Table of Contents

Unit -3 Creating Types in C#

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

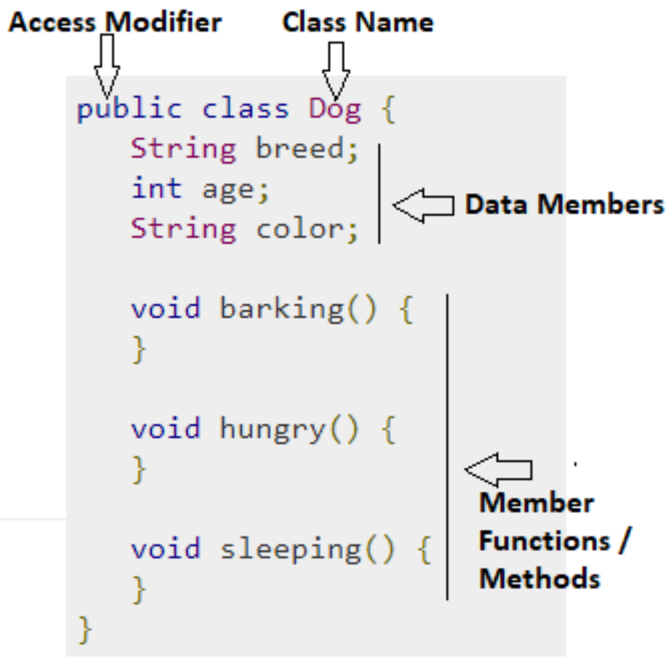
Classes

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, { }.

<u>General form of a class is shown below:</u>



```
public class Dog {
    String breed;
    int age;
    String color;

    void barking() {
    }

    void hungry() {
    }

    void sleeping() {
    }
}
```

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:

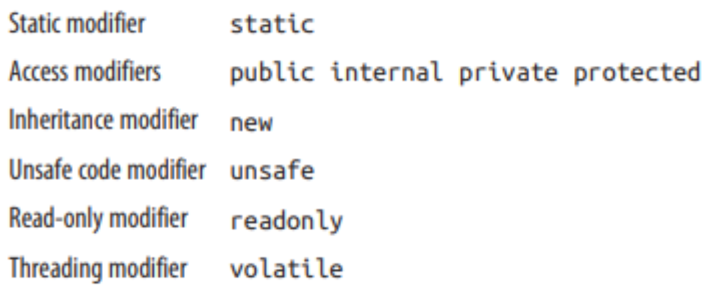
Tree t = new Tree ().

Fields

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

```
class Person
{
    public string name;
    public int age = 30;
}
```

Fields allow the following modifiers:



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

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.

```
static readonly int pi=3.14;
```

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 com- prises its name and parameter types in order (but not the parameter names, nor the return type).

Methods allow the following modifiers:

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

Expression-bodied methods

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

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

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

```
void Test (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 Add (int x) {……}
void Add (double x) {……}
void Add (int x, float y) {……}
void Add (float x, float 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 Add (int x) {……}
float Add (int x) {……}     //compile-time error

void Add (int[] x) {……}
void Add (params int x) {……}  //compile-time error
```

Constructors

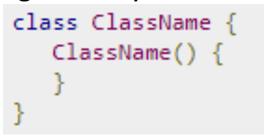
A *constructor* in C# is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- 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:

```
using System;
class Cons
    {
        private int sum;
        //instance constructor
        public Cons(int a, int b)
        {
            sum = a + b;
            Console.WriteLine("Sum = {0}",sum);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Cons(10,20);  //constructor is called
        }
    }
```

Output:

```
Sum = 30
```

Instance constructors allow the following modifiers:

public internal private protected

Overloaded Constructors

A class or struct may overload constructors. When more than one constructor with the same name is defined in the same class, they are called overloaded, if the parameters are different for each constructor.

```
using System;
class Cons
    {
        private int sum;

        public Cons(int a, int b)
        {
            sum = a + b;
            Console.WriteLine("Sum = {0}",sum);
        }

        public Cons(int a, int b, int c)
        {
            sum = a + b + c;
            Console.WriteLine("Sum = {0}", sum);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Cons(10,20);     //prints 30
            new Cons(10,20,30);  //prints 60
        }
    }
```

Output:

```
Sum = 30
Sum = 60
```

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

Example

```
using System;
class Cons
    {
        public Cons()
        {
            Console.WriteLine("I am inside Constructor");
        }

        static Cons()  //will be called only once
        {
            Console.WriteLine("I am inside Static Constructor");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Cons();  //both consturctor called
            new Cons();
            //static constructor will not be called now
        }
    }
```

Output:

```
I am inside Static Constructor
I am inside Constructor
I am inside Constructor
```

Destructor

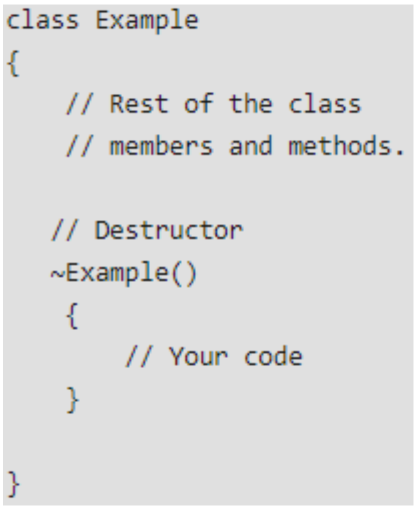
Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's

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
```

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

he 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; }
}
```

Example of Automatic Property

```
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();
    }
}
```

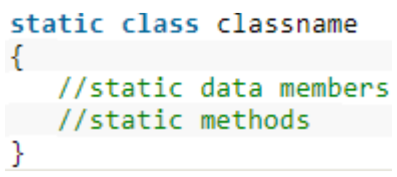
Static Classes

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.

<u>Syntax of static class</u>



```
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.

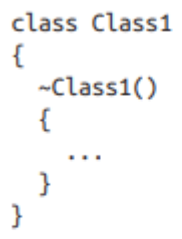
<u>Example</u>

```
using System;
static class Person
    {
        //static data members
        public static int id;
        public static string name;

        //static method
        public static void Display()
        {
            Console.Write("Id={0}\tName={1}",id,name);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //object creation not required
            //accessing data members
            Person.id = 001;
            Person.name = "Ram";
            //accessing methods
            Person.Display();
        }
    }
```

```
Output:
Id=001    Name=Ram
```

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()
  {
    ...
  }
}
```

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

<u>Here is an example of declaring and calling struct:</u>

```
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**
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified

protected	Access is limited to within the class definition and any class that inherits from the class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	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.
private protected	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
```

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 C# 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 C# 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.

Important facts about inheritance in C#

- **Default Superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass(single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because C# does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by C#.

- **Inheriting Constructors:** A subclass inherits all the members (fields, methods) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has properties(get and set methods) for accessing its private fields, then a subclass can inherit.

The syntax of C# 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.

1. Single Inheritance

Single Inheritance refers to a child and parent class relationship where a class extends the another class. In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

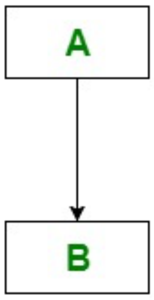


Fig: Single Inheritance

Syntax of Single Inheritance:

```
    class A   //base class
    {
        //data members & methods
    }

    class B : A   //derived class
    {
        //data members & methods
    }
```

Example:

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

    class B : A
    {
        public void test()
        {
            Console.WriteLine("Value of a is: "+a);
            Console.WriteLine("Value of b is: " +b);
        }
    }

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

Output:

```
Value of a is: 10
Value of b is: 5
```

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.
In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

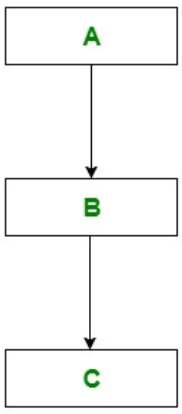


Fig: Multilevel Inheritance

Syntax of Multilevel Inheritance:

```
    class A   //base class
    {
        //data members & methods
    }

    class B : A   //derived class
        //also base class of C
    {
        //data members & methods
    }

    class C : B   //derived class
    {
        //data members & methods
    }
```

Example:

```
using System;
class A
    {
        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 B : A
    {
        public void Add()
        {
            base.c = base.a + base.b;
            Console.WriteLine("Sum="+base.c);
        }
    }
    class C : B
    {
        public void Sub()
        {
            base.c = base.a - base.b;
            Console.WriteLine("Difference=" + base.c);
        }
    }

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

            Console.ReadLine();
        }
    }
```

Output:

```
Value of a is: 20
Value of b is: 5
Sum=25
Difference=15
```

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.

In this inheritance one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.

