

PARTIAL CLASS

- Many developers need access to the same class, then having the class in multiple files can be beneficial.
- The **partial** keywords allow a class to span multiple source files.
- A partial type must have the same accessibility.
- If the partial type is sealed or abstract then the entire class will be sealed and abstract.

Advantages

- Multiple developers can work simultaneously in the same class in different files.
- When you were working with automatically generated code, the code can be added to the class without having to recreate the source file like in Visual studio.
- You can also maintain your application in an efficient manner by compressing large classes into small ones.
- With the help of a partial class concept, you can split the UI of the design code and the business logic code to read and understand the code.

EXAMPLE

```
public partial class Employee
{
    public void DoWork()
    {
    }
}
```

```
public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

STATIC CLASS

- A static class is declared using the "static" keyword.
- If the class is declared as static then the compiler never creates an instance of the class.
- All the member fields, properties and functions must be declared as **static**
- They are accessed by the class name directly not by a class instance

```
public static class staticDemo
{
    //static fields
    public static int a=10,b=15,sum;

    //static method
    public static void Add()
    {
        sum =a+b;
    }
}

//function calling directly
    staticDemo.Add();
```

ABSTRACT CLASS

- C# allows both classes and functions to be declared abstract using the **abstract** keyword.
- We can't create an instance of an abstract class.
- An abstract member has a signature but no function body and they must be overridden in any non-abstract derived class.//abstract class
- It may contains non-abstract member.


```
public abstract class Employess
{
    //abstract method with no implementation
    public abstract void displayData();
}

//derived class
public class DE : Employess
{
    //abstract class method implementation
    public override void displayData()
    {
        Console.WriteLine("Abstract class method");
    }
}
```

SEALED CLASS

- Sealed classes cannot be inherited.
- You can create an instance of a sealed class.
- A sealed class is used to prevent further refinement through inheritance.


```
sealed class SealedClass
```

```
{
```

```
    void myfunv();
```

```
}
```

```
public class test :SealedClass//wrong. will give compilation error
```

```
{
```

```
}
```

METHOD TYPES

- **static:** accessible through class name. not the object or instance.
- **virtual:** Method may be overridden in derived class.
- **abstract:** Method must be overridden in non-abstract derived class(permitted only on abstract class).
- **override:** Method overrides a base class method.

Inheritance

- It is the process by which objects of one class acquire properties of objects of another class.
- Inheritance describes the ability to create new classes based on an existing class.
- Inheritance is implemented by colon(:).

Inheritance Example

```
public class Person
{
    public string FirstName
    {
        get;
        set;
    }
    public string FirstName
    {
        get;
        set;
    }
}
```

```
public class Employee : Person
{
    public string Email
    {
        get;
        set;
    }
}
```

- // "Employee" class, now contains all properties of "Person" class
- // the "FirstName" and "LastName" properties in "Person" class are automatically carried into this class.

```
Employee e = new Employee();
```

```
e.FirstName = "Ram";
```

```
e.LastName = "Shreshtha";
```

```
e.Email = "test@test.com";
```

```
//The FirstName and LastName are properties of Person but can use in Employee object.
```


POLYMORPHISM

- When a message can be processed in different ways is called polymorphism.
- Simply it means one name ,multiple forms.
- It allows to invoke methods of derived class through base class reference during runtime.
- It has the ability for classes to provide different implementations of methods that are called through the same name.
- Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

Polymorphism Is Of Two Types:

- I. Compile time polymorphism/Overloading
- II. Runtime polymorphism/Overriding

Compile Time Polymorphism

Compile time polymorphism is method and operators overloading. It is also called early binding.

In method overloading, method performs the different task at the different input parameters.

CONTD...

Runtime Time Polymorphism

- Runtime time polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, we change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.

EXAMPLE OF METHOD OVERLOADING (COMPILE TIME POLYMORPHISM)

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a+b;
    }
    public int Add(int a, int b, int c)
    {
        return a+b+c;
    }
}
```

CONTD...

```
Calculator c = new Calculator();
```

```
int sum = c.Add(5,6);
```


```
int sum2=c.Add(7,8,9);
```

EXAMPLE OF OVERRIDING

```
public class Parent
{
    public virtual string Meth1()
    {
        return "MyBase-Meth1";
    }
    public virtual string Meth2()
    {
        return "MyBase-Meth2";
    }
}
```


CONTD...

```
public class Child : Parent
{
    public override string Meth1()
    {
        return "MyDerived-Meth1";
    }
    public new string Meth2()
    {
        return "MyDerived-Meth2";
    }
}
```



CONTD...

```
Child c = new Child();
```

```
Parent p = (Parent) c;
```

```
MessageBox.Show(p.Meth1());
```

```
MessageBox.Show(p.Meth2());
```

OUTPUT

- MyDerived-Meth I
- MyBase-Meth I

Interfaces

- An interface is similar to a class, but it provides a specification rather than an implementation for its members.
- An interface declaration is like a class declaration, but it provides no implementation for its members.
- Interface members are by default **abstract** and **public**.
- A class (or struct) can implement multiple interfaces. In contrast, a class can inherit from only a single class.
- These members will be implemented by the classes and structs that implement the interface.
- An interface can contain only methods, properties, events, and indexers.
- An interface cannot contain a constructor (as it cannot be used to create objects)

CONTD...

```
using System;
public interface IShape
{
    void draw();
}
public class Rectangle : IShape
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
```

CONTD...

```
public class Circle : IShape
{
    public void draw()
    {
        Console.WriteLine("drawing circle");
    }
}
```


CONTD...

```
class Program
{
    static void Main(string[] args)
    {
        IShape shape;

        shape = new Rectangle(); // Create a object
        shape.draw();           // drawing rectangle

        shape = new Circle();

        shape.draw();           // drawing circle
    }
}
```

Why And When To Use Interfaces

- To achieve security - hide certain details and only show the important details of an object (interface).
- C# does not support "multiple inheritance" (a class can only inherit from one base class).
- However, it can be achieved with interfaces, because the class can implement multiple interfaces.
- Note: To implement multiple interfaces, separate them with a comma .

```
interface IFirstInterface
{
    void myMethod();// interface method
}
interface ISecondInterface
{
    void myOtherMethod();// interface method
}
class DemoClass : IFirstInterface, ISecondInterface // Implement multiple interfaces
{
    public void myMethod()
    {
        Console.WriteLine("Some text..");
    }
    public void myOtherMethod()
    {
        Console.WriteLine("Some other text...");
    }
}
```

CONTD...

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        DemoClass myObj = new DemoClass();
```

```
        myObj.myMethod();
```

```
        myObj.myOtherMethod();
```

```
    }
```

```
}
```

Enums

- An **enum** is a special value type that lets you specify a group of named numeric constants.
- To define an enumeration type, use the **enum** keyword and specify the names of enum members:
- `public enum BorderSide { Left, Right, Top, Bottom }`

We can use this **enum** type as follows:

```
BorderSide topSide = BorderSide.Top;
```

```
bool isTop = (topSide == BorderSide.Top); // true
```

CONTD...

Enum Conversions

- You can convert an enum instance to and from its underlying integral value with an explicit cast:
- `int i = (int) BorderSide.Left;`
- `BorderSide side = (BorderSide) i;`

Nested Types

A nested type is declared within the scope of another type.

```
public class TopLevel
{
    public class Nested { } // Nested class
    public enum Color { Red, Blue, Tan } // Nested enum
}
```


Generics

- C# has two separate mechanisms for writing code that is reusable across different types: **inheritance** and **generics**.
- Whereas inheritance expresses reusability with a base type, generics express reusability with a “template” that contains “placeholder” types.
- Generics, when compared to inheritance, can increase type safety and reduce casting and boxing

Generic Types

- A generic type declares type parameters—placeholder types to be filled in by the consumer of the generic type, which supplies the type arguments.
- Here is a generic type `Stack<T>`, designed to stack instances of type `T`. `Stack<T>` declares a single type parameter `T`:

CONTD...

- Generics exist to write code that is reusable across different types.

```
class DataStore<T>
```

```
{
```

```
    public T Data { get; set; }
```

```
}
```

```
DataStore<string> store = new DataStore<string>();
```

```
DataStore<string> store = new DataStore<string>();
```

```
class DataStore<T>
```

```
{
```

```
    public T Data { get; set; }
```

```
}
```

© TutorialsTeacher.com

```
store.Data = "Hello World!";
```

```
//store.Data = 123; // compile-time error
```

Generic Methods

- A generic method declares type parameters within the signature of a method

```
using System;
using System.Collections.Generic;
namespace GenericApp
{
    class Program
    {
        static void Swap<T>(ref T a, ref T b)
        {
            T temp = a;
            a = b;
            b = temp;
        }
        static void Main(string[] args)
        {
            int a = 40, b = 60;
            Swap<int>(ref a, ref b);
            Console.WriteLine("After swap: {0}, {1}", a, b);
        }
    }
}
```


Generic Constraints

- By default, a type parameter can be substituted with any type whatsoever. Constraints can be applied to a type parameter to require more specific type arguments.

These are the possible constraints:

- `where T : base-class` // Type argument must be a value type
- `where T : interface` // Type argument must implement from `<interface> interface`.
- `where T : class` // Type argument must be a reference type
- `where T : struct` // Value-type constraint (excludes Nullable types)
- `where T : new()` // Type argument must have a public parameterless constructor.
- `where T : U` // There are two type arguments T and U. T must be inherit from U.