What is design Pattern in c#?

Design Patterns are categorized into 3 types and they are:

1. Creational Design Patterns.
2. Structural Design Patterns.
3. Behavioral Design Patterns.

What are Creational Design Patterns?

These patterns deal with the process of objects creation. The flowing are the different types of Creational Design patterns.

1. Abstract Factory Pattern - Createinstances of several classes belonging to different families.
2. Factory Pattern - Create instances ofderived classes.
3. Builder Pattern - Separates an objectconstruction from its representation.
4. Lazy Pattern - Create a duplicate objector clone of the object.
5. Prototype Pattern - Specify the kind ofobjects to create using a prototypical instance, and create new objects bycopying this prototype.
6. Singleton Pattern - Ensures that a classcan have only one instance.

What are Structural Design Patterns?

These patterns deal with the composition of objects structures. The flowing are the different types of Structural Design patterns.

1. Adapter Pattern - Interfaces of classesvary depending on the requirement.
2. Bridge Pattern - Class level abstractionis separated from its implementation.
3. Composite Pattern - Individual objects &a group of objects are treated similarly in this approach.
4. Decorator Pattern - Functionality isassigned to an object.
5. Facade Pattern - A common interface iscreated for a group of interfaces sharing a similarity.
6. Flyweight Pattern - The concept ofsharing a group of small sized objects.
7. Proxy Pattern - When an object iscomplex and needs to be shared, its copies are made. These copies are calledthe proxy objects.

What are Behavioral Design Patterns?

These patterns deal with the process of communication, managing relationships, and responsibilities between objects. The flowing are the different types of Behavioral Design patterns.

1. Chain Or Responsibilities Pattern - Inthis pattern, objects communicate with each other depending on logicaldecisions made by a class.
2. Command Pattern - In this pattern,objects encapsulate methods and the parameters passed to them.
3. Observer Pattern - Define a one-to-manydependency between objects so that when one object changes state, all itsdependents are notified and updated automatically.
4. Interpreter Pattern - A way to includelanguage elements in a program.

5. Iterator Pattern - Provide a way toaccess the elements of an aggregate object sequentially without exposing itsunderlying representation.
6. Mediator Pattern - Define an object thatencapsulates how a set of objects interact. In other words, it definessimplified communication between classes.
7. Memento Pattern - Without violatingencapsulation, capture and externalize an object's internal state so thatthe object can be restored to this state later.
8. State Pattern - Allow an object to alterits behavior when its internal state changes. The object will appear tochange its class.
9. Strategy Pattern - Define a family ofalgorithms, encapsulate each one, and make them interchangeable. Strategylets the algorithm vary independently from clients that use it.
10. Visitor Pattern - Defines a newoperation to a class without change.
11. Template Method Pattern - Defer theexact steps of an algorithm to a subclass.

Singleton Design Pattern In C#

Design patterns are the vital part of software design and architecture. Earlier, I wrote an article on basics of Design Patterns in C#. Now it it time to discuss Singleton Design Pattern in C#. If you're new to design patterns, I recommend reading this article: Design Patterns In C#.

What is Singleton Design Pattern?

1. Ensures a class has only one instance and provides a global point of access to it.
2. A singleton is a class that only allows a single instance of itself to be created, and usually gives simple access to that instance.
3. Most commonly, singletons don't allow any parameters to be specified when creating the instance, since a second request of an instance with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.)

There are various ways to implement the Singleton Pattern in C#. The following are the common characteristics of a Singleton Pattern.

- A single constructor, that is private and parameterless.
- The class is sealed.
- A static variable that holds a reference to the single created instance, if any.
- A public static means of getting the reference to the single created instance, creating one if necessary.

Advantages of Singleton Pattern

The advantages of a Singleton Pattern are:

1. Singleton pattern can be implemented interfaces.
2. It can be also inherit from other classes.
3. It can be lazy loaded.
4. It has Static Initialization.
5. It can be extended into a factory pattern.

6. It helps to hide dependencies.
7. It provides a single point of access to a particular instance, so it is easy to maintain.

Disadvantages of Singleton Pattern

The disadvantages of a Singleton Pattern are:

1. Unit testing is more difficult (because it introduces a global state into an application).
2. This pattern reduces the potential for parallelism within a program, because to access the singleton in a multi-threaded system, an object must be serialized (by locking).

Singleton class vs. Static methods

The following conpares Singleton class vs. Static methods:

1. A Static Class cannot be extended whereas a singleton class can be extended.
2. A Static Class can still have instances (unwanted instances) whereas a singleton class prevents it.
3. A Static Class cannot be initialized with a STATE (parameter), whereas a singleton class can be.
4. A Static class is loaded automatically by the CLR when the program or namespace containing the class is loaded.

