Memory and Resource Management

Estimated time for completion: 60 minutes

Goals:

- Understand IDisposable and the "using" construct
- Understand how the garbage collector uses generations to optimize collection
- Understand how the garbage collector compacts memory
- Understand how to use the performance counters to monitor GC usage

---

Overview

This activity involves performing various experiments designed to highlight the generational nature of the CLR garbage collector and how to effectively deal with memory and non-memory resources.

Part 1- Using the "using" construct

*In this activity you will fix an application that is misbehaving by applying the **using** pattern.*

1. In the before/snotepad directory you will find snotepad.sln which implements a simple notepad application. Open it and run it.
2. Try opening a text file, editing it, and saving it. You should notice that if you open a file, edit it, then try to save it that you get an exception. Similarly if you create a new file you'll be able to save it once, but if you hit save a second time you'll get an exception.
3. On the **Tools** menu you'll find a **Collect Garbage** option, which calls `GC.Collect`. Try editing a file and saving it. If you get an exception select **Tools | Collect Garbage** and try saving again. You should have a pretty good idea at this point what's wrong with this application - your job is to fix it.
4. **You should find it relatively easy to fix this application. Try to fix it yourself or read the steps below if you have difficulty.** Hint: the functions you need to focus on are `OpenFile` and `SaveFile` in **Form1.cs**.
5. The reason the application is misbehaving is that `OpenFile` and `SaveFile` create `StreamReader/StreamWriter` objects but neglect to `Close` them.
6. You can fix this by just adding explicit calls to `Close` but a better solution is to add `using` blocks to the two functions. Provide them now. The code is provided below if you get stuck:

```
void OpenFile(string path)
 {
  try
  {
   using (StreamReader reader = new StreamReader(path))
   {
    textBox1.Text = reader.ReadToEnd();
   }

   Dirty = false;
   CurrentPath = path;
  }
  catch(Exception e)
  {
   MessageBox.Show(e.Message);
  }
 }

 void SaveFile(string path)
 {
  try
  {
   using (StreamWriter writer = new StreamWriter(path, false))
   {
    writer.Write(textBox1.Text);
    writer.Flush();
   }

   Dirty = false;
   CurrentPath = path;
  }
  catch (Exception e)
  {
   MessageBox.Show(e.Message);
  }
 }
```

7. Try running the application now. You should be able to open, edit, and save files normally.

Part 2- Implementing IDisposable: Object Pooling

Object Pooling *is a useful idea for many scenarios. One of the primary uses is for dealing with resources that are scarce and/or have a high cost of creation. Rather than create a new object every time libraries can implement pools from which clients borrow temporarily, promising to return them when finished so other code can use the connection. ADO.NET for example uses object pooling to share database connections (which are precious and expensive to establish) without requiring clients to be aware of the details.*

In this part of the lab you will work with a simple object pool and see the important role `IDisposable` *plays in making it work seamlessly.*

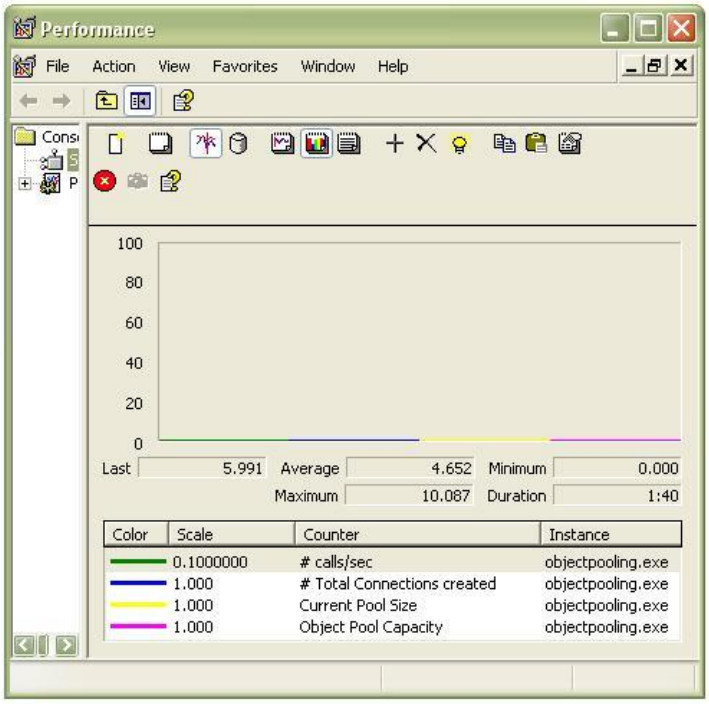
1. Open the ObjectPooling project from the before\pooling directory. This project consists of a single form that can be used to spawn worker threads. Each thread runs function `WorkerThread` that simply sits in a loop making calls to a class called **BasicConnection** (in BasicConnection.cs). The code looks like this:

```
void WorkerThread()
{
 while (!Cancelled)
 {
  BasicConnection conn = new BasicConnection();
  conn.DoWork("Hello");
  conn.DoWork("Goodbye");
 }
}
```

