

UNIT-II

1. Classes.

Ans: Here are the different programs that are related to the following concepts:

- **Simple class:**

```
namespace MyNameSpace
{
    class Program
    {
        static void Main(string[] args)
        {
            // Code here
        }
    }
}
```

- **Class with fields and constants:**

```
namespace MyNameSpace
{
    class Program
    {
        // Private attribute
        private readonly string? myPrivate = "Private attribute";

        // Public attribute
        public static string? myPublic = "Public attribute";

        // const attribute
        public const int myConst = 10;

        // Static attribute
        public static int myStatic = 20;

        // Main method
        static void Main(string[] args)
        {
            // Accessing private attribute
            Program obj = new();
            Console.WriteLine(obj.myPrivate);

            // Accessing static attribute
            Console.WriteLine(myStatic);

            // Accessing const attribute
            Console.WriteLine(myConst);

            // Accessing public attribute
            Console.WriteLine(myPublic);
        }
    }
}
```

- **Class with methods, constructor, deconstructor, destructor, and object initialization:**

```

namespace MyNameSpace
{
    class Program
    {
        // Field
        public int myField;

        // Constructor without a parameter
        public Program()
        {
            Console.WriteLine("Constructor called");
            this.myField = 0;
        }

        // Constructor with a parameter
        public Program(int myField)
        {
            Console.WriteLine("Constructor called");
            this.myField = myField;
        }

        // Deconstructor
        public void Deconstruct(out int myField)
        {
            Console.WriteLine("Deconstructor called");
            myField = this.myField;
        }

        // Destructor
        ~Program()
        {
            Console.WriteLine("Destructor called");
        }

        // Method
        public void MyMethod()
        {
            Console.WriteLine("My field value: " + this.myField);
        }

        // Main method
        static void Main(string[] args)
        {
            // Create an object for the class
            Program myObject = new Program();
            myObject.MyMethod();

            // Create an object for the class with a parameter
            Program myObject2 = new Program(10);
            myObject2.MyMethod();

            // Deconstruct the object

```

```

myObject.Deconstruct(out int myField);
Console.WriteLine("My field value: " + myField);

// Force garbage collection and trigger the destructor
GC.Collect();
}
}

```

- **Class with get set property accessors:**

```

namespace MyNameSpace
{
    class Program
    {
        // Field
        public int myField;

        public int GetSetMyField
        {
            get { return myField; }
            set { myField = value; }
        }

        // readonly field
        public int InitOnly { get; init; } = 20;

        // Main method
        static void Main(string[] args)
        {
            // Create an object of the Program class
            Program myObj = new()
            {
                // Set the value of the field
                GetSetMyField = 10
            };

            // Get the value of the field
            int x = myObj.GetSetMyField;

            // Print the value of the field
            Console.WriteLine(x);

            // myObj.InitOnly = 20; // Error because it is readonly

            // Print the value of the readonly field
            Console.WriteLine(myObj.InitOnly);
        }
    }
}

```

- **Class with a default Indexer and a custom Indexer:**

```

namespace MyNameSpace
{
    class Program

```

```

{
    readonly string[] sentence = "My name is vineeth.".Split(' ');

    // Custom Indexer Implementation
    public string this[int index]
    {
        get
        {
            return sentence[index];
        }

        set
        {
            sentence[index] = value;
        }
    }

    // Main method
    static void Main(string[] args)
    {
        string s = "Hello";
        string? s2 = null;

        // Default Indexer
        Console.WriteLine(s[0]); // H
        Console.WriteLine(s2?[0]); // null

        // Custom Indexer
        Program obj = new();
        Console.WriteLine(obj[0]); // My
        obj[0] = "Our";
        Console.WriteLine(obj[0]); // Our
    }
}

```

- **Class with static constructors, static fields, constants, finalizers, partial classes, partial methods, and the nameof operator:**

```

namespace MyNameSpace
{
    // Partial class declaration
    partial class PartialClass
    {
        // Partial method
        static partial void PartialMethod();
    }

    // Partial class implementation
    partial class PartialClass
    {
        // Partial method implementation
        static partial void PartialMethod()
        {

```

```

        Console.WriteLine("Partial method implemented");
    }
}

class Program
{
    // static constructor
    static Program()
    {
        Console.WriteLine("Main class initialized");
    }

    // static field
    public static int Count = 0;

    // Constant
    public const int MaxCount = 100;

    // Finalizer or Destructor
    ~Program()
    {
        Console.WriteLine("Main class destroyed");
    }

    static void Main(string[] args)
    {
        // Print static field
        Console.WriteLine("Count: " + Count);

        // Print constant
        Console.WriteLine("MaxCount: " + MaxCount);

        // Printing the name of the variable
        Console.WriteLine(nameof(Count));

        // Partial method cannot be called directly
        // PartialClass.PartialMethod();
    }
}
}

```

2. Inheritance.

Ans: Here is a simple program for inheritance:

Program.cs:

```

class BaseClass
{
    public static void Method()
    {
        Console.WriteLine("BaseClass.Method");
    }
}

// Single inheritance

```

```

class DerivedClass : BaseClass
{
    public static new void Method()
    {
        Console.WriteLine("DerivedClass.Method");
    }
}

// Multilevel inheritance
class DerivedDerivedClass : DerivedClass
{
    public static new void Method()
    {
        Console.WriteLine("DerivedDerivedClass.Method");
    }
}

// Hierarchical inheritance
class AnotherDerivedClass : BaseClass
{
    public static new void Method()
    {
        Console.WriteLine("AnotherDerivedClass.Method");
    }
}

// Multiple inheritance
interface I1
{
    void Method();
}

interface I2
{
    void Method();
}

class MultipleInheritance : I1, I2
{
    void I1.Method()
    {
        Console.WriteLine("I1.Method");
    }

    void I2.Method()
    {
        Console.WriteLine("I2.Method");
    }
}

class Program
{
    static void Main()
    {
        BaseClass.Method(); // BaseClass.Method
        DerivedClass.Method(); // DerivedClass.Method
    }
}

```

```

        DerivedDerivedClass.Method(); // DerivedDerivedClass.Method
        AnotherDerivedClass.Method(); // AnotherDerivedClass.Method
        MultipleInheritance multipleInheritance = new();
        ((I1)multipleInheritance).Method(); // I1.Method
        ((I2)multipleInheritance).Method(); // I2.Method
    }
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
BaseClass.Method
DerivedClass.Method
DerivedDerivedClass.Method
AnotherDerivedClass.Method
I1.Method
I2.Method

```

3. Object type, boxing, unboxing, static and runtime type checking, the GetType method, the typeof operator, and the ToString method.

Ans: Here's a program for object type, boxing, unboxing, static and runtime type checking, the GetType method, the typeof operator, and the ToString method.:

Program.cs:

```

namespace ObjectTypeDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Demonstrating object type and casting
            object obj1 = "Hello";
            object obj2 = 123;

            string str = (string)obj1; // Downcast to string
            int num = (int)obj2;        // Downcast to int

            Console.WriteLine(str); // Output: Hello
            Console.WriteLine(num); // Output: 123

            // Demonstrating boxing and unboxing
            int x = 10;
            object obj = x;    // Boxing

            int y = (int)obj; // Unboxing

            Console.WriteLine(y); // Output: 10

            // Demonstrating static and runtime type checking
            object obj3 = "Hello";

            if (obj3 is string) // Static type checking
            {
                string str2 = (string)obj3; // Runtime type checking
                Console.WriteLine("String value: " + str2);
            }

            // Demonstrating GetType method and typeof operator
            object obj4 = 123;

```

```

        Type type1 = obj4.GetType(); // Using GetType method
        Type type2 = typeof(int);    // Using typeof operator

        Console.WriteLine("Type of obj4: " + type1); // Output: System.Int32
        Console.WriteLine("Type of int: " + type2);   // Output: System.Int32

        // Demonstrating ToString method
        int number = 456;
        string str3 = number.ToString(); // Calling ToString on int

        Console.WriteLine("String representation of number: " + str3); // Output:
456
    }
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Hello
123
10
String value: Hello
Type of obj4: System.Int32
Type of int: System.Int32
String representation of number: 456

```

4. Structs.

Ans: Here is a simple program for structs:

Program.cs:

```

namespace StructsDemo
{
    // Define a struct for Point with X and Y coordinates
    public struct Point
    {
        // Fields X and Y are public
        public int X;
        public int Y;

        // Parameterized constructor to initialize X and Y
        public Point(int x, int y)
        {
            X = x;
            Y = y;
        }

        // Method to reset X, Y coordinates (attempting to modify fields X, Y)
        public static void ResetXY()
        {
            Console.WriteLine("Resetting X and Y coordinates");
            // X = 10;
            // Y = 20;
        }
    }

    class Program
    {

```



```

static void Main(string[] args)
{
    // Create a Point struct instance using parameterized constructor
    Point point1 = new(5, 10);

    // Display the initial values of X and Y coordinates
    Console.WriteLine("Initial Point coordinates:");
    Console.WriteLine("X: " + point1.X);
    Console.WriteLine("Y: " + point1.Y);

    // Call the ResetX method
    Point.ResetXY();

    // Attempting to modify X, Y fields outside the struct
    point1.X = 10;
    point1.Y = 20;

    // Display the values of X and Y after attempted reset
    Console.WriteLine("Point coordinates after attempted reset:");
    Console.WriteLine("X: " + point1.X);
    Console.WriteLine("Y: " + point1.Y);
}
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Initial Point coordinates:
X: 5
Y: 10
Resetting X and Y coordinates
Point coordinates after attempted reset:
X: 10
Y: 20

```

5. Access modifiers.

Ans: Here is a simple program for Access modifiers:

Test.cs:

```

class Test
{
    // Access modifiers
    // public: Accessible from any other class
    // private: Accessible only from the same class
    // protected: Accessible only from the same class or derived classes
    // internal: Accessible only from the same assembly
    // protected internal: Accessible only from the same assembly or derived
classes
    // private protected: Accessible only from the same assembly and derived
classes

    public static int myInt = 10;
    private static readonly string myString = "Hello";
    protected static double myDouble = 3.14;
    internal static bool myBool = true;
    protected static internal char myChar = 'A';
    private static protected float myFloat = 4.269f;
}

```

ProgramTest.cs:

```
class ProgramTest : Test
{
    static void Main(string[] args)
    {
        Console.WriteLine(Test.myInt);
        // Console.WriteLine(Test.myString);
        Console.WriteLine(Test.myDouble);
        Console.WriteLine(Test.myBool);
        Console.WriteLine(Test.myChar);
        Console.WriteLine(Test.myFloat);
    }
}
```

Program.cs:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Test.myInt);
        // Console.WriteLine(Test.myString);
        // Console.WriteLine(Test.myDouble);
        Console.WriteLine(Test.myBool);
        Console.WriteLine(Test.myChar);
        // Console.WriteLine(Test.myFloat);
    }
}
```

6. Interfaces.

Ans: Here is a simple program for Interfaces:

Program.cs:

```
// Define an interface
interface IShape
{
    double CalculateArea();
    double CalculatePerimeter();
}

// Implement the interface in a class
class Circle : IShape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }

    public double CalculatePerimeter()
    {
        return 2 * Math.PI * Radius;
    }
}
```

```

    }
}

// Implement the interface in another class
class Rectangle : IShape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public double CalculateArea()
    {
        return Width * Height;
    }

    public double CalculatePerimeter()
    {
        return 2 * (Width + Height);
    }
}

class Program
{
    public static void Main()
    {
        // Create instances of Circle and Rectangle
        Circle circle = new(5);
        Rectangle rectangle = new(4, 6);

        // Use the interface to call the methods
        Console.WriteLine("Circle Area: " + circle.CalculateArea());
        Console.WriteLine("Circle Perimeter: " + circle.CalculatePerimeter());

        Console.WriteLine("Rectangle Area: " + rectangle.CalculateArea());
        Console.WriteLine("Rectangle Perimeter: " + rectangle.CalculatePerimeter());
    }
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Circle Area: 78.53981633974483
Circle Perimeter: 31.41592653589793
Rectangle Area: 24
Rectangle Perimeter: 20

```

7. Enums.

Ans: Here is a simple program for Enums:

Program.cs:

```

class Program
{
    // Define an enum for days of the week
    enum DaysOfWeek

```

```

{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

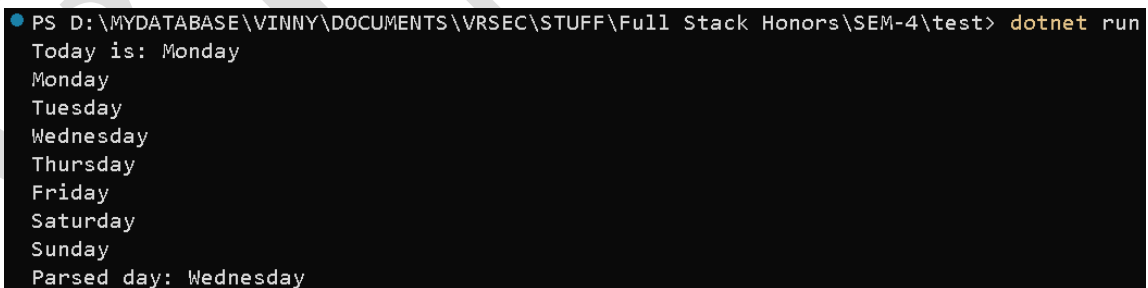
static void Main()
{
    // Accessing enum values
    DaysOfWeek today = DaysOfWeek.Monday;
    Console.WriteLine("Today is: " + today);

    // Enum iteration
    foreach (DaysOfWeek day in Enum.GetValues(typeof(DaysOfWeek)))
    {
        Console.WriteLine(day);
    }

    // Enum parsing
    string userInput = "Wednesday";
    if (Enum.TryParse(userInput, out DaysOfWeek parsedDay))
    {
        Console.WriteLine("Parsed day: " + parsedDay);
    }
    else
    {
        Console.WriteLine("Invalid input");
    }
}
}

```

Output:



```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Today is: Monday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Parsed day: Wednesday

```

8. Nested types.

Ans: Here is a simple program for Nested types:

Program.cs:

```

namespace NestedTypesDemo
{
    // Define a top-level class
    public class TopLevel
    {
        private readonly int topLevelPrivateField = 10;

        // Define a nested class
    }
}

```

```

    public class Nested
    {
        // Nested class can access private members of the enclosing class
        public static void AccessEnclosingPrivateMember(TopLevel topLevel)
        {
            Console.WriteLine("Accessing top-level private field from nested
class: " + topLevel.topLevelPrivateField);
        }
    }

    // Define a nested enum
    public enum Color
    {
        Red,
        Blue,
        Green
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Accessing nested class from outside the enclosing class
        TopLevel.Nested.AccessEnclosingPrivateMember(new TopLevel());

        // Accessing nested enum from outside the enclosing class
        TopLevel.Color color = TopLevel.Color.Blue;
        Console.WriteLine("Accessing nested enum from outside the enclosing
class: " + color);
    }
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Accessing top-level private field from nested class: 10
Accessing nested enum from outside the enclosing class: Blue

```

9. Generics.

Ans: Here is a simple program for Generics:

Program.cs:

```

class GenericClass<T>(T value)
{
    private readonly T genericField = value;

    public T GenericMethod(T parameter)
    {
        Console.WriteLine($"Generic field: {genericField}");
        Console.WriteLine($"Generic parameter: {parameter}");
        return parameter;
    }
}

class Program
{

```

```

public static void Main()
{
    GenericClass<int> intGenericClass = new(10);
    int result = intGenericClass.GenericMethod(20);
    Console.WriteLine($"Result: {result}");

    GenericClass<string> stringGenericClass = new("Hello");
    string strResult = stringGenericClass.GenericMethod("World");
    Console.WriteLine($"Result: {strResult}");
}
}

```

Output:

```

PS D:\MYDATABASE\VINNY\DOCUMENTS\VRSEC\STUFF\Full Stack Honors\SEM-4\test> dotnet run
Generic field: 10
Generic parameter: 20
Result: 20
Generic field: Hello
Generic parameter: World
Result: World

```

10. ASP.NET page structure.

Ans: Following are the important elements of an ASP.NET page:

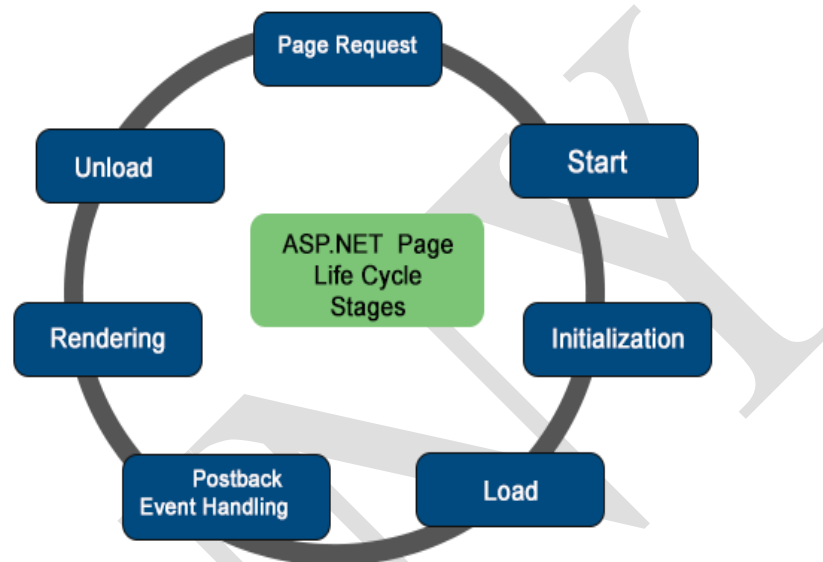
- **Directives:** These are instructions that are processed on the server before the page is sent to the client. They start with <%@ and end with %>. For example, the Page directive is used to specify page-specific attributes used by the ASP.NET page parser and compiler.
Ex: <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs" Inherits="WebApplication1.WebForm1" %>
- **Code Declaration Blocks:** These are sections of code that are executed on the server. They are enclosed in <% and %>.
Ex: <% string greeting = "Hello, ASP.NET!"; %>
- **ASP.NET Controls:** These are server-side components that encapsulate user-interface and other related functionality. They are processed on the server and output HTML to the browser.
Ex: <asp:Label ID="Label1" runat="server" Text="Hello, ASP.NET!" />
- **Code Render Blocks:** These are blocks of code that are executed on the server and their result is sent to the client. They are enclosed in <%= and %>.
Ex: <%= DateTime.Now %>
- **Server-side Comments:** These are comments that are ignored by the server and not sent to the client. They are enclosed in <%-- and --%>.
Ex: <%-- This is a server-side comment. It will not be sent to the client. --%>
- **Server-side Include Directives:** These directives allow you to include the content of another file in the current page. They are processed on the server before the page is sent to the client.
Ex: <!-- #include file="header.ascx" -->
- **Literal Text and HTML Tags:** These are sent to the client as is. They are not processed by the server.
Ex: <h1>Welcome to our website!</h1> <p>This is some text.</p>

11. Page Life Cycle.

Ans: The ASP.NET page life cycle is a series of events that happen from the time an ASP.NET page is requested to the time it is rendered to the browser and then unloaded. Understanding these events is crucial for effectively programming your ASP.NET web applications. Here are the main stages of the ASP.NET page life cycle:

- 1) **Page Request:** The page request occurs before the page life cycle begins. When the page is requested by a user, ASP.NET determines whether the page needs to be parsed and compiled or a cached version of the page can be sent in response without running the page.
- 2) **Start:** In this stage, page properties such as Request and Response are set. At this point, the page also determines whether the request is a postback or a new request and sets the IsPostBack property. The page also sets the UICulture property.

- 3) **Initialization:** During this stage, controls on the page are available and each control's UniqueID property is set. A master page and themes are also applied to the page if applicable.
- 4) **Load:** If the current request is a postback, control properties are loaded with information recovered from view state and control state.
- 5) **Postback Event Handling:** If the request is a postback, any event handlers are called.
- 6) **Rendering:** Before rendering, the view state is saved for the page and all controls. During the rendering stage, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream of the page's Response property.
- 7) **Unload:** The Unload event is raised after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded, and all cleanup is performed.



12. HTML Server Controls.

Ans: HTML Server Controls are standard HTML tags understood by the server. They are like HTML controls, but they include a `runat="server"` attribute, which means they can be manipulated on the server side. They become instances of the `HtmlControl` class, which gives them server-side properties, methods, and events. Here's a table of some common HTML controls that can be used as HTML Server Controls in ASP.NET:

HTML Control	Description
<code><input type="text" runat="server" /></code>	A text box where users can input text.
<code><input type="password" runat="server" /></code>	A text box where users can input password. The input is masked.
<code><input type="radio" runat="server" /></code>	A radio button which allows users to select one option from a group.
<code><input type="checkbox" runat="server" /></code>	A checkbox which allows users to select multiple options from a group.
<code><input type="button" runat="server" /></code>	A clickable button.
<code><input type="submit" runat="server" /></code>	A submit button that submits the form data to the server.
<code><input type="reset" runat="server" /></code>	A reset button that resets all form fields to their initial values.
<code><select runat="server" /></code>	A drop-down list that allows users to select an option from a list.
<code><textarea runat="server" /></code>	A multi-line text input control.
<code><button runat="server" /></code>	A clickable button, which can contain text or images.
<code><form runat="server" /></code>	A container for form elements.
<code></code>	An image.
<code></code>	A hyperlink.
<code><table runat="server" /></code>	A table.
<code><div runat="server" /></code>	A division or a section in an HTML document.
<code></code>	An inline container used to mark up a part of a text, or a part of a document.

13. Web Server Controls.

Ans: Web Server Controls are special ASP.NET tags understood by the server. Like HTML Server Controls, they are also processed on the server side and can be manipulated in your server-side code. However, Web Server Controls provide a higher level of abstraction and functionality compared to HTML Server Controls. Here's a table of some common Web Server Controls in ASP.NET:

Web Server Control	Description
<code><asp:TextBox runat="server" /></code>	A text box where users can input text.
<code><asp:Button runat="server" /></code>	A clickable button.
<code><asp:Label runat="server" /></code>	A control that displays text to the user.
<code><asp:DropDownList runat="server" /></code>	A drop-down list that allows users to select an option from a list.
<code><asp:CheckBox runat="server" /></code>	A checkbox which allows users to select multiple options from a group.
<code><asp:RadioButton runat="server" /></code>	A radio button which allows users to select one option from a group.
<code><asp:ListBox runat="server" /></code>	A list box that allows users to select one or more options.
<code><asp:Image runat="server" /></code>	A control that displays an image.
<code><asp:HyperLink runat="server" /></code>	A control that displays a hyperlink.
<code><asp:Calendar runat="server" /></code>	A calendar control that allows users to select dates.
<code><asp:GridView runat="server" /></code>	A control that displays data in a grid format.
<code><asp:DataList runat="server" /></code>	A control that displays data in a format that you can customize using templates.
<code><asp:Repeater runat="server" /></code>	A control that displays a repeated list of items that are bound to the control.
<code><asp:FileUpload runat="server" /></code>	A control that allows users to upload files.
<code><asp:CheckBoxList runat="server" /></code>	A group of checkbox controls that allows users to select multiple options.
<code><asp:RadioButtonList runat="server" /></code>	A group of radio button controls that allows users to select one option.

14. Web User Controls.

Ans: Web User Controls in ASP.NET are reusable page fragments that can be embedded in multiple ASP.NET web pages. They are like complete web pages but are used for repeating blocks of HTML and server-side code. Web User Controls are saved with .ascx extension and can contain static HTML, Web Server Controls, and code for event handlers, just like an ASP.NET page. However, they cannot be requested independently; they must be included in an ASP.NET page using a UserControl directive.

Ex:

MyUserControl.ascx:

```
<%@ Control Language="C#" AutoEventWireup="true" %>
<asp:Label ID="Label1" runat="server" Text="Hello from the User Control!" />
```

Default.aspx:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<%@ Register Src="MyUserControl.ascx" TagPrefix="uc" TagName="MyControl" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <uc:MyControl runat="server" ID="UserControl1" />
    </form>
</body>
</html>
```


15. Validation controls.

Ans: Validation controls in ASP.NET are used to validate user input on both the client-side and server-side. They ensure that the user input matches the expected format before the form data is submitted, reducing the need for manual error checking and exception handling. Here's a table of some common Validation Controls in ASP.NET:

Validation Control	Description
<code><asp:RequiredFieldValidator runat="server" /></code>	Ensures the user does not skip an entry.
<code><asp:CompareValidator runat="server" /></code>	Compares the user's entry against a constant value, another control's value, or checks if the entry is of a specific data type.
<code><asp:RangeValidator runat="server" /></code>	Checks if the user's entry falls within a certain range of values.
<code><asp:RegularExpressionValidator runat="server" /></code>	Checks if the user's entry matches a pattern defined by a regular expression.
<code><asp:CustomValidator runat="server" /></code>	Allows you to write your own function to validate the user's entry.
<code><asp:ValidationSummary runat="server" /></code>	Displays a summary of all validation errors in a page.

16. Custom web controls.

Ans: Custom Web Controls, also known as Custom Server Controls, are controls that you create according to your own requirements. They are more complex than User Controls because they are created as compiled code (DLL files), which makes them easier to share and reuse across multiple projects. Custom Web Controls are derived from the `System.Web.UI.Control` class or from one of the existing Web Server Controls. They encapsulate their own rendering logic and can have their own properties, methods, and events.

Ex:

```
using System.Web.UI;  
using System.Web.UI.WebControls;
```

```
public class MyCustomControl : WebControl  
{  
    protected override void RenderContents(HtmlTextWriter output)  
    {  
        output.Write("Hello, this is a custom web control!");  
    }  
}
```

Usage:

```
<%@ Register Namespace="MyNamespace" Assembly="MyAssembly" TagPrefix="custom" %>  
<custom:MyCustomControl runat="server" />
```