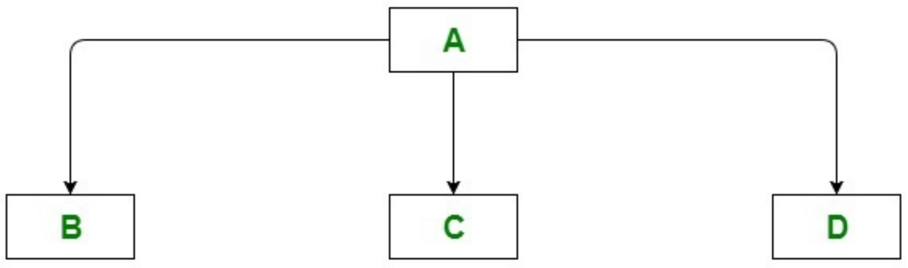


Fig: Hierarchical Inheritance

Syntax of Hierarchical Inheritance:

```
    class A   //base class
    {
        //data members & methods
    }
    class B : A   //derived class
    {
        //data members & methods
    }

    class C : A   //derived class
    {
        //data members & methods
    }
```

Example:

```
using System;
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();
        }
    }
```

Output:

```
Area of Triangle=25
Area of Rectangle=50
```

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.

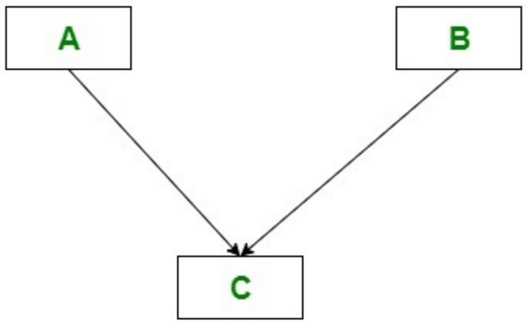


Fig: Multiple Inheritance

Syntax of Multiple Inheritance:

```
    interface A
    {
        //methods (only signature)
    }

    class B
    {
        //data members & methods
    }

    class C : B,A
    {
        //data members & methods
    }
```

Example:

```
using System;
interface A
    {
        //doesn't contain data members
        //method only contain signature
        int CalculateArea();
        int CalculatePerimeter();
    }

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

    class C : A, B
    {
        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)
        {
            C obj = new C();
            //int area=obj.CalculateArea();
            //int peri=obj.CalculatePerimeter();

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

Output:
```
Area of Rectangle=50
Perimeter of Rectangle=50
```

5. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since C# doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In C#, we can achieve hybrid inheritance only through Interfaces.

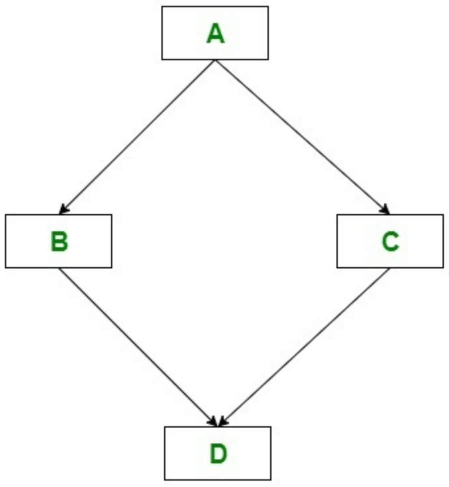


Fig: Hybrid Inheritance

Syntax of Hybrid Inheritance:

```
interface A
    {
        //code here
    }

    class B : A
    {
        //code here
    }

    class C : B
    {
        //code here
    }

    class D : C,A
    {
        //code here
    }
```

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.

Syntax of Interface in C#

```
   interface interface_name
    {
        //method signature
    }

    class class_name : interface_name
    {
        //method implementation
    }
```

Example:

```
using System;
interface A
    {
        // doesn't contain fields
        // only contains method signature
        void GetData(int l,int b);
        int CalculateArea();
        int CalculatePerimeter();
    }

    class B : A
    {
        int l, b;
      //method implementation

      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)
        {
            B obj = new B();
            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.

Syntax of Abstract Class

```
abstract class class_name
    {
        //data members & properties
        //methods
    }
```

Example:

```
using System;
abstract class A
    {
        public void MessageA()
        {
            Console.WriteLine("Running from abstract class");
        }
    }

    class B : A
    {
        public void MessageB()
        {
            Console.WriteLine("Running from another class");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            B obj = new B();
            obj.MessageA();
            obj.MessageB();
        }
    }
```

