

## Unit -3 Creating Types in C# [12 Hrs]

Classes; Constructors and Destructors; this Reference; Properties; Indexers; Static Constructors and Classes; Finalizers; Dynamic Binding; Operator Overloading; Inheritance; Abstract Classes and Methods; base Keyword; Overloading; Object Type; Structs; Access Modifiers; Interfaces; Enums; Generics

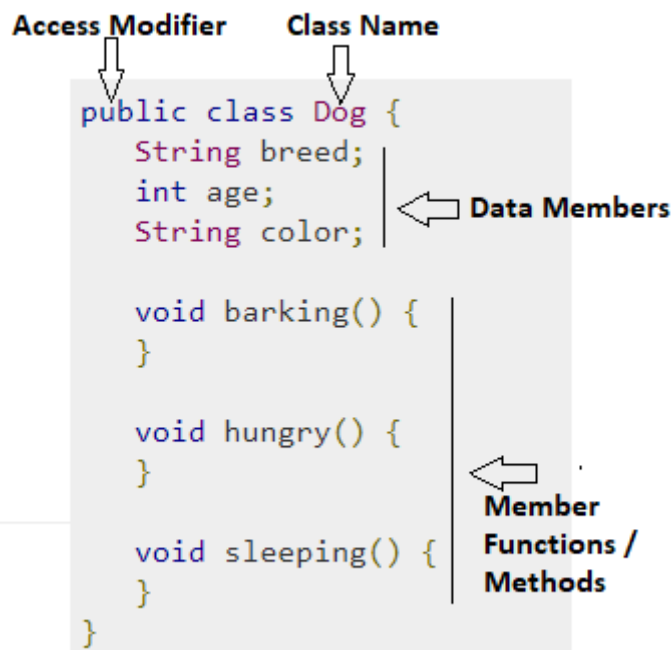
### Classes

A class, in the context of C#, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access.
2. **Class name**: The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any)**: The name of the class's parent (superclass), if any, preceded by the (:). A class can only extend (subclass) one parent.
4. **Interfaces(if any)**: A comma-separated list of interfaces implemented by the class, if any, preceded by the (:). A class can implement more than one interface.
5. **Body**: The class body surrounded by braces, { }.

General form of a class is shown below:



```
Access Modifier  Class Name
  ↓              ↓
public class Dog {
    String breed;
    int age;
    String color;
    void barking() {
    }
    void hungry() {
    }
    void sleeping() {
    }
}
```

← **Data Members**

← **Member Functions / Methods**

### Object

An object is a combination of data and procedures working on the available data. An object has a state and behavior. The state of an object is stored in fields (variables), while methods

(functions) display the object's behavior. Objects are created from templates known as classes. In C#, an object is created using the keyword "new". Object is an instance of a class. There are three steps to creating a Java object:

1. Declaration of the object
2. Instantiation of the object
3. Initialization of the object

When a object is declared, a name is associated with that object. The object is instantiated so that memory space can be allocated. Initialization is the process of assigning a proper initial value to this allocated space. The properties of Java objects include:

- One can only interact with the object through its methods. Hence, internal details are hidden.
- When coding, an existing object may be reused.
- When a program's operation is hindered by a particular object, that object can be easily removed and replaced.

A new object t from the class "tree" is created using the following syntax:

**Tree t = new Tree ().**

## **Fields**

A field is a variable that is a member of a class or struct. For example:

```
class Octopus
{
    string name;
    public int Age = 10;
}
```

Fields allow the following modifiers:

Static modifier	<code>static</code>
Access modifiers	<code>public internal private protected</code>
Inheritance modifier	<code>new</code>
Unsafe code modifier	<code>unsafe</code>
Read-only modifier	<code>readonly</code>
Threading modifier	<code>volatile</code>

### **The readonly modifier**

The readonly modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the enclosing type's constructor.

```
public int Age = 10;
static readonly int legs = 8, eyes = 2;
```

## Methods

A method performs an action in a series of statements. A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type. A method can specify a void return type, indicating that it doesn't return any value to its caller.

A method can also output data back to the caller via ref/out parameters. A method's signature must be unique within the type. A method's signature comprises its name and parameter types in order (but not the parameter names, nor the return type).

Methods allow the following modifiers:

Static modifier	<code>static</code>
Access modifiers	<code>public internal private protected</code>
Inheritance modifiers	<code>new virtual abstract override sealed</code>
Partial method modifier	<code>partial</code>
Unmanaged code modifiers	<code>unsafe extern</code>
Asynchronous code modifier	<code>async</code>

## Expression-bodied methods

A method that comprises a single expression, such as the following:

```
int Foo (int x) { return x * 2; }
```

can be written more tersely as an expression-bodied method. A fat arrow replaces the braces and return keyword:

```
int Foo (int x) => x * 2;
```

Expression-bodied functions can also have a void return type:

```
void Foo (int x) => Console.WriteLine (x);
```

## Overloading methods

A type may overload methods (have multiple methods with the same name), as long as the signatures are different. For example, the following methods can all coexist in the same type:

```
void Foo (int x) {...}  
void Foo (double x) {...}  
void Foo (int x, float y) {...}  
void Foo (float x, int y) {...}
```

However, the following pairs of methods cannot coexist in the same type, since the return type and the params modifier are not part of a method's signature:

```
void Foo (int x) {...}  
float Foo (int x) {...}           // Compile-time error  
  
void Goo (int[] x) {...}  
void Goo (params int[] x) {...}  // Compile-time error
```

## Constructors

A *constructor* in C# is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

All classes have constructors, whether you define one or not, because C# automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

### Default Constructor

The *default constructor* is a constructor that is automatically generated in the absence of explicit constructors (i.e. no user defined constructor). The automatically provided constructor is called sometimes a *nullary* constructor.

Following is the syntax of a default constructor –

```
class ClassName {  
    ClassName() {  
    }  
}
```

### Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class Panda  
{  
    string name;           // Define field  
    public Panda (string n) // Define constructor  
    {  
        name = n;         // Initialization code (set up field)  
    }  
}  
...  
  
Panda p = new Panda ("Petey"); // Call constructor
```

Instance constructors allow the following modifiers:

public   internal   private   protected

## Overloaded Constructors

A class or struct may overload constructors. To avoid code duplication, one constructor may call another, using **this** keyword:

```
using System;

public class Wine
{
    public decimal Price;
    public int Year;
    public Wine (decimal price) { Price = price; }
    public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

When one constructor calls another, the called constructor executes first.

## Destructor

Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.

### **Important Points:**

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a [constructor](#) because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

### Syntax

```
class Example
{
    // Rest of the class
    // members and methods.

    // Destructor
    ~Example()
    {
        // Your code
    }
}
```

**Example:**

```
class ConsDes
{
    //constructor
    public ConsDes(string message)
    {
        Console.WriteLine(message);
    }

    public void test()
    {
        Console.WriteLine("This is a method");
    }

    //destructor
    ~ConsDes()
    {
        Console.WriteLine("This is a destructor");
        Console.ReadKey();
    }
}

class Construct
{
    static void Main(string[] args)
    {
        string msg = "This is a constructor";
        ConsDes obj = new ConsDes(msg);
        obj.test();
    }
}
```

**Output:**

```
This is a constructor
This is a method
This is a destructor
```

**Static Constructor**

In c#, **Static Constructor** is used to perform a particular action only once throughout the application. If we declare a [constructor](#) as **static**, then it will be invoked only once irrespective of number of class instances and it will be called automatically before the first instance is created.

Generally, in c# the static constructor will not accept any access modifiers and parameters. In simple words we can say it's a parameter less.

Following are the properties of static constructor in c# programming language.

- Static constructor in c# won't accept any parameters and access modifiers.

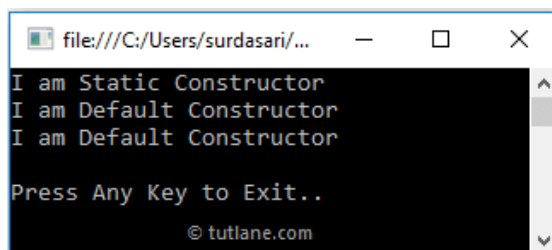
- The static constructor will invoke automatically, whenever we create a first instance of class.
- The static constructor will be invoked by CLR so we don't have a control on static constructor execution order in c#.
- In c#, only one static constructor is allowed to create.

### C# Static Constructor Syntax

```
class User
{
    // Static Constructor
    static User()
    {
        // Your Custom Code
    }
}
```

### Example

```
class User
{
    // Static Constructor
    static User()
    {
        Console.WriteLine("I am Static Constructor");
    }
    // Default Constructor
    public User()
    {
        Console.WriteLine("I am Default Constructor");
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Both Static and Default constructors will invoke for
first instance
        User user = new User();
        // Only Default constructor will invoke
        User user1 = new User();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```



## **The this Reference**

The “this” keyword in C# is used to refer to the current instance of the class. It is also used to differentiate between the method parameters and class fields if they both have the same name.

Another usage of “this” keyword is to call another constructor from a constructor in the same class.

Here, for an example, we are showing a record of Students i.e: id, Name, Age, and Subject. To refer to the fields of the current class, we have used the “this” keyword in C#:

```
public Student(int id, String name, int age, String subject) {
    this.id = id;
    this.name = name;
    this.subject = subject;
    this.age = age;
}
```

Let us see the complete example to learn how to work with the “this” keyword in C#:

```
using System;

class Student {
    public int id, age;
    public String name, subject;

    public Student(int id, String name, int age, String subject) {
        this.id = id;
        this.name = name;
        this.subject = subject;
        this.age = age;
    }

    public void showInfo() {
        Console.WriteLine(id + " " + name + " " + age + " " + subject);
    }
}

class StudentDetails {
    public static void Main(string[] args) {
        Student std1 = new Student(001, "Jack", 23, "Maths");

        std1.showInfo();
    }
}
```

### **Output:**

```
001 Jack 23 Maths
```



## Properties

**Properties** look like fields from the outside, but internally they contain logic, like methods do. **Properties** are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors(get and set)** through which the values of the private fields can be read, written or manipulated.

Usually, inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

```
using System;
class MyClass
{
    private int x;
    public void SetX(int i)
    {
        x = i;
    }
    public int GetX()
    {
        return x;
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.SetX(10);
        int xVal = mc.GetX();
        Console.WriteLine(xVal);
    }
}
```

### Output:

10

## Automatic Properties

The most common implementation for a property is a getter and/or setter that simply reads and writes to a private field of the same type as the property. An automatic property declaration instructs the compiler to provide this implementation. We can improve the first example in this section by declaring `CurrentPrice` as an automatic property:

```
public class Stock
{
    ...
    public decimal CurrentPrice { get; set; }
}
```

```

class Chk
{
    public int a { get; set; }
    public int b { get; set; }
    public int sum
    {
        get { return a + b; }
    }
}

class Test
{
    static void Main()
    {
        Chk obj = new Chk();
        obj.a = 10;
        obj.b = 5;
        Console.WriteLine("Sum of "+obj.a+" and "+obj.b+" = "+obj.sum);
        Console.ReadKey();
    }
}

```

**Output:**

Sum of 10 and 5 = 15

## **Indexers**

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index argument rather than a property name.

The string class has an indexer that lets you access each of its char values via an int index:

```

string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'

```

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s).

C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Defining an indexer allows you to create a class like that can allows its items to be accessed an array. Instances of that class can be accessed using the [] array access operator.

```
<modifier> <return type> this [argument list]
{
    get
    {
        // your get block code
    }
    set
    {
        // your set block code
    }
}
```

In the above code:

**<modifier>**

can be private, public, protected or internal.

**<return type>**

can be any valid C# types.

**this**

this is a special keyword in C# to indicate the object of the current class.

**[argument list]**

The formal-argument-list specifies the parameters of the indexer.

Following program demonstrates how to use an indexer.

