**Fundamental concepts of Object Oriented Programming.**

* **Encapsulation** – Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.

**Public** – Visible in the current and referencing assembly.

**Private** – Visible inside the current class.

**Protected** – Visible inside the current as well as in the derived class.

**Internal** – Visible inside the containing assembly.

**Internal protected** – Visible inside the containing assembly, also in the descendent of the current class.

* **Abstract class** – An abstract class is a special type of class that cannot be instantiated. An abstract class is designed to be inherited by subclasses that either implement or override its methods.

It is a class whose object cannot be instantiated. The class can only be inherited. It should contain at least one method abstract. It is denoted by the keyword *abstract.*

1. **Public abstract** **class** NonAbstractMethod
2. {
3. **public** **string** getDcn()
4. {
5. **return** "PS20100301A0012";
6. }
7. **public** **abstract** **void** getSeqID();
8. }
9. **Public class** Utilize : NonAbstractMethod
10. {
11. **public** **override** **void** getSeqID()
12. {
13. //Method Logic
14. }
15. }

* **Inheritance** –Inheritance is the ability to create a new class with attributes and behaviours from an existing class. The newly created class is the derived (or child) class and the existing class is the base (or parent) class.

public class Animal

{

    public void See()

    {

        Console.WriteLine("All animals can see");

    }

  public void Hear()

    {

        Console.WriteLine("All animals have ears");

    }

}

public class Cat : Animal

{

    public void Talk()

    {

        Console.WriteLine("A cat purrs");

    }

}

public static void Main(string[] args)

    {

        Cat obj = new Cat();

        obj.Hear();

obj.See();

     }

* **Polymorphism** – The name means, one name, many forms. It is achieved by having multiple methods with the same name but different implementations.

**Overloading** (Compile Time Polymorphism / Static Binding / Early Binding)

Public void show\_method(int a, int b)

{

//do something

}

Public void show\_method(string str1, string str2)

{

//do something

}

**Overriding** (Runtime Time Polymorphism / Dynamic Binding / Late Binding) 

Public class Net

{

public virtual void Act()

{

Console.WriteLine(‘’Net Act");

}

}

Public class Perl : Net

{

public override void Act()

{

Console.WriteLine("Perl Act");

}

}

* **Interface** – An Interface is a class with no implementation. The only thing that it contains is the declaration of methods, properties, and events.

Public interface IZoo

{

Public void See();

Public string Hear();

}

public abstract class Animal : IZoo

{

    public abstract void Talk()

    {

        Console.WriteLine("All animals can talk");

    }

    public void See()

    {

        Console.WriteLine("All animals can see");

    }

  public void Hear()

    {

        Console.WriteLine("All animals have ears");

    }

}

* **Partial class** – C# provides the ability to have a single class implementation in multiple files using the partial modifier [keyword](https://www.tutorialsteacher.com/csharp/csharp-keywords).

**classfile1.cs**

public partial class MyPartialClass

{

public void Method1(int val)

{

Console.WriteLine(val);

}

}

**classfile2.cs**

public partial class MyPartialClass

{

public void Method2(int val)

{

Console.WriteLine(val);

}

}

* **Sealed class** – Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, this class cannot be inherited. To access the member of sealed class, we need to create the object of class.

sealed class SealedClass

{

    public int Add(int a, int b)

    {

        return a + b;

    }

}

Public static void Main(string[] args)

    {

        SealedClass slc = new SealedClass();

        int total = slc.Add(6, 4);

        Console.WriteLine("Total = " + total.ToString());

    }

* **Static class -** A static class can only contain static data members, static methods, and a static constructor. It is not allowed to create objects of the static class. Static classes are sealed by default; means it does not allow inheritance.

Public static class Author

{

    public static string Name = "Bappaditya";

    public static void Details()

    {

        Console.WriteLine("The details of Author is:");

    }

}

public static void Main(String[] args)

{

        Console.WriteLine(Author.Name);

        Author.Details();

     }

-------------------------------------------------------

Public class Program

{

static void MethodA()

{

Console.WriteLine("Static method");

}

void MethodB()

{

Console.WriteLine("Instance method");

}

}

Public static void Main()

{

Program.MethodA();

Program programInstance = new Program();

programInstance.MethodB();

}

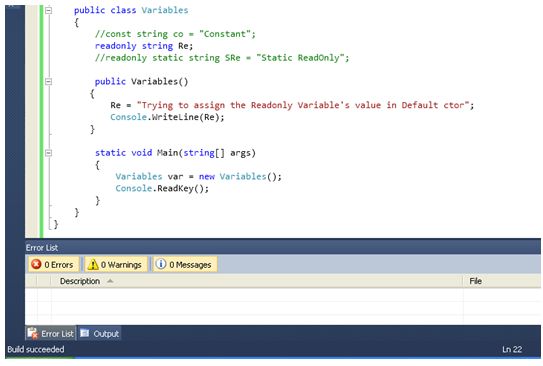
* **Difference between static, const and readonly**

**Const**  
Constant fields are defined at the time of declaration in the code snippet, because once they are defined they can't be modified. By default, a constant is static, so you can't define them static from your side.

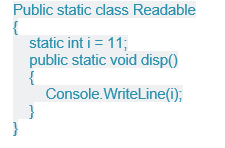




**Readonly**A Readonly field can be initialized either at the time of declaration or within the constructor of the same class. We can also change the value of a Readonly at runtime or assign a value to it at runtime (but in a non-static constructor only).



**Static**  
The static keyword is used to declare a static member. Static field can be initialized at the time of declaration or anywhere within the class.



* **Difference between ref and out**

**Ref Keyword**While using the Ref keyword we need to initialize the parameter before passing to metod.

int x=10;

Foo(ref int x); // OK

int y;

Foo(ref int y); // Error: y should be initialized before calling the method

1. **int** intRef = 2;
2. **private** **static** **int** MethodCall(**ref** **int** intRef)
3. {
4. **return** intRef;
5. }

**Out Keyword**  
  
While using Out keyword we need to initialize the parameter before returning from the calling method.

1. **int** intOut;
2. **private** **static** **int** MethodCall(**out** **int** intOut)
3. {
4. **return** intOut = 3;
5. }

* **How to restrict a method to be inherited?**

Making the method Static method.

* **What is the difference between Continue and Break Statement**

Break statement breaks the loop. It makes the control of the program to exit from the loop. Continue statement makes the control of the program to exit only the current iteration. It does not break the loop.

* **Where do we use interface and abstract class?**

C# does not support multiple inheritances, interfaces are mainly used to implement the multiple inheritances. As a class can implement more than one interface and only inherit from one abstract class. An interface is mainly used only when we do not require the implementation of methods or functionalities.

|  |  |
| --- | --- |
| **ABSTRACT CLASS** | **INTERFACE** |
| It contains both declaration and definition part. | It contains only a declaration part. |
| Multiple inheritance is not achieved by abstract class. | Multiple inheritance is achieved by interface. |
| It contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). | It does not contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). |
| It can contain static members. | It does not contain static members. |
| It can contain different types of access modifiers like public, private, protected etc. | It only contains public access modifier because everything in the interface is public. |
| The performance of an abstract class is fast. | The performance of interface is slow because it requires time to search actual method in the corresponding class. |
| A class can only use one abstract class. | A class can use multiple interface. |
| Abstract class can contain methods, fields, constants, etc. | Interface can only contain methods. |

| **DISPOSE** | **FINALIZE** |
| --- | --- |
| The method dispose( ) is defined in the interface IDisposable interface. | The method finalize( ) id defined in java.lang.object class. |
| public void dispose( ){ // dispose code here } | protected void finalize( ){ // finalization code here } |
| The method dispose( ) is invoked by the user. | The method finalize( ) is invoked by the garbage collector. |
| Method dispose( ) is used to free unmanaged resources whenever it is invoked. | Method finalize( ) is used to free unmanaged resources before the object is destroyed. |
| The method dispose( ) is to be implemented whenever there is a close( ) method. | The method finalize( ) is to be implemented for unmanaged resources. |
| The method dispose( ) is declared as public. | The method finalize( ) is declared as private. |
| The method dispose( ) is faster and instantly disposes an object. | The method finalize is slower as compared to dispose. |

* **Define ABC concept in WCF**

**Address:**Where exactly the service hosted? In other words, what is the location of the service? An address could be an IP Address, server name, URL and so on.

**Binding:**It is about how the messages are handled in the service side and the client side.

**Contract**: The contract is an agreement between the client and the server about what the structure and content of messages being exchanged.

* An **address** that indicates where the endpoint can be found / service hosted.
* A **binding** that specifies how a client can communicate with the endpoint.
* A **contract** is just an interface between client and server where client and server communicate each other.