1. Write a C# program to demonstrate five types of constructor in C#.

Theory: A constructor is a special member of a class that is automatically called when an object is created. It is mainly used to initialize data members of a class. C# supports different types of constructors such as: Default Constructor – Initializes object with default values. Parameterized Constructor – Accepts parameters to initialize data members. Copy Constructor– Copies values from one object to another. Static Constructor – Initializes static members of the class and runs only once. Private Constructor – Restricts object creation from outside the class.

In [7]:
```
// LAb 1

using System;

class Student
{
    public int Id;
    public string Name;
    public static string College;

    static Student()
    {
        College = "ABC Engineering College";
        Console.WriteLine("Static Constructor called");
    }

    public Student()
    {
        Id = 0;
        Name = "Not Assigned";
        Console.WriteLine("Default Constructor called");
    }

    public Student(int id, string name)
    {
        Id = id;
        Name = name;
        Console.WriteLine("Parameterized Constructor called");
    }

    public Student(Student s)
    {
        Id = s.Id;
        Name = s.Name;
        Console.WriteLine("Copy Constructor called");
    }

    public void Display()
    {
        Console.WriteLine("Id: " + Id + ", Name: " + Name + ", College: " + College);
    }
}

class Singleton
{
    private static Singleton instance;
```

```
    private Singleton()
    {
        Console.WriteLine("Private Constructor called");
    }

    public static Singleton GetInstance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}

// Top-level code
Student s1 = new Student();
s1.Display();

Student s2 = new Student(101, "Rahul");
s2.Display();

Student s3 = new Student(s2);
s3.Display();

Singleton obj1 = Singleton.GetInstance();
Singleton obj2 = Singleton.GetInstance();

// Student details (added at the end)
Console.WriteLine("\nStudent Details:");
Console.WriteLine("\nName: Mohit Tharu");
Console.WriteLine("\nRoll no: 117");

Console.ReadLine();
```

```
Static Constructor called
Default Constructor called
Id: 0, Name: Not Assigned, College: ABC Engineering College
Parameterized Constructor called
Id: 101, Name: Rahul, College: ABC Engineering College
Copy Constructor called
Id: 101, Name: Rahul, College: ABC Engineering College
Private Constructor called

Student Details:

Name: Mohit Tharu

Roll no: 117
```

2. WAP in C# to demonstrate concept of Auto Property and Read-Only Property.

Theory: Properties in C# provide a flexible way to read, write, or compute values of private fields. Auto Property allows automatic creation of getter and setter without explicitly declaring a backing variable. Read-Only Property allows data to be read but not modified after initialization.These properties improve code readability and data security.

In [11]:
```
using System;
```

```
class Student
{
    // Auto Properties
    public string Name { get; set; }
    public int RollNo { get; set; }

    // Read-Only Property
    public string College { get; }

    // Constructor
    public Student(string name, int rollNo)
    {
        Name = name;
        RollNo = rollNo;
        College = "Patan Multiple Campus";
    }

    public void Display()
    {
        Console.WriteLine("Student Details:");
        Console.WriteLine("Name: " + Name);
        Console.WriteLine("Roll No: " + RollNo);
        Console.WriteLine("College: " + College);
    }
}

// Top-level code
Student s = new Student("Mohit Tharu", 117);
s.Display();

Console.ReadLine();
```

```
Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

3. WAP in C# to demonstrate Jagged Array.

Theory:A jagged array is an array of arrays in which each inner array can have a different size. Unlike multidimensional arrays, jagged arrays provide flexibility in storing uneven data and are commonly used when rows contain varying numbers of elements.

In [19]:
```
using System;

// Declaring Jagged Array
int[][] marks = new int[3][];

// Initializing Jagged Array
marks[0] = new int[] { 80, 85, 90 };
marks[1] = new int[] { 75, 88 };
marks[2] = new int[] { 92, 89, 95, 91 };

// Displaying Jagged Array elements
Console.WriteLine("Jagged Array Elements:");

for (int i = 0; i < marks.Length; i++)
{
    Console.Write("Row " + (i + 1) + ": ");
    for (int j = 0; j < marks[i].Length; j++)
```

