CBