1. ***With a neat block diagram explain the functions of the various building blocks in the architecture of the .NET framework***

**1. Applications**

Applications are the end products developed using the .NET framework. They make use of the class library and run under the CLR.

* They can be of different types: **Web Applications, Windows Applications, Mobile Applications, or Console Applications**.
* Applications can be developed in multiple languages such as **C#, VB.NET, or F#**.
* The same application can work across different platforms due to the **Intermediate Language (IL)**.
* Applications interact with the user and provide solutions to real-world problems.
* They form the topmost layer of the .NET architecture.

**2. .NET Framework Class Library (FCL / BCL)**

The **Class Library** provides a rich collection of reusable types and functions.

* **FCL (Framework Class Library):** A set of classes, interfaces, and APIs that developers can use directly.
* **BCL (Base Class Library):** Core part of FCL which provides fundamental features.
* It contains classes for **File I/O, String handling, Collections, and Networking**.
* Provides **Database connectivity through ADO.NET** and supports **XML processing**.
* Saves development time by offering pre-built and tested functionalities, ensuring consistency across languages.

**3. Common Language Runtime (CLR)**

The **CLR** is the heart of the .NET framework, also known as the execution engine.

* It converts **Intermediate Language (IL)** into machine code using the **Just-In-Time (JIT) compiler**.
* Performs **Automatic Garbage Collection** to manage memory efficiently.
* Provides **Exception Handling** to catch and handle runtime errors.
* Ensures **Security and Type Safety** so that unsafe code cannot harm the system.
* Allows **language interoperability**, meaning code written in different .NET languages can work together.

**4. Common Language Infrastructure (CLI)**

The **CLI** is a specification that defines how the .NET runtime should work.

* It uses **Intermediate Language (IL)** for platform independence.
* **Assemblies and Metadata** contain information about the program, such as version and type definitions.
* Provides **cross-language integration**, so developers can combine C#, VB.NET, and other languages in the same project.
* Makes programs **portable**, so they can run on different platforms with a suitable runtime.
* Ensures that all .NET languages follow common rules and standards.

**5. Key Features of .NET Architecture**

* **Language Interoperability:** Programs can be developed in multiple languages and still work together.
* **Managed Execution:** All code runs under the supervision of CLR.
* **Rich Class Library:** Large set of reusable classes and APIs.
* **Automatic Memory Management:** Handled by Garbage Collector.
* **Security and Reliability:** Provides code access security and type safety.

**Conclusion**

The **.NET Framework architecture** is designed to provide a secure, consistent, and language-independent development platform.  
The combination of **Applications, Class Library, CLR, and CLI** ensures that developers can create robust, scalable, and high-performance applications.

1. ***What are value types and reference types in c#? Explain ref, out with examples.***

**Value Types and Reference Types in C#**

**1. Value Types**

* Value types store the actual **data directly** in memory.
* They are stored in the **stack**.
* When assigned to another variable, a **copy** of the value is made.
* Examples: int, float, double, char, bool, struct, enum.

**Example:**

int a = 10;

int b = a; // b gets a copy of a

b = 20;

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

Console.WriteLine(b); // Output: 20

Here, changing b does not affect a because they hold **separate copies**.

**2. Reference Types**

* Reference types store a **reference (address)** to the actual data.
* They are stored in the **heap**, while the reference is on the stack.
* When assigned to another variable, both point to the **same object**.
* Examples: class, object, string, array, delegate.

**Example:**

class Student {

public string name;

}

Student s1 = new Student();

s1.name = "John";

Student s2 = s1; // s2 refers to the same object as s1

s2.name = "Mike";

Console.WriteLine(s1.name); // Output: Mike

Here, s1 and s2 refer to the **same object in memory**.

**3. Difference Between Value Types and Reference Types**

| **Value Types** | **Reference Types** |
| --- | --- |
| Store data directly. | Store reference (address). |
| Stored in **Stack**. | Stored in **Heap** (reference in stack). |
| Copy is created on assignment. | Reference is shared on assignment. |
| Examples: int, float, struct. | Examples: class, string, array. |

**4. The ref Keyword**

* Used to **pass arguments by reference**.
* The variable must be **initialized before** it is passed.
* Any changes inside the method affect the original variable.

**Example:**

class Program {

static void Square(ref int x) {

x = x \* x;

}

static void Main() {

int num = 5;

Square(ref num);

Console.WriteLine(num); // Output: 25

}

}

Here, the value of num is modified inside the method because it was passed using ref.