`BasicConnection` in turn is an extremely simple class that simulates a resource that is expensive to acquire - like a database connection:

```
class BasicConnection
{
 public BasicConnection()
 {
  Thread.Sleep(2000);
 }

 public void DoWork(string message)
 {
  PerfmonCounters.CallsPerSecond.Increment();
  Thread.Sleep(5); //simulate work
 }
}
```

2. The intended usage for this program is to run it and then kick off performance monitor to watch the behavior of the application. Run the application and hit the "Start" button. Five threads will be started in response (an animation in the status bar will run as long as there are child threads running).
3. Now hit the **Perfmon>>** button. This will launch perfmon.msc, which should load up perfmon in Histogram Mode. It should look something like this:



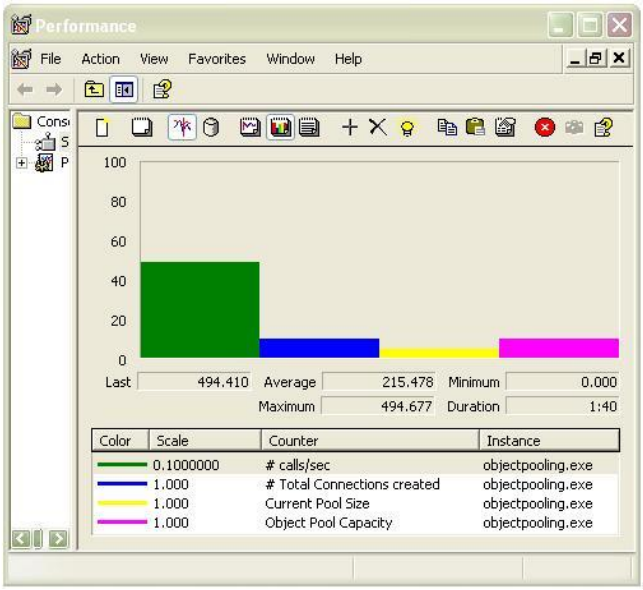
Perfmon isn't doing much yet, but the value you want to look at is **# calls/sec** which is recording the throughput of the application. You should see a value of something like eight calls a second which is pretty miserably bad. The reason for this should be obvious from the code snippets above - each thread can only do one call every two seconds because it is paying a high initialization cost. This is a good example of a place where object pooling makes sense. **Your goal now is to implement a pooled object class to improve performance without forcing the client application to be involved.**

4. A simple object pool implementation has been provided for you in ConnectionPool.cs. Add it to the project by right-clicking on the project in the solution explorer and clicking **Add...| Existing Item**. This class is fairly complex but you will not have to modify it at all. All you have to concern yourself with is the public API of the class which consists of the following functions:

```
static int Capacity { get; set; }
static BasicConnection GetConnection();
static void DisposeConnection(BasicConnection conn);
```

5. First the easy bit - add some code to `trackBar1_Scroll` to set the `Capacity` of the object pool to the trackbar value. This will allow you to dynamically adjust the pool size while the application is running.
6. You should see a file in your project called PooledConnection.cs. This is where you will define the class that will perform object pooling on behalf of the application. Note that in real-world code `PooledConnection` would generally live in an assembly with `BasicConnection` and `ReallySimpleConnectionPool`, both of which would be marked as `internal`. For simplicity's sake you're just implementing it all in a single application.

7. Define a new class called `PooledConnection`. In order to implement object pooling `PooledConnection` should do the following:
    a. Hold a private field of type `BasicConnection`. Acquire one from the object pool in the default constructor or in a field initializer.
    b. Implement a `DoWork` function that takes a string and simply delegates it to the `BasicConnection` object.
    c. Implement a Finalizer that calls `ReallySimpleConnectionPool.DisposeConnection`. You probably should do this in a `try / catch` block, in case any exceptions get thrown.
8. Now go into `Form1.WorkerThread` and change it to use `PooledConnection` instead of `BasicConnection`. Run the app again. Is it better, or worse?
9. You should see the application just hanging with a throughput of zero. If you place breakpoints in your code you should see the same behavior you saw in snotepad - when you run GC.Collect a few objects will run but then will hang again waiting for finalization.
10. You should know what to do at this point - implement `IDisposable` of course! Implement IDisposable on `PooledConnection`, keeping the following in mind:
    a. `Dispose` should return the connection back to the pool.
    b. `Dispose` should call `GC.SuppressFinalize` to prevent the finalizer from running.
    c. Calling `DoWork` after `Dispose` should result in an `ObjectDisposedException` being thrown.
    d. Calling `Dispose` multiple times should have no effect.
11. Now change the code in WorkerThread to use a using block. Run the application again. You should see tremendously improved throughput, similar to this:



12. You're done writing code. Do you see how `IDisposable` plays a pivotal role in enabling object pooling to work? Play around with the pool a bit. What happens if you shrink the pool to be smaller than the number of threads? What happens if you make it much bigger? How much throughput can you wring out of the system by adjusting the number of threads and the size of the object pool?

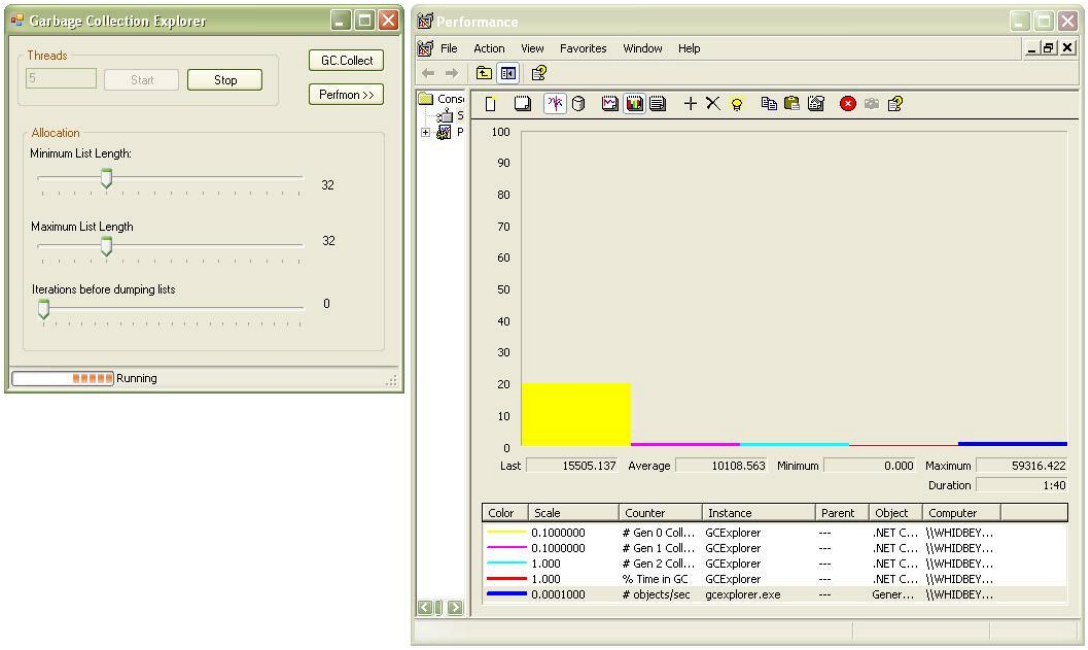
Part 3- Generations and "Midlife Crisis" (optional)

The generational nature of the .NET garbage collection makes it more responsive on average, but it can have some subtle and important impact on your code. Throughput of server applications often depends on speed of memory allocation. The generational nature of the .NET Garbage Collector means that allocating ***n*** *bytes of memory will take varying amounts of time. If you allocate 1 million objects one at a time, immediately setting them to* `null` *the collector only has to do Gen0 collections, which are relatively cheap. If you instead allocate a linked list of 500,000 objects, set it to* `null`*, then allocate another 500,000 it will probably take longer, even though both programs allocated the same amount of memory.*

This activity is based on that idea: ***it's a very different thing to allocate one million objects one at a time than it is to allocate a list with one million objects in it.*** *Toward that end, this application spawns threads that sit in a loop allocating simple linked lists of objects. Each thread allocates linked lists of varying lengths (which you can control from the UI). These lists are then held for a few loop iterations before they are overwritten, releasing them to the garbage collector.*

An application has been provided for you, so you won't need to write any code for this experiment. What you will do is experiment with the garbage collector to see how patterns of allocation can have profound effects on application behavior and throughput.

1. The mechanics of this application are very similar to the previous part of the activity. Open the GCExplorer project and run it. Click the **Perfmon>>** button to kick off Performance Monitor, and click **Start** to begin the process. You should see something like this:



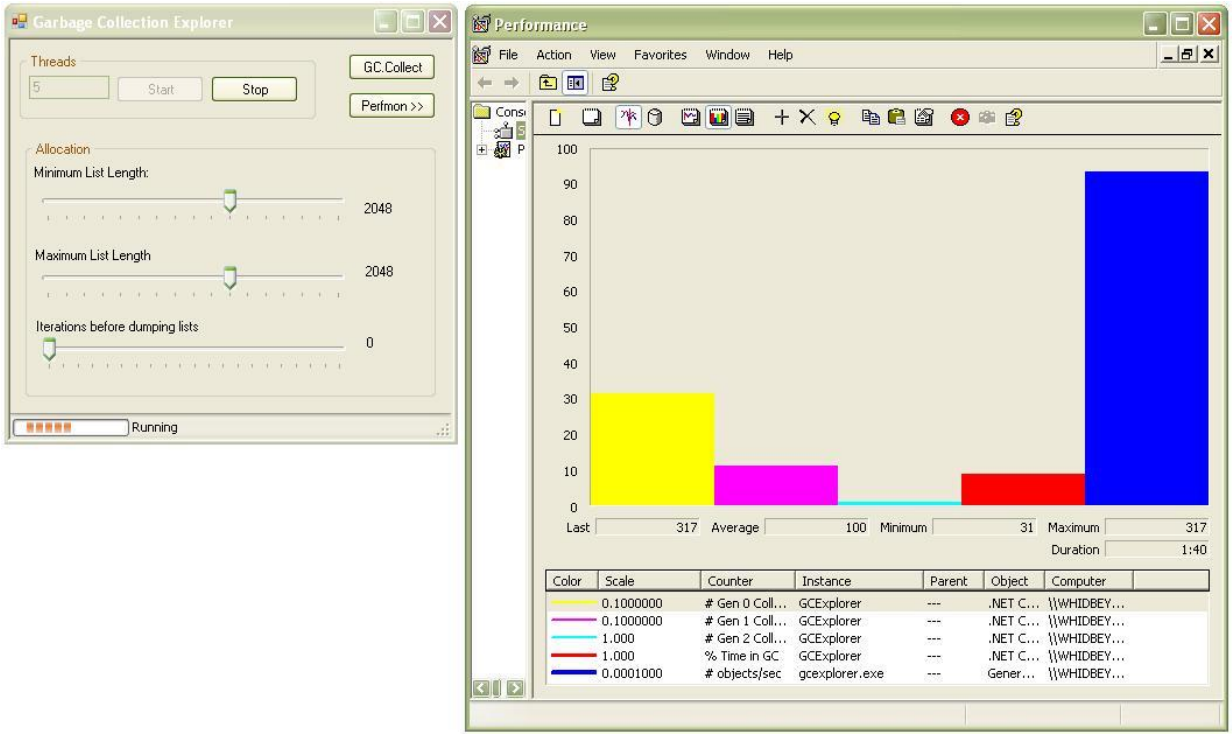
2. You can control the number of threads to run. Each thread runs a loop that looks approximately like this:

```
void WorkerThread()
{
 Random r = new Random();
 Node[] lists = new Node[100];
 int n = 0;

 long lastThreshold = 0;

 while (!Cancelled)
 {
  long MaxPow = Interlocked.Read(ref MaxPower);
  long MinPow = Interlocked.Read(ref MinPower);

  MinPow = Math.Min(MinPow, MaxPow);

  long pow = ..Generate a random number between 2^MinPow and 2^MaxPow..

  long threshold = Interlocked.Read(ref DumpThreshold);
  ...

  Node head = AllocateList((long)Math.Pow(2.0, pow));

  if (threshold > 0)
   lists[n % threshold] = head;
  n = n+1;
 }
}
```

See Form1.cs for the full implementation. This loop allocates lists of varying length based on the values of the sliders in the UI. It stores some number of them in the `lists` array and overwrites entries occasionally based on the value of the third slider in the UI.

