4. The `private _manager` object, marked `static`, is used inside a critical section (using `lock`) to make sure it is thread safe. Thread safety is very important here. Otherwise, two threads might simultaneously call `GetInstance()` and, on finding the `EmailManager instance (_manager)` null, will both try to create an instance, thereby creating two instances of the class. The `lock` keyword helps us make sure that once a thread enters the region, no other thread can do so until the first thread exits, making our code thread safe. We pass the `EmailManager`'s type in order to lock the statement using the `typeof` operator to define the scope of the lock statement.

An important point to note is that in the above code we have to make sure that the type used in the `typeof()` command is not publicly accessible, otherwise the scope would be affected. It is better to create a `private` object within our class to use as a reference object in the `lock` statement, as in:

```
private static object forLock = new object();
public static EmailManager GetInstance()
         {
                  // Use 'Lazy initialization'
                  if (_manager == null)
                  {
       //ensure thread safety using locks
       lock(typeof(forLock)
            {
                               _manager = new EmailManager();
                       }
      }
               return _manager;
         }
```

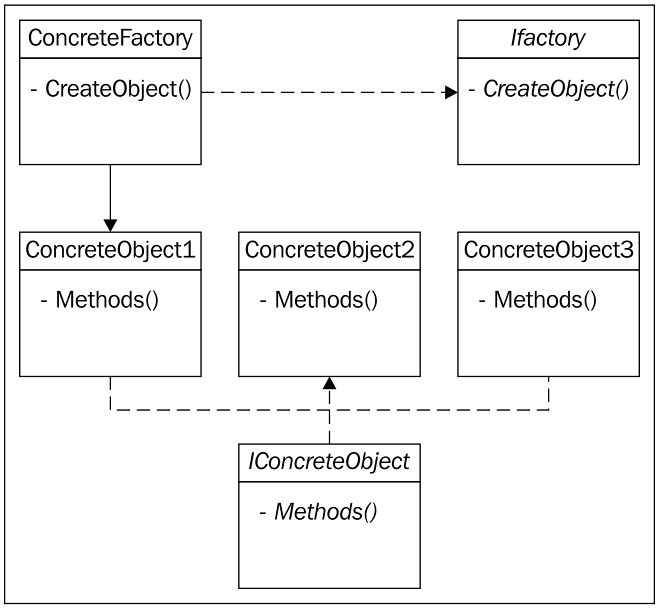
So the above code can be used for implementing a Singleton design pattern in ASP.NET effectively and safely. Now we will move to another famous design pattern—the Factory method.

Factory Method

The Factory design pattern is another heavily-used pattern in ASP.NET applications to help introduce loose coupling and remove dependencies in the code. The Factory method is a creational design pattern that is used to create objects without any prior knowledge of the type of the object. We delegate the responsibilities of creating the actual objects to subclasses or separate factory classes. This can be accomplished by using interfaces or abstract classes.

Why do we need this design pattern? Change is one thing that we cannot avoid in software development. No matter how strictly we jot down specifications of the software we are making, there will always be some changes in the future. And each such change might affect the application code base. So there is no way of avoiding changes, and unless our applications are designed to adapt to these changes, we will be spending more time and money on changing the application code each time a change is requested. The factory design helps us make our applications "change-friendly".

Here is a representative class diagram showing how a Factory pattern might be used in an object model:



We have `IConcreteObject`, an interface to be implemented by all of the `ConcreteObject` classes. We use this interface to abstract all methods.

We have three concrete object classes implementing the `IConcreteObject`, `ConcreteObject1`, `ConcreteObject2`, and `ConcreteObject3` interfaces.

Now, we have an interface named `Ifactory` and the `ConcreteFactory` class is implementing that interface and is responsible for creating the concrete objects (`ConcreteObject1`, `ConcreteObject2`, or `ConcreteObject3`). It is up to the concrete factory as to which class is to be instantiated.

So the responsibility for creating the object is delegated to the `ConcreteFactory` class. Let us first understand the basic Factory design , through the use of a simple practical example.