Table of Contents

1. Explain properties how can it be used to achieve encapsulation give suitable program c#

In C#, **properties** are a way to encapsulate fields in a class. They provide a mechanism to control access to the data by allowing you to add logic for getting and setting values, thus supporting encapsulation. Encapsulation ensures that the internal representation of an object is hidden from the outside, exposing only what is necessary through controlled access.

Key Features of Properties in C#:

- **Encapsulation:** Properties allow you to expose data in a controlled manner. You can restrict access to the internal state while still allowing controlled manipulation.
- **Validation:** You can include logic in the `get` and `set` accessors to validate data before assigning it.
- **Read-Only/Write-Only:** Properties can be configured to be read-only, write-only, or both.
- **Abstraction:** Properties hide the implementation details, making it easier to change the internal logic without affecting external code.

```
using System;

public class Student
{
    // Private fields to store data
    private string name;
    private int age;

    // Public property for Name with basic encapsulation
    public string Name
    {
        get { return name; } // Getter to access the private field
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
            {
                name = value; // Assign value only if it's not null or empty
            }
            else
            {
                throw new ArgumentException("Name cannot be null or empty");
            }
        }
    }

    // Public property for Age with validation logic
    public int Age
    {
        get { return age; } // Getter to retrieve the private field
        set
        {
            if (value > 0 && value <= 120) // Validating age is within a
reasonable range
            {
                age = value;
            }
            else
            {
                throw new ArgumentException("Age must be between 1 and 120");
            }
        }
    }

    // Method to display student details
    public void DisplayStudentInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of the Student class
        Student student = new Student();

        // Set properties using the public setters
        student.Name = "Alice";
        student.Age = 22;

        // Get properties using the public getters
        Console.WriteLine($"Student Name: {student.Name}");
        Console.WriteLine($"Student Age: {student.Age}");

        // Display student details
        student.DisplayStudentInfo();

        // Uncommenting the following lines will throw exceptions
        // student.Name = ""; // Throws exception: Name cannot be null or empty
        // student.Age = -5;  // Throws exception: Age must be between 1 and 120
    }
}
```

- Hiding Data with Private Fields: The fields name and age are private, preventing direct access from outside the class.
- Controlled Access through Properties:
- The Name and Age properties expose the private fields in a controlled manner.
- Validation logic in the set accessor ensures the internal state remains valid.
- Abstraction: External code interacts with the Name and Age properties without knowing the details of how data is stored or validated internally.
- Read-Only/Write-Only: By implementing only get or set, you can create read-only or write-only properties.

2. Explain type safety in c#? create c# program to create class named person with name age citizenship number and gender and the class should contain suitable methods to store and display data

Type safety in C# ensures that objects and variables are accessed only in ways that are compatible with their defined data types. This prevents errors that might occur due to type mismatches, ensuring that the program behaves predictably.

Features of Type Safety in C#:

1. **Compile-Time Checking:** Errors due to type mismatches are caught at compile time.
2. **No Buffer Overflow:** C# prevents direct memory manipulation, reducing risks like buffer overflow.
3. **Strongly Typed Language:** Every variable and object has a defined type, which cannot be overridden.
4. **Casting Rules:** Explicit or implicit casting is required when converting between types, and invalid casts are not allowed.
5. **Generics:** Generics in C# provide type safety for collections, preventing runtime errors.

Person class program

```
using System;

public class Person
{
    // Private fields for encapsulation
    private string name;
    private int age;
    private string citizenshipNumber;
    private string gender;

    // Method to store data
    public void StoreData(string name, int age, string citizenshipNumber, string
gender)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Name cannot be empty.");
        if (age <= 0 || age > 120)
            throw new ArgumentException("Age must be between 1 and 120.");

        if (string.IsNullOrWhiteSpace(citizenshipNumber))
            throw new ArgumentException("Citizenship Number cannot be empty.");

        if (gender.ToLower() != "male" && gender.ToLower() != "female" &&
gender.ToLower() != "other")
            throw new ArgumentException("Gender must be Male, Female, or
Other.");

        this.name = name;
        this.age = age;
        this.citizenshipNumber = citizenshipNumber;
        this.gender = gender;
    }

    // Method to display data
    public void DisplayData()
    {
        Console.WriteLine("=== Person Details ===");
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Age: {age}");
        Console.WriteLine($"Citizenship Number: {citizenshipNumber}");
        Console.WriteLine($"Gender: {gender}");
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of the Person class
        Person person = new Person();

        // Store data in the object
        person.StoreData("John Doe", 30, "AB123456", "Male");

        // Display the data
        person.DisplayData();


    }
}
```

3. Custom Exception

```
using System;

// Custom exception class
public class InvalidAgeException : Exception
{
    // Default constructor
    public InvalidAgeException() : base("Age is invalid. It must be between 1 and 
120.") { }

    // Constructor with a custom message
    public InvalidAgeException(string message) : base(message) { }

    // Constructor with a custom message and inner exception
    public InvalidAgeException(string message, Exception innerException)
        : base(message, innerException) { }
}

// Person class for demonstration
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    // Method to set age with validation
    public void SetAge(int age)
    {
        if (age <= 0 || age > 120)
        {
            throw new InvalidAgeException($"Invalid age: {age}. Age must be 
between 1 and 120.");
        }
        Age = age;
    }

    public void Display()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}

// Program to demonstrate custom exception
class Program
{
    static void Main()
    {
        try
        {
            // Create a Person object
            Person person = new Person();
            person.Name = "Alice";

            // Set an invalid age to trigger the custom exception
            person.SetAge(150);

            // Display the person's details
            person.Display();
        }
        catch (InvalidAgeException ex)
        {
            // Handle the custom exception
            Console.WriteLine("Custom Exception Caught:");
            Console.WriteLine(ex.Message);
        }
        catch (Exception ex)
        {
            // Catch any other exceptions
            Console.WriteLine("An unexpected error occurred:");
            Console.WriteLine(ex.Message);
        }
        finally
        {
            Console.WriteLine("Execution completed. Cleaning up resources if
necessary.");
        }
    }
}
```

4. what do you mean by delegate how to change delegate into event

A delegate in C# is a type that represents references to methods with a specific signature. Delegates are often used to define callback methods, enable event-driven programming, and pass methods as arguments. Essentially, a delegate acts as a type-safe function pointer.

Key Features of Delegates:

- Type-Safe: Ensures that the method signature matches the delegate declaration.
- Multicast: Can hold references to multiple methods.
- Anonymous Methods: Can work with lambda expressions or inline code.
- Used for Events: Delegates form the foundation of events in C#.

An **event** is a special kind of delegate that provides a way to notify subscribers when something happens. While delegates can be called directly, events add an extra layer of restriction and abstraction, ensuring they can only be invoked from within the class or struct that declared them.

Differences Between Delegates and Events:

Feature	Delegate	Event
Invocation	Can be invoked directly.	Cannot be invoked directly outside the declaring class.
Usage	Function pointers and callbacks.	Specifically for event-driven programming.
Accessibility	Subscribers can replace invocation list.	Subscribers can only add/remove handlers.

Delegate to an event

To convert a delegate into an event, you use the event keyword in its declaration. This restricts direct invocation of the delegate, allowing only subscription and unsubscription (+= and -=) from outside the declaring class.

```
using System;

public class Publisher
{
    // Declare a delegate
    public delegate void Notify(string message);

    // Declare a public delegate instance
    public Notify OnNotify; // Can be accessed and invoked from outside

    public void TriggerEvent()
    {
        // Check if the delegate has any subscribers
        if (OnNotify != null)
        {
            OnNotify("Event triggered using delegate!");
        }
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of Publisher
        Publisher publisher = new Publisher();

        // Subscribe to the delegate
        publisher.OnNotify += DisplayMessage;

        // Trigger the event
        publisher.TriggerEvent();

        // Direct invocation of delegate from outside (allowed for delegates)
        publisher.OnNotify("Direct invocation using delegate!");
    }

    // Method to handle notifications
    static void DisplayMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

5. binary operator overloading in suitable example

```
using System;

public class ComplexNumber
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public ComplexNumber(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    // Overloading the + operator
    public static ComplexNumber operator +(ComplexNumber c1, ComplexNumber c2)
    {
        return new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }

    // Overriding ToString for display purposes
    public override string ToString()
    {
        return $"{Real} + {Imaginary}i";
    }
}

class Program
{
    static void Main()
    {
        ComplexNumber c1 = new ComplexNumber(3, 4);
        ComplexNumber c2 = new ComplexNumber(1, 2);

        // Using the overloaded + operator
        ComplexNumber result = c1 + c2;
        Console.WriteLine($"Result: {result}"); // Output: Result: 4 + 6i
    }
}
```

6. What is a Lambda Expression?

A **lambda expression** is a concise way to represent an anonymous function. It's particularly useful for defining small, inline functions, often used with delegates, LINQ, and event handling.

Importance of Lambda Expressions:

1. **Conciseness:** Reduces boilerplate code.
2. **Improves Readability:** Simplifies code by expressing functionality inline.
3. **Higher-Order Functions:** Works seamlessly with methods like Where, Select, etc.
4. **Flexibility:** Supports inline operations for complex operations.

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

class Program
{
    static void Main()
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

        // Using a lambda expression to filter even numbers
        var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();

        Console.WriteLine("Even Numbers: " + string.Join(", ", evenNumbers)); //
Output: Even Numbers: 2, 4, 6
    }
}
```

7. LINQ vs SQL

Aspect	LINQ (Language-Integrated Query)	SQL (Structured Query Language)
Integration	Integrated into C# and other .NET languages.	Used to query relational databases.
Type Safety	Supports compile-time type checking.	Not type-safe; syntax errors are detected at runtime.
Scope	Works with in-memory collections (e.g., arrays, lists).	Works with databases (e.g., SQL Server, MySQL).
Syntax	Query is written in C# syntax.	Query is written in SQL syntax.
Execution	Queries are executed in-memory or deferred.	Queries are executed directly on the database server.

Aspect	LINQ (Language-Integrated Query)	SQL (Structured Query Language)
Extensibility	Can query various data sources like XML, objects, and SQL databases.	Limited to databases only.

8. Explain different types of validation in ASP.NET

Form validation is the process of ensuring that user inputs meet the required criteria before they are submitted to the server. In ASP.NET, validation can be performed both on the client-side and server-side. ASP.NET provides built-in **validation controls** to handle common validation scenarios.

Client-Side Validation:

- Validation occurs in the browser before the form is submitted to the server.
- Provides immediate feedback to the user, improving the user experience.
- Implemented using JavaScript.

Server-Side Validation:

- Validation occurs on the server after the form is submitted.
- Ensures security since client-side validation can be bypassed.
- Always used as a fallback for critical validations.

Control	Purpose
RequiredFieldValidator	Ensures a value is entered for a required field.
RangeValidator	Checks if a value falls within a specified range (e.g., numbers, dates).
CompareValidator	Compares the value of one control with another or with a constant value (e.g., password matching).
RegularExpressionValidator	Ensures input matches a specific regular expression pattern (e.g., email, phone number).
CustomValidator	Allows custom validation logic to be written in server-side or client-side code.
ValidationSummary	Displays a summary of all validation errors in one place.

Example of form validation

ASPX code

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="ValidationExample.aspx.cs" Inherits="ValidationExample" %>

<!DOCTYPE html>
<html>
<head>
    <title>ASP.NET Form Validation</title>
</head>
<body>
    <form id="form1" runat="server">
        <h2>User Registration Form</h2>
        <div>
            <label>Username:</label>
            <asp:TextBox ID="txtUsername" runat="server"></asp:TextBox>
            <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
runat="server"
                ControlToValidate="txtUsername" ErrorMessage="Username is
required!" ForeColor="Red"></asp:RequiredFieldValidator>
        </div>

        <div>
            <label>Age:</label>
            <asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
            <asp:RangeValidator ID="RangeValidator1" runat="server"
                ControlToValidate="txtAge" MinimumValue="18" MaximumValue="60"
                Type="Integer" ErrorMessage="Age must be between 18 and 60!"
ForeColor="Red"></asp:RangeValidator>
        </div>

        <div>
            <label>Email:</label>
            <asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
            <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
runat="server"
                ControlToValidate="txtEmail"
                ValidationExpression="^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$"
                ErrorMessage="Enter a valid email!"
ForeColor="Red"></asp:RegularExpressionValidator>
        </div>

        <div>
            <label>Password:</label>
            <asp:TextBox ID="txtPassword" runat="server"
TextMode="Password"></asp:TextBox>
            <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
runat="server"
                ControlToValidate="txtPassword" ErrorMessage="Password is
required!" ForeColor="Red"></asp:RequiredFieldValidator>
        </div>

        <div>
            <label>Confirm Password:</label>
            <asp:TextBox ID="txtConfirmPassword" runat="server"
TextMode="Password"></asp:TextBox>
            <asp:CompareValidator ID="CompareValidator1" runat="server"
                ControlToValidate="txtConfirmPassword"
ControlToCompare="txtPassword"
```

```
                ErrorMessage="Passwords do not match!"
ForeColor="Red"></asp:CompareValidator>
        </div>

        <div>
            <asp:Button ID="btnSubmit" runat="server" Text="Submit"
OnClick="btnSubmit_Click" />
        </div>

        <asp:ValidationSummary ID="ValidationSummary1" runat="server"
ForeColor="Red" />
    </form>
</body>
</html>
```

C# code

```
using System;

public partial class ValidationExample : System.Web.UI.Page
{
    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        if (Page.IsValid)
        {
            Response.Write("Form submitted successfully!");
        }
        else
        {
            Response.Write("There are validation errors on the form.");
        }
    }
}
```

9. C# program to store id name age salary of 5 employees in list and

- **write linq query for selecting name address of employeee whose name start with letter R age greater than 25 in descending order based on age**
- **LINQ to select wmployee with highest salary**

```
using System;
using System.Collections.Generic;
using System.Linq;
public class Employee
{
    public int Id { get; set; } // Property to store employee ID
    public string Name { get; set; } // Property to store employee name
    public int Age { get; set; } // Property to store employee age
    public double Salary { get; set; } // Property to store employee salary
    public string Address { get; set; } // Property to store employee address
}

class Program
{
    static void Main()
    {
        // Creating a list of 5 employees
        List<Employee> employees = new List<Employee>
        {
            new Employee { Id = 1, Name = "Ramesh", Age = 30, Salary = 50000,
Address = "Address 1" },
            new Employee { Id = 2, Name = "Suresh", Age = 22, Salary = 40000,
Address = "Address 2" },
            new Employee { Id = 3, Name = "Ravi", Age = 28, Salary = 45000,
Address = "Address 3" },
            new Employee { Id = 4, Name = "Amit", Age = 25, Salary = 35000,
Address = "Address 4" },
            new Employee { Id = 5, Name = "Rajesh", Age = 32, Salary = 60000,
Address = "Address 5" }
        };

        // LINQ query to filter employees
        var result = employees
            .Where(e => e.Name.StartsWith("R") && e.Age > 25) // Conditions: Name
starts with 'R', Age > 25
            .OrderByDescending(e => e.Age) // Sorting in descending order of Age
            .Select(e => new { e.Name, e.Address }); // Selecting Name and
Address

        // Displaying the filtered employees
        Console.WriteLine("Employees whose names start with 'R' and are older
than 25 (sorted by age):");
        foreach (var employee in result) // Iterating through the filtered
employees
        {
            Console.WriteLine($"Name: {employee.Name}, Address:
{employee.Address}"); // Displaying employee details
        }

        // LINQ query to select the employee with the maximum salary
        var maxSalaryEmployee = employees
```

```
            .OrderByDescending(e => e.Salary) // Sorting employees by salary in
descending order
            .FirstOrDefault(); // Selecting the first employee (with the highest
salary)

        // Displaying the employee with the maximum salary
        if (maxSalaryEmployee != null) // Checking if an employee was found
        {
            Console.WriteLine("\nEmployee with the maximum salary:");
            Console.WriteLine($"Name: {maxSalaryEmployee.Name}, Salary:
{maxSalaryEmployee.Salary}, Address: {maxSalaryEmployee.Address}"); // Displaying
employee details
        }
    }
}
```

10. Delegates and Multicasting. Program to calculate sum and product using delegates

A delegate in C# is a type-safe object that can reference methods with a specific signature. It allows methods to be passed as arguments or assigned to variables, enabling flexible and reusable code.

What is Multicast Delegation?

- A multicast delegate is a delegate that references multiple methods.
- When the delegate is invoked, all the methods in its invocation list are executed sequentially.

```
using System;

public class Calculator
{
    // Method to calculate sum
    public void CalculateSum(int a, int b)
    {
        Console.WriteLine($"Sum: {a + b}");
    }

    // Method to calculate product
    public void CalculateProduct(int a, int b)
    {
        Console.WriteLine($"Product: {a * b}");
    }
}

public class Program
{
    // Define a delegate that accepts two integers
    public delegate void CalculationDelegate(int x, int y);
    static void Main()
    {
        Calculator calc = new Calculator();

        // Create delegate instances for the methods
        CalculationDelegate delSum = calc.CalculateSum;
        CalculationDelegate delProduct = calc.CalculateProduct;

        // Multicasting: Combine both delegates into one
        CalculationDelegate multicastDel = delSum + delProduct;

        // Invoke the multicast delegate
        Console.WriteLine("Performing calculations using multicast delegate:");
        multicastDel(10, 5); // Executes both CalculateSum and CalculateProduct

        // Removing a method from the multicast delegate
        multicastDel -= delProduct;

        Console.WriteLine("\nAfter removing product calculation:");
        multicastDel(10, 5); // Executes only CalculateSum
    }
}
```

11. Language interoperability of C# in points? What is optional parameter? Enum parameter? c# program that stores values in two enumerators department and college it uses two fuction to display data in department and college enum

Language interoperability in .NET allows different languages to:

Work Together:

Code written in one language can interact with another (e.g., calling a C# library from VB.NET).

CLR Support:

All .NET languages compile to **Intermediate Language (IL)**, making them interoperable.

Common Type System (CTS):

Ensures data types across languages are consistent (e.g., int in C# is equivalent to Integer in VB.NET).

Optional Parameter

Optional parameters allow a method to have default values if arguments are not passed.

void PrintMessage(string message = "Hello, World!")

```
{
  Console.WriteLine(message);
}
```

ENUM example

```
using System;

enum Department { ComputerScience, Mechanical, Civil, Electrical }
enum College { Engineering, Medical, Arts, Commerce }

class Program
{
    static void DisplayDepartment()
    {
        Console.WriteLine("Departments:");
        foreach (var dept in Enum.GetValues(typeof(Department)))
        {
            Console.WriteLine(dept);
        }
    }

    static void DisplayCollege()
    {
        Console.WriteLine("\nColleges:");
        foreach (var col in Enum.GetValues(typeof(College)))
        {
            Console.WriteLine(col);
        }
    }

    static void Main()
    {
        DisplayDepartment();
        DisplayCollege();
    }
}
```

12. C# program that demostrate how base keywork is used to call member funciton and constructor of parent class also another c# program to handle index out of range and invalid cast exception

```
class Parent
{
    public Parent()
    {
        Console.WriteLine("Parent Constructor Called");
    }

    public void ParentMethod()
    {
        Console.WriteLine("Parent Method Called");
    }
}

class Child : Parent
{
    public Child() : base()
    {
        Console.WriteLine("Child Constructor Called");
    }

    public void CallParentMethod()
    {
        base.ParentMethod();
    }
}

class Program
{
    static void Main()
    {
        Child child = new Child();
        child.CallParentMethod();
    }
}
```

Exception handling

```
using System;

class Program
{
    static void Main()
    {
        try
        {
            int[] numbers = { 1, 2, 3 };
            Console.WriteLine(numbers[5]); // IndexOutOfRangeException
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }

        try
        {
            object str = "Hello";
            int num = (int)str; // InvalidCastException
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
    }
}
```

13. What is query syntax in linq explain select contains order by where in LINQ

Query Syntax is a declarative way to write LINQ queries using keywords like from, where, select, etc.

Common LINQ Keywords:

- select: Projects the result of a query.
- where: Filters the data based on a condition.
- order by: Sorts the result (ascending by default).
- contains: Checks if a collection contains a specific element.

Program using LINQ implementing above concepts

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

class Program
{
    static void Main()
    {
        List<string> fruits = new List<string> { "Apple", "Banana", "Cherry",
"Mango", "Raspberry" };

        // Query Syntax
        var filteredFruits = from fruit in fruits
                             where fruit.Contains("a") // Filter fruits with 'a'
                             orderby fruit ascending   // Sort alphabetically
                             select fruit;

        Console.WriteLine("Filtered Fruits:");
        foreach (var fruit in filteredFruits)
        {
            Console.WriteLine(fruit);
        }
    }
}
```

14. What is an event? Explain with suitable program how event is handled in c#? explain concept of generic delegate with program

An event in C# is a mechanism used to implement a publisher-subscriber model. It enables communication between objects. Events are typically used to signal state changes or user actions (e.g., button clicks in a GUI).

- Publisher: The object that triggers the event.
- Subscriber: The object that listens and reacts to the event.

An event is based on delegates, allowing methods to be attached and called when the event occurs.

Event handling code

```
using System;

public class Publisher
{
    // Declare an event using EventHandler
    public event EventHandler EventOccurred;

    // Method to raise the event
    public void TriggerEvent()
    {
        Console.WriteLine("Publisher: Triggering the event...");
        EventOccurred?.Invoke(this, EventArgs.Empty); // Raise the event if there
are subscribers
    }
}

public class Subscriber
{
    // Method to handle the event
    public void OnEventOccurred(object sender, EventArgs e)
    {
        Console.WriteLine("Subscriber: Event received and handled.");
    }
}

class Program
{
    static void Main()
    {
        // Create publisher and subscriber instances
        Publisher publisher = new Publisher();
        Subscriber subscriber = new Subscriber();
        // Subscribe to the event
        publisher.EventOccurred += subscriber.OnEventOccurred;

        // Trigger the event
        publisher.TriggerEvent();

        // Unsubscribe from the event (optional)
        publisher.EventOccurred -= subscriber.OnEventOccurred;
    }
}
```

Generic Delegate

A **generic delegate** allows you to define delegates with type parameters, making them reusable for multiple types. It is part of the **System namespace** and includes delegates like Func, Action, and Predicate.

Common Generic Delegates:

1. **Func**:
    - Represents a method that returns a value.
    - Example: Func<int, int, int> add = (a, b) => a + b;
2. **Action**:
    - Represents a method that does not return a value.
    - Example: Action<string> greet = name => Console.WriteLine($"Hello, {name}!");
3. **Predicate**:
    - Represents a method that returns a boolean value.
    - Example: Predicate<int> isEven = num => num % 2 == 0;

Generic delegate example

```
using System;

class Program
{
    // Method to add two integers
    public static int Add(int a, int b)
    {
        return a + b;
    }

    // Method to check if a number is even
    public static bool IsEven(int number)
    {
        return number % 2 == 0;
    }

    static void Main()
    {
        // Using Func (delegate with return type)
        Func<int, int, int> addFunc = Add;
        int sum = addFunc(5, 10);
        Console.WriteLine($"Sum: {sum}");

        // Using Predicate (delegate with a boolean return type)
        Predicate<int> isEvenPredicate = IsEven;
        bool isEven = isEvenPredicate(4);
        Console.WriteLine($"Is 4 even? {isEven}");

        // Using Action (delegate with no return type)
        Action<string> greetAction = name => Console.WriteLine($"Hello,
{name}!");
        greetAction("Alice");
    }
}
```

Aspect	Delegate	Event
Definition	A reference to a method or set of methods.	A mechanism for subscribing to notifications.
Access	Can be called directly by any object.	Can only be invoked by the owning class.
Use	Primarily used for callbacks.	Used to signal state changes or actions.

15. write c# code to overload ++ and == operator

```
using System;

public class Counter
{
    public int Value { get; set; }

    // Overloading the ++ (increment) operator
    public static Counter operator ++(Counter counter)
    {
        counter.Value++;
        return counter;
    }

    // Overloading the == (equality) operator
    public static bool operator ==(Counter counter1, Counter counter2)
    {
        if (ReferenceEquals(counter1, null) || ReferenceEquals(counter2, null))
            return false;
        return counter1.Value == counter2.Value;
    }

    // Overriding Equals and GetHashCode to use == operator properly
    public override bool Equals(object obj)
    {
        if (obj is Counter)
            return this == (Counter)obj;
        return false;
    }

    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }
}

class Program
{
    static void Main()
    {
        Counter counter1 = new Counter { Value = 5 };
        Counter counter2 = new Counter { Value = 5 };

        // Using overloaded ++ operator
        counter1++;
        Console.WriteLine($"Counter1 Value: {counter1.Value}");  // Output: 6

        // Using overloaded == operator
        Console.WriteLine($"Are Counter1 and Counter2 equal? {counter1 ==
counter2}");  // Output: False
    }
}
```

16. Explain various server control in asp.net with program, list any five contextual keywork in c# what are other types of keyword than contextual list that too

ASP.NET provides several server-side controls that allow you to build dynamic web pages with rich functionality. Some of the common server controls are:

- Button: Represents a clickable button.
- Label: Displays text on the page.
- TextBox: Allows user input.

- DropDownList: Allows the user to select from a list of options.
- GridView: Displays data in a tabular format, useful for displaying large sets of data.

