Singleton Pattern

Estimated time for completion: 45 minutes

Overview:

In this lab you will develop a general purpose implementation of the singleton pattern and then put your new found knowledge into creating a single instance windows form. Time permitting you will then look at developing a thread scoped version of a singleton

Goals:

- Develop and put to use the singleton pattern
- Create a thread scoped singleton

Lab Notes:

Part 1: Create a basic Singleton implementation

In this part you will create and test a basic singleton implementation, you will then use this implementation in the remaining parts of the exercise.

1. Launch Visual Studio and create an empty console project, call the project BasicSingleton and set the location to C:\labs\work\Singleton\before
2. Now add a new class to the project called Singleton, and implement a basic singleton class. Remember that you are having to control instantiation and so ensure the type has a private constructor and a public static accessor to obtain a reference to the single instance

```
public class Singleton
{
    private static Singleton singleton = null;
    private Singleton(){}
    public static Singleton GetInstance()
    {
        if(singleton == null)
          singleton = new Singleton();

        return singleton;
    }
}
```

3. When you are happy with your basic implementation, validate that it has the desired singleton behavior. You could use object.ReferenceEquals to ensure you always get back the same reference and that it is not null. Finally check to make sure you cannot call new on your new type
4. Once you are happy it is working re-visit it and ensure that you are happy it is thread safe. There should be no possible race conditions resulting in multiple instances being created. How might you test this ?

Part 2: Single Instance form

In this part you will refactor an existing piece of code to make use of a singleton. The piece of code in question is a windows forms application that wishes to have only one instance of a Find dialog. Currently this has been implemented in a less than ideal way, by simply holding onto the form via a static reference. Whilst the application exhibits singleton style behavior it does not force developers to do so.

1. Load the SharingForms.sln into visual studio and familiarize yourself with the project. Run the application, select File, New and see that a new window appears. Now select "Edit, Find", this should show a find dialog box. Enter some text and hit Find (note it does not actually find anything). Next time you select the Find option the same piece of text is displayed in the "Find What" text box. This should be the case irrespective of the top level window you issue the find from.
2. Examine the code inside SexyNotepadForm.cs by right clicking on the file inside the solution explorer and selecting View Code. You will see a method called findToolStripMenuItem_Click. Inside here is the logic to implement the shared form behavior, notice that whilst it works the technique does not prevent the creation of multiple forms. Now look at SearchForm.cs this is the type you need to turn into a singleton if you are to implement the desired behavior using the Singleton pattern.
3. Refactor the SearchForm class to use the Singleton pattern. Making the SearchForm constructor private and providing the necessary static field and GetInstance method.
4. Refactor the SexyNotepadForm.cs method findToolStripMenuItem_Click to use the new singleton logic. Compile and test your new code.

Part 3: Thread Scoped Singleton

Whilst it is often necessary to have a single shared resource for an entire application this can also hinder performance in a multi threaded application. It is therefore sometimes more desirable to have a singleton per thread. The code in this example demonstrates that a single log file being utilized by multiple threads could be a hindrance to application throughput, further it could also result in logic becoming serialized thus effecting scalability.

1. Load the LoggingThrouput.sln and familiarize yourself with the project. The file Program.cs is used to exercise a Singleton logging implementation with multiple threads. The Logger.cs file contains a singleton implementation that implements centralized logging. All logging is directed to a file called log.txt, in order to ensure that messages are not interleaved the LogMsg method serializes each write to a file using the lock keyword. This has an effect of throttling throughput, run the application and make a note of how long it takes. Note if you are running on a single core machine you will need to add an additional Thread.Sleep after the actual logging operation, but before releasing the lock in order to see the effect.
2. You could reduce contention in this application by giving each thread its own file. Do this by refactoring the Logger type to support this behaviour, without changing anything inside Program.cs. You will need to make Logger.GetInstance() to return an instance that is bound to the calling thread. You can do this by decorating the static instance with the

[ThreadStatic] attribute, you will also need to ensure that you create a unique file for each thread, you could use Thread.CurrentThread.ManagedThreadId as a prefix or suffix to the log filename .

3. When you are happy with your refactoring of Logger re-run the application, and you should see faster throughput

---

Part 4: Create a Generic Singleton

Creating the most optimal singleton implementation is not that straight forward. Rather than keep re-implementing the singleton mechanics, create a Generic Singleton based on your original implemention.

4. Create a class called Singleton that has a single type argument call it T, used to represent the type of the instance the singleton is wrapping up. This class will contain the appropriate static methods used to control the creation of a single instance of T, so as to allow the following programming model

```
class Highlander : Singleton<Highlander>
{
   private Highlander()
   {

   }
   // Instance methods for Highlander class
}

Highlander theOnlyOne = Highlander.GetInstance();
```

5. If you were to unit test a Singleton are there any issues with this approach, and how would you resolve them.