Output:

```
Running from abstract class
Running from another class
```

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

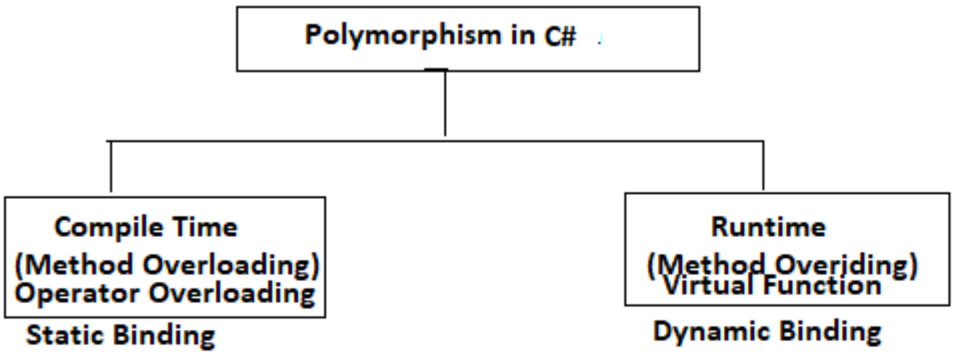
```
using System;
abstract class A
    {
        public abstract int AddData(int a,int b);
    }

    class B : A
    {
        public override int AddData(int a,int b)
        {
            return a + b;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            B obj = new B();
            int res=obj.AddData(20, 30);
            Console.WriteLine("Result is :"+res);
        }
    }
```

Output:

```
Result is : 50
```

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

```
using System;
class Addition
    {
       public void Sum(int a, int b)
        {
            int s = a + b;
            Console.WriteLine("Sum of two integer numbers:" + s);
        }

        public void Sum(double a, double b)
        {
            double d = a + b;
            Console.WriteLine("Sum of two double numbers:" + d);
        }

        public void Sum(int a, int b, int c)
        {
            int t = a + b + c;
            Console.WriteLine("Sum of three integer numbers:" + t);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Addition a = new Addition();
            a.Sum(10, 20);
            a.Sum(10.25,9.23);
            a.Sum(50, 100, 200);
        }
    }
```

Output:

```
Sum of two integer numbers: 30
Sum of two double numbers: 19.48
Sum of three integer numbers: 350
```

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 overriden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overriden 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

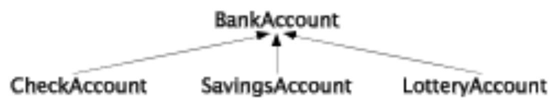
```
class Vir
    {
        public virtual void message()
        {
            Console.WriteLine("This is test");
        }
    }

    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();
        }
    }
```

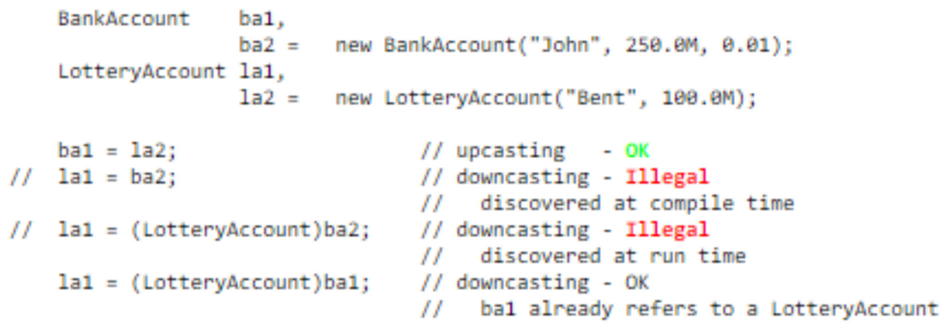
Output:
```
This is test1
```

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
```

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.

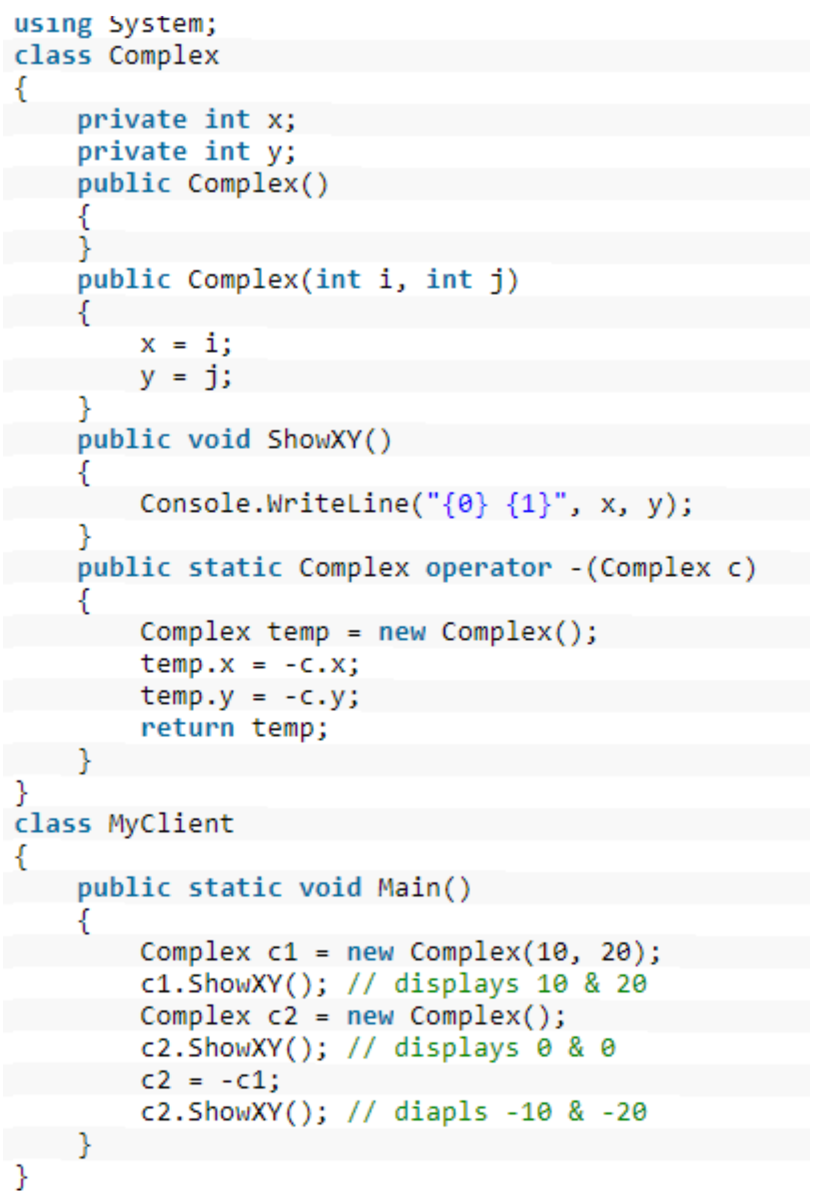
Syntax:
```
access specifier  className  operator Operator_symbol (parameters)
	{
		// Code
	}
```

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(); // dislplays 30 & 50
    }
}
```

Sealed 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.

Features of Sealed Class

1. A sealed class is completely opposite to an abstract class.
2. This sealed class cannot contain abstract methods.
3. It should be the bottom-most class within the inheritance hierarchy.

4. A sealed class can never be used as a base class.
5. This sealed class is specially used to avoid further inheritance.
6. The keyword sealed can be used with classes, instance methods, and properties.

Syntax of sealed class

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

Example 1

```
using System;
sealed class A
    {
       //code here
    }

class B : A  //compiler shows error.
{            //we cannot inherit base class

}
```

Example 2

```
using System;
sealed class Test
    {
        public void Message()
        {
            Console.WriteLine("Running form sealed class.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test obj = new Test();
            obj.Message();
        }
    }
```

Output:

```
Running form sealed class.
```

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.

```
using System;
class Base {
   public virtual void Test() { ... }
}

class Subclass1 : Base {
   public sealed override void Test() { ... }
}

class Subclass2 : Subclass1 {
   public override void Test() { ... } // Does not compile!
   // If `Subclass1.Test` was not sealed, it would've compiled
correctly.
}
```

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();
        }
    }
```

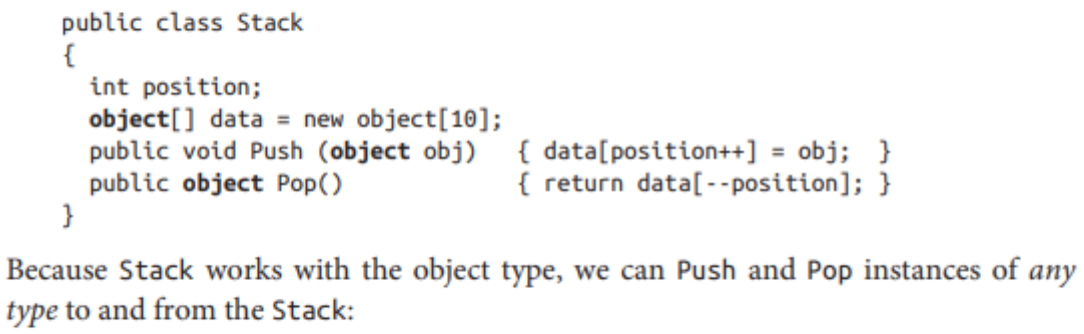
```
Output:
I am inside base class
I am inside derived class
```

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
  }
}
```

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();
    }
}
```

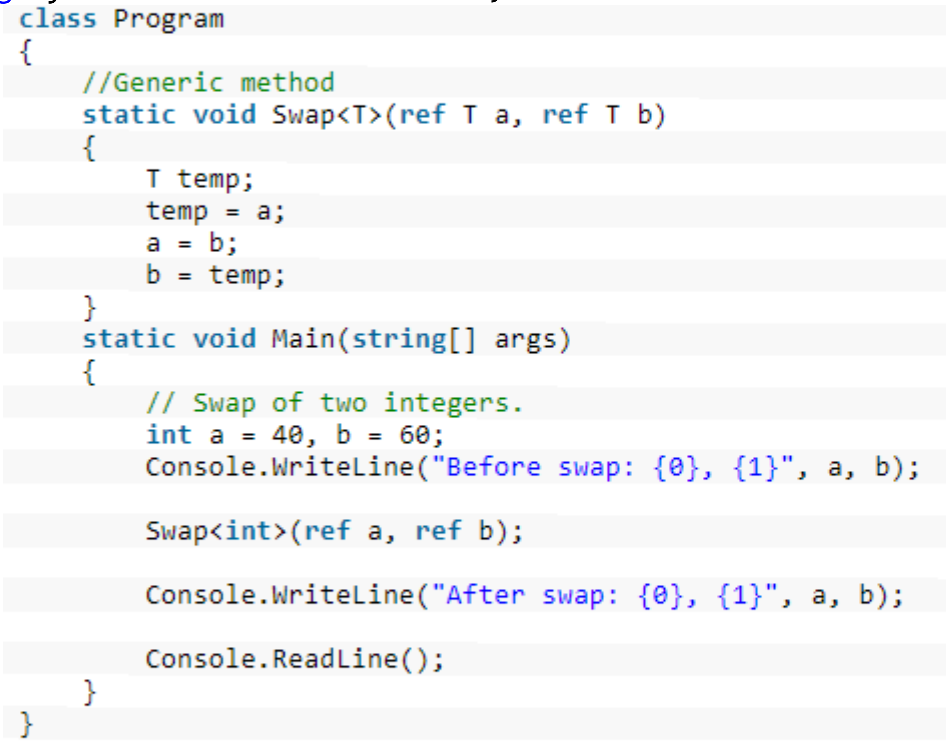
Output

```
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 = newStack < 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
```

Exercise

Objective Questions

1. Which of the following is used to define the member of a class externally?
   a) : b) ::
   c) # d) none of the mentioned

2. Which of the following statements about objects in "C#" is correct?
   a) Everything you use in C# is an object, including Windows Forms and controls
   b) Objects have methods and events that allow them to perform actions
   c) All objects created from a class will occupy equal number of bytes in memory
   d) All of the mentioned

3. "A mechanism that binds together code and data in manipulates, and keeps both safe from outside interference and misuse. In short it isolates a particular code and data from all other codes and data. A well-defined interface controls access to that particular code and data."
   a) Abstraction
   b) Polymorphism
   c) Inheritance
   d) Encapsulation

4. Correct way of declaration of object of the following class is?

   ```
   class name
   ```

   a) name n = new name();
   b) n = name();
   c) name n = name();
   d) n = new name();

5. The data members of a class by default are?
   a) protected, public
   b) private, public
   c) private
   d) public

6. Number of constructors a class can define is?
   a) 1
   b) 2
   c) Any number
   d) None of the mentioned

7. Correct statement about constructors in C#.NET is?
   a) Constructors can be overloaded
   b) Constructors are never called explicitly
   c) Constructors have same name as name of the class
   d) All of the mentioned

8. Which of the following statements is correct about constructors in C#.NET?
   a) A constructor cannot be declared as private
   b) A constructor cannot be overloaded
   c) A constructor can be a static constructor
   d) None of the mentioned

9. What is the return type of constructors?
   a) int
   b) float
   c) void
   d) none of the mentioned

10. Which method has the same name as that of its class?
   a) delete
   b) class
   c) constructor
   d) none of the mentioned

11. Which operator among the following signifies the destructor operator?
   a) ::
   b) :
   c) ~
   d) &

12. Operator used to free the memory when memory is allocated?
   a) new
   b) free
   c) delete
   d) none of the mentioned

13. Select wrong statement about destructor in C#?
   a) A class can have one destructor only
   b) Destructors cannot be inherited or overloaded
   c) Destructors can have modifiers or parameters
   d) All of the mentioned

14. What is the return type of destructor?
   a) int
   b) void
   c) float
   d) none of the mentioned

15. How many values does a function return?
   a) 0
   b) 2
   c) 1
   d) any number of values

16. The capability of an object in Csharp to take number of different forms and hence display behaviour as according is known as ___________
   a) Encapsulation
   b) Polymorphism

c) Abstraction
d) None of the mentioned

17. Which of the following keyword is used to change data and behavior of a base class by replacing a member of the base class with a new derived member?
a) Overloads
b) Overrides
c) new
d) base

18. Correct way to overload +operator?
a) public sample operator + (sample a, sample b)
b) public abstract operator + (sample a,sample b)
c) public static sample operator + (sample a, sample b)
d) all of the mentioned

19. Wrong statement about run time polymorphism is?
a) The overridden base method should be virtual, abstract or override
b) An abstract method is implicitly a virtual method
c) An abstract inherited property cannot be overridden in a derived class
d) Both override method and virtual method must have same access level modifier

20. Choose the wrong statement about structures in C#.NET?
a) Structures can be declared within a procedure
b) Structures can implement an interface but they cannot inherit from another structure
c) Structure members cannot be declared as protected
d) A structure cannot be empty

21. When does a structure variable get destroyed?
a) When no reference refers to it, it will get garbage collected
b) Depends on whether it is created using new or without new operator
c) As variable goes out of the scope
d) Depends on either we free its memory using free() or delete()

22. What will be the output of the following C# code?

