C# Notes

**================Object Creation in C#================**

**public class MyClass**

**{**

**private int field1 = 10;** *// Field initialized directly*

**public int Property1 { get; set; } = 20;** // Property initialized with default value

**public string Property2 { get; set; }**

**public MyClass()**

**{**

**Property2 = "Initialized in constructor";** // Property initialized **in constructor**

**}**

**}**

* **Initialization Sequence**:
  1. **Memory Allocation**: Memory is allocated for an instance of MyClass. In heap, while variable in stack as a reference
  2. **Field Initialization**: field1 is initialized to 10.
  3. **Property Initialization**: Property1 is initialized to 20 using property initializer.
  4. **Constructor Initialization**: Constructor sets the value of Property2 to "Initialized in constructor".
  5. **Constructor also calls any base class constructor before itself.**

**================Early vs Late Binding================**

**EARLY**

It recognizes and checks the [methods](https://www.geeksforgeeks.org/c-methods/), or [properties](https://www.geeksforgeeks.org/c-properties/)during compile time. In this binding, the compiler already knows about what kind of object it is and what are the methods or properties it holds,

The performance of early binding is fast and it is easy to code.

Geeks g = new Geeks();

**// Calling the method of Geeks class**

        g.details("Ankita", "C#");

**// Calling "mymethod()" gives error**

**// because this method does not**

**// belong to class Geeks or compiler**

**// does not know mymethod() at compile time**

        g.mymethod();

**LATE**

the compiler does not know about what kind of object it is and what are the methods or properties it holds, here the objects are dynamic objects. **The performance of late binding is slower than early binding because it requires lookups at run-time.**

  dynamic obj = 4;

        dynamic obj1 = 5.678;

**// Display the type of objects**

        Console.WriteLine("The type of the objects are :");

**// GetType() method is**

**// used to get the type**

        Console.WriteLine(obj.GetType());

        Console.WriteLine(obj1.GetType());

#### 

#### 

#### Impact on performance, maintainability, and extensibility

**Performance:**

* Early binding has better performance since the method or property calls are resolved at compile time, and the runtime does not need to perform any additional lookups or type checks.
* Late binding has a performance overhead because the runtime must perform additional work to determine the method or property to be called
* **Maintainability:**
* Early binding results in more maintainable code because the types and method signatures are known at compile time, enabling better error detection, type safety, and IntelliSense support in IDEs. This makes it easier to identify and fix issues during development.
* Late binding can lead to less maintainable code because errors may not be detected until runtime, making it harder to identify and fix issues.

**================REFLECTION================**

Using reflection, you can, for instance, load a class dynamically from an Assembly, test whether a given type has a specific member, and even create code dynamically.

### ***Why Is Reflection Used?***

You need to use Reflection when you want to inspect the contents of an assembly. For example, you can get all members of the object by typing “.” before an object when viewing your Visual Studio editor [IntelliSense.](https://code.visualstudio.com/docs/editor/intellisense)

### ***Is Reflection In C# Slow?***

Generally speaking, code that utilizes reflection is slower than code written in a conventional manner.

But is this a problem? Most of the time, it shouldn’t be. Reflection should be employed when the problem cannot be solved using traditional methods. Consequently, if you use reflection only in specific scenarios, your code should remain performant.

When you write a C# program that uses reflection, **you can use either the TypeOf operator or the GetType() method to get the object’s type.**

## ***A Simple Use Case of C# Reflection***

Reflection can be used to create applications called type browsers which allow users to select types and then read the data provided about them. [This example](https://msdn.microsoft.com/en-us/library/ms173183(v=vs.90).aspx) illustrates how to use the static method GetType to find the Type of a variable:

**// Using GetType to obtain type information:**

int i = 42;

System.Type type = i.GetType();

System.Console.WriteLine(type);

The above example results in the following output: System.Int32

### **Examples of Reflection in C#**

[Implementing reflection in C#](http://www.csharp-examples.net/reflection-examples/) requires a two-step process. You first get the “type” object, then use the type to browse members such as “methods” and “properties.”

This is how you would create instances of DateTime class from the system assembly:

// create instance of class DateTime

DateTime dateTime = (DateTime)Activator.CreateInstance(typeof(DateTime));

To access the sample class Calculator from Test.dll assembly, the Calculator class should be defined as the following:

namespace Test

{

public class Calculator

{

public Calculator() { ... }

private double \_number;

public double Number { get { ... } set { ... } }

public void Clear() { ... }

private void DoClear() { ... }

public double Add(double number) { ... }

public static double Pi { ... }

public static double GetPi() { ... }

}

}

Then, you can use reflection to load the Test.dll assembly:

// dynamically load assembly from file Test.dll

Assembly testAssembly = Assembly.LoadFile(@"c:\Test.dll");

To create an instance of the calculator class:

**// get type of class Calculator from just loaded assembly**

Type calcType = testAssembly.GetType("Test.Calculator");

**// create instance of class Calculator**

object calcInstance = Activator.CreateInstance(calcType);

And access its members (the following examples illustrate getting values for the public double Number property):

// get info about property: public double Number

PropertyInfo numberPropertyInfo = calcType.GetProperty("Number");

// get value of property: public double Number

double value = (double)numberPropertyInfo.GetValue(calcInstance, null);

// set value of property: public double Number

numberPropertyInfo.SetValue(calcInstance, 10.0, null);

### **How Reflection in C# Works**

The main class for reflection is the System.Type class, which is a partial abstract class representing a type in the [Common Type System](https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system) (CTS). When you use this class, you can find the types used in a module and namespace and also determine if a given type is a reference or value type. You can parse the corresponding metadata tables to look through these items

#### Situations where reflection is beneficial or necessary

1. **Plugin architectures**: Reflection allows you to load external types and create instances of those types at runtime, which is essential for implementing plugin architectures, where the types and their implementations are not known at compile time.

Assembly pluginAssembly = Assembly.LoadFrom("Plugin.dll");

Type pluginType = pluginAssembly.GetType("PluginNamespace.PluginClass");

object pluginInstance = Activator.CreateInstance(pluginType);

1. **Dependency Injection**: Reflection can be used to create instances of types and inject dependencies, enabling the use of inversion of control (IoC) and dependency injection (DI) patterns.

public class Container

{

private Dictionary<Type, Type> \_registrations = new Dictionary<Type, Type>();

public void Register<TInterface, TImplementation>()

{

\_registrations[typeof(TInterface)] = typeof(TImplementation);

}

public TInterface Resolve<TInterface>()

{

Type implementationType = \_registrations[typeof(TInterface)];

return (TInterface)Activator.CreateInstance(implementationType);

}

}

1. **Late binding**: Reflection enables you to call methods, access properties, and manipulate fields at runtime, even if their names and types are not known at compile time. This can be useful in scenarios where the types must be loaded dynamically or when working with third-party libraries that may change their APIs.

Type targetType = typeof(TargetClass);

MethodInfo targetMethod = targetType.GetMethod("MethodName");

object targetInstance = Activator.CreateInstance(targetType);

targetMethod.Invoke(targetInstance, new object[] { /\* arguments \*/ });

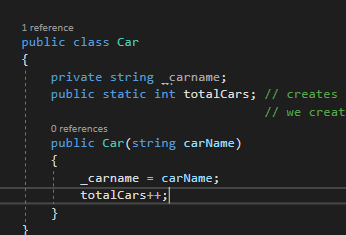
**================Static================**

**Why we need Static ?**

 A static class can’t be instantiated. A static member has only one copy of the member, regardless of the number of instances.  If multiple instances of a class are created, the last updated value of a static member will be available to all instances.

**Static Constructor**

* Static constructors can't be parameterized.
* Static Constructor is used to initializing static data members of the class.



Car names will be new always but totalCount will have only one memory location and will not be initialized with every obj of Car.

**Usage 1**

//Static Example

Car car1 = new Car("Swift");

Car car2 = new Car("BMW");

// we creaated 2 obj so it will create 2 obj but each obj will have same memory location

// for totalCars var in Car class as it is static

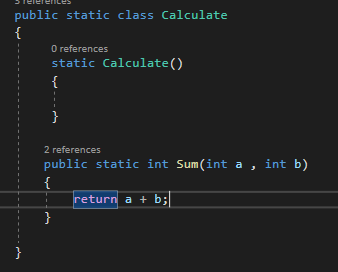
int totalCars = Car.totalCars;

Console.WriteLine(totalCars);

To call static member we need ClassName.property directly without creating an Object.

**Usage 2 –**

We can create a Support class just like **Math** class in c#. and use it again and again..





It will always give new result and just like Math class we made this Calculate class.

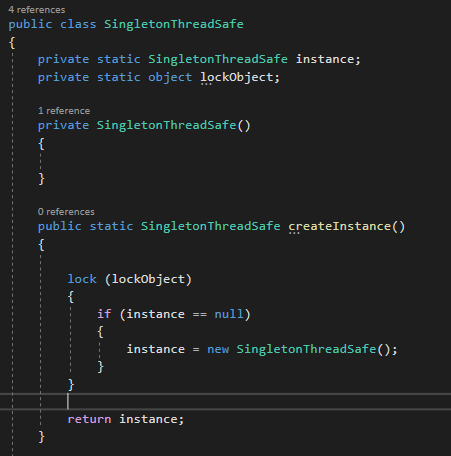
The C# Math class is static because it contains methods that perform mathematical operations without needing to store or retrieve data that is unique to a particular instance of the class.

