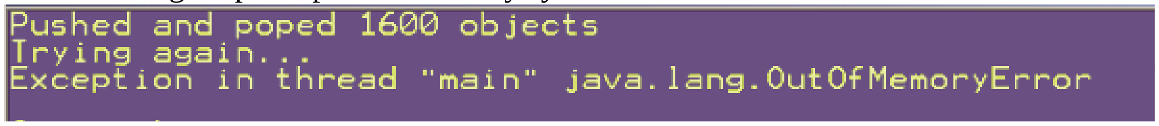
Study the code above carefully and see if you can find a memory leak!

Here is an example code that exercises the above Stack:

```
class Stuff
{
     private int[] vals = new int[10000];
}

class User
{
     public static void main(String[] args)
     {
          try
          {
               final int testSize = 1600;
               Stack s = new Stack(testSize);

               for (int i = 0; i < testSize; i++)
                    s.push(new Stuff());
               for (int i = 0; i < testSize; i++)
                    s.pop();
```

```
                              System.out.println("Pushed and poped " +
                                   testSize + " objects");
                              System.out.println("Trying again...");
                              for (int i = 0; i < testSize; i++)
                                   s.push(new Stuff());
                              for (int i = 0; i < testSize; i++)
                                   s.pop();
                              }
                              catch(Throwable t)
                              {
                                   System.out.println(t);
                              }
          }
}
```

The following output is produced on my system:



```
Pushed and poped 1600 objects
Trying again...
Exception in thread "main" java.lang.OutOfMemoryError
```

You can modify the value of testSize to get the same response on your system depending on your system's configurations.

Now, modify the pop method of the Stack as follows and try again:

```
     public Object pop()
     {
          if (size == 0) throw new EmptyStackException();
          Object result = elements[--size];
          elements[size] = null;
          return result;
     }
```

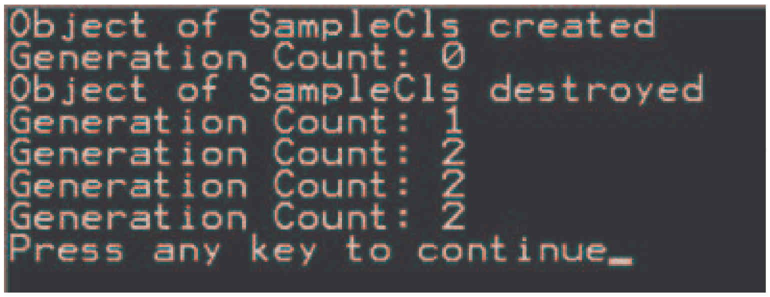
You will notice that the OutOfMemoryError does not occur any more. **It is important to release references to obsolete objects as soon as you figure that you do not need them[2].**

AGC in .NET

In the .NET environment, AGC is provided for managed objects. There are two kinds of objects in .NET – *value objects* and *reference objects*. Value objects are created on the stack while the reference objects are created on the heap. The heap in .NET is a managed heap. .NET employs a lazy GC as well. However, in order to make the garbage collection sweep more efficient, it partitions the heap based on a *generation count*. A generation count may be defined as the number of times an object has seen or survived the garbage collection sweep. .NET does not keep the generation count beyond the value of 2; in other words, an object's generation count may be 0, 1 or 2. The objects on the heap are grouped based on their generation count. When a garbage collection sweep occurs, it first tries to reclaim as many unreachable objects as possible from the group with a generation count of 0 and 1. If it still needs more memory to be reclaimed, then it looks for unreachable objects within the group with a generation count of 2. The basic logic behind this heuristics is that objects tend to live shorter; if they live for a while, they may live longer. This makes sense if we consider the fact that objects created locally within function/methods do not live too long. However, objects that are created and passed around from one method to another tend to live longer. Here is an example that illustrates the generation count and garbage collection sweep:

```
// C# sample
public class SampleCls
{
     public SampleCls()
     {
          Console.WriteLine("Object of
               SampleCls created");
     }
     ~SampleCls()
     {
          Console.WriteLine("Object of
               SampleCls destroyed");
     }
}

class GCTester
{
// Build in release mode to see the garbage collection work
// immediately.

     static void Main(string[] args)
     {
          SampleCls sampleClsObject = new SampleCls();
          Object obj = new Object();

          for (int i = 0; i < 5; i++)
          {
               Console.WriteLine("Generation Count: " +
                    GC.GetGeneration(obj));

               GC.Collect();
               GC.WaitForPendingFinalizers();
          }
     }
}
```

The output produced by the above code is:



```
Object of SampleCls created
Generation Count: 0
Object of SampleCls destroyed
Generation Count: 1
Generation Count: 2
Generation Count: 2
Generation Count: 2
Press any key to continue_
```

Did you notice that the object of SampleCls was destroyed even though we are still within the main method where the sampleClsObject reference is still in scope? This is a feature of .NET where they perform aggressive garbage collection. Even though the reference is still in scope, if you are not using it any more then the object becomes a candidate for GC faster than in Java.

Just like in Java, the garbage collection in .NET could be much later even though the object may be unreachable. So, relying on the Finalize method (written in C# using the C++ destructor syntax ~) is not a good idea.

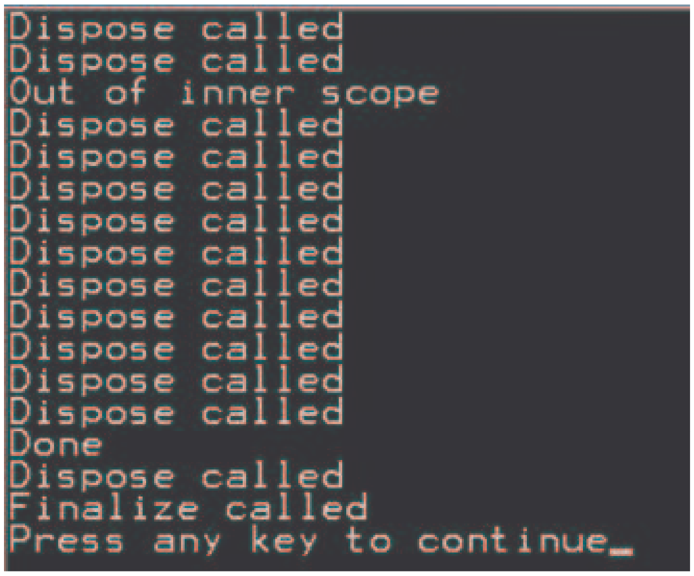
What's the solution to release resources in .NET?

In the discussion earlier about Java, we said "If your class uses a resource and you want it to be freed immediately then provide a method that will free up the resources when called. You may name this method as close, cleanup, destroy, release, etc." The same is the case in .NET, except that they have provided a much more methodical approach to this issue. In .NET, a new interface named IDisposable has been introduced. The IDisposable interface has a method called Dispose. You will simply implement the IDisposable interface. Frameworks like Windows Forms check to see if a control implements the IDisposable interface, and call the Dispose method on it. Similarly, in your application, similarly, you can check if that interface is supported by a class you are using and call the Dispose method on the object when it is no longer needed. There is also added advantage to this approach in .NET.

The execution of the Finalize method will slow down the garbage collection process. Turning it off on objects that do not have any resource to be cleaned up (or on objects that have been disposed properly) will provide better efficiency. This is made possible by a method SuppressFinalize() of the GC class. An example below illustrates this concept:

```
// Garbage.cs
using System;

namespace UsageOfDispose
{
     public class Garbage : IDisposable
     {
          private bool disposed = false;

          public void Dispose()
          {
               if (disposed == true)
                    throw new
                         ObjectDisposedException(
                              "Already Disposed");
               disposed = true;

               // What ever cleanup
               Console.WriteLine("Dispose called");
               GC.SuppressFinalize(this);
          }

          ~Garbage()
          {
               Dispose();
               Console.WriteLine("Finalize called");
          }
     }
}

// PlayingWithFinalize.cs
using System;

namespace UsageOfDispose
{
     class PlayingWithFinalize
     {
          static void Main(string[] args)
          {
               Garbage g1 = new Garbage();

               g1.Dispose();
               g1 = null;

               {
                    Garbage g2 = new Garbage();
                    Garbage g2a = new Garbage();
                    //...
                    g2.Dispose();
               }
               Console.WriteLine("Out of inner scope");

               for (int i = 0; i < 10; i++)
               //for (int i = 0; i < 1000; i++)
               {
                    Garbage g3 = new Garbage();
                    //...
                    g3.Dispose();
                }

                Console.WriteLine("Done");
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }
        }
}
```

The output from the above program is:



```
Dispose called
Dispose called
Out of inner scope
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Dispose called
Done
Dispose called
Finalize called
Press any key to continue_
```

Notice that the program creates thirteen objects. In main, Dispose is being called properly on 12 of these objects and not being invoked on one. Notice that the Finalize method is being called on only that object that we missed to call Dispose() on.

Conclusion

There is no double that Automatic Garbage Collection simplifies our code. Those with experience in C++ would probably agree with me more on this. However, when it comes to releasing resources, unfortunately, AGC does not help. You still have to rely on calling methods on the objects at an appropriate time - destructor in C++, a resource releasing method in Java, Dispose in .NET. In addition, we also need to release references on obsolete objects to avoid any memory leaks.

References

1. Ken Arnold, James Gosling, David Holmes, *The Java Programming Language, Third Edition*. Addison-Wesley, Boston, MA, 2000. ISBN:0-201-70433-1.
2. Joshua Block, *Effective Java Programming Language Guide*. Addison-Wesley, Boston, MA, 2001. ISBN:0-201-31005-8.
3. Microsoft Developer Network (MSDN). http://msdn.microsoft.com