**5. The out Keyword**

* Also used to **pass arguments by reference**.
* The variable **need not be initialized before** passing.
* The method must **assign a value** before returning.

**Example:**

class Program {

static void Divide(int a, int b, out int quotient, out int remainder) {

quotient = a / b;

remainder = a % b;

}

static void Main() {

int q, r;

Divide(10, 3, out q, out r);

Console.WriteLine("Quotient: " + q); // Output: 3

Console.WriteLine("Remainder: " + r); // Output: 1

}

}

Here, q and r are assigned values inside the method and then used outside.

**Conclusion**

* **Value types** store data directly, while **reference types** store a reference to data.
* The **ref keyword** is used to pass variables **by reference** when they are already initialized.
* The **out keyword** is used to return multiple values from a method, and the variables need not be initialized before passing.

1. ***Write in detail on boxing and unboxing with example.***

**Boxing and Unboxing in C#**

**Introduction**

C# is an object-oriented language that supports both **value types** (like int, float, struct) and **reference types** (like object, class).  
Since object is the base class of all types in C#, there is sometimes a need to **convert value types into reference types** and vice versa.  
This is done using **Boxing** and **Unboxing**.

**1. Boxing**

* **Definition:** Boxing is the process of **converting a value type into an object type** (reference type).
* The value is stored in the **heap memory**.
* It allows value types (like int, float) to be treated as objects.
* This happens **implicitly** (automatically) or can be done explicitly.

**Example of Boxing:**

int num = 100; // Value type

object obj = num; // Boxing - num is converted into object

Console.WriteLine("Value of object: " + obj);

Here, the integer num is **boxed** into the reference variable obj.

**2. Unboxing**

* **Definition:** Unboxing is the process of **extracting the value type from an object type**.
* The value is copied from the **heap back to the stack**.
* It must be done **explicitly** (type casting required).
* If the cast is invalid, it will throw an exception.

**Example of Unboxing:**

object obj = 200; // Boxing

int num = (int)obj; // Unboxing

Console.WriteLine("Value of num: " + num);

Here, the object obj is **unboxed** back into the integer num.

**3. Combined Example of Boxing and Unboxing**

using System;

class Program {

static void Main() {

int x = 50; // Value type

object obj = x; // Boxing

int y = (int)obj; // Unboxing

Console.WriteLine("Original value: " + x);

Console.WriteLine("Boxed value: " + obj);

Console.WriteLine("Unboxed value: " + y);

}

}

**Output:**

Original value: 50

Boxed value: 50

Unboxed value: 50

**4. Difference Between Boxing and Unboxing**

| **Boxing** | **Unboxing** |
| --- | --- |
| Converts **value type → reference type**. | Converts **reference type → value type**. |
| Performed **implicitly** by compiler. | Must be done **explicitly** with casting. |
| Stores data in the **heap**. | Retrieves data back to the **stack**. |
| Example: object obj = x; | Example: int y = (int)obj; |
| May cause **performance overhead**. | Also costly if done frequently. |

**5. Conclusion**

* **Boxing** and **Unboxing** provide a way to work with value types as objects.
* **Boxing** happens when a value type is stored in an object or interface type.
* **Unboxing** happens when the value is extracted back from the object.
* Though useful, frequent boxing/unboxing should be avoided as it causes **performance overhead**.

1. ***Write a short note on the following i) Simple array ii) Multi-dimensional Array iii) Jagged array***

**Arrays in C#**

An **array** is a collection of elements of the same type stored in **contiguous memory locations**.  
In C#, arrays are objects and their size is fixed after creation.  
There are three main types: **Simple Array, Multi-Dimensional Array, and Jagged Array**.

**i) Simple Array (Single-Dimensional Array)**

* Stores elements in a **single row**.
* Accessed using **one index**.
* Index starts from **0**.

**Syntax:**

datatype[] arrayName = new datatype[size];

**Example:**

int[] marks = new int[5] { 10, 20, 30, 40, 50 };

for (int i = 0; i < marks.Length; i++)

Console.WriteLine(marks[i]);

**Output:**

10

20

30

40

50

**ii) Multi-Dimensional Array**

* Stores elements in **rows and columns** (like a matrix).
* Accessed using **two or more indices**.
* Declared with a comma [,].

**Syntax:**

datatype[,] arrayName = new datatype[rows, columns];

**Example:**

int[,] matrix = new int[2, 3] {

{ 1, 2, 3 },

{ 4, 5, 6 }

};

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 3; j++) {

Console.Write(matrix[i, j] + " ");

}

Console.WriteLine();

}

**Output:**

1 2 3

4 5 6

**iii) Jagged Array**

* An **array of arrays**, meaning each row can have **different lengths**.
* More flexible than multi-dimensional arrays.
* Declared with double square brackets [][].

**Syntax:**

datatype[][] arrayName = new datatype[rows][];

**Example:**

int[][] jagged = new int[3][];

jagged[0] = new int[] { 1, 2 };

jagged[1] = new int[] { 3, 4, 5 };

jagged[2] = new int[] { 6 };

for (int i = 0; i < jagged.Length; i++) {

for (int j = 0; j < jagged[i].Length; j++) {

Console.Write(jagged[i][j] + " ");

}

Console.WriteLine();

}

**Output:**

1 2

3 4 5

6

**Conclusion**

* **Simple Array** → One row, fixed size.
* **Multi-Dimensional Array** → Rows & columns (matrix).
* **Jagged Array** → Array of arrays, rows can be uneven.

***5. Write a c# program to read names from the console, sort it on the ascending order and again print it on the console.***

*using System;*

*class SortNames*

*{*

*static void Main()*

*{*

*Console.Write("Enter the number of names: ");*

*int n = int.Parse(Console.ReadLine());*

*string[] names = new string[n];*

*// Input names*

*Console.WriteLine("Enter the names:");*

*for (int i = 0; i < n; i++)*

*{*

*names[i] = Console.ReadLine();*

*}*

*// Sort names in ascending order*

*Array.Sort(names);*

*// Print sorted names*

*Console.WriteLine("\nNames in Ascending Order:");*

*foreach (string name in names)*

*{*

*Console.WriteLine(name);*

*}*

*Console.ReadLine();*

*}*

*}*

***6. Explain Sealed class, Sealed method and new method with suitable examples.***

**1) Sealed Class**

**Definition:**

* A **sealed class** is a class that cannot be inherited.
* Declared using the sealed keyword.
* Useful when you want to **restrict inheritance** for security, performance, or design reasons.

**Syntax:**

sealed class ClassName

{

// Members

}

**Example:**

sealed class A

{

public void Show()

{

Console.WriteLine("This is a sealed class.");

}

}

// ❌ Error: Cannot inherit from sealed class

// class B : A { }

**2) Sealed Method**

**Definition:**

* A **sealed method** is a method that cannot be overridden further in derived classes.
* Used with sealed override.
* Allows overriding **only once** and then locks it.

**Syntax:**

class BaseClass

{

public virtual void MethodName() { }

}

class DerivedClass : BaseClass

{

public sealed override void MethodName()

{

// Implementation

}

}

**Example:**

class A

{

public virtual void Display()

{

Console.WriteLine("Base class Display method.");

}

}

class B : A

{

public sealed override void Display()

{

Console.WriteLine("Derived class sealed Display method.");

}

}

class C : B

{

// ❌ Error: Cannot override sealed method

// public override void Display() { }

}

**3) New Method**

**Definition:**

* A **new method** hides a method of the base class with the same name.
* Declared with the new keyword.
* It does **not override**, but **hides** the base class method.

**Syntax:**

class BaseClass

{

public void MethodName() { }

}

class DerivedClass : BaseClass

{

public new void MethodName() { }

}

**Example:**

class A

{

public void Show()

{

Console.WriteLine("Base class Show method.");

}

}

class B : A

{

public new void Show()

{

Console.WriteLine("Derived class new Show method.");

}

}

class Test

{

static void Main()

{

A obj1 = new A();

obj1.Show(); // Base class Show method

B obj2 = new B();

obj2.Show(); // Derived class new Show method

A obj3 = new B();

obj3.Show(); // Base class Show method

}

}

**Comparison Table**

| **Feature** | **Keyword Used** | **Effect on Inheritance** |
| --- | --- | --- |
| **Sealed Class** | sealed class | Prevents inheritance |
| **Sealed Method** | sealed override | Prevents further overriding |
| **New Method** | new | Hides base class method, not override |

***7. Define static and instance methods, also write it the difference between them***

**1) Static Method**

**Definition:**

* A **static method** belongs to the **class itself**, not to any specific object.
* It is declared using the static keyword.
* Can be called using the **class name directly**, without creating an object.
* Cannot access instance variables or instance methods directly.