**================SINGLETON PATTERN================**

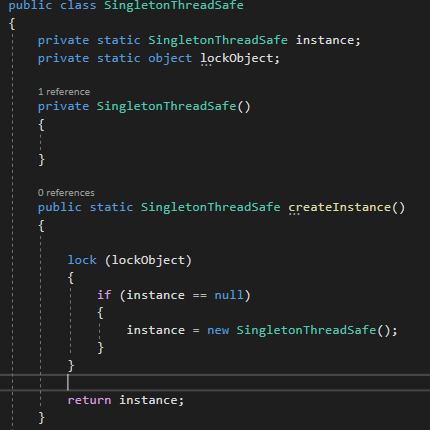
A singleton is a class that allows only a single instance of itself to be created and usually gives simple access to that instance.

Typically a requirement of singletons is that they are created lazily - i.e. that the instance isn't created until it is first needed.

**Basic singleton**



**Thread safe Singleton**



Unfortunately, performance suffers as a lock is acquired every time the instance is requested.

Note that **instead of locking on typeof(Singleton**) as some versions of this implementation do**, I lock on the value of a static variable which is private to the class.** *Locking on objects which other classes can access and lock on (such as the type) risks performance issues and even deadlocks*. Wherever possible, only lock on objects specifically created for the purpose of locking, or which document that they are to be locked on for specific purposes (e.g. for waiting/pulsing a queue). Usually such objects should be private to the class they are used in. This helps to make writing thread-safe applications significantly easier.

**Lock -** the lock keyword is used to manage access to a block of code that can be executed by multiple threads at the same time. It's a way to ensure that only one thread can access a particular code block at a time, preventing a race condition.

**USAGE**

* + - 1. Can be used for classes that require one obj and do a specific job only. **Like FileLogger**

## **Singleton class vs Static methods**

The following compares Singleton class vs. Static methods,

1. A Static Class cannot be extended whereas a singleton class can be extended. It can implement interfaces, while Static class cant .
2. A Static Class cannot be initialized whereas a singleton class can be.
3. A Static class is loaded automatically by the CLR when the program containing the class is loaded.
4. Static is a keyword and Singleton is a pattern.
5. Singleton uses Static internally but not vice versa

**The advantages of a Singleton Pattern are,**

1. Singleton pattern can implement interfaces.
2. Can be lazy-loaded and has Static Initialization.
3. It helps to hide dependencies.
4. It provides a single point of access to a particular instance, so it is easy to maintain.

## **Disadvantages of Singleton Design Pattern**

The disadvantages of a Singleton Pattern are,

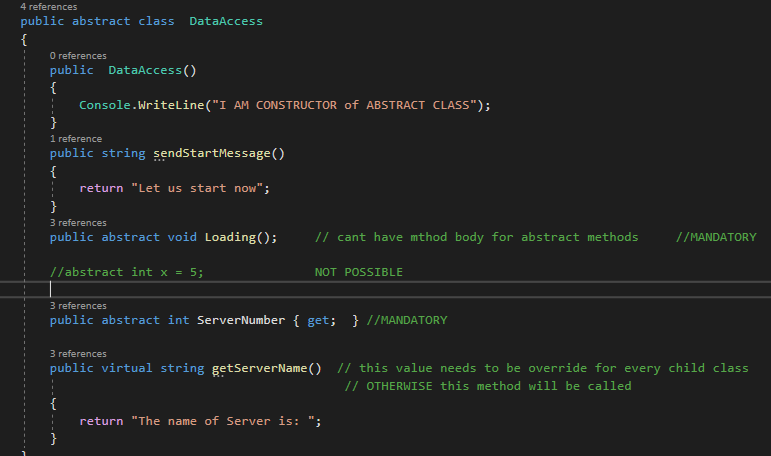
1. Unit testing is a bit difficult as it introduces a global state into an application
2. Reduces the potential for parallelism within a program by locking.

