Lab 9
Due: Aug 4, 2023 at the end of class

1 Lab 9 Specification

1.1 Overview

The purpose of this lab is to explore C#'s built-in garbage collector and analyze a language system's performance. The skeleton code at `https://classroom.github.com/a/wK-gMDHK` is the start of this week's lab. C# has an explicit call to run the garbage collector, which is very convenient to see its effects. Some language systems run a garbage collector as needed, but give programmers no way of running it manually. Since hardware and software setups are not identical, there will be significant differences in different people's final results.

If you get an error on Mac: `Debugger operation failed: Synchronous operation cancelled!`, go to System Preferences → Security & Privacy → Privacy tab → Automation. There should be a Visual Studio entry on the right. Make sure Terminal is checked.

1.1.1 Goals

1. Practice invoking C#'s garbage collector
2. Practice modifying C# code
3. Practice analyzing data in Excel

1.2 Garbage Collector Parts

The following three parts examine different behaviors in the garbage collector. For all three, you will write some code and print out some timing results. For each part, you will also plot the timing results to visually show the effect that the garbage collector is having. The `Main` method is located in **GC.cs**.

1.2.1 Collect All Garbage

In **Part1.cs**, allocate 8000 **LargeObject**s (using the **new** keyword), but don't store their references. Print out the timer after each allocation. Modify your code to run the default garbage collector after each 1000 allocations. Use the **System.GC.Collect()** call to invoke the garbage collector. For the timer, you don't need to create a new stopwatch each iteration – just print details of the currently running one.

Copy the command line output to Excel. In Excel, calculate the time differences between each sequential output (time deltas) to see how much time elapsed between one allocation and the next. Create a **Scatter chart** of the results (tick number vs time deltas).

This should highlight when the program took extra time between allocations, as the program was working on garbage collection routines (or other things) at the same time. There are other demands on your CPU so you may see unrelated spikes. If you see one or two tremendous outliers, delete them manually so that your data is actually scattered in the y range.

Save the Excel file in the same directory as your source files and name it **part1.xlsx**.

1.2.2 Specified Generations

In **Part2.cs**, create a new array with a size of 8000 of **LargeObject**s. Then store a new **LargeObject** in each element of the array. Print the timer after each allocation, just like in Part 1. Run the garbage collector similarly to before – every 1000 allocations. This time, call **System.GC.Collect(0)** to run the garbage collector only on the Generation 0 objects. The generation number is how many times an allocated block has survived garbage collections. You will have to modify the call in **GC.cs** to invoke this code.

In a separate Excel file from part 1, chart the resulting data points (tick number vs time deltas). You should see the increasing time impacts with each garbage collection as the garbage collector has more allocations to consider, but none of the memory can be freed.

Save the Excel file in the same directory as your source files and name it **part2.xlsx**.

1.2.3 Detecting Collections

In **Part3.cs**, allocate 80,000 **LargeObject**s, with a size of 40 rather than their default. Like in Part 1, don't save their references. Don't invoke the garbage collector at all. Print the timer out after each **10 allocations**. Since you are filling up the heap with objects that can be deallocated, garbage collections should occasionally trigger during allocations.

For this part, chart the time deltas again. You should see that most allocations take about the same amount of time, but there are isolated performance hits. Some of these are garbage collection events! From a chart like this, we could infer the heap size and the runtime impact of garbage collection on a program. If you see one or two outliers right near the beginning of the run, set them to an average value so the chart better fills the y axis.

Save the Excel file in the same directory as your source files and name it **part3.xlsx**.

2 Submission Check

Make sure that your Excel files are correctly included in your submission. Visual Studio should automatically adds new files to your repository if it detects them, but you will still have to commit/push the changes once the files have been added. If you are unsure about your submission, check the available files on the Github website.

3 Grading

Be professional. Make results easy to understand and grade. Include only those parts to be graded. Write comments in each class and for specific methods if they are confusing or complicated.