Singleton

“Highlander Pattern”

DEVELOPMENTOR

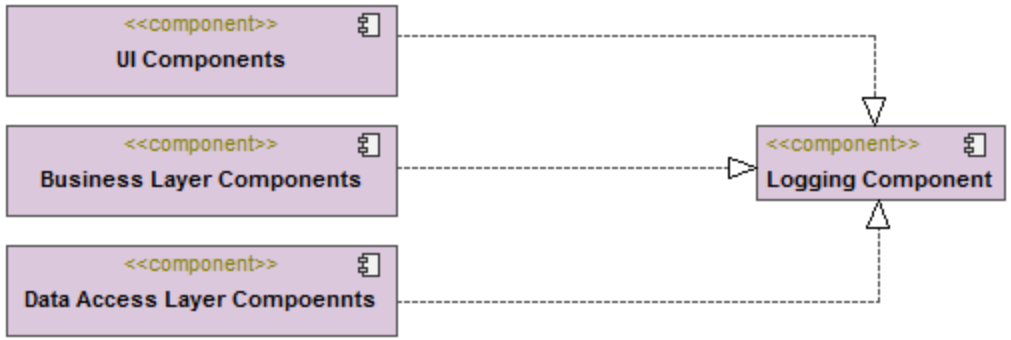
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

There can be only one

- There are some objects we want to share system wide
  - Expensive to create
  - Represent a single instance entity
  - Shared state
- Examples
  - Cache
  - Logging
  - Preference settings
  - Serial Interface

Problem

- Allow the application to write diagnostic error messages anywhere in the app to a central log file



Could I use global variables (a.k.a. statics)?

- Provides a single point of shared access.
- Does not enforce single instance
  - programmers have to know about the global variable and not accidentally instantiate their own copy.
- When do the global objects get created ?

```
public static class Global
{
      public static readonly Logger Logger = new Logger();
      public static readonly Cache Cache = new Cache();

}
```

So how about using a static class ?

- Provides a single point of shared access.
- Ensures only one “instance”
- What if I want to derive from a non static class?
- What if I want to provide a different implementation at runtime?
- What if you want multiple instances? (Perhaps one per thread)

```
public static class StaticFileLogger {
   private static readonly StreamWriter logStream =
                                          new StreamWriter(@"log.txt");

   public static void LogMsg(string msg) {
            logStream.WriteLine("{0},{1}", DateTime.Now, msg);
   }
}
```

Ensuring only one instance

- Using static classes would result in a fragile design
- We really need to use objects to maximum use of OO technology
- Re-state the problem
  - We need to prevent use of new multiple times for our type
  - We need a way to always obtain the same instance
- Possible Solution
  - Encapsulate object creation
    - Make the object constructor private
    - Provide a static method that will manage instance creation

Simple Singleton

```
public class Logger
{
    //Single instance created internally
    private static Logger _instance = new Logger();

    // type constructor private,
    // clients can not create an instance
    private Logger(){  }

    //Shared instance returned to clients
    public static Logger GetInstance() {
        return _instance;
    }

    // Instance methods
    public void LogMsg(string msg) { ... }
}
```

Singleton client

```
static void Main(string[] args)
{
  // Logger logger = new Logger() will not compile
  //
  Logger logger = Logger.GetInstance();

  logger.LogMsg("Hello World");
}
```

How does it work ?

- Relying on the CLR guarantee that the type constructor will be called only once
- There is only one way to obtain an instance outside the type
  - GetInstance()
- Any Issues?
  - When does the instance of Logger get created?
  - Answer: That depends
    - Release or Debug
    - beforeFieldInit attribute
  - What happens if an exception fires during construction?