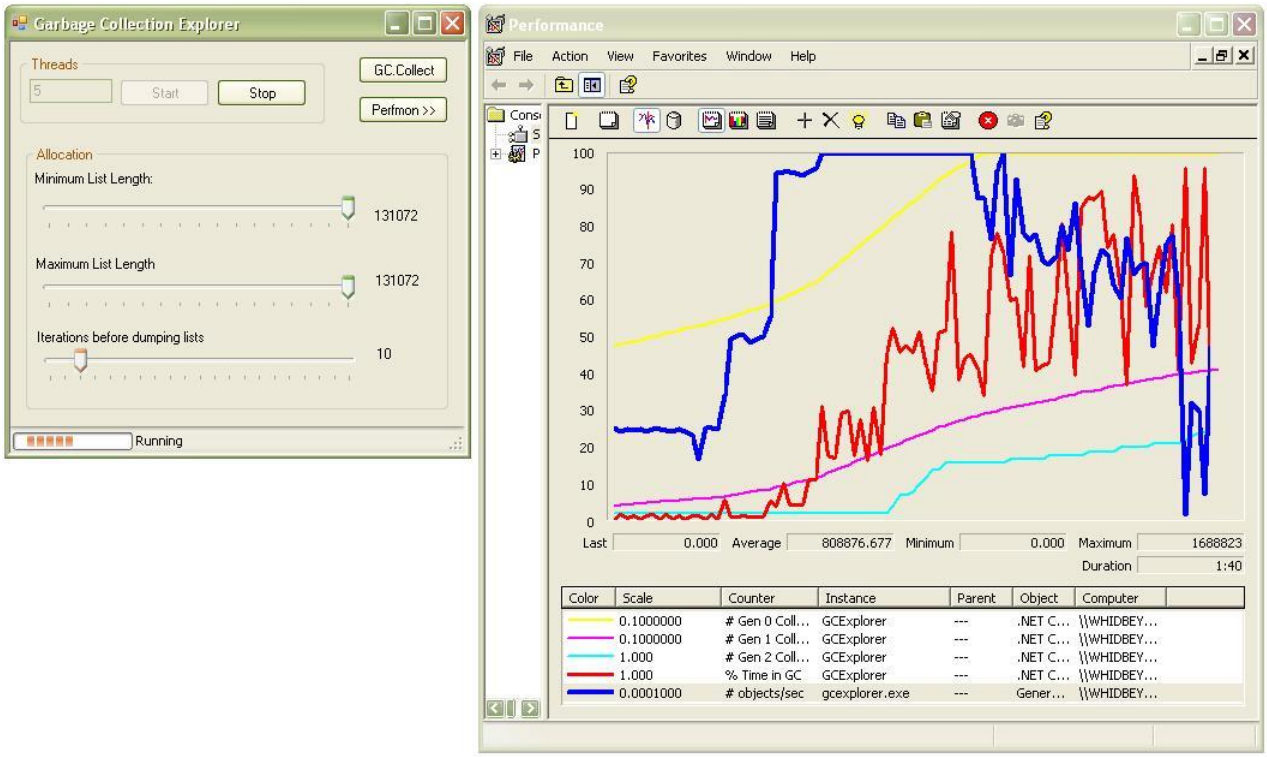
When you first start this program, 100% of the memory allocated is collected in Gen0. This is very easy on the garbage collector.

3. If you slide the sliders to the right a little (but stay below about 4k objects) you can see the garbage collector work a little harder. In the picture below you can see that %time in GC has gone up, but we see very healthy (and stable) throughput and a nice distribution of Gen0, Gen1, and Gen2 collections.



4. If you keep ratcheting the size of the lists eventually some of the objects in the list are promoted into Gen1 (or even Gen2) before the loop has finished adding items to the list. If you increase the length of time the lists are held before they are dumped you increase memory pressure on the program even further, which has a double whammy effect.

First, the increased memory demands force the system to scrounge more memory. Assuming there's enough memory around this should be a temporary problem. The second (and potentially worse) problem is that pushing the lists into Gen2 and then freeing them means the system spends more time doing Gen2 collections. The results can be seen here:



This shows the result of slowly increasing the minimum list size trackbar up to the maximum and then increasing the number of iterations lists are held before being released. Throughput (the blue line) jumps every time the list length is increased until about halfway across the graph when the number of Gen2 collections starts growing at an alarming rate. Gen2 collections are a lot more expensive than Gen0 or Gen1 collections, so the program spends a lot more time in GC (the red line). As you can see, this wreaks havoc with the number of objects per second being allocated, which in a real system would destroy throughput. The moral of the story here is: **avoid promoting objects into Gen2 and then abandoning them**.

5. Play with the application (but be careful - you can bury your system with it!). Once the program is running can you predict the effect moving any given slider will have? What about increasing or decreasing the number of threads?
6. GCExplorer allocates linked lists of type `Node` which doesn't do anything except take up space. What happens if you add a finalizer (even a do-nothing finalizer) to class `Node`? How does that affect promotion rates? What effect does it have on memory usage and throughput?