```

class Program
{
    class IndexerClass
    {
        private string[] names = new string[10];
        public string this[int i]
        {
            get
            {
                return names[i];
            }
            set
            {
                names[i] = value;
            }
        }
    }
    static void Main(string[] args)
    {
        IndexerClass Team = new IndexerClass();
        Team[0] = "Rocky";
        Team[1] = "Teena";
        Team[2] = "Ana";
        Team[3] = "Victoria";
        Team[4] = "Yani";
        Team[5] = "Mary";
        Team[6] = "Gomes";
        Team[7] = "Arnold";
        Team[8] = "Mike";
        Team[9] = "Peter";
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(Team[i]);
        }
        Console.ReadKey();
    }
}

```

### Difference between Indexers and Properties

Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	Properties can be static as well as instance members.
A get accessor of an indexer has the same formal parameter list as the indexer.	A get accessor of a property has no parameters.
A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	A set accessor of a property contains the implicit value parameter.

## **Static Classes**

A C# static class is a class that can't be instantiated. The sole purpose of the class is to provide blueprints of its inherited classes. A static class is created using the "static" keyword in C#. A static class can contain static members only. You can't create an object for the static class.

### **Advantages of Static Classes**

1. If you declare any member as a non-static member, you will get an error.
2. When you try to create an instance to the static class, it again generates a compile time error, because the static members can be accessed directly with its class name.
3. The static keyword is used before the class keyword in a class definition to declare a static class.
4. A static class members are accessed by the class name followed by the member name.

### **Syntax of static class**

```
static class classname
{
    //static data members
    //static methods
}
```

### **Static members of a class**

If we declare any members of a class as static we can access it without creating object of that class.

#### **Example**

```
class MyCollege
{
    //static fields
    public static string CollegeName;
    public static string Address;

    //static constructor
    static MyCollege()
    {
        CollegeName = "ABC College of Technology";
        Address = "Hyderabad";
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyCollege.CollegeName);
        Console.WriteLine(MyCollege.Address);
        Console.Read();
    }
}
```

## Example of static class

```
// Creating static class
// Using static keyword
static class Author {

    // Static data members of Author
    public static string A_name = "Ankita";
    public static string L_name = "CSharp";
    public static int T_no = 84;

    // Static method of Author
    public static void details()
    {
        Console.WriteLine("The details of Author is:");
    }
}

// Driver Class
public class GFG {

    // Main Method
    static public void Main()
    {

        // Calling static method of Author
        Author.details();

        // Accessing the static data members of Author
        Console.WriteLine("Author name : {0} ", Author.A_name);
        Console.WriteLine("Language : {0} ", Author.L_name);
        Console.WriteLine("Total number of articles : {0} ",
                           Author.T_no);
    }
}
```

### Output

The details of Author is:  
Author name : Ankita  
Language : CSharp  
Total number of articles : 84

## Finalizers

Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the ~ symbol:

```
class Class1
{
    ~Class1()
    {
        ...
    }
}
```

## Inheritance

**Inheritance** is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allow to **inherit** the features (**fields and methods**) of another class.

The process by which one class acquires the properties (data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship. Inheritance is used in java for the following:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

## Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

## The syntax of Java Inheritance

```
class Subclass-name : Superclass-name  
{  
    //methods and fields  
}
```

The **:** indicates that you are making a new class that derives from an existing class.

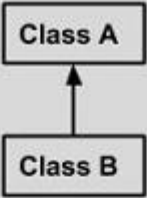
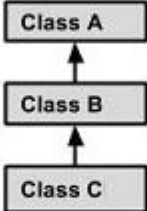
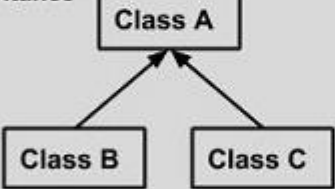
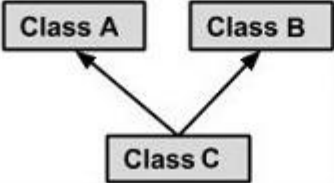
The meaning of "extends" is to increase the functionality.

In the terminology of C#, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

## Types of inheritance in C#

On the basis of class, there can be **three** types of inheritance in java: **single, multilevel and hierarchical**.

In C# programming, **multiple and hybrid inheritance** is supported through **interface** only.

<b>Single Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A] </pre>	<pre> public class A {     ..... } public class B extends A {     ..... } </pre>
<b>Multi Level Inheritance</b>  <pre> graph BT     C[Class C] --&gt; B[Class B]     B --&gt; A[Class A] </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends B {.....} </pre>
<b>Hierarchical Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]     C[Class C] --&gt; A </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends A {.....} </pre>
<b>Multiple Inheritance</b>  <pre> graph BT     C[Class C] --&gt; A[Class A]     C --&gt; B[Class B] </pre>	<pre> public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance </pre>

**Note:** Codes are written in Java, So please change the syntax of the program according to C# language.

## 1. Single Inheritance

**Single Inheritance** refers to a child and parent class relationship where a class extends the another class.

```

class A
{
    public int a=10, b=5;
}

class B : A
{
    int a=30, b=5;
    public void test()
    {
        Console.WriteLine("Value of a is: "+a);
        Console.WriteLine("Value of a is: " + base.a);
    }
}

```



```
//driver class
class Inherit
{
    static void Main(string[] args)
    {
        B obj = new B();
        obj.test();
        Console.ReadLine();
    }
}
```

## **2. Multilevel Inheritance**

Multilevel inheritance refers to a child and parent class relationship where a class extends the child class. For example, class C extends class B and class B extends class A.