How to Implement Singleton Pattern in your code

There are many way to implement a Singleton Pattern in C#.

1. No Thread Safe Singleton.
2. Thread-Safety Singleton.
3. Thread-Safety Singleton using Double-Check Locking.
4. Thread-Safe Singleton without using locks and no lazy instantiation.
5. Fully lazy instantiation.
6. Using .NET 4's Lazy<T> type.

1. No Thread Safe Singleton

Explanation of the following code:

1. The following code is not thread-safe.
2. Two different threads could both have evaluated the test (if instance == null) and found it to be true, then both creates instances, which violates the singleton pattern.
3. Note that in fact the instance may already have been created before the expression is evaluated, but the memory model doesn't guarantee that the new value of instance will be seen by other threads unless suitable memory barriers have been passed.

```
// Bad code! Do not use!
```

```
public sealed class Singleton
{
    //Private Constructor.
    private Singleton()
    {
    }
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

2. Thread Safety Singleton

Explanation of the following code:

1. This implementation is thread-safe.
2. In the following code, the thread is locked on a shared object and checks whether an instance has been created or not.
3. This takes care of the memory barrier issue and ensures that only one thread will create an instance.
4. For example: Since only one thread can be in that part of the code at a time, by the time the second thread enters it, the first thread will have created the instance, so the expression will evaluate to false.
5. The biggest problem with this is performance; performance suffers since a lock is required every time an instance is requested.

```
public sealed class Singleton
{
    Singleton()
    {
    }
    private static readonly object padlock = new object();
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
}
```

3. Thread Safety Singleton using Double Check Locking

Explanation of the following code:

1. In the following code, the thread is locked on a shared object and checks whether an instance has been created or not with double checking.

```
public sealed class Singleton
{
    Singleton()
    {
    }
    private static readonly object padlock = new object();
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (padlock)
                {
                    if (instance == null)
                    {
                        instance = new Singleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

4. Thread Safe Singleton without using locks and no lazy instantiation

Explanation of the following code:

1. The preceding implementation looks like very simple code.
2. This type of implementation has a static constructor, so it executes only once per Application Domain.
3. It is not as lazy as the other implementation.

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    // Explicit static constructor to tell C# compiler
    // not to mark type as beforefieldinit
    static Singleton()
    {
    }
    private Singleton()
    {
    }
    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }
}
```

5. Fully lazy instantiation

Explanation of the following code:

1. Here, instantiation is triggered by the first reference to the static member of the nested class, that only occurs in Instance.
2. This means the implementation is fully lazy, but has all the performance benefits of the previous ones.
3. Note that although nested classes have access to the enclosing class's private members, the reverse is not true, hence the need for instance to be internal here.
4. That doesn't raise any other problems, though, as the class itself is private.
5. The code is more complicated in order to make the instantiation lazy.

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    // Explicit static constructor to tell C# compiler
    // not to mark type as beforefieldinit
    static Singleton()
    {
    }
    private Singleton()
    {
    }
    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }
}
```

6. Using .NET 4's Lazy<T> type

Explanation of the following code:

1. If you're using .NET 4 (or higher) then you can use the System.Lazy<T> type to make the laziness really simple.
2. All you need to do is pass a delegate to the constructor that calls the Singleton constructor, which is done most easily with a lambda expression.
3. It also allows you to check whether or not the instance has been created with the IsValueCreated property.

```
public sealed class Singleton
{
    private Singleton()
    {
    }
    private static readonly Lazy<Singleton> lazy = new Lazy<Singleton>(() => new Singleton());
    public static Singleton Instance
    {
        get
        {
            return lazy.Value;
        }
    }
}
```

Example

The final example is here:

```
namespace Singleton
{
    class Program
    {
        static void Main(string[] args)
        {
            Calculate.Instance.ValueOne = 10.5;
            Calculate.Instance.ValueTwo = 5.5;
            Console.WriteLine("Addition : " + Calculate.Instance.Addition());
            Console.WriteLine("Subtraction : " + Calculate.Instance.Subtraction());
            Console.WriteLine("Multiplication : " + Calculate.Instance.Multiplication());

            Console.WriteLine("Division : " + Calculate.Instance.Division());
            Console.WriteLine("\n----------------------\n");
            Calculate.Instance.ValueTwo = 10.5;
            Console.WriteLine("Addition : " + Calculate.Instance.Addition());
            Console.WriteLine("Subtraction : " + Calculate.Instance.Subtraction());
            Console.WriteLine("Multiplication : " + Calculate.Instance.Multiplication());

            Console.WriteLine("Division : " + Calculate.Instance.Division());
            Console.ReadLine();
        }
    }
    public sealed class Calculate
    {
        private Calculate()
        {
        }
        private static Calculate instance = null;
        public static Calculate Instance
        {
            get
            {
                if (instance == null)
                {
                    instance = new Calculate();
                }
                return instance;
            }
        }
        public double ValueOne { get; set; }
        public double ValueTwo { get; set; }
        public double Addition()
        {
            return ValueOne + ValueTwo;
        }
        public double Subtraction()
        {
            return ValueOne - ValueTwo;
        }
        public double Multiplication()
        {
            return ValueOne * ValueTwo;
        }
        public double Division()
        {
            return ValueOne / ValueTwo;
        }
    }
}
```