```
    {
        Console.Write(marks[i][j] + " ");
    }
    Console.WriteLine();
}

// Student details (added at the end)
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Jagged Array Elements:
Row 1: 80 85 90
Row 2: 75 88
Row 3: 92 89 95 91

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

4. WAP to demonstrate Indexer in C#:a) When index is of int type b) When index is of other than int type

Theory: An Indexer allows an object of a class to be accessed like an array. It uses the this keyword and can accept different types of parameters. Int type indexer allows access using numeric index values. Non-int type indexer (like string) allows access using keys or names. Indexers improve data encapsulation and make object access easier.

In [24]:
```
using System;

// (a) Indexer with int type
class IntIndexer
{
    private int[] numbers = new int[5];

    public int this[int index]
    {
        get { return numbers[index]; }
        set { numbers[index] = value; }
    }
}

// (b) Indexer with non-int type (string)
class StringIndexer
{
    private string[] names = new string[3];

    public string this[string key]
    {
        get
        {
            if (key == "first") return names[0];
            else if (key == "second") return names[1];
```

```
            else if (key == "third") return names[2];
            else return "Invalid Key";
        }
        set
        {
            if (key == "first") names[0] = value;
            else if (key == "second") names[1] = value;
            else if (key == "third") names[2] = value;
        }
    }
}

// Top-level code
IntIndexer obj1 = new IntIndexer();
obj1[0] = 10;
obj1[1] = 20;
obj1[2] = 30;

Console.WriteLine("Indexer with int type:");
Console.WriteLine(obj1[0]);
Console.WriteLine(obj1[1]);
Console.WriteLine(obj1[2]);

StringIndexer obj2 = new StringIndexer();
obj2["first"] = "Apple";
obj2["second"] = "Banana";
obj2["third"] = "Mango";

Console.WriteLine("\nIndexer with non-int type:");
Console.WriteLine(obj2["first"]);
Console.WriteLine(obj2["second"]);
Console.WriteLine(obj2["third"]);

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Indexer with int type:
10
20
30

Indexer with non-int type:
Apple
Banana
Mango

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

5. WAP to demonstrate:

```
    # a) The use of base keyword to access base class fields
    # b) The use of base keyword to call base class methods
    # c) The use of base keyword to call base class constructor
```

Theory: The base keyword in C# is used to access members of the base (parent) class from a derived (child) class. Uses of base keyword: Access base class fields when they are hidden by derived class members. Call base class methods from the derived class.

In [30]:
```
using System;

// Base class
class Person
{
    protected string name;

    // Base class constructor
    public Person(string name)
    {
        this.name = name;
        Console.WriteLine("Base class constructor called");
    }

    // Base class method
    public void ShowName()
    {
        Console.WriteLine("Name from base class: " + name);
    }
}

// Derived class
class Student : Person
{
    public int RollNo;

    // Derived class constructor calling base class constructor
    public Student(string name, int rollNo) : base(name)
    {
        RollNo = rollNo;
        Console.WriteLine("Derived class constructor called");
    }

    public void Display()
    {
        // (a) Access base class field using base keyword
        Console.WriteLine("Accessing base class field using base keyword: " + base.name);

        // (b) Call base class method using base keyword
        base.ShowName();

        Console.WriteLine("Roll No: " + RollNo);
    }
}

// Top-level code
Student s = new Student("Mohit Tharu", 117);
s.Display();

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Base class constructor called
Derived class constructor called
Accessing base class field using base keyword: Mohit Tharu
Name from base class: Mohit Tharu
Roll No: 117

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

6. Program to show a) method overriding and method hiding/shadowing in C# b) dynamic polymorphism using method overriding.

Theory: Method Overriding occurs when a derived class provides a new implementation of a base class method using virtual and override keywords. Method Hiding (Shadowing) occurs when a derived class defines a method with the same name as the base class method using the new keyword. Dynamic Polymorphism means that the method call is resolved at runtime based on the object type, achieved using method overriding.