```
class A5
{
    public int a, b, c;

    public void ReadData(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    public void Display()
    {
        Console.WriteLine("Value of a is: "+a);
        Console.WriteLine("Value of b is: " + b);
    }
}

class A6 : A5
{
    public void Add()
    {
        base.c = base.a + base.b;
        Console.WriteLine("Sum="+base.c);
    }
}

class A7 : A6
{
    public void Sub()
    {
        base.c = base.a - base.b;
        Console.WriteLine("Difference=" + base.c);
    }
}
```

```

    }

    class Level
    {
        static void Main()
        {
            A7 obj = new A7();
            obj.ReadData(20,5);
            obj.Display();
            obj.Add();
            obj.Sub();

            Console.ReadLine();
        }
    }
}

```

### **3. Hierarchical Inheritance**

Hierarchical inheritance refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```

class Polygon
{
    public int dim1,dim2;
    public void ReadDimension(int dim1, int dim2)
    {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }
}

class Rectangle : Polygon
{
    public void AreaRec()
    {
        base.ReadDimension(10,5);
        int area = base.dim1 * base.dim2;
        Console.WriteLine("Area of Rectangle="+area);
    }
}

class Traingle : Polygon
{
    public void AreaTri()
    {
        base.ReadDimension(10,5);
        double area = 0.5*base.dim1 * base.dim2;
        Console.WriteLine("Area of Triangle=" + area);
    }
}

```

```

    }
}

//driver class
class Hier
{
    static void Main()
    {
        Traingle tri = new Traingle();
        //tri.ReadDimension(10,5);
        tri.AreaTri();

        Rectangle rec = new Rectangle();
        //rec.ReadDimension(10,7);
        rec.AreaRec();

        Console.ReadLine();
    }
}

```

#### 4. Multiple Inheritance

When one class extends more than one classes then this is called multiple inheritance. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance.

C# doesn't allow multiple inheritance. We can use **interfaces** instead of **classes** to achieve the same purpose.

```

interface IA
{
    // doesn't contain fields
    int CalculateArea();
    int CalculatePerimeter();
}

class CA
{
    public int l, b;
    public void ReadData(int l,int b)
    {
        this.l = l;
        this.b = b;
    }
}

class BB : CA, IA
{
    public int CalculateArea()
    {
        ReadData(10,5);
        int area=l*b;
    }
}

```

```

        return area;
    }

    public int CalculatePerimeter()
    {
        ReadData(15, 10);
        int peri = 2*(l+b);
        return peri;
    }
}

//driver class
class Inter
{
    static void Main(string[] args)
    {
        BB obj = new BB();
        //int area=obj.CalculateArea();
        //int peri=obj.CalculatePerimeter();

        Console.WriteLine("Area of Rectangle=" + obj.CalculateArea());
        Console.WriteLine("Perimeter of Rectangle=" +
            obj.CalculatePerimeter());

        Console.ReadKey();
    }
}

```

## **Interface in C#**

An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties. The reason interfaces only provide declarations is because they are inherited by structs and classes, that must provide an implementation for each interface member declared.

Like a class, an interface can have methods and properties, but the methods declared in interface are by default abstract (only method signature, no body).

- ❖ Interfaces specify what a class must do and not how. It is the blueprint of the class.
- ❖ An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
- ❖ If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

### **Why do we use interface?**

- It is used to achieve total abstraction.
- Since C# does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

```

interface IA
{
    // doesn't contain fields
    void GetData(int l,int b);
    int CalculateArea();
    int CalculatePerimeter();
}

class BB : IA
{
    int l, b;
    public void GetData(int l, int b)
    {
        this.l = l;
        this.b = b;
    }
    public int CalculateArea()
    {
        int area=l*b;
        return area;
    }

    public int CalculatePerimeter()
    {
        int peri = 2*(l+b);
        return peri;
    }
}

//driver class
class Inter
{
    static void Main(string[] args)
    {
        BB obj = new BB();
        obj.GetData(10,5);
        Console.WriteLine("Area of Rectangle=" + obj.CalculateArea());
        Console.WriteLine("Perimeter of Rectangle=" +
            obj.CalculatePerimeter());

        Console.ReadKey();
    }
}

```

**Output:**

Area of Rectangle=50  
 Perimeter of Rectangle=30

## Abstract Classes

A class declared as abstract can never be instantiated. Instead, only its concrete sub-classes can be instantiated.

If a class is defined as abstract then we can't create an instance of that class. By the creation of the derived class object where an abstract class is inherit from, we can call the method of the abstract class.

For example,

```
abstract class mc1 {
    public int add(int a, int b) {
        return (a + b);
    }
}
class mc1: mc1 {
    public int mul(int a, int b) {
        return a * b;
    }
}
class test {
    static void Main(string[] args) {
        mc1 ob = new mc1();
        int result = ob.add(5, 10);
        Console.WriteLine("the result is {0}", result);
    }
}
```

## Abstract Members

An Abstract method is a method without a body. The implementation of an abstract method is done by a derived class. When the derived class inherits the abstract method from the abstract class, it must override the abstract method. This requirement is enforced at compile time and is also called dynamic polymorphism.

**Abstract members are used to achieve total abstraction.**

The syntax of using the abstract method is as follows:

<access-modifier>abstract<return-type>method name (parameter)

The abstract method is declared by adding the abstract modifier the method.

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

```

public class TestAbstract
{
    public static void Main()
    {
        Rectangle s = new Rectangle();
        s.draw();
    }
}

```

**Output:**

drawing ractangle...

**Another Example**

```

abstract class test1 {
    public int add(int i, int j) {
        return i + j;
    }
    public abstract int mul(int i, int j);
}
class test2: test1 {
    public override int mul(int i, int j) {
        return i * j;
    }
}
class test3: test1 {
    public override int mul(int i, int j) {
        return i - j;
    }
}
class test4: test2 {
    public override int mul(int i, int j) {
        return i + j;
    }
}
class myclass {
    public static void main(string[] args) {
        test2 ob = new test4();
        int a = ob.mul(2, 4);
        test1 ob1 = new test2();
        int b = ob1.mul(4, 2);
        test1 ob2 = new test3();
        int c = ob2.mul(4, 2);
        Console.WriteLine("{0},{1},{2}", a, b, c);
        Console.ReadLine();
    }
}

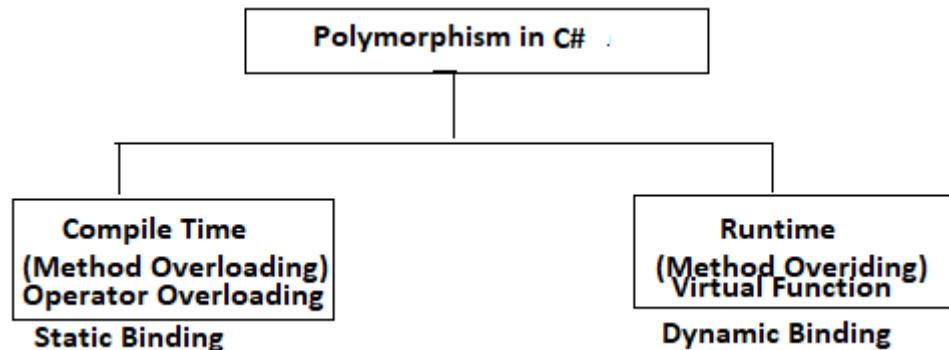
```

In the above program, one method i.e. mul can perform various functions depending on the value passed as parameters by creating an object of various classes which inherit other classes. Hence we can achieve dynamic polymorphism with the help of an abstract method.