**Syntax:**

class ClassName

{

public static void MethodName()

{

// code

}

}

**Example:**

class Calculator

{

public static int Add(int a, int b)

{

return a + b;

}

}

class Test

{

static void Main()

{

// Calling static method without object

Console.WriteLine(Calculator.Add(5, 10));

}

}

**2) Instance Method**

**Definition:**

* An **instance method** belongs to a specific **object (instance)** of a class.
* Declared without the static keyword.
* Requires an object of the class to be invoked.
* Can access both instance variables and static variables.

**Syntax:**

class ClassName

{

public void MethodName()

{

// code

}

}

**Example:**

class Student

{

public string name;

public void ShowName() // instance method

{

Console.WriteLine("Student name: " + name);

}

}

class Test

{

static void Main()

{

Student s1 = new Student();

s1.name = "Ravi";

s1.ShowName(); // called using object

}

}

**3) Difference Between Static and Instance Methods**

| **Feature** | **Static Method** | **Instance Method** |
| --- | --- | --- |
| **Keyword** | Declared using static | Declared without static |
| **Belongs To** | Class itself | Individual object (instance) |
| **Access** | Called using **ClassName.Method()** | Called using **object.Method()** |
| **Object Required?** | ❌ Not required | ✅ Required |
| **Access to Members** | Can access only static members directly | Can access both static & instance members |
| **Example** | Calculator.Add(5, 10) | student.ShowName() |

***8. What is constructor? Illustrate how the superclass constructors are called in their base classes.***

**1) Constructor**

**Definition:**

* A **constructor** is a special method in a class that is **automatically called** when an object is created.
* It is used to **initialize the data members** of a class.
* A constructor has the **same name as the class** and does **not return any value** (not even void).

**Syntax:**

class ClassName

{

public ClassName() // Constructor

{

// Initialization code

}

}

**Example:**

class Student

{

public string name;

// Constructor

public Student(string n)

{

name = n;

}

public void Show()

{

Console.WriteLine("Name: " + name);

}

}

class Test

{

static void Main()

{

Student s1 = new Student("Ravi"); // constructor called automatically

s1.Show();

}

}

**2) Calling Superclass Constructors**

* In **inheritance**, when a derived (child) class object is created:
  1. The **base class constructor** is called first.
  2. Then the **derived class constructor** is called.
* This ensures that the **base part of the object is initialized before the derived part**.
* The base keyword is used to explicitly call a specific constructor of the base class.

**Syntax:**

class BaseClass

{

public BaseClass()

{

Console.WriteLine("Base class constructor called.");

}

}

class DerivedClass : BaseClass

{

public DerivedClass() : base() // calling base constructor

{

Console.WriteLine("Derived class constructor called.");

}

}

**Example:**

class Person

{

public Person()

{

Console.WriteLine("Person (base) constructor called.");

}

}

class Student : Person

{

public Student() : base() // calls base class constructor

{

Console.WriteLine("Student (derived) constructor called.");

}

}

class Test

{

static void Main()

{

Student s1 = new Student();

}

}

**Output:**

Person (base) constructor called.

Student (derived) constructor called.

**3) Key Points**

1. Constructors are **special methods** for object initialization.
2. If a base class has a constructor, it is **always executed before** the derived class constructor.
3. The base keyword is used to call **parameterized constructors** of the base class.
4. If no constructor is defined, C# provides a **default constructor** automatically.
5. Ensures **proper initialization order** in inheritance hierarchy.

***9. What is interface how it is different from abstract class? Illustrate how to accomplish with an example***

**1) Definition of Interface**

* An **interface** in C# is a collection of **abstract methods, properties, events, or indexers** that does not contain implementation.
* It defines a **contract** that a class or struct must follow.
* A class that implements an interface must provide implementation for **all members** of the interface.
* Declared using the interface keyword.

**2) Syntax**

interface IShape

{

void Draw(); // abstract method (no body)

}

class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing Circle");

}

}

**3) Example**

using System;

interface IShape

{

void Draw();

}

class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing Circle");

}

}

class Rectangle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing Rectangle");

}

}

class Test

{

static void Main()

{

IShape s1 = new Circle();

s1.Draw(); // Drawing Circle

IShape s2 = new Rectangle();

s2.Draw(); // Drawing Rectangle

}

}

**4) Difference between Interface and Abstract Class**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Keyword** | abstract | interface |
| **Implementation** | Can have **both implemented and abstract** methods | Cannot have method implementation (only definitions, except default interface methods in C# 8+) |
| **Multiple Inheritance** | A class can inherit **only one** abstract class | A class can implement **multiple** interfaces |
| **Variables** | Can have fields (variables) | Cannot have fields (only properties, events, indexers) |
| **Access Modifiers** | Can use access modifiers (public, protected) | All members are **public** by default |
| **Constructor** | Can have constructors | Cannot have constructors |

**5) Key Points**

1. **Interface** defines a contract → classes must follow it.
2. **Abstract class** can provide partial implementation + contract.
3. **Interfaces support multiple inheritance** (a class can implement many interfaces).
4. Use **abstract class** when classes are closely related; use **interface** for unrelated classes that just need a common behavior.