ASPX code

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebApplication1._Default" %>

<!DOCTYPE html>
<html>
<head>
    <title>ASP.NET Server Controls</title>
</head>
<body>
    <form id="form1" runat="server">
        <h3>ASP.NET Server Controls Example</h3>

        <asp:Label ID="Label1" runat="server" Text="Enter your name: " />
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br/><br/>

        <asp:Button ID="Button1" runat="server" Text="Greet"
OnClick="Button1_Click" /><br/><br/>

        <asp:Label ID="Label2" runat="server" Text="Greeting will appear here."
/>
    </form>
</body>
</html>
```

C#

```
using System;

namespace WebApplication1
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Button1_Click(object sender, EventArgs e)
        {
            // Display greeting message in the label
            Label2.Text = "Hello, " + TextBox1.Text;
        }
    }
}
```

Contextual keyword

Contextual keywords in C# are reserved words that have special meaning in specific contexts, but can still be used as identifiers in other contexts. These keywords are not reserved globally.

Five Contextual Keywords in C#:

- async – Used to define an asynchronous method.
- await – Used to pause the execution of an asynchronous method.
- yield – Used to return an element to the caller in an iterator method.
- is – Used to check the runtime type of an object.
- in – Used to specify that a parameter is passed by reference but cannot be modified.

Other Types of Keywords (Non-Contextual):

- abstract
- class
- interface
- public
- private
- static
- new
- void
- int
- string

These are reserved keywords and cannot be used as identifiers in any context.

17. c# program to initialize and display jagged array element with sum of each row

```
using System;

class Program
{
    static void Main()
    {
        // Initialize a jagged array
        int[][] jaggedArray = new int[3][]
        {
            new int[] { 1, 2, 3 },
            new int[] { 4, 5 },
            new int[] { 6, 7, 8, 9 }
        };

        // Display the elements and sum of each row
        for (int i = 0; i < jaggedArray.Length; i++)
        {
            int rowSum = 0;
            Console.Write($"Row {i + 1}: ");
            foreach (var num in jaggedArray[i])
            {
                Console.Write(num + " ");
                rowSum += num;
            }
            Console.WriteLine($"| Sum: {rowSum}");
        }
    }
}
```

18. what is string interpolation differentiate Indexers and properties

String Interpolation allows you to embed expressions directly inside string literals. It provides a more readable and convenient syntax for formatting strings.

```
int age = 25;
string name = "John";
Console.WriteLine($"Name: {name}, Age: {age}");
```

Feature	Indexer	Property
Definition	An indexer allows objects to be accessed like arrays using an index.	A property allows controlled access to the data of an object.
Syntax	Uses `this` keyword and index parameters.	Uses a getter and setter to access or modify values.
Usage	Typically used for objects that store collections or elements in an indexed way.	Used for more general purpose value encapsulation.
Example	`public int this[int index] { get; set; }`	`public int Age { get; set; }`
Use Case	When you need to allow access to data like an array or collection.	To provide a way to set or get a value with validation or computation.

19. Explain basic generics with code

Generics in C# allow you to define classes, methods, or interfaces with placeholders for data types, making code reusable for different data types. You define the type only when you create an instance of the class or call the method.

```
using System;

public class GenericClass<T>
{
    private T value;

    public T Value
    {
        get { return value; }
        set { this.value = value; }
    }

    public void DisplayValue()
    {
        Console.WriteLine("Value: " + value);
    }
}

class Program
{
    static void Main()
    {
        // Using generic class with int type
        GenericClass<int> intInstance = new GenericClass<int>();
        intInstance.Value = 42;
        intInstance.DisplayValue(); // Output: Value: 42

        // Using generic class with string type
        GenericClass<string> stringInstance = new GenericClass<string>();
        stringInstance.Value = "Hello, Generics!";
        stringInstance.DisplayValue(); // Output: Value: Hello, Generics!
    }
}
```

20. Program to calculate simple interest in in one page with form and displaying it to another asp page

Form.aspx (form accepting PTR)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Input.aspx.cs"
Inherits="WebApplication1.Input" %>

<!DOCTYPE html>
<html>
<head>
    <title>Simple Interest Calculator</title>
</head>
<body>
    <h3>Enter the details to calculate Simple Interest:</h3>
    <form id="form1" runat="server">
        <asp:Label ID="Label1" runat="server" Text="Principal Amount: " />
        <asp:TextBox ID="txtPrincipal" runat="server" />
        <br /><br />
        <asp:Label ID="Label2" runat="server" Text="Rate of Interest: " />
        <asp:TextBox ID="txtRate" runat="server" />
        <br /><br />

        <asp:Label ID="Label3" runat="server" Text="Time (in years): " />
        <asp:TextBox ID="txtTime" runat="server" />
        <br /><br />

        <asp:Button ID="btnCalculate" runat="server" Text="Calculate"
OnClick="btnCalculate_Click" />
    </form>
</body>
</html>
```

Input.aspx.cs (doing caclulations)

```
using System;

namespace WebApplication1
{
    public partial class Input : System.Web.UI.Page
    {
        protected void btnCalculate_Click(object sender, EventArgs e)
        {
            double principal = Convert.ToDouble(txtPrincipal.Text);
            double rate = Convert.ToDouble(txtRate.Text);
            double time = Convert.ToDouble(txtTime.Text);

            // Calculate simple interest
            double simpleInterest = (principal * rate * time) / 100;

            // Pass the calculated value to the result page
            Response.Redirect("Result.aspx?interest=" + simpleInterest);
        }
    }
}
```

Result.aspx(showing result)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Result.aspx.cs"
Inherits="WebApplication1.Result" %>

<!DOCTYPE html>
<html>
<head>
    <title>Simple Interest Result</title>
</head>
<body>
    <h3>Simple Interest Calculation Result:</h3>
    <asp:Label ID="lblResult" runat="server" Text="Result will be displayed
here." />
</body>
</html>
```

Result.aspx.cs(fetching result)

```
using System;

namespace WebApplication1
{
    public partial class Result : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            // Retrieve the interest value from the query string and display it
            string interest = Request.QueryString["interest"];
            lblResult.Text = "Calculated Simple Interest: " + interest;
        }
    }
}
```

21. What is system exception write program to read balance and withdraw amount from keyboard if withdraw less than balance throw application error with message all in CLI small basic code in c# dont try getting fancy with extra things

An unwanted or unexpected event, which occurs during the execution of a program i.e at runtime, that disrupts the normal flow of the program's instructions. Sometimes during the execution of the program, the user may face the possibility that the program may crash that si called system exception

Withdraw program with exception

```
using System;

class Program
{
    static void Main()
    {
        try
        {
            // Read balance and withdrawal amount
            Console.Write("Enter balance: ");
            double balance = Convert.ToDouble(Console.ReadLine());

            Console.Write("Enter withdrawal amount: ");
            double withdrawAmount = Convert.ToDouble(Console.ReadLine());

            // Check if withdrawal is less than balance
            if (withdrawAmount > balance)
            {
                throw new ApplicationException("Withdrawal amount exceeds 
balance.");
            }

            // Update balance after withdrawal
            balance -= withdrawAmount;
            Console.WriteLine($"Transaction successful. New balance is: 
{balance}");
        }
        catch (ApplicationException ex)
        {
            // Handle custom exception
            Console.WriteLine($"Error: {ex.Message}");
        }
        catch (Exception ex)
        {
            // Handle other exceptions
            Console.WriteLine($"Unexpected error: {ex.Message}");
        }
    }
}
```

22. Define any 5 standard LINQ standard operator and give c# program to compute aggregate salary of five employee display employee record in descending order with respect ot salary of employee using LINQ

Here are **five standard LINQ operators** in C#:

1. **Where()** - Filters a collection based on a condition.
2. **Select()** - Projects each element of a collection into a new form.
3. **OrderBy()** - Sorts elements of a collection in ascending order.
4. **Sum()** - Computes the sum of a collection of numbers.
5. **Average()** - Computes the average of a collection of numbers

Salary ordering employee codes

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

class Program
```

```
{
    // Employee class to hold employee details
    public class Employee
    {
        public int EmployeeID { get; set; }
        public string Name { get; set; }
        public double Salary { get; set; }
    }

    static void Main()
    {
        // Create a list of employees
        List<Employee> employees = new List<Employee>
        {
            new Employee { EmployeeID = 1, Name = "John", Salary = 3000 },
            new Employee { EmployeeID = 2, Name = "Alice", Salary = 5000 },
            new Employee { EmployeeID = 3, Name = "Bob", Salary = 4000 },
            new Employee { EmployeeID = 4, Name = "Charlie", Salary = 6000 },
            new Employee { EmployeeID = 5, Name = "David", Salary = 3500 }
        };

        // Compute total salary using LINQ Sum() operator
        double totalSalary = employees.Sum(e => e.Salary);
        Console.WriteLine($"Total Salary of all Employees: {totalSalary}");

        // Display employee records ordered by salary in descending order using
OrderByDescending()
        var sortedEmployees = employees.OrderByDescending(e => e.Salary);

        Console.WriteLine("\nEmployee Records (Sorted by Salary in Descending
Order):");
        foreach (var employee in sortedEmployees)
        {
            Console.WriteLine($"ID: {employee.EmployeeID}, Name: {employee.Name},
Salary: {employee.Salary}");
        }
    }
}
```

23. Different Types of Delegates in C# Write a C# program to create a class Time which represents time. The class should have three fields for hours, minutes and seconds. It should have constructor to initialize hours, minutes and seconds and method displayTime() to print current time. Overload following operators:

a) + (add two time objects on 24 hours clock)

b) (compare two time objects)

There are three main types of delegates in C#:

1. **Single-Cast Delegates**: A delegate that points to a single method. When invoked, it calls one method.
2. **Multi-Cast Delegates**: A delegate that points to multiple methods. When invoked, it calls all the methods in the delegate's invocation list.
3. **Generic Delegates**: Delegates that are defined using the Action, Func, or Predicate types. These delegates are more flexible and can be used for methods with different signatures.

Time overloading program in c#

```
using System;

class Time
{
    public int Hours { get; set; }
    public int Minutes { get; set; }
    public int Seconds { get; set; }

    // Constructor to initialize time
    public Time(int hours, int minutes, int seconds)
    {
        Hours = hours;
        Minutes = minutes;
        Seconds = seconds;
    }

    // Method to display time
    public void DisplayTime()
    {
        Console.WriteLine($"{Hours:D2}:{Minutes:D2}:{Seconds:D2}");
    }

    // Overload + operator to add two time objects on a 24-hour clock
    public static Time operator +(Time t1, Time t2)
    {
        int totalSeconds = (t1.Hours * 3600 + t1.Minutes * 60 + t1.Seconds) +
                           (t2.Hours * 3600 + t2.Minutes * 60 + t2.Seconds);
        totalSeconds %= 86400; // To ensure the time stays within 24 hours

        return new Time(totalSeconds / 3600, (totalSeconds % 3600) / 60,
totalSeconds % 60);
    }

    // Overload == operator to compare two time objects
    public static bool operator ==(Time t1, Time t2)
    {
        return t1.Hours == t2.Hours && t1.Minutes == t2.Minutes && t1.Seconds ==
t2.Seconds;
    }

    // Overload != operator to compare two time objects
    public static bool operator !=(Time t1, Time t2)
    {
        return !(t1 == t2);
    }

    // Overload Equals method for comparison
    public override bool Equals(object obj)
    {
        if (obj is Time t)
        {
            return this == t;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return Hours * 3600 + Minutes * 60 + Seconds;
    }
}

class Program
{
    static void Main()
    {
        // Create two time objects
        Time time1 = new Time(4, 30, 45); // 4:30:45
        Time time2 = new Time(3, 40, 30); // 3:40:30

        // Display individual times
        Console.WriteLine("Time 1:");
        time1.DisplayTime();
        Console.WriteLine("Time 2:");
        time2.DisplayTime();

        // Add two time objects
        Time time3 = time1 + time2;
        Console.WriteLine("Time after addition:");
        time3.DisplayTime();

        // Compare two time objects
        if (time1 == time2)
        {
            Console.WriteLine("Time 1 is equal to Time 2.");
        }
        else
        {
            Console.WriteLine("Time 1 is not equal to Time 2.");
        }
    }
}
```

24. Explain ExecuteScalar() and ExecuteReader() and execute non query C# Program to Insert Customer Records and Display Customers with Balance Greater Than 5000

ExecuteScalar()

- **Purpose**: ExecuteScalar() is used when a query returns a **single value**. This can be a result from aggregate functions like COUNT(), SUM(), AVG(), or a single field in a result set. It returns the value of the first column of the first row in the result set.
- **Usage**: It is typically used for queries that return a single piece of data, such as the total number of rows in a table or a sum of a particular column.

ExecuteReader()

- **Purpose:** ExecuteReader() is used when a query returns multiple rows and columns, such as when selecting data from a table. It returns a SqlDataReader object that can be used to read the result set row by row.
- **Usage:** It is typically used for SELECT queries that retrieve multiple records from the database. It requires you to read through the result set using methods like Read().

ExecuteNonQuery()

- Purpose: ExecuteNonQuery() is used for SQL statements that do not return data (e.g., INSERT, UPDATE, DELETE, or CREATE TABLE). It executes the query but does not return any data. Instead, it returns the number of rows affected (if applicable).
- Usage: It is typically used for queries that modify the database or execute operations without expecting any results (other than the number of affected rows).

PROGRAM FOR GREATER THAN 5000 salary

SQL CODE

```
CREATE TABLE Customer (
    Account_no INT PRIMARY KEY,
    Name NVARCHAR(100),
    Address NVARCHAR(200),
    Balance DECIMAL(10, 2)
);
```

C# CODE

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString =
"Server=your_server_name;Database=Bank;Integrated Security=True;";

        // Insert customer records into the database
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();

            string insertQuery = "INSERT INTO Customer (Account_no, Name,
Address, Balance) VALUES " +
                                 "(1, 'Alice', '123 Main St', 4500), " +
                                 "(2, 'Bob', '456 Oak St', 6000), " +
                                 "(3, 'Charlie', '789 Pine St', 7000), " +
                                 "(4, 'David', '101 Maple St', 4000), " +
                                 "(5, 'Eva', '202 Birch St', 8000)";

            SqlCommand command = new SqlCommand(insertQuery, connection);
            command.ExecuteNonQuery();
        }

        // Display customers with balance greater than 5000
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();

            string selectQuery = "SELECT Account_no, Name, Address, Balance FROM
Customer WHERE Balance > 5000";
            SqlCommand command = new SqlCommand(selectQuery, connection);
            SqlDataReader reader = command.ExecuteReader();

            Console.WriteLine("Customers with balance greater than 5000:");
            while (reader.Read())
            {
                Console.WriteLine($"Account No: {reader["Account_no"]}, Name:
{reader["Name"]}, Address: {reader["Address"]}, Balance: {reader["Balance"]}");
            }
        }
    }
}
```

25. Asp.net basic program to populate database named student with an aspx form taking data and it must have two button add, update add will populate but update will upadate with respect to id given and another page to show the data adjacent having delete button to delete record database query only using LINQ facility of ASP.NET

SQL for database

```
CREATE DATABASE StudentDB;

USE StudentDB;

CREATE TABLE Students (
    StudentID INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),
    Age INT,
    Address NVARCHAR(200)
);
```

input.aspx(form that either will add or update the data in the database)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Input.aspx.cs"
Inherits="WebApplication1.Input" %>

<!DOCTYPE html>
<html>
<head>
    <title>Student Form</title>
</head>
<body>
    <h3>Enter Student Details</h3>
    <form id="form1" runat="server">
        <asp:Label ID="Label1" runat="server" Text="Student ID: " />
        <asp:TextBox ID="txtStudentID" runat="server" />
        <br /><br />

        <asp:Label ID="Label2" runat="server" Text="Name: " />
        <asp:TextBox ID="txtName" runat="server" />
        <br /><br />

        <asp:Label ID="Label3" runat="server" Text="Age: " />
        <asp:TextBox ID="txtAge" runat="server" />
        <br /><br />

        <asp:Label ID="Label4" runat="server" Text="Address: " />
        <asp:TextBox ID="txtAddress" runat="server" />
        <br /><br />

        <asp:Button ID="btnAdd" runat="server" Text="Add" OnClick="btnAdd_Click"
/>
        <asp:Button ID="btnUpdate" runat="server" Text="Update"
OnClick="btnUpdate_Click" />
    </form>
</body>
</html>
```

Input.aspx.cs (code to update or add new data to the student database)

```
using System;
using System.Linq;

namespace WebApplication1
{
    public partial class Input : System.Web.UI.Page
    {
        protected void btnAdd_Click(object sender, EventArgs e)
        {
            using (StudentDBContext db = new StudentDBContext())
            {
                // Add a new student
                var student = new Student
                {
                    Name = txtName.Text,
                    Age = Convert.ToInt32(txtAge.Text),
                    Address = txtAddress.Text
                };
                db.Students.Add(student);
                db.SaveChanges();
            }

            // Redirect to Display page to show updated data
            Response.Redirect("Display.aspx");
        }

        protected void btnUpdate_Click(object sender, EventArgs e)
        {
            using (StudentDBContext db = new StudentDBContext())
            {
                // Find the student by ID
                int studentId = Convert.ToInt32(txtStudentID.Text);
                var student = db.Students.SingleOrDefault(s => s.StudentID ==
studentId);

                if (student != null)
                {
                    // Update student details
                    student.Name = txtName.Text;
                    student.Age = Convert.ToInt32(txtAge.Text);
                    student.Address = txtAddress.Text;
                    db.SaveChanges();
                }
            }

            // Redirect to Display page to show updated data
            Response.Redirect("Display.aspx");
        }
    }
}
```

Display.aspx (displaying all student data in a page with delete button)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Display.aspx.cs"
Inherits="WebApplication1.Display" %>

<!DOCTYPE html>
<html>
<head>
    <title>Display Students</title>
</head>
<body>
    <h3>List of Students</h3>
    <table border="1">
        <tr>
            <th>Student ID</th>
            <th>Name</th>
            <th>Age</th>
            <th>Address</th>
            <th>Action</th>
        </tr>
        <asp:Repeater ID="StudentRepeater" runat="server">
```

```
            <ItemTemplate>
                <tr>
                    <td><%# Eval("StudentID") %></td>
                    <td><%# Eval("Name") %></td>
                    <td><%# Eval("Age") %></td>
                    <td><%# Eval("Address") %></td>
                    <td>
                        <asp:Button ID="btnDelete" runat="server" Text="Delete"
                                    CommandArgument='<%# Eval("StudentID") %>'
OnClick="btnDelete_Click" />
                    </td>
                </tr>
            </ItemTemplate>
        </asp:Repeater>
    </table>
</body>
</html>
```

Display.aspx.cs (code to fetch data from database and delete the data indie)

```
using System;
using System.Linq;

namespace WebApplication1
{
    public partial class Display : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                LoadStudentData();
            }
        }

        private void LoadStudentData()
        {
            using (StudentDBContext db = new StudentDBContext())
            {
                // Get all students and bind to Repeater
                var students = db.Students.ToList();
                StudentRepeater.DataSource = students;
                StudentRepeater.DataBind();
            }
        }

        protected void btnDelete_Click(object sender, EventArgs e)
        {
            int studentId = Convert.ToInt32(((Button)sender).CommandArgument);

            using (StudentDBContext db = new StudentDBContext())
            {
                // Find and delete the student
                var student = db.Students.SingleOrDefault(s => s.StudentID ==
studentId);
                if (student != null)
                {
                    db.Students.Remove(student);
                    db.SaveChanges();
                }
            }

            // Refresh the page to reflect the changes
            LoadStudentData();
        }
    }
}
```

Student.cs student entity class

```
public class Student
{
    public int StudentID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }
}
```

Studentdbcontext.cs (connection to database)

```
using System.Data.Entity;

public class StudentDBContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    public StudentDBContext() : base("name=StudentDBConnection")
    {
    }
}
```

26. Differentiate error and exception

- **Error**: An error is a critical issue that arises from system-level problems, like hardware failures or missing resources. These are typically unrecoverable (e.g., `OutOfMemoryError` in C#).

- **Exception**: An exception is an event caused by logical issues in the code (e.g., `NullReferenceException`). These are recoverable using structured exception handling (try-catch).

27. Matrix multiplication program

```
using System;

class MatrixMultiplication
{
    static void Main()
    {
        int[,] A = { { 1, 2 }, { 3, 4 } };
        int[,] B = { { 5, 6 }, { 7, 8 } };
        int[,] C = new int[2, 2];

        for (int i = 0; i < 2; i++)
        {
            for (int j = 0; j < 2; j++)
            {
                C[i, j] = 0;
                for (int k = 0; k < 2; k++)
                {
                    C[i, j] += A[i, k] * B[k, j];
                }
            }
        }

        for (int i = 0; i < 2; i++)
        {
            for (int j = 0; j < 2; j++)
                Console.Write(C[i, j] + " ");
            Console.WriteLine();
        }
    }
}
```

28. C# generic class and method example?