## **Polymorphism**

**Polymorphism in C#** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.



## **Method Overloading**

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists. In order to overload a method, the argument lists of the methods must differ in either of these:

### **a) Number of parameters.**

add(int, int)  
add(int, int, int)

### **b) Data type of parameters.**

add(int, int)  
add(int, float)

### **c) Sequence of Data type of parameters.**

add(int, float)  
add(float, int)



## Example of Method Overloading

```
public class Methodoverloading
{
    public int add(int a, int b) //two int type Parameters method
    {
        return a + b;
    }
    public int add(int a, int b,int c) //three int type Parameters with same method
    {
        return a + b+c;
    }
    public float add(float a, float b,float c,float d) //four float type Parameters
    {
        return a + b+c+d;
    }
}
```

## Method Overriding

Method overriding in C# allows programmers to create base classes that allows its inherited classes to override same name methods when implementing in their class for different purpose. This method is also used to enforce some must implement features in derived classes.

### **Important points:**

- Method overriding is only possible in derived classes, not within the same class where the method is declared.
- Base class must use the virtual or abstract keywords to declare a method. Then only can a method be overridden

Here is an example of method overriding.

```
public class Account
{
    public virtual int balance()
    {
        return 10;
    }
}

public class Amount : Account
{
    public override int balance()
    {
        return 500;
    }
}
```

```

class Test
{
    static void Main()
    {
        Amount obj = new Amount();
        int balance = obj.balance();
        Console.WriteLine("Balance is: "+balance);
        Console.ReadKey();
    }
}

```

**Output:**

Balance is 500

### **Virtual Method**

A virtual method is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived the class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overridden in the derived class using the override keyword.

### **Features of virtual method**

- By default, methods are non-virtual. We can't override a non-virtual method.
- We can't use the virtual modifier with the static, abstract, private or override modifiers.
- **If class is not inherited, behaviour of virtual method is same as non-virtual method, but in case of inheritance it is used for method overriding.**

### **Behaviour of virtual method without inheritance – same as non virtual**

```

class Vir
{
    public virtual void message()
    {
        Console.WriteLine("This is test");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Vir obj = new Vir();
        obj.message();
        Console.ReadKey();
    }
}

```

## Behaviour of virtual method with inheritance – used for overriding

```
class Vir
{
    public virtual void message()
    {
        Console.WriteLine("This is test");
    }
}

class Vir1 : Vir
{
    public override void message()
    {
        Console.WriteLine("This is test1");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Vir1 obj = new Vir1();
        obj.message();
        Console.ReadKey();
    }
}
```

### Output:

This is test1

## Upcasting and Downcasting

**Upcasting** converts an object of a specialized type to a more general type. **An upcast operation creates a base class reference from a subclass reference.**

For example:

```
Stock msft = new Stock();
Asset a = msft; // Upcast
```

**Downcasting** converts an object from a general type to a more specialized type. **A downcast operation creates a subclass reference from a base class reference.**

For example:

```
Stock msft = new Stock();
Asset a = msft; // Upcast
Stock s = (Stock)a; // Downcast
```



```

BankAccount    ba1,
                ba2 = new BankAccount("John", 250.0M, 0.01);
LotteryAccount la1,
                la2 = new LotteryAccount("Bent", 100.0M);

    ba1 = la2;           // upcasting - OK
//    la1 = ba2;         // downcasting - Illegal
                        // discovered at compile time
//    la1 = (LotteryAccount)ba2; // downcasting - Illegal
                        // discovered at run time
    la1 = (LotteryAccount)ba1; // downcasting - OK
                        // ba1 already refers to a LotteryAccount
  
```

### The as operator

The as operator performs a downcast that evaluates to null (rather than throwing an exception) if the downcast fails:

```

Asset a = new Asset();
Stock s = a as Stock; // s is null; no exception thrown
  
```

### The is operator

The is operator tests whether a reference conversion would succeed; in other words, whether an object derives from a specified class (or implements an interface). It is often used to test before downcasting.

```

if (a is Stock)
    Console.WriteLine (((Stock)a).SharesOwned);
  
```

## Operator Overloading

The concept of overloading a function can also be applied to operators. Operator overloading gives the ability to use the same operator to do various operations. It provides additional capabilities to C# operators when they are applied to user-defined data types. It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.

Only the predefined set of C# operators can be overloaded. To make operations on a user-defined data type is not as simple as the operations on a built-in data type. To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement. An operator can be overloaded by defining a function to it. The function of the operator is declared by using the operator keyword.

### Syntax:

```

access specifier className operator Operator_symbol (parameters)
{
    // Code
}
  
```

The following table describes the overloading ability of the various operators available in C# :

OPERATORS	DESCRIPTION
+, -, !, ~, ++, --	unary operators take one operand and can be overloaded.
+, -, *, /, %	Binary operators take two operands and can be overloaded.
==, !=, =	Comparison operators can be overloaded.
&&,	Conditional logical operators cannot be overloaded directly
+=, -=, *=, /=, %=, =	Assignment operators cannot be overloaded.

### Overloading Unary Operators

The following program overloads the **unary - operator** inside the class Complex.