Lazy Initialization

- To implement lazy initialization can explicitly define a constructor

```
public class Logger
{
     // removed new Logger()  from here
     // and defined an explicit type
     // constructor
     private static readonly Logger instance;

     static Logger()
     {
         instance = new Logger();
     }

     ...
}
```

Explicit type constructor

- In both cases CLR will generate a type constructor
  - The difference being the beforeFieldInit attribute
- beforeFieldInit attribute says it is OK to initialize type any time before use
  - Allows the CLR to be provide more efficient initialization
- Without beforeFieldInit CLR initializes the type at its first use
  - The CLR still needs to guarantee that the constructor is only called once
  - Has to ensure that every static reference checks for initialisation
  - Has to ensure that check is done in thread safe manner

Issues with a Singleton

- .NET Singleton is scoped on an App Domain not process
- The singleton object lives forever
  - Potential stateful behaviour across unit tests
  - Consider creating private method to reset singleton
- Difficult to mock
  - Singleton.Instance is tightly coupled
  - Consider
    - Making the constructor protected, virtual methods to allow stubbing
    - Creating private method to SetInstance, invoke using reflection
- Consider if you really need one before creating one

Who wants to live forever?

- One motivation of the singleton is to reuse a resource intensive object
- Cached DataSet
  - Takes a long time to load
  - If already loaded allow others to use the same copy
  - If no longer used becomes a candidate for garbage collection
- After it has no more references we should release it but how…?
  - DIY Ref counting
  - Weak References

Weak reference Singleton

- Single instance held by WeakReference
- GC will not keep object alive if no strong references

```
public class CachedEntity
{
      private static WeakReference _instance = new WeakReference(null);

      private CachedEntity() { // Load data  }

      public static CachedEntity GetInstance() {
          CachedEntity strongInstance = (CachedEntity)_instance.Target;

          if (strongInstance == null) {
              strongInstance = new CachedEntity();
              _instance = new WeakReference(strongInstance);
          }
          return strongInstance;
      }
}
```

Introducing the Generic Singleton

- Implementing Singleton mechanics many times tedious and error prone
- Consider building generic versions

```
public class Singleton<T> where T:class {
    private static T instance;

    public static T GetInstance()
    {
        . . .
        return instance;
    }
}
```

Using the Generic Singleton

```
public sealed class Highlander : Singleton<Highlander> {
   private Highlander()   {
     Console.WriteLine("Highlander created");
   }
}

class Program {
   static void Main(string[] args) {
     Highlander highlander = Highlander.GetInstance();
     Debug.Assert(object.ReferenceEquals(highlander,
                  Highlander.GetInstance()));

     // Will not compile
     // Highlander h2 = new Highlander();
    }
}
```

Thread safe lazy initialization

- Static constructors for lazy instantiation not always applicable
- What are the issues in this code?
  - Isn't this just the same as the CLR implementing lazy init?

```
public class Logger
{
    private static Logger instance = null;
    private static object initLock = new object();

    public static Logger GetInstance()
    {
        lock (initLock)
        {
            if (instance == null)
                instance = new Logger();
        }

        return instance;
    }
    ...
}
```

Optimised thread safe lazy initialization

- This technique is known as Double Check Locking
  - only pay the expense of synchronization when the object has not been created

```
public static Logger GetInstance()
{
   if (instance == null)
        CreateInstance();

    return instance;
}

private static void CreateInstance() {
       lock (initLock) {
           if (instance == null)
                instance = new Logger();
        }
}
```

A single instance per thread

- Instance shared across multiple threads may require synchronisation
  - For high contention object this can have an effect on throughput
  - can therefore be beneficial having one instance per thread
- Examine the logging example
  - multiple threads logging through a single stream will require synchronization
  - turn on high level of diagnostics and this could severely change behaviour

- WARNING...nothing to stop a reference being passed from one thread to another.

Thread scoped Singleton

```
public class ThreadedLogger
{
      [ThreadStatic]
      private static ThreadedLogger instance;

      private StreamWriter logStream;

      private ThreadedLogger() {
          logStream = new StreamWriter(String.Format("log{0}.txt",
                                  Thread.CurrentThread.ManagedThreadId));
      }

      public static ThreadedLogger GetInstance() {
          if (instance == null) {
              instance = new ThreadedLogger();
          }
          return instance;
      }
         ...
}
```

Marks storage a thread based

No need for any locking since
Storage is on a per thread basis

Summary

- When you need a single instance of an object use a singleton
- May use a static class
  - but this will result in a fragile solution
- Consider creating a Generic Singleton for each Singleton variant
- Don't overuse the singleton pattern
- Agree on standard names for Singleton variants
  - WeakSingleton
  - ThreadScopedSingleton