***10. What is late binding? How is it achieved? Explain with a suitable example.***

**1) Definition**

* **Late Binding** means that the **method to be called is determined at runtime** (not at compile time).
* It is also called **dynamic polymorphism**.
* Achieved in C# using **virtual methods and overriding**, or by using the **dynamic keyword**.
* Opposite of **early binding**, where method resolution happens at compile time.

**2) How it is Achieved**

Late Binding in C# can be achieved by:

1. Using **Virtual Methods** and **Method Overriding** (with virtual and override).
2. Using the **dynamic keyword** (method binding happens at runtime).

**3) Example – Using Virtual and Override**

using System;

class Shape

{

public virtual void Draw() // virtual method

{

Console.WriteLine("Drawing Shape");

}

}

class Circle : Shape

{

public override void Draw() // override method

{

Console.WriteLine("Drawing Circle");

}

}

class Test

{

static void Main()

{

Shape s; // base class reference

s = new Circle(); // refers to derived object

s.Draw(); // Late Binding → Output: Drawing Circle

}

}

**Explanation:**

* s is a reference of base class Shape, but points to Circle.
* At runtime, the CLR decides to call Circle.Draw() instead of Shape.Draw().
* This is **late binding** (decision taken at runtime).

**4) Example – Using Dynamic Keyword**

using System;

class Demo

{

public void Show()

{

Console.WriteLine("Hello from Show()");

}

}

class Test

{

static void Main()

{

dynamic obj = new Demo();

obj.Show(); // Late binding (resolved at runtime)

}

}

**5) Key Points**

1. Late Binding = **method resolution at runtime**.
2. Achieved by **virtual + override** or **dynamic keyword**.
3. Provides **flexibility**, but slightly slower than early binding.
4. Commonly used in **polymorphism** and when working with **COM objects / reflection**.

***11. Explain the user defined exception in c#. Describe the syntax with an example.***

**1) Definition**

* In C#, an **exception** is an error that occurs during program execution.
* Along with built-in exceptions (DivideByZeroException, NullReferenceException, etc.), we can create our own exceptions — called **User Defined Exceptions**.
* A **user defined exception** is a class created by the programmer that **inherits from the base class Exception**.
* Used when built-in exceptions do not describe the error condition properly.

**2) Syntax**

// Step 1: Define a custom exception class

class MyException : Exception

{

public MyException(string message) : base(message)

{

}

}

// Step 2: Use it in program

class Test

{

static void Main()

{

try

{

// Raise custom exception

throw new MyException("This is a user defined exception.");

}

catch (MyException ex)

{

Console.WriteLine("Caught Exception: " + ex.Message);

}

}

}

**3) Example – User Defined Exception**

using System;

class AgeException : Exception

{

public AgeException(string message) : base(message)

{

}

}

class Student

{

public int age;

public void SetAge(int a)

{

if (a < 18)

{

// throw user defined exception

throw new AgeException("Age must be 18 or above.");

}

else

{

age = a;

Console.WriteLine("Age set to: " + age);

}

}

}

class Test

{

static void Main()

{

try

{

Student s = new Student();

s.SetAge(15); // invalid age

}

catch (AgeException ex)

{

Console.WriteLine("Error: " + ex.Message);

}

}

}

**Output:**

Error: Age must be 18 or above.

**4) Key Points**

1. User defined exceptions are created by **inheriting from System.Exception**.
2. Constructor usually passes a **message** to the base Exception class.
3. Raised using throw new MyException("message").
4. Handled using try-catch block.
5. Useful for **application-specific errors** (like Invalid Age, Insufficient Balance, etc.).

***12. Explain in brief about the Life Cycle Events of ASP.NET.***

**1) Definition**

* The **ASP.NET Page Life Cycle** describes the sequence of events that occur from the **time a page request is made** in the browser until the page is **rendered and unloaded**.
* Understanding these events helps developers write code at the correct stage of execution (e.g., initialization, loading data, rendering UI, cleanup).