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
    {
    }
    public Complex(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine("{0} {1}", x, y);
    }
    public static Complex operator -(Complex c)
    {
        Complex temp = new Complex();
        temp.x = -c.x;
        temp.y = -c.y;
        return temp;
    }
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex();
        c2.ShowXY(); // displays 0 & 0
        c2 = -c1;
        c2.ShowXY(); // diapls -10 & -20
    }
}
```

## Overloading Binary Operators

An overloaded binary operator must take two arguments; at least one of them must be of the type class or struct, in which the operation is defined.

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
    }
    public Complex(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine("{0} {1}", x, y);
    }
    public static Complex operator +(Complex c1, Complex c2)
    {
        Complex temp = new Complex();
        temp.x = c1.x + c2.x;
        temp.y = c1.y + c2.y;
        return temp;
    }
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex(20, 30);
        c2.ShowXY(); // displays 20 & 30
        Complex c3 = new Complex();
        c3 = c1 + c2;
        c3.ShowXY(); // displays 30 & 50
    }
}
```

## Sealing Functions and Classes

### C# 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. In C#, the sealed modifier is used to declare a class as sealed. If a class is derived from a sealed class, compiler throws an error.

**If you have ever noticed, structs are sealed. You cannot derive a class from a struct.**

```
// Sealed class
sealed class SealedClass{
}
```

```

using System;
class Class1
{
    static void Main(string[] args)
    {
        SealedClass sealedCls = new SealedClass();
        int total = sealedCls.Add(4, 5);
        Console.WriteLine("Total = " + total.ToString());
    }
}
// Sealed class
sealed class SealedClass
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}

```

### **Sealed Methods and Properties**

You can also use the sealed modifier on a method or a property that overrides a virtual method or property in a base class.

This enables you to allow classes to derive from your class and prevent other developers that are using your classes from overriding specific virtual methods and properties.

```

class X
{
    protected virtual void F()
    {
        Console.WriteLine("X.F");
    }
    protected virtual void F2()
    {
        Console.WriteLine("X.F2");
    }
}
class Y : X
{
    sealed protected override void F()
    {
        Console.WriteLine("Y.F");
    }
    protected override void F2()
    {
        Console.WriteLine("X.F3");
    }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    //
    protected override void F()
    {
        Console.WriteLine("C.F");
    }
    // Overriding F2 is allowed.
    protected override void F2()
    {
        Console.WriteLine("Z.F2");
    }
}

```

## The base Keyword

We can use the base keyword to access the fields of the base class within derived class. It is useful if base and derived classes have the same fields.

**If derived class doesn't define same field, there is no need to use base keyword.** Base class field can be directly accessed by the derived class.

```
using System;
public class Animal{
    public string color = "white";
}
public class Dog: Animal
{
    string color = "black";
    public void showColor()
    {
        Console.WriteLine(base.color); //displays white
        Console.WriteLine(color); //displays black
    }
}
public class TestBase
{
    public static void Main()
    {
        Dog d = new Dog();
        d.showColor();
    }
}
```

The base keyword has two uses:

- To call a base class constructor from a derived class constructor.
- To call a base class method which is overridden in the derived class.

## Calling a base class constructor from a derived class constructor

```
class Base
{
    public Base(int a, int b)
    {
        Console.WriteLine("Value of a={0} and b={1}",a,b);
    }
}
class Derived : Base
{
    public Derived(int x,int y):base(x,y)
    {
```



```

        Console.WriteLine("Value of x={0} and y={1}", x, y);
    }
}
class BaseEx
{
    static void Main(){
        new Derived(10,5);
        Console.ReadKey();
    }
}

```

**Output:**

Value of a=10 and b=5  
Value of x=10 and y=5

**Calling a base class method which is overridden in derived class**

```

class Base
{
    public virtual void BaseMethod()
    {
        Console.WriteLine("I am inside base class");
    }
}
class Derived : Base
{
    public override void BaseMethod()
    {
        base.BaseMethod();
        Console.WriteLine("I am inside derived class");
    }
}
class BaseEx
{
    static void Main(){
        Derived obj = new Derived();
        obj.BaseMethod();
        Console.ReadKey();
    }
}

```

**Output:**

I am inside base class  
I am inside derived class

## The object Type

**object (System.Object) is the ultimate base class for all types. Any type can be upcast to object.**

To illustrate how this is useful, consider a general-purpose stack. A stack is a data structure based on the principle of LIFO—"Last-In First-Out." A stack has two operations: push an object on the stack, and pop an object off the stack.

**Here is a simple implementation that can hold up to 10 objects:**

```
public class Stack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj) { data[position++] = obj; }
    public object Pop()           { return data[--position]; }
}
```

Because Stack works with the object type, we can Push and Pop instances of *any type* to and from the Stack:

## Boxing and Unboxing

**Boxing is the act of converting a value-type instance to a reference-type instance.** The reference type may be either the object class or an interface.

In this example, we box an int into an object:

```
int x = 9;
object obj = x; // Box the int
```

**Unboxing reverses the operation, by casting the object back to the original value type:**

```
int y = (int)obj; // Unbox the int
```

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type, and throws an `InvalidCastException` if the check fails.

For instance, the following throws an exception, because long does not exactly match int:

```
object obj = 9;           // 9 is inferred to be of type int
long x = (long) obj;      // InvalidCastException
```

The following succeeds, however:

```
object obj = 9;
long x = (int) obj;
```

As does this:

```
object obj = 3.5;         // 3.5 is inferred to be of type double
int x = (int) (double) obj; // x is now 3
```

## The GetType Method and typeof Operator

All types in C# are represented at runtime with an instance of System.Type. There are two basic ways to get a System.Type object:

- Call GetType on the instance.
- Use the typeof operator on a type name.

**GetType** is evaluated at runtime; **typeof** is evaluated statically at compile time (when generic type parameters are involved, it's resolved by the Just-In-Time compiler).