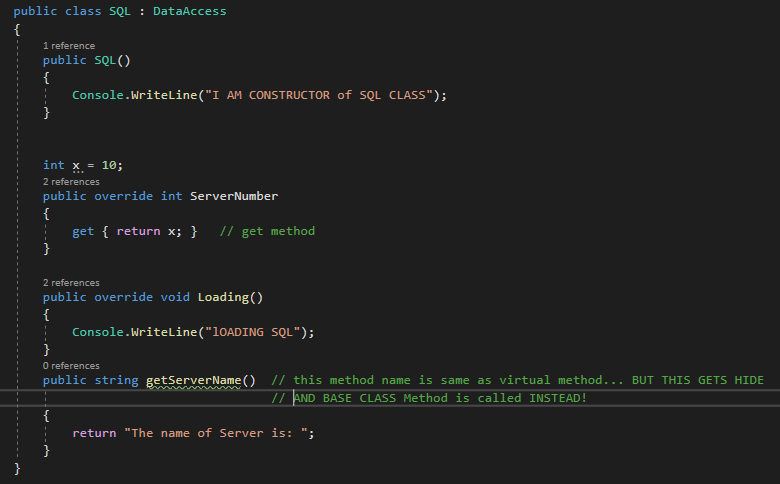
**================ABSTRACT======================**

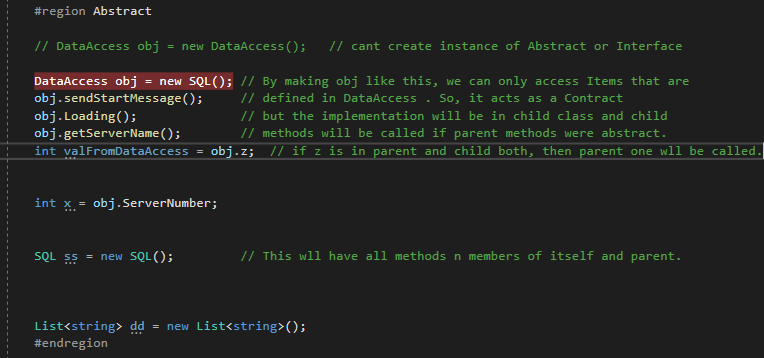
**-Abstract** means that something is missing in that class or method or property. So, if we put **abstract**  in method, they cant have method body. It cant be used on variables like int or string….BUT CAN BE USED on properties. **Virtual** keyword can also be used on properties and methods only.

**-**It is Blend of **INTERFACE and Normal Class**

-Abstract keyword is opposite of sealed access modifier. If we have sealed on class then we cant make that class abstract… bcz Abstract wants you to implement it.







***--An abstract class has a protected constructor (by default) allowing derived types to initialize it.***

***--Constructor of Abstract class is called first then Base class!*** *Its same for normal classes also.*

**WHY CONSTRUCTOR IN ABSTRACT CLASS but not in Interface???**

In abstract class, we have an instance variable, abstract methods, and non-abstract methods. We need to initialize the non-abstract methods and instance variables, therefore abstract classes have a constructor. An instance variable is a variable which is declared in a class but outside of constructors, methods, or blocks

-NO OBJECT directly or Indirectly is created of ABSTRACT CLASS!!!

In-Depth Analysis

* **Object Creation**:
  + An object is created when you instantiate a class using the new keyword. For example:

**MyClass obj = new MyClass();**

Here, new MyClass() triggers the creation of an object of type MyClass on the heap.

* **Constructor Invocation**:
  + Once the object is created, its constructor is invoked.
  + Constructors are responsible for initializing the newly created object. They can initialize fields, properties, and perform any necessary setup.
  + In C#, constructors are automatically called when an object is instantiated using new.
* **Sequence of Events**:
* Memory allocation on the heap occurs first, where space is reserved for the object.
* All fields that are outside methods or constructor are initialized.
* The constructor is then called to initialize this allocated memory.
* All fields and properties are initialized according to the logic provided in the constructor.
* When you instantiate a concrete class that derives from an abstract class, memory is allocated on the heap for the derived class instance.
* During this process, the constructor chain is invoked. This means that the constructor of the abstract class, if accessible, can be invoked by the constructor of the derived class to initialize any members defined in the abstract class.

**\*IMPORTANT**

**WHEN WE WRITE ->** AbstractClass obj = new DerviedClass();

IInterface obj = new DerivedClass();

Then obj will only refer to methods or properties or fields that are in Abstract class only and not in DerviedClass… if DerivedClass has a implementation of something that was also in AbstractClass then only DerivedClass implementation will be called.

**ADVANTAGE-**

* 1. Does not let us create Obj of that class just like Interface. So no one can directly call any method of it from outside as Abstract class has normal methods also apart from Abstract methods.

**++I , i++ ??**

**Why to use get inside a methiid?**

**What is get ; set;**

**What is IApp = new Facebook();**

**List vs Ilist**

**Singleton Pattern**

**https://www.youtube.com/watch?v=r6Y0SmbufmU**