In [33]:
```
using System;

// Base class
class Animal
{
    public virtual void Sound()
    {
        Console.WriteLine("Animal makes a sound");
    }

    public void Eat()
    {
        Console.WriteLine("Animal eats food");
    }
}

// Derived class
class Dog : Animal
{
    // Method Overriding
    public override void Sound()
    {
        Console.WriteLine("Dog barks");
    }

    // Method Hiding / Shadowing
    public new void Eat()
    {
        Console.WriteLine("Dog eats bones");
    }
}

// Top-level code
Animal a1 = new Animal();
Animal a2 = new Dog();   // Dynamic polymorphism
Dog d = new Dog();
```

```
Console.WriteLine("Method Overriding & Dynamic Polymorphism:");
a1.Sound();   // Base class method
a2.Sound();   // Derived class method (runtime binding)

Console.WriteLine("\nMethod Hiding / Shadowing:");
a2.Eat();     // Base class method
d.Eat();      // Derived class method

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Method Overriding & Dynamic Polymorphism:
Animal makes a sound
Dog barks

Method Hiding / Shadowing:
Animal eats food
Dog eats bones

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

7. WAP to illustrate the concept of a) Abstract class b) Interface c) Multiple inheritance using interface in C#

Theory: An abstract class contains abstract methods (without body) and non-abstract methods. It cannot be instantiated and must be inherited. An interface contains only method declarations. It supports full abstraction. Multiple inheritance is not supported using classes in C#, but it is achieved using multiple interfaces.

In [36]:
```
using System;

// (a) Abstract class
abstract class Shape
{
    public abstract void Draw();

    public void Info()
    {
        Console.WriteLine("This is a shape");
    }
}

// (b) Interface
interface IPrintable
{
    void Print();
}
```

```
// Another interface (for multiple inheritance)
interface IColor
{
    void Color();
}

// (c) Multiple inheritance using interface
class Circle : Shape, IPrintable, IColor
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }

    public void Print()
    {
        Console.WriteLine("Printing Circle");
    }

    public void Color()
    {
        Console.WriteLine("Circle color is Red");
    }
}

// Top-level code
Circle c = new Circle();
c.Info();     // Abstract class method
c.Draw();     // Abstract method implementation
c.Print();    // Interface method
c.Color();    // Multiple inheritance using interface

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
This is a shape
Drawing Circle
Printing Circle
Circle color is Red

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

8. WAP program that contains: a) Structure (struct) b) Enumeration (enum) c) Partial class

Theory: A Structure (struct) is a value type used to group related variables of different data types. An Enumeration (enum) is a user-defined data type that consists of named constants. A Partial class allows a class to be divided into multiple parts, which can be defined in the same or different files.

In [39]:
```
using System;
```

```
// (a) Structure
struct StudentStruct
{
    public string Name;
    public int RollNo;

    public void Display()
    {
        Console.WriteLine("Struct Student:");
        Console.WriteLine("Name: " + Name);
        Console.WriteLine("Roll No: " + RollNo);
    }
}

// (b) Enumeration
enum Department
{
    ComputerScience,
    InformationTechnology,
    Electronics
}

// (c) Partial class (Part 1)
partial class Student
{
    public string Name;
    public int RollNo;
}

// Partial class (Part 2)
partial class Student
{
    public void Show()
    {
        Console.WriteLine("\nPartial Class Student:");
        Console.WriteLine("Name: " + Name);
        Console.WriteLine("Roll No: " + RollNo);
    }
}

// Top-level code

// Using Structure
StudentStruct ss;
ss.Name = "Mohit Tharu";
ss.RollNo = 117;
ss.Display();

// Using Enumeration
Department dept = Department.ComputerScience;
Console.WriteLine("\nEnum Department:");
Console.WriteLine("Department: " + dept);

// Using Partial Class
Student s = new Student();
s.Name = "Mohit Tharu";
s.RollNo = 117;
s.Show();

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
```

```
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Struct Student:
Name: Mohit Tharu
Roll No: 117

Enum Department:
Department: ComputerScience

Partial Class Student:
Name: Mohit Tharu
Roll No: 117

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

9. WAP to illustrate the concept of: a) Delegate b) Multicast delegate c) Func Delegate d) Action Delegate e) Anonymous Method f) Event in C#.

Theory: Delegate: A type that references a method with a specific signature. It allows methods to be passed as parameters. Multicast Delegate: A delegate that references multiple methods and calls them in sequence. Func Delegate: A generic delegate that returns a value. Action Delegate: A generic delegate that does not return a value. Anonymous Method: A method without a name, defined inline using delegate keyword. Event: A mechanism that allows a class to notify other classes when something happens, typically using delegates.

In [44]:
```
using System;

// (a) Delegate
delegate void MyDelegate(string message);

// Publisher class for Event
class EventDemo
{
    // (f) Event
    public event MyDelegate MyEvent;

    public void RaiseEvent()
    {
        if (MyEvent != null)
            MyEvent("Event has been triggered");
    }
}

class ProgramDemo
{
    // Methods for delegate
    public static void Method1(string msg)
    {
        Console.WriteLine("Method1: " + msg);
    }
```

```
    public static void Method2(string msg)
    {
        Console.WriteLine("Method2: " + msg);
    }
}

// Top-level code

// (a) Delegate
MyDelegate d1 = ProgramDemo.Method1;
d1("Simple Delegate");

// (b) Multicast Delegate
MyDelegate d2 = ProgramDemo.Method1;
d2 += ProgramDemo.Method2;
Console.WriteLine("\nMulticast Delegate:");
d2("Hello");

// (c) Func Delegate
Func<int, int, int> add = (a, b) => a + b;
Console.WriteLine("\nFunc Delegate Result: " + add(10, 20));

// (d) Action Delegate
Action<string> action = msg => Console.WriteLine("Action Delegate: " + msg);
action("Welcome");

// (e) Anonymous Method
MyDelegate anon = delegate (string msg)
{
    Console.WriteLine("Anonymous Method: " + msg);
};
anon("Hello Anonymous");

// (f) Event
EventDemo ev = new EventDemo();
ev.MyEvent += ProgramDemo.Method1;
Console.WriteLine("\nEvent Example:");
ev.RaiseEvent();

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Method1: Simple Delegate

Multicast Delegate:
Method1: Hello
Method2: Hello

Func Delegate Result: 30
Action Delegate: Welcome
Anonymous Method: Hello Anonymous

Event Example:
Method1: Event has been triggered

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

10. WAP which use any a) Non generic collection b) Generic Collection

Theory: Non-generic collections: Collections that can store any type of object. Examples: ArrayList, Hashtable, Queue, Stack. They are not type-safe. Generic collections: Collections that store specific types. Examples: List, Dictionary<TKey, TValue>, Queue. They are type-safe and provide better performance.

In [47]:
```
using System;
using System.Collections;
using System.Collections.Generic;

// Top-level code

// (a) Non-Generic Collection (ArrayList)
ArrayList list = new ArrayList();
list.Add("Mohit");
list.Add(117);
list.Add(75.5);

Console.WriteLine("Non-Generic Collection (ArrayList):");
foreach (var item in list)
{
    Console.WriteLine(item);
}

// (b) Generic Collection (List<T>)
List<string> names = new List<string>();
names.Add("Mohit Tharu");
names.Add("Rahul");
names.Add("Sita");

Console.WriteLine("\nGeneric Collection (List<string>):");
foreach (string name in names)
{
    Console.WriteLine(name);
}

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
```

```
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Non-Generic Collection (ArrayList):
Mohit
117
75.5

Generic Collection (List<string>):
Mohit Tharu
Rahul
Sita

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

11. Program to demonstrate the use of Generic Class with Generic field and method.

Theory: Generic Class: A class that can operate on any data type specified at object creation. Generic Field: A class member whose type is generic. Generic Method: A method that can work with any data type, independent of the class's type or other types. Benefits: Type :safety ,Code ,reusability

In [50]:
```
using System;

// Generic Class
class MyGeneric<T>
{
    // Generic Field
    public T data;

    // Constructor
    public MyGeneric(T value)
    {
        data = value;
    }

    // Generic Method
    public void ShowData<U>(U info)
    {
        Console.WriteLine("Generic Field Value: " + data);
        Console.WriteLine("Generic Method Value: " + info);
    }
}

// Top-level code
MyGeneric<int> obj1 = new MyGeneric<int>(100);
obj1.ShowData<string>("Hello Generic");

MyGeneric<string> obj2 = new MyGeneric<string>("Mohit Tharu");
obj2.ShowData<int>(117);

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
```

```
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Generic Field Value: 100
Generic Method Value: Hello Generic
Generic Field Value: Mohit Tharu
Generic Method Value: 117

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

12. WAP to take input from keyboard and write them to a file

Theory:File Handling: Writing to a file in C# can be done using classes from System.IO namespace such as StreamWriter or FileStream. Keyboard Input: The Console.ReadLine() method reads input from the user. Combining these, we can read user input and save it to a text file.

In [57]:
```
using System;
using System.IO;

// File path
string filePath = "studentdata.txt";

// Simulated input (used when keyboard input is not supported)
string input = "This is my first file handling program in C#";

// Writing to file
using (StreamWriter writer = new StreamWriter(filePath))
{
    writer.WriteLine("User Input:");
    writer.WriteLine(input);
}

Console.WriteLine("Data written to file successfully!");
Console.WriteLine("Written Content:");
Console.WriteLine(input);

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Data written to file successfully!
Written Content:
This is my first file handling program in C#

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

13. WAP to demonstrate the concept of LINQ

Theory: LINQ allows querying collections (arrays, lists, XML, databases) directly in C# using query syntax or method syntax. Benefits: ->Simplifies data querying ->Strongly typed, compile-time checking ->Works on different data source.

In [62]:
```
using System;
using System.Linq;
using System.Collections.Generic;

// Data source
List<int> numbers = new List<int> { 10, 25, 30, 45, 60, 75 };

// LINQ Query Syntax
var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;

// LINQ Method Syntax
var greaterThanThirty = numbers.Where(n => n > 30);

Console.WriteLine("LINQ Query Syntax (Even Numbers):");
foreach (var n in evenNumbers)
{
    Console.WriteLine(n);
}

Console.WriteLine("\nLINQ Method Syntax (Numbers > 30):");
foreach (var n in greaterThanThirty)
{
    Console.WriteLine(n);
}

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
LINQ Query Syntax (Even Numbers):
10
30
60

LINQ Method Syntax (Numbers > 30):
45
60
75

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

14. WAP to demonstrate Lamda Expressions in C#.

Theory: Lambda Expression is an anonymous function used to create delegates or expression tree types.
Syntax: (parameters) => expression Often used with LINQ or generic delegates like Func and Action

In [65]:
```
using System;
using System.Collections.Generic;
using System.Linq;

// Data source
List<int> numbers = new List<int> { 5, 10, 15, 20, 25, 30 };

// Lambda Expression with Where()
var evenNumbers = numbers.Where(n => n % 2 == 0);

// Lambda Expression with Select()
var squares = numbers.Select(n => n * n);

Console.WriteLine("Even Numbers using Lambda Expression:");
foreach (var n in evenNumbers)
{
    Console.WriteLine(n);
}

Console.WriteLine("\nSquares using Lambda Expression:");
foreach (var s in squares)
{
    Console.WriteLine(s);
}

// Lambda Expression with Action
Action<string> greet = msg => Console.WriteLine("Message: " + msg);
greet("Hello from Lambda Expression");

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Even Numbers using Lambda Expression:
10
20
30

Squares using Lambda Expression:
25
100
225
400
625
900
Message: Hello from Lambda Expression

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

15. WAP to

```
# a) demonstrate exception handling in C# using try, catch and finally
blocks.
# b) deal with throw keyword in exception handling
# c) demonstrate custom exception handling
```

Theory: Exception Handling: Mechanism to handle runtime errors without crashing the program. try block: Code that may generate exception is placed here.

In [68]:

```
using System;

// (c) Custom Exception
class AgeException : Exception
{
    public AgeException(string message) : base(message)
    {
    }
}

class ExceptionDemo
{
    // Method using throw keyword
    public static void CheckAge(int age)
    {
        if (age < 18)
        {
            // (b) throw keyword
            throw new AgeException("Age must be 18 or above.");
        }
        else
        {
            Console.WriteLine("Valid Age: " + age);
        }
    }
}

// Top-level code
try
{
```

```
    // (a) try-catch-finally
    Console.WriteLine("Enter your age:");
    int age = 16;   // fixed value for online compiler

    ExceptionDemo.CheckAge(age);
}
catch (AgeException ex)
{
    Console.WriteLine("Custom Exception Caught: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("General Exception: " + ex.Message);
}
finally
{
    Console.WriteLine("Finally block executed");
}

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Enter your age:
Custom Exception Caught: Age must be 18 or above.
Finally block executed

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

16. WAP a) to use built-in attributes in C#. b) to create and use custom attribute in C#.

Theory: Attributes: Metadata added to classes, methods, or properties to provide extra information. Built-in attributes: Predefined by .NET (e.g., [Obsolete], [Serializable]). Custom attributes: User-defined attributes by inheriting from System.Attribute.

In [73]:
```
using System;
using System.Reflection;

// (b) Custom Attribute
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
class InfoAttribute : Attribute
{
    public string Author { get; }
    public string Version { get; }

    public InfoAttribute(string author, string version)
    {
        Author = author;
        Version = version;
    }
}
```

```
// Using Built-in and Custom Attributes
[Obsolete("This class is obsolete. Use NewDemo instead.")]
[Info("Mohit Tharu", "1.0")]
class Demo
{
    [Info("Mohit Tharu", "1.1")]
    public void Show()
    {
        Console.WriteLine("Demo class Show() method");
    }
}

// Top-level code
Demo d = new Demo();
d.Show();

// Reading Custom Attribute using Reflection
Type type = typeof(Demo);
object[] attributes = type.GetCustomAttributes(typeof(InfoAttribute), false);

foreach (InfoAttribute attr in attributes)
{
    Console.WriteLine("\nCustom Attribute Details:");
    Console.WriteLine("Author: " + attr.Author);
    Console.WriteLine("Version: " + attr.Version);
}

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Demo class Show() method

Custom Attribute Details:
Author: Mohit Tharu
Version: 1.0

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

```
(31,1): warning CS0618: 'Demo' is obsolete: 'This class is obsolete. Use NewDemo instead.'

(31,14): warning CS0618: 'Demo' is obsolete: 'This class is obsolete. Use NewDemo instead.'

(35,20): warning CS0618: 'Demo' is obsolete: 'This class is obsolete. Use NewDemo instead.'
```

17. WAP to demonstrate asynchronous programming in C# using async and await keywords.

Theory: Asynchronous programming allows a program to perform tasks without blocking the main thread. async keyword: Declares a method as asynchronous. await keyword: Waits for the completion of

an asynchronous task without blocking the main thread. Benefits: Improved responsiveness in UI applications Efficient use of system resources Better performance for I/O-bound tasks

In [78]:
```
using System;
using System.Threading.Tasks;

class AsyncDemo
{
    // Asynchronous method
    public static async Task LongTask()
    {
        Console.WriteLine("Long task started...");
        await Task.Delay(3000);   // Simulates time-consuming work
        Console.WriteLine("Long task completed.");
    }
}

// Top-level code
Console.WriteLine("Program started");

// Calling async method
Task t = AsyncDemo.LongTask();

// Other work while async task is running
Console.WriteLine("Doing other work...");

// Wait for async task to complete
t.Wait();

Console.WriteLine("Program ended");

// Student details
Console.WriteLine("\nStudent Details:");
Console.WriteLine("Name: Mohit Tharu");
Console.WriteLine("Roll No: 117");
Console.WriteLine("College: Patan Multiple Campus");

Console.ReadLine();
```

```
Program started
Long task started...
Doing other work...
Long task completed.
Program ended

Student Details:
Name: Mohit Tharu
Roll No: 117
College: Patan Multiple Campus
```

In [ ]: