1. What are the access modifiers allowed in C#?

In C#, access modifiers determine the visibility and accessibility of classes, methods, and other members. The main access modifiers in C# are:

- `public` – Accessible from anywhere in the program.
- `private` – Accessible only within the same class.
- `protected` – Accessible within the same class and derived (child) classes.
- `internal` – Accessible within the same assembly (project).
- `protected internal` – Accessible within the same assembly and in derived (child) classes, even if they are in a different assembly.
- `private protected` – Accessible within the same class and its derived (child) classes, but only if they are in the same assembly.

```
class Example
{
public int publicVar; // Accessible anywhere
private int privateVar; // Accessible only within this class
protected int protectedVar; // Accessible in this class and derived classes
internal int internalVar; // Accessible within the same assembly
protected internal int protectedInternalVar; // Accessible within the same assembly or derived classes
private protected int privateProtectedVar; // Accessible in this class and derived classes within the same assembly
}
```

```
class program
{
public string name = “Rajesh”;
}
class Main
{
static void Main(string[] args)
{
program myObj = new program();
Console.WriteLine(myObj.model);
```

```
}
}
```

2. Explain interfaces and write a C# program for extending and combining interfaces.

An interface in C# is like a contract that defines a set of methods, properties, or events that a class must implement. Unlike classes, interfaces cannot have fields or method implementations.

Features of Interfaces:

- **No Implementation:** Interfaces only define method signatures, not their implementation.
- **Multiple Inheritance:** A class can implement multiple interfaces, solving the issue of multiple inheritance in C#.
- **Polymorphism:** Allows different classes to be treated the same if they implement the same interface.

An interface can inherit from another interface (extending it) or a class can implement multiple interfaces (combining them).

Example

```
using System;

interface IStudent1
{
void Read();
}
interface IStudent2 : IStudent1
{
void Write();
}

interface IStudent3 : IStudent2
{
void Listen();
}

// Class implementing IStudent3 (which includes IStudent1 & IStudent2)
class Student : IStudent3
{
public void Read()
{
Console.WriteLine("Student is reading.");
}
```

```
public void Write()
{
Console.WriteLine("Student is writing.");
}

public void Listen()
{
Console.WriteLine("Student is listening.");
}
}

class Program
{
static void Main()
{
Student student = new Student();
student.Read(); // From IStudent1
student.Write(); // From IStudent2
student.Listen(); // From IStudent3
}
}
```

3. What are Delegate? How are they useful? Explain.

A delegate in C# is a type-safe function pointer that can hold references to methods with a specific signature. It allows methods to be passed as parameters, assigned to variables, and executed dynamically at runtime.

Why Are Delegates Useful?

Delegates are useful because they let you:

1. **Call methods dynamically**: You can decide which method to call at runtime, making your code flexible.
2. **Handle events**: In apps with buttons or other controls, delegates let you link an event (like a button click) to a method that should be executed.
3. **Pass methods as arguments**: You can pass methods to other methods, which is useful for customizing behavior.
4. **Call multiple methods**: One delegate can call multiple methods at the same time.

Example:

```
using System;
```

```
// Define a delegate that takes two integers and returns an integer
delegate int MathOperation(int a, int b);

class Program
{
// Methods matching the delegate signature
static int Add(int x, int y)
{
return x + y;
}

static int Multiply(int x, int y)
{
return x * y;
}

static void Main()
{

//instances of delegate and pass methods as variable
MathOperation operation1 = new MathOperation(Add);
MathOperation operation2 = new MathOperation(Multiply);

// Using delegates to call methods
Console.WriteLine("Addition: " + operation1(5, 3)); // Output: 8
Console.WriteLine("Multiplication: " + operation2(5, 3)); // Output: 15
}
}
```

4. Define Constructor? Write C# program to illustrate the concept of Constructor overloading.

A constructor in C# is a special method that is used to initialize objects when they are created. It has the same name as the class and does not have a return type. Constructors are automatically called when an object of a class is instantiated.

Constructor Overloading:

Constructor overloading is the process of having multiple constructors in a class with the same name but different parameters. This allows objects to be initialized in different ways.

Example:

```
using System;

class Student
{
public string Name;
public int Age;

// Constructor with one parameter
public Student(string n)
{
Name = n; // Corrected from 'name' to 'n'
Age = 0; // Default age
}

// Constructor with two parameters
public Student(string n, int a)
{
Name = n;
Age = a;
}

public void DisplayInfo()
{
Console.WriteLine("Name is: " + Name + " and age is: " + Age);
}
}

class Program
{
static void Main()
{
// Creating objects using overloaded constructors
Student student1 = new Student("Alice");
Student student2 = new Student("Bob", 21);

// Displaying information
student1.DisplayInfo();
student2.DisplayInfo();
}
}
```

Output:

Name is: Alice and age is: 0
Name is: Bob and age is: 21

5.Write a program in c# to find greatest number among three numbers?

```
//Using Else if ladder statements

using System;

class Program
{
static void Main()
{
// Input three numbers
Console.Write("Enter the first number: ");
int num1 = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the second number: ");
int num2 = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the third number: ");
int num3 = Convert.ToInt32(Console.ReadLine());

// Find the highest number using else if
if (num1 >= num2 && num1 >= num3)
{
Console.WriteLine("The highest number is: " + num1);
}
else if (num2 >= num1 && num2 >= num3)
{
Console.WriteLine("The highest number is: " + num2);
}
else
{
Console.WriteLine("The highest number is: " + num3);
}
}
}
```

```
//using Math.max() function. It include two paramethers only.

using System;

class Program
{
static void Main()
{
// Input three numbers
Console.Write("Enter the first number: ");
int num1 = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the second number: ");
int num2 = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the third number: ");
int num3 = Convert.ToInt32(Console.ReadLine());

// Find the highest number using Math.Max
int highest = Math.Max(num1, Math.Max(num2, num3)); //first find
max in num2 and 3 then after compare num1 and new max one.

// Output the result
Console.WriteLine("The highest number is: " + highest);
}
}
```

6.What is inheritance? Explain different types of inheritance with examples.

Inheritance is a process of creating a child class(derived class) from an existing class (base class). Inheritance is an Object-Oriented Programming (OOP) concept where one class (called the derived class) inherits the properties and methods from another class (called the base class). Inheritance allows the derived class to reuse code from the base class, reducing redundancy and improving code reusability.

In C#, inheritance is implemented using the `:`, where the derived class inherits from the base class.

Types of Inheritance in C#

1. **Single Inheritance:** A class inherits from only **one** base class.

```
using System;
// Base class A
class A
{
// Method in class A
public void AMethod()
{
Console.WriteLine("Method of class A");
}
}
// Class B inherits from class A (Single-Level Inheritance)
class B : A
{
// Method in class B
public void BMethod()
{
Console.WriteLine("Method of class B");
}
}
class Program
{
static void Main(string[] args)
{
// Creating an object of class B
B objB = new B();
// Calling methods from class A and B
objB.AMethod(); // Inherited from class A
objB.BMethod(); // Defined in class B
```

```
}
}
```

3. Hierarchical Inheritance: Multiple classes inherit from a **single base class**.

```
using System;

// Base class A
class A
{
// Method in class A
public void AMethod()
{
Console.WriteLine("Method of class A");
}
}

// Class B inherits from A
class B : A
{
// Method in class B
public void BMethod()
{
Console.WriteLine("Method of class B");
}
}

// Class C inherits from A
class C : A
{
// Method in class C
public void CMethod()
{
Console.WriteLine("Method of class C");
}
}

class Program
{
static void Main(string[] args)
{
// Creating object of class B
B objB = new B();
```

```
objB.AMethod(); // Inherited from class A
objB.BMethod(); // Defined in class B

Console.WriteLine("————————");

// Creating object of class C
C objC = new C();
objC.AMethod(); // Inherited from class A
objC.CMethod(); // Defined in class C
}
}
```

2. **Multilevel Inheritance:** A class inherits from another class, which itself is derived from a base class.

```
using System;
// Base class A
class A
{
// Method in class A
public void AMethod()
{
Console.WriteLine("Method of class A");
}
}
// Class B inherits from class A (First Level)
class B : A
{
// Method in class B
public void BMethod()
{
Console.WriteLine("Method of class B");
}
}

// Class C inherits from class B (Second Level)
class C : B
{
// Method in class C
public void CMethod()
{
Console.WriteLine("Method of class C");
```

```
}
}

class Program
{
static void Main(string[] args)
{
// Creating an object of class C
C objC = new C();
// Calling methods from class A, B, and C
objC.AMethod(); // Inherited from class A
objC.BMethod(); // Inherited from class B
objC.CMethod(); // Defined in class C
}
}
```

4.Multiple Inheritance (Using Interfaces): C# **does not support multiple inheritance** with classes, but it **supports multiple inheritance using interfaces**.

```
using System;

// Interface 1
interface IA
{
void AMethod();
}

// Interface 2
interface IB
{
void BMethod();
}

// Class C implementing both interfaces IA and IB
class C : IA, IB
{
public void AMethod()
{
Console.WriteLine("Method of Interface A");
}

public void BMethod()
{
```

```
Console.WriteLine("Method of Interface B");
}
}

class Program
{
static void Main(string[] args)
{
// Creating object of class C
C objC = new C();

objC.AMethod(); // Implemented from interface IA
objC.BMethod(); // Implemented from interface IB
}
}
```

What is static class in C#

A **static class** is a class that:

- Cannot be instantiated (you can't create objects from it).
- Can only contain **static members** (methods, fields, properties, etc.).
- Is often used to create utility or helper methods that don't need object state.

```
using System;
namespace demo
{
public static class program
{
public static int Add(int a, int b)
{
return a + b;
}

public static int Multiply(int a, int b)
{
return a * b;
}
}

class Program
{
```

```
static void Main(string[] args)
{
int sum = MathHelper.Add(10, 5);
int product = MathHelper.Multiply(4, 3);

Console.WriteLine("Sum: " + sum);
Console.WriteLine("Product: " + product);
}
}
}
```

Define constructor with example

A **constructor** is a special method in a class that:

- Has the same name as the class.
- Has **no return type** (not even `void`).
- Is automatically called **when an object is created**.
- Is used to **initialize fields or set up an object**.

```
using System;

namespace ConstructorExample
{
public class Student
{
public string Name;
public int Age;

// Constructor
public Student(string name, int age)
{
Name = name;
Age = age;
}

// Method to display info
public void DisplayInfo()
{
Console.WriteLine("Name: " + Name);
Console.WriteLine("Age: " + Age);
```

```
}
}

class Program
{
static void Main(string[] args)
{
// Creating object and calling constructor
Student student1 = new Student("Rajesh", 20);
student1.DisplayInfo();
}
}
}
```

Explain linq with example and describe its importance ?

LINQ (Language Integrated Query) is a feature in C# that lets you **query data** from different data sources like:

- Collections (e.g., List, Array)
- Databases (with Entity Framework)
- XML, JSON, etc.

Importance of LINQ

- **Readable**: Makes code shorter and easier to read.
- **Type-safe**: Compile-time checking helps catch errors early.
- **Integrated**: No need to learn new query languages for different data sources.
- **Versatile**: Works with objects, collections, XML, databases, etc.
- **Declarative**: You describe *what* to do, not *how* to do it.
- **Interact:** Communicate with database by C# program, no need to know sql knowledge.
- Filtering: Linq Supports filtering, ordering, sorting.
- Crud Operations: Linq supports crud operations from the database.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SimpleLinqExample
{
class Program
{
```

```
static void Main(string[] args)
{
List<int> numbers = new List<int> { 10, 15, 20, 25, 30, 35 };

var evenNumbers = from n in numbers
where n % 2 == 0
select n;

Console.WriteLine("Even numbers:");
foreach (var num in evenNumbers)
{
Console.WriteLine(num);
}
}
}
}
```

What is Razor in C#?

Razor is a **server-side markup syntax** used in **ASP.NET MVC** or **ASP.NET Core** to embed **C# code** into **HTML**.

- It's used in `.cshtml` files (C# HTML).
- Razor allows clean mixing of HTML and C#.
- Razor is used to **dynamically generate web pages on the server**

Why Razor is Important

Feature	Benefit
Clean Syntax	You write HTML + C# together without clunky syntax
Server-Side	Razor runs on the server and sends pure HTML to the browser
Fast & Lightweight	Minimal syntax and fast execution
Type-Safe	Compile-time checking of code in Razor views

This program displays the current date and a list of names using Razor in an `.cshtml` file.

```
@{
var today = DateTime.Now.ToLongDateString();
```

```
var names = new List<string> { “Rajesh”, “Sita”, “Hari” };
}

<!DOCTYPE html>
<html>
<head>
<title>Simple Razor Page</title>
</head>
<body>
<h2>Welcome to Razor Page</h2>
<p>Today’s date: @today</p>

<h3>Student List:</h3>
<ul>
@foreach (var name in names)
{
<li>@name</li>
}
</ul>
</body>
</html>
```


Rajesh Parajuli

+977-9847546279
parajulirajesh2072@gmail.com

Useful Links

- Home
- Services
- Contact Me
- Terms & Conditions

Worked with

- BidhyaTech
- JK Arts
- MastaSoftsolution
- Ghodaghodi Multiple Campus

Location

Ghodaghodi Municipality-1, Kailali Nepal