How to upload and download files in ASP.NET Core MVC.

Solution

In an empty project, update the *Startup* class to add services and middleware for MVC.

```
public void ConfigureServices(
        IServiceCollection services)
    {
        services.AddSingleton<IFileProvider>(
            new PhysicalFileProvider(
                Path.Combine(Directory.GetCurrentDirectory(), "wwwroot"
)));

        services.AddMvc();
    }

          public void Configure(
              IApplicationBuilder app,
              IHostingEnvironment env)
          {
              app.UseMvc(routes =>
              {
                  routes.MapRoute(
                      name: "default",
                      template: "{controller=Home}/{action=Index}/{id?}
");
              });
          }
```

Add a Controller and action methods to upload and download the file.

```
[HttpPost]
  public async Task<IActionResult> UploadFile(IFormFile file)
  {
      if (file == null || file.Length == 0)
          return Content("file not selected");

      var path = Path.Combine(
                  Directory.GetCurrentDirectory(), "wwwroot",
                  file.GetFilename());

            using (var stream = new FileStream(path, FileMode.Create))

            {
                await file.CopyToAsync(stream);
            }

            return RedirectToAction("Files");
        }

        public async Task<IActionResult> Download(string filename)
        {
            if (filename == null)
                return Content("filename not present");

            var path = Path.Combine(
                           Directory.GetCurrentDirectory(),
                           "wwwroot", filename);

            var memory = new MemoryStream();
            using (var stream = new FileStream(path, FileMode.Open))
            {
                await stream.CopyToAsync(memory);
            }
            memory.Position = 0;
            return File(memory, GetContentType(path), Path.GetFileName(
path));
        }
```

Add a Razor page with HTML form to upload a file.

```
<form asp-controller="Home" asp-action="UploadFile" method="post"
      enctype="multipart/form-data">

    <input type="file" name="file" />
    <button type="submit">Upload File</button>
</form>
```

Discussion Uploading

ASP.NET Core MVC model binding provides *IFormFile* interface to upload one or more files. The HTML form must have the *encoding* type set to *multipart/form-data* and an *input* element with *type*attribute set to *file*.

You could also upload multiple files by receiving a list of *IFormFile* in action method and setting *input* element with *multiple* attribute.

```
// In Controller
[HttpPost]
public async Task<IActionResult> UploadFiles(List<IFormFile> files)

// In HTML
<input type="file" name="files" multiple />
```

You could also have *IFormFile* as a property on model received by the action method.

```
public class FileInputModel
 {
     public IFormFile FileToUpload { get; set; }
 }
 [HttpPost]
 public async Task<IActionResult> UploadFileViaModel(FileInputModel mod
el)
```

Note

The name on input elements must match action parameter name (or model property name) for model binding to work. This is no different than model binding of simple and complex types.

Downloading

Action method needs to return *FileResult* with either a *stream*, *byte[],* or virtual path of the file. You will also need to know the *content-type* of the file being downloaded. Here is a sample (quick/dirty) utility method.

```
      private string GetContentType(string path)
         {
             var types = GetMimeTypes();
             var ext = Path.GetExtension(path).ToLowerInvariant();
             return types[ext];
         }

         private Dictionary<string, string> GetMimeTypes()
         {
                   return new Dictionary<string, string>
                   {
                       {".txt", "text/plain"},
                       {".pdf", "application/pdf"},
                       {".doc", "application/vnd.ms-word"},
                       {".docx", "application/vnd.ms-word"},
                       {".xls", "application/vnd.ms-excel"},
                       {".xlsx", "application/vnd.openxmlformats
       officedocument.spreadsheetml.sheet"},
                       {".png", "image/png"},
                       {".jpg", "image/jpeg"},
                       {".jpeg", "image/jpeg"},
                       {".gif", "image/gif"},
                       {".csv", "text/csv"}
                   };
               }
```