```
{
     struct abc
     {
         int i;
     }
     class Program
     {
         static void Main(string[] args)
         {
             abc x = new abc();
             abc z;
             x.i = 10;
             z = x;
             z.i = 15;
```

```
                console.Writeline(x.i + "   " + y.i)
            }
        }
    }
```

a) 10 10
b) 10 15
c) 15 10
d) 15 15

23. The modifier used to define a class which does not have objects of its own but acts as a base class for its subclass is?
a) Sealed
b) Static
c) New
d) Abstract

24. Choose the correct statements among the following:
a) An abstract method does not have implementation
b) An abstract method can take either static or virtual modifiers
c) An abstract method can be declared only in abstract class
d) All of the mentioned

25. Which of the following modifiers is used when an abstract method is redefined by a derived class?
a) Overloads
b) Override
c) Base
d) Virtual

26. Which among the following cannot be used as a datatype for an enum in C#.NET?
a) short
b) double
c) int
d) all of the mentioned

27. Wrong statement about enum used in C#.NET is?
a) An enum can be declared inside a class
b) An object cannot be assigned to an enum variable
c) An enum can be declared outside a class
d) An enum can have Single and Double values

28. Which keyword is used for correct implementation of an interface in C#.NET?
a) interface
b) Interface
c) intf
d) Intf

29. Which of the following is the correct way of implementing an interface addition by class maths?

a) class maths : addition {}
b) class maths implements addition {}
c) class maths imports addition {}
d) none of the mentioned

30. Access specifiers which can be used for an interface are?
a) Public
b) Protected
c) Private
d) All of the mentioned

31. The number of levels of inheritance are?
a) 5
b) 4
c) 3
d) 2

32. Select the class visibility modifiers among the following:
a) Private, protected, public, internal
b) Private, protected, public, internal, protected internal
c) Private, protected, public
d) All of the mentioned

33. In Inheritance concept, which of the following members of base class are accessible to derived class members?
a) static
b) protected
c) private
d) shared

34. A class member declared protected becomes member of subclass of which type?
a) public member
b) private member
c) protected member
d) static member

35. Which form of inheritance is not supported directly by C# .NET?
a) Multiple inheritance
b) Multilevel inheritance
c) Single inheritance
d) Hierarchical inheritance

36. If no access modifier for a class is specified, then class accessibility is defined as?
a) public
b) protected
c) private
d) internal

37. The process of defining two or more methods within the same class that have same name but different parameters list?
    a) Method overloading
    b) Method overriding
    c) Encapsulation
    d) None of the mentioned

38. Which of these can be overloaded?
    a) Constructors
    b) Methods
    c) Both Constructors & Methods
    d) None of the mentioned

39. Operators that can be overloaded are?
    a) ||
    b) ‘+=’
    c) +
    d) []

40. Correct method to define + operator is?
    a) public sample operator +(int a, int b)
    b) public abstract operator +(int a, int b)
    c) public static sample operator +(int a, int b)
    d) public abstract sample operator +(int a, int b)

41. Which keyword is used to declare a base class method while performing overriding of base class methods?
    a) this
    b) virtual
    c) override
    d) extend

42. The process of defining a method in a subclass having same name & type signature as a method in its superclass is known as?
    a) Method overloading
    b) Method overriding
    c) Method hiding
    d) None of the mentioned

43. Which of the given modifiers can be used to prevent Method overriding?
    a) Static
    b) Constant
    c) Sealed
    d) final

44. Select the correct statement from the following?
    a) Static methods can be a virtual method
    b) Abstract methods can be a virtual method
    c) When overriding a method, the names and type signatures of the override method must be the same as the virtual method that is being overridden
    d) We can override virtual as well as nonvirtual methods

45. Which of the following cannot be used to declare a class as a virtual?
    a) Methods
    b) Properties
    c) Events
    d) Fields

46. Where the properties can be declared?
    a) Class
    b) Struct
    c) Interface
    d) All of the mentioned

47. Select the modifiers which can be used with the properties?
    a) Private
    b) Public
    c) Protected Internal
    d) All of the mentioned

48. Select the correct statement about properties of read and write in C#.NET?
    a) A property can simultaneously be read or write only
    b) A property cannot be either read only or write only
    c) A write only property will only have get accessor
    d) A read only property will only have get accessor

49. What is meant by the term generics?
    a) parameterized types
    b) class
    c) structure
    d) interface

50. Select the type argument of an open constructed type?
    a) Gen<int>
    b) Gen<T>
    c) Gen<>
    d) None of the mentioned

51. Which among the given classes is present in System.Collection.Generic.namespace?
    a) Stack
    b) Tree
    c) Sorted Array
    d) All of the mentioned

52. Which of these type parameters is used for generic methods to return and accept any type of object?
    a) K
    b) N
    c) T
    d) V

53. Which of the given statements are valid about generics in .NET Framework?
   a) generics are useful in collection classes in .NET framework
   b) generics delegates are not allowed in C#.NET
   c) generics is a not language feature
   d) all of the mentioned

54. Which statement is valid for the following C# code snippet?

```
public class Generic<T>
{
    public T Field;
}
class Program
{
    static void Main(string[] args)
    {
        Generic<String> g = new Generic<String>();
        g.Field = "Hi";
        Console.WriteLine(g.Field);
    }
}
```

   a) Compile time error
   b) Generic being a keyword cannot be used as a class name
   c) Runtime error
   d) Code runs successfully

55. Choose the keyword which declares the indexer?
   a) base
   b) this
   c) super
   d) extract

Answer Key:

1. b	2. d	3. d	4. a	5. c
6. c	7. d	8. c	9. d	10. c
11. c	12. c	13. c	14. d	15. c
16.b	17.c	18.d	19.c	20.a
21.c	22.b	23.d	24.d	25.b
26.b	27.d	28.a	29.a	30.a
31.b	32.b	33.b	34.d	35.a
36.c	37.a	38.c	39.c	40.c
41.b	42.b	43.c	44.c	45.d
46.d	47.d	48.d	49.a	50.c
51.a	52.c	53.a	54.d	55.b

Subjective Questions

1. What do you mean by class and object? Explain with example.
2. Create a simple class named Person that contains basic information like name, age, gender, etc. Your class should also contain functions/methods for storing and displaying data.
3. Explain read-only modifier with example.
4. Differentiate constructor and destructor with example.
5. What do you mean by static constructor? How it is different than other types of constructors? Explain with example.
6. Explain the use of this keyword in C# with example.
7. What do you mean by properties in C#? Explain with example.
8. Explain automatic property in C# with example.
9. Explain indexers with the help of suitable example.
10. What do you mean by static class? Explain with suitable program.
11. Differentiate single and multilevel inheritance with example.
12. Differentiate multilevel and hierarchical inheritance with example.
13. Differentiate multilevel and multiple inheritance with example.
14. What is multiple inheritance? How it is achieved in C#? Explain with suitable example.
15. What do you mean by interface? Explain use of interface in C# with example.
16. What do you mean by abstraction? Explain abstract class and methods with example.
17. Differentiate static and dynamic binding with example.
18. What do you mean by method overloading? Explain with example.
19. What do you mean by method overriding? Explain the use of virtual function for method overriding.
20. What is operator overloading? What are different operators that can be overloaded in C#? Explain.
21. Write a program in C# to overload unary operators.
22. Write a program in C# to overload binary operators.
23. What do you mean by sealed class? Explain with example.
24. What do you mean by sealed method? Explain with example.
25. Explain the use of base keyword in C# with example.
26. Differentiate class and struct. Write a C# program using struct.
27. Explain different access supported by C# with example.
28. What id enum? Why it is used in C#? Explain with the help of program.
29. What are different types of generic classes supported in C#? Explain with example.
30. What do you mean by constructor? Explain different types of constructors in detail along with programs.
31. What do you mean by inheritance? Explain different types of inheritance along with suitable programs.
32. What is polymorphism? Explain compile time and runtime polymorphism with program in detail.
33. What do you mean by generics? Write a C# program to create generic class and generic methods.