**2) Major Life Cycle Events**

Here are the important events in order:

**1. Page Request**

* Occurs when the user requests an ASP.NET page.
* ASP.NET decides whether to **compile the page** or serve it from cache.

**2. Start**

* Page properties like **Request** and **Response** objects are set.
* Page determines whether it is a **postback** (form submitted) or a **new request**.

**3. Initialization (Page\_Init)**

* Controls on the page are initialized.
* Each control gets a **unique ID**.
* No data is loaded at this stage.

**4. Load (Page\_Load)**

* Page and its controls are loaded with data.
* **Most of the code to populate controls** (like dropdowns, grids) is written here.

**5. PostBack Event Handling**

* If the request is a **postback**, the events (like button click, dropdown change) are handled here.

**6. PreRender**

* Final changes to the page or its controls can be made **before rendering**.
* Last event where you can modify data.

**7. Render**

* ASP.NET generates the **HTML output** and sends it to the browser.

**8. Unload**

* Cleanup stage.
* Used to **release resources** (like database connections, file handles).
* Page is discarded from memory.

**3) Diagram – Page Life Cycle**

Page Request → Start → Initialization → Load → Postback Event Handling → PreRender → Render → Unload

**4) Example (Key Events in Code)**

protected void Page\_Init(object sender, EventArgs e)

{

Response.Write("Page Initialization<br/>");

}

protected void Page\_Load(object sender, EventArgs e)

{

Response.Write("Page Load<br/>");

}

protected void Page\_PreRender(object sender, EventArgs e)

{

Response.Write("Page PreRender<br/>");

}

protected void Page\_Unload(object sender, EventArgs e)

{

// Cleanup code here

}

**5) Key Points**

1. ASP.NET Life Cycle has **well-defined stages**.
2. Page\_Init → control initialization.
3. Page\_Load → load data into controls.
4. PostBack Event Handling → handle user actions.
5. PreRender → last modification.
6. Unload → cleanup resources.

***13. Brief about three ways to program in ASP.net.***

**1. Web Forms**

* **Definition:** Web Forms are event-driven and allow rapid development using a drag-and-drop interface.
* **Features:**
  1. Uses controls like TextBox, Button, GridView, etc.
  2. Supports event handling (e.g., Button\_Click).
  3. Provides ViewState to maintain data between requests.
  4. Fast for small to medium applications.
  5. Suitable for developers familiar with Windows Forms.

**Example:**

<asp:Button ID="btnSubmit" runat="server" Text="Click" OnClick="btnSubmit\_Click" />

**2. MVC (Model–View–Controller)**

* **Definition:** MVC is a design pattern that separates an application into three components: Model, View, and Controller.
* **Features:**
  1. Clear separation of concerns.
  2. Easy for unit testing and large applications.
  3. Model = Business Logic, View = UI, Controller = Request Handling.
  4. Provides full control over HTML, CSS, and JavaScript.
  5. Popular for enterprise-level web apps.

**Example:**

public class HomeController : Controller {

public ActionResult Index() {

return View();

}

}

**3. ASP.NET Core (Modern Approach)**

* **Definition:** ASP.NET Core is a cross-platform, open-source framework for building web apps and APIs.
* **Features:**
  1. Works on Windows, Linux, and macOS.
  2. High performance and lightweight.
  3. Supports dependency injection.
  4. Unified model for building web apps, REST APIs, and services.
  5. Suitable for cloud-based and modern apps.

**Example:**

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello ASP.NET Core!");

app.Run();

**Summary:**

* **Web Forms** → Quick development, event-driven, simple apps.
* **MVC** → Structured, testable, scalable apps.
* **ASP.NET Core** → Modern, cross-platform, high performance apps.

***14. Explain in detail about ASP intrinsic objects and web forms in ASP.NET***

**ASP.NET Intrinsic Objects**

Intrinsic objects are built-in objects provided by ASP.NET to handle requests, responses, sessions, and application data. They are created automatically by the ASP.NET runtime and available in all web forms.

**Major Intrinsic Objects:**

1. **Request Object**
   * Used to get information sent by the client (browser) to the server.
   * Can read query string, form data, cookies, and headers.
   * **Example:**
   * string name = Request.QueryString["username"];
2. **Response Object**
   * Used to send output (text, HTML, cookies, files) to the client browser.
   * **Example:**
   * Response.Write("Welcome to ASP.NET!");
3. **Server Object**
   * Provides utility methods such as URL encoding and error handling.
   * **Example:**
   * string encodedUrl = Server.UrlEncode("my page.aspx");
4. **Session Object**
   * Stores user-specific data on the server for the duration of a session.
   * Each user gets a unique session ID.
   * **Example:**
   * Session["UserName"] = "John";
5. **Application Object**
   * Used to store data that can be shared among all users of the application.
   * **Example:**
   * Application["Visitors"] = (int)Application["Visitors"] + 1;

✅ These objects help manage **user input, output, state management, and communication** between client and server.

**Web Forms in ASP.NET**

Web Forms is a programming model in ASP.NET used to build dynamic, data-driven web applications with a **drag-and-drop, event-driven approach**.

**Features of Web Forms:**

1. **Event-driven Programming** – Handles user actions like button clicks, text changes, etc.
2. **Rich Server Controls** – Provides controls like TextBox, DropDownList, GridView.
3. **ViewState** – Maintains values of controls between postbacks.
4. **Code-Behind Model** – Separates UI (ASPX file) and logic (C# code-behind file).
5. **Rapid Development** – Easy to build apps with minimal coding.

**Syntax Example of a Web Form**

**ASPX Page (Frontend):**

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

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

<asp:Label ID="lblMessage" runat="server"></asp:Label>

**Code-Behind (C#):**

protected void btnSubmit\_Click(object sender, EventArgs e)

{

lblMessage.Text = "Hello, " + txtName.Text;

}

✅ Here, the TextBox takes input, Button triggers an event, and Label displays the output.

**Conclusion**

* **ASP.NET Intrinsic Objects** (Request, Response, Session, Server, Application) provide built-in functionality for handling user input, server response, and state management.
* **Web Forms** is an event-driven framework that simplifies web development by using controls, ViewState, and a code-behind model.

Together, intrinsic objects and web forms make ASP.NET a **powerful and developer-friendly framework** for building interactive web applications.

***15. Explain 3 types of button controls available in ASP.NET.***

**1. Button Control (<asp:Button>)**

* **Definition:**  
  The standard button control is used to submit a form or trigger an event on the server.
* **Features:**
  1. Raises the Click event when pressed.
  2. Can perform server-side processing.
  3. Can be customized with text and style.
  4. Frequently used for actions like **submit, save, login, or update**.
  5. Default button type in ASP.NET.

**Syntax Example:**

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

**Code-Behind (C#):**

protected void btnSubmit\_Click(object sender, EventArgs e)

{

Response.Write("Button clicked!");

}

**2. LinkButton Control (<asp:LinkButton>)**

* **Definition:**  
  A LinkButton looks like a **hyperlink** but works like a button.
* **Features:**
  1. Raises server-side events when clicked.
  2. Useful for navigation menus or links with server-side logic.
  3. Supports styles (can look like normal text link).
  4. Does not redirect directly—it posts back and executes server code.
  5. Helpful in **GridView, Repeater controls** for edit/delete links.

**Syntax Example:**

<asp:LinkButton ID="lnkDetails" runat="server" Text="View Details" OnClick="lnkDetails\_Click" />

**Code-Behind (C#):**

protected void lnkDetails\_Click(object sender, EventArgs e)

{

Response.Write("LinkButton clicked!");

}

**3. ImageButton Control (<asp:ImageButton>)**

* **Definition:**  
  An ImageButton displays an **image** instead of text and works like a button.
* **Features:**
  1. Displays an image as clickable button.
  2. Raises Click and Command events.
  3. Returns mouse coordinates (X and Y) where the user clicked.
  4. Useful for image-based navigation (e.g., icons for submit, search).
  5. Can improve UI design and user experience.

**Syntax Example:**

<asp:ImageButton ID="imgBtn" runat="server"

ImageUrl="~/images/submit.png"

OnClick="imgBtn\_Click" />

**Code-Behind (C#):**

protected void imgBtn\_Click(object sender, ImageClickEventArgs e)

{

Response.Write("ImageButton clicked at X=" + e.X + " Y=" + e.Y);

}

**Conclusion**

* **Button** → Standard clickable button for form submission.
* **LinkButton** → Looks like a hyperlink but executes server-side code.
* **ImageButton** → Displays an image as a button and can capture click coordinates.

👉 These three button controls allow ASP.NET developers to design **user-friendly and functional web applications** with different styles and behaviors.

***16. What is a Master Page in ASP.NET? Explain the concept with appropriate examples.***

**Definition**

A **Master Page** in ASP.NET defines a **common layout** (template) that can be shared across multiple web pages in a web application. It allows developers to maintain a consistent design (such as header, footer, and navigation menu) across all pages.

* A master page has the extension **.master**.
* The content pages are linked to the master page and insert their own content into placeholders.
* At runtime, ASP.NET merges the master page with the content page into a single webpage.

**Key Features of Master Page**

1. **Consistent Layout** – Ensures the same look (header, footer, menu) on all pages.
2. **ContentPlaceHolder** – Special control where content pages insert their unique data.
3. **Code Reusability** – Common code written once can be reused in multiple pages.
4. **Easy Maintenance** – Changes in master page automatically apply to all linked pages.
5. **Separation of Layout and Content** – Designers handle master layout, while developers focus on content.

**Structure of a Master Page**

The master page defines the **common parts of the UI** and places a ContentPlaceHolder for child pages to insert their content.

**Example: Master Page (Site.master)**

<%@ Master Language="C#" AutoEventWireup="true" %>

<!DOCTYPE html>

<html>

<head runat="server">

<title>My Website</title>

</head>

<body>

<div style="background-color: lightblue; padding:10px;">

<h1>My Website Header</h1>

<hr />

</div>

<!-- Placeholder for child page content -->

<asp:ContentPlaceHolder ID="MainContent" runat="server"></asp:ContentPlaceHolder>

<div style="background-color: lightgray; padding:10px; margin-top:20px;">

<p>Footer: © 2025 My Website</p>

</div>

</body>

</html>

**Structure of a Content Page**

The content page uses the MasterPageFile attribute to link with the master page. It then provides content for the ContentPlaceHolder.

**Example: Content Page (Home.aspx)**

<%@ Page Title="Home" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" %>

<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">

<h2>Welcome to the Home Page</h2>

<p>This is specific content for the Home Page.</p>

</asp:Content>

**How it Works**

1. **Master Page** defines the layout.
2. **Content Page** provides content for the placeholders.
3. At runtime, ASP.NET merges them into one page.

**Advantages of Master Page**

* Maintains **consistent design** across pages.
* Reduces **duplication of HTML and code**.
* Makes **large applications easier to maintain**.
* Supports **nested master pages** (master page inside another).

**Conclusion**

Master Pages in ASP.NET are like **templates** that define a standard design for web applications. Content pages plug their unique information into placeholders while reusing the same layout. This leads to **consistency, reusability, and maintainability** in web development.

***17. Define user control? How can you create and use a user control in an ASP.NET?***

**Definition**

A **User Control** in ASP.NET is a reusable component that contains both **UI (HTML/ASP.NET controls)** and **code (C# or VB.NET)**.

* It has the extension **.ascx**.
* Similar to a web page but **cannot run on its own** – it must be hosted inside another ASP.NET page.
* Useful when a piece of functionality (like header, footer, login form, navigation menu) is needed on multiple pages.

**Advantages of User Controls**

1. **Reusability** – Can be used across multiple pages in the project.
2. **Modularity** – Divides a web page into smaller, manageable parts.
3. **Maintainability** – Changes in the control automatically reflect in all pages where it is used.
4. **Encapsulation** – Combines UI and related logic in one component.
5. **Rapid Development** – Saves development time for repetitive UI.

**Steps to Create and Use a User Control**

**1. Create a User Control (.ascx file)**

* Right-click project → Add New Item → Web User Control.
* Design UI with HTML/ASP.NET controls.
* Add logic in code-behind file.

**Example: MyControl.ascx**

<%@ Control Language="C#" AutoEventWireup="true" %>

<asp:Label ID="lblMessage" runat="server" Text="Enter Name: "></asp:Label>

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

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

<asp:Label ID="lblOutput" runat="server"></asp:Label>

**Code-Behind (MyControl.ascx.cs):**

protected void btnSubmit\_Click(object sender, EventArgs e)

{

lblOutput.Text = "Hello, " + txtName.Text;

}

**2. Register the User Control on an ASPX Page**

* Use the @Register directive to register the control.

**Example (Default.aspx):**

<%@ Register TagPrefix="uc" TagName="MyControl" Src="~/MyControl.ascx" %>

**3. Use the Control in the Page**

* Insert the user control like a normal ASP.NET control.

**Example:**

<uc:MyControl ID="uc1" runat="server" />

**How it Works**

1. The **User Control (MyControl.ascx)** contains reusable UI + logic.
2. The **ASPX Page** registers and embeds this control.