**System.Type** has properties for such things as the type's name, assembly, base type, and so on. For example:

```
using System;

public class Point { public int X, Y; }

class Test
{
    static void Main()
    {
        Point p = new Point();
        Console.WriteLine (p.GetType().Name);           // Point
        Console.WriteLine (typeof (Point).Name);        // Point
        Console.WriteLine (p.GetType() == typeof(Point)); // True
        Console.WriteLine (p.X.GetType().Name);         // Int32
        Console.WriteLine (p.Y.GetType().FullName);     // System.Int32
    }
}
```

## Structs

A struct is similar to a class, with the following key differences:

- A struct is a value type, whereas a class is a reference type.
- A struct does not support inheritance

A struct can have all the members a class can, except the following:

- A parameter less constructor
- Field initializers
- A finalizer
- Virtual or protected members

### Here is an example of declaring and calling struct:

```
public struct Point
{
    int x, y;
    public Point (int x, int y) { this.x = x; this.y = y; }
}

...
Point p1 = new Point ();          // p1.x and p1.y will be 0
Point p2 = new Point (1, 1);     // p1.x and p1.y will be 1
```

The next example generates three compile-time errors:

```
public struct Point
{
    int x = 1;                      // Illegal: field initializer
    int y;
    public Point() {}               // Illegal: parameterless constructor
    public Point (int x) {this.x = x;} // Illegal: must assign field y
}
```

Changing struct to class makes this example legal.

### Access Modifiers

Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself. For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.

Access Modifiers	Inside Assembly		Outside Assembly	
	With Inheritance	With Type	With Inheritance	With Type
Public	✓	✓	✓	✓
Private	X	X	X	X
Protected	✓	X	✓	X
Internal	✓	✓	X	X
Protected Internal	✓	✓	✓	X

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

In C# there are 6 different types of Access Modifiers.

Modifier	Description
<b>public</b>	There are no restrictions on accessing public members.
<b>private</b>	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
<b>protected</b>	Access is limited to within the class definition and any class that inherits from the class
<b>internal</b>	Access is limited exclusively to classes defined within the current project assembly
<b>protected internal</b>	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
<b>private protected</b>	Access is limited to the containing class or types derived from the containing class within the current assembly.

## Examples

Class2 is accessible from outside its assembly; Class1 is not:

```
class Class1 {}           // Class1 is internal (default)
public class Class2 {}
```

ClassB exposes field x to other types in the same assembly; ClassA does not:

```
class ClassA { int x;      } // x is private (default)
class ClassB { internal int x; }
```

Functions within Subclass can call Bar but not Foo:

```
class BaseClass
{
    void Foo() {}           // Foo is private (default)
    protected void Bar() {}
}

class Subclass : BaseClass
{
    void Test1() { Foo(); }  // Error - cannot access Foo
    void Test2() { Bar(); }  // OK
}
```

## Restrictions on Access Modifiers

```
class BaseClass { protected virtual void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} } // OK
class Subclass2 : BaseClass { public override void Foo() {} } // Error
```

(An exception is when overriding a protected internal method in another assembly, in which case the override must simply be protected.)

The compiler prevents any inconsistent use of access modifiers. For example, a subclass itself can be less accessible than a base class, but not more:

```
internal class A {}
public class B : A {}           // Error
```

## Enums in C#

Enum in C# language is a value type with a set of related named constants often referred to as an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type, which is user-defined. Enums type can be an integer (float, int, byte, double etc.) but if you use beside int, it has to be cast.

Enum is used to create numeric constants in .NET framework. All member of the enum are of enum type. There must be a numeric value for each enum type.

**The default underlying type of the enumeration elements is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.**

```
// making an enumerator 'month'
enum month
{

    // following are the data members
    jan,
    feb,
    mar,
    apr,
    may

}

class Program {

    // Main Method
    static void Main(string[] args)
    {

        // getting the integer values of data members..
        Console.WriteLine("The value of jan in month " +
            "enum is " + (int)month.jan);
        Console.WriteLine("The value of feb in month " +
            "enum is " + (int)month.feb);
        Console.WriteLine("The value of mar in month " +
            "enum is " + (int)month.mar);
        Console.WriteLine("The value of apr in month " +
            "enum is " + (int)month.apr);
        Console.WriteLine("The value of may in month " +
            "enum is " + (int)month.may);

    }
}
```

### Output

```
The value of jan in month enum is 0
The value of feb in month enum is 1
The value of mar in month enum is 2
The value of apr in month enum is 3
The value of may in month enum is 4
```

## Generics

Generics in C# and .NET procedure many of the benefits of strongly-typed collections as well as provide a higher quality of and a performance boost for code.

Generics are very similar to C++ templates but having a slight difference in such a way that the source code of C++ templates is required when a template is instantiated with a specific type and .NET Generics are not limited to classes only. In fact, they can also be implemented with Interfaces, Delegates and Methods.

The detailed specification for each collection is found under the **System.Collection.Generic** namespace.

### Generic Classes

The **Generic class** can be defined by putting the **<T>** sign after the class name. It isn't mandatory to put the "T" word in the Generic type definition. You can use any word in the TestClass<> class declaration.

```
public class TestClass<T> { }
```

The **System.Collection.Generic** namespace also defines a number of classes that implement many of these key interfaces. The following table describes the core class types of this namespace.

Generic class	Description
Collection<T>	The basis for a generic collection Comparer compares two generic objects for equality
Dictionary<TKey, TValue>	A generic collection of name/value pairs
List<T>	A dynamically resizable list of Items
Queue<T>	A generic implementation of a first-in, first-out (FIFO) list
Stack<T>	A generic implementation of a last-in, first-out (LIFO) list

```
using System.Collections.Generic;
class Test<T>
{
    T[] t=new T[5];
    int count = 0;
    public void addItem(T item)
    {
        if (count < 5)
        {
            t[count] = item;
            count++;
        }
        else
        {
            Console.WriteLine("Overflow exists");
        }
    }
}
```

```

    public void displayItem()
    {
        for (int i = 0; i < count; i++)
        {
            Console.WriteLine("Item at index {0} is {1}",i,t[i]);
        }
    }
}

class GenericEx
{
    static void Main()
    {
        Test<int> obj = new Test<int>();
        obj.addItem(10);
        obj.addItem(20);
        obj.addItem(30);
        obj.addItem(40);
        obj.addItem(50);
        //obj.addItem(60); //overflow exists
        obj.displayItem();
        Console.ReadKey();
    }
}

```

### **Output**

```

Item at index 0 is 10
Item at index 1 is 20
Item at index 2 is 30
Item at index 3 is 40
Item at index 4 is 50

```

### **Generic Methods**

The objective of this example is to build a swap method that can operate on any possible data type (value-based or reference-based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference via ref keyword.



```

using System.Collections.Generic;
class Program
{
    //Generic method
    static void Swap<T>(ref T a, ref T b)
    {
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
    static void Main(string[] args)
    {
        // Swap of two integers.
        int a = 40, b = 60;
        Console.WriteLine("Before swap: {0}, {1}", a, b);

        Swap<int>(ref a, ref b);

        Console.WriteLine("After swap: {0}, {1}", a, b);

        Console.ReadLine();
    }
}

```

### Output

Before swap: 40, 60  
After swap: 60, 40

## Dictionary

Dictionaries are also known as maps or hash tables. It represents a data structure that allows you to access an element based on a key. One of the significant features of a dictionary is faster lookup; you can add or remove items without the performance overhead.