```
class GenericClass<T>
{
    public void Display(T value)
    {
        Console.WriteLine("Value: " + value);
    }
}

class Program
```

```
{
    static void Main()
    {
        GenericClass<int> obj = new GenericClass<int>();
        obj.Display(42);

        GenericClass<string> obj2 = new GenericClass<string>();
        obj2.Display("Hello");
    }
}
```

29. Differwent exception classes

- System.Exception (Base class)
- ArgumentNullException
- DivideByZeroException
- IndexOutOfRangeException
- InvalidOperationException

30. Alphabetic sort in list<>

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

class Program
{
    static void Main()
    {
        List<string> names = new List<string> { "Alice", "John", "Bob" };
        names.Sort();

        foreach (string name in names)
            Console.WriteLine(name);
    }
}
```

31. Selecting record with Kathmandu location assuming id name address and course being the attributes ADO.NET way

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = "your_connection_string";
        using (SqlConnection conn = new SqlConnection(connectionString))
        {
            conn.Open();
            SqlCommand cmd = new SqlCommand("SELECT * FROM Students WHERE Address 
= 'Kathmandu'", conn);
            SqlDataReader reader = cmd.ExecuteReader();

            while (reader.Read())
                Console.WriteLine(reader["Name"]);
        }
    }
}
```

32. Validated form method in aspx forms

<asp:TextBox ID="txtName" runat="server" />

<asp:TextBox ID="txtAge" runat="server" />

<asp:RequiredFieldValidator ControlToValidate="txtName" ErrorMessage="Name required" runat="server" />

<asp:RangeValidator ControlToValidate="txtAge" MinimumValue="18" MaximumValue="30" Type="Integer" ErrorMessage="Age 18-30 only" runat="server" />

<asp:Button ID="btnSubmit" runat="server" Text="Submit" />

33. Elements of ASP.NET

- Web Forms
- MVC
- Razor Pages
- Web API

34. C# program to demonstrate LINQ concatenate and union?

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

class Program
{
    static void Main()
    {
        List<int> list1 = new List<int> { 1, 2, 3 };
        List<int> list2 = new List<int> { 3, 4, 5 };

        var concat = list1.Concat(list2);
        var union = list1.Union(list2);

        Console.WriteLine("Concatenate: " + string.Join(", ", concat));
        Console.WriteLine("Union: " + string.Join(", ", union));
    }
}
```

35. What is an Indexer?

An **indexer** in C# allows an object to be indexed in a way similar to an array. It's a special type of property that enables classes or structs to be accessed using square brackets [].

Example:

```
public class SampleCollection
{
    private string[] arr = new string[100];
    public string this[int index]
    {
        get { return arr[index]; }
        set { arr[index] = value; }
    }
}

var collection = new SampleCollection();
collection[0] = "Hello";
Console.WriteLine(collection[0]); // Output: Hello
```

36. What is Generics, and why are Generics type-safe?

Generics in C# allow you to define classes, methods, and structures with a placeholder for a data type. It improves code reusability and type safety.

Why Generics are Type-Safe? Generics enforce compile-time type checking. This means you can't accidentally use the wrong type in a generic method or class, reducing runtime errors.

Example:

```
List<int> numbers = new List<int>();
numbers.Add(10); // Allowed
// numbers.Add("Hello"); // Compile-time error
```

37. What is the role of the base keyword in C#?

The base keyword is used to:

- Access the base class constructor.
- Access base class methods, properties, or fields.

Example:

```
public class Parent
{
    public Parent() { Console.WriteLine("Parent Constructor"); }
    public void Display() { Console.WriteLine("Parent Method"); }
}
public class Child : Parent
{
    public Child() : base() { Console.WriteLine("Child Constructor"); }
    public void Show() { base.Display(); }
}
```

38. Explain LINQ and List Five LINQ Standard Operators:

LINQ (Language Integrated Query) simplifies querying data from different sources like collections, databases, or XML.

Five Standard Operators:

- Where: Filters a sequence.
- Select: Projects values into a new form.
- OrderBy: Sorts data.
- GroupBy: Groups data.
- Aggregate: Performs operations like sum, average.

39. What is a Delegate? Explain Types of Delegates in C#:

A delegate is a type-safe function pointer that holds references to methods with the same signature.

Types:

- Single-cast delegate
- Multi-cast delegate
- Anonymous delegate
- Generic delegate (e.g., Func, Action)

---

40. Difference between DataReader, DataSet, DataAdapter:

Feature	**DataReader**	**DataSet**	**DataAdapter**
Nature	Forward-only, read-only	In-memory data storage	Bridge between DB and DataSet
Connectivity	Connected	Disconnected	Used in disconnected scenarios
Performance	Faster	Slower	Moderate

Example

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = "your_connection_string";
        using (SqlConnection conn = new SqlConnection(connectionString))
        {
            conn.Open();
            SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Students",
conn);
            DataSet ds = new DataSet();
            adapter.Fill(ds);

            DataTable dt = ds.Tables[0];
            foreach (DataRow row in dt.Rows)
                Console.WriteLine(row["Name"]);
        }
    }
}
```

41. What is an Event, and How Is It Handled in C#?

An **event** is a notification mechanism to respond to actions like a button click.

Handling Events:

```
public delegate void Notify();
public class Process
{
    public event Notify OnStart;
    public void Start() { OnStart?.Invoke(); }
}

Process process = new Process();
process.OnStart += () => Console.WriteLine("Process Started!");
process.Start();
```

---

42. Difference between Hashtable and Dictionary in C#: also what is string interpolation?

Feature	Hashtable	Dictionary
Type-Safe	No	Yes

Feature	**Hashtable**	**Dictionary**
Performance	Slower	Faster
Key/Value Types	Object	Strongly typed

String Interpolation simplifies formatting strings using placeholders.
Example:
csharp
Copy code

```
int age = 25;
Console.WriteLine($"I am {age} years old.");
```