.Net offers several dictionary classes, for instance **Dictionary<TKey, TValue>**. The type parameters TKey and TValue represent the types of the keys and the values it can store, respectively.

```

using System.Collections.Generic;
public class Program
{
    static void Main(string[] args)
    {
        //define Dictionary collection
        Dictionary<int, string> dObj = new Dictionary<int,
            string>(5);
        //add elements to Dictionary
        dObj.Add(1, 1, "Tom");
        dObj.Add(2, "John");
        dObj.Add(3, "Maria");
    }
}

```

```

dObj.Add(4, "Max");
dObj.Add(5, "Ram");

//print data
for (int i = 1; i <= dObj.Count; i++)
{
    Console.WriteLine(dObj[i]);
}
Console.ReadKey();
}
}

```

## Queues

Queues are a special type of container that ensures the items are being accessed in a FIFO (first in, first out) manner. Queue collections are most appropriate for implementing messaging components. We can define a Queue collection object using the following syntax:

```
Queue qObj = new Queue();
```

The Queue collection property, methods and other specification definitions are found under the Sysyem.Collection namespace. The following table defines the key members;

System.Collection.Queue Members	Definition
Enqueue()	Add an object to the end of the queue.
Dequeue()	Removes an object from the beginning of the queue.
Peek()	Return the object at the beginning of the queue without removing it.

```

using System.Collections.Generic;
class Program {
    static void Main(string[] args) {
        Queue < string > queue1 = new Queue < string > ();
        queue1.Enqueue("MCA");
        queue1.Enqueue("MBA");
        queue1.Enqueue("BCA");
        queue1.Enqueue("BBA");

        Console.WriteLine("The elements in the queue are:");
        foreach(string s in queue1) {
            Console.WriteLine(s);
        }

        queue1.Dequeue(); //Removes the first element that
                           enter in the queue here the first element is MCA
        queue1.Dequeue(); //Removes MBA
    }
}

```

```

        Console.WriteLine("After removal the elements in the
                           queue are:");
        foreach(string s in queue1) {
            Console.WriteLine(s);
        }
    }
}

```

## Stacks

A Stack collection is an abstraction of LIFO (last in, first out). We can define a Stack collection object using the following syntax:

```
Stack qObj = new Stack();
```

The following table illustrates the key members of a stack;

System.Collection.Stack Members	Definition
Contains()	Returns true if a specific element is found in the collection.
Clear()	Removes all the elements of the collection.
Peek()	Previews the most recent element on the stack.
Push()	It pushes elements onto the stack.
Pop()	Return and remove the top elements of the stack.

```

class Program {
    static void Main(string[] args) {
        Stack < string > stack1 = new Stack < string > ();
        stack1.Push("*****");
        stack1.Push("MCA");
        stack1.Push("MBA");
        stack1.Push("BCA");
        stack1.Push("BBA");
        stack1.Push("*****");
        stack1.Push("***Courses***");
        stack1.Push("*****");
        Console.WriteLine("The elements in the stack1 are
                           as:");
        foreach(string s in stack1) {
            Console.WriteLine(s);
        }

        //For remove/or pop the element pop() method is used
        stack1.Pop();
        stack1.Pop();
        stack1.Pop();
        Console.WriteLine("After removal/or pop the element
                           the stack is as:");
        //the element that inserted in last is remove firstly.
    }
}

```

```

        foreach(string s in stack1) {
            Console.WriteLine(s);
        }
    }
}

```

## List

List<T> class in C# represents a strongly typed list of objects. List<T> provides functionality to create a list of objects, find list items, sort list, search list, and manipulate list items. In List<T>, T is the type of objects.

### Adding Elements

```

// Dynamic ArrayList with no size limit
List<int> numberList = new List<int>();
numberList.Add(32);
numberList.Add(21);
numberList.Add(45);
numberList.Add(11);
numberList.Add(89);
// List of string
List<string> authors = new List<string>(5);
authors.Add("Mahesh Chand");
authors.Add("Chris Love");
authors.Add("Allen O'Neill");
authors.Add("Naveen Sharma");
authors.Add("Monica Rathbun");
authors.Add("David McCarter");

// Collection of string
string[] animals = { "Cow", "Camel", "Elephant" };
// Create a List and add a collection
List<string> animalsList = new List<string>();
animalsList.AddRange(animals);
foreach (string a in animalsList)
    Console.WriteLine(a);

```

### Remove Elements

```

// Remove an item
authors.Remove("New Author1");

// Remove 3rd item
authors.RemoveAt(3);

// Remove all items
authors.Clear();

```

### Sorting

```

authors.Sort();

```

### Other Methods

```
authors.Insert(1,"Shaijal"); //insert item at index 1  
authors.Count;           //returns total items
```

### Array List

C# ArrayList is a non-generic collection. The ArrayList class represents an array list and it can contain elements of any data types. The ArrayList class is defined in the System.Collections namespace. An ArrayList is dynamic array and grows automatically when new items are added to the collection.

```
ArrayList personList = new ArrayList();
```

### Insertion

```
personList.Add("Sandeep");
```

### Removal

```
// Remove an item  
personList.Remove("New Author1");
```

```
// Remove 3rd item  
personList.RemoveAt(3);
```

```
// Remove all items  
personList.Clear();
```

### Sorting

```
personList.Sort();
```

### Other Methods

```
personList.Insert(1,"Shaijal"); //insert item at index 1  
personList.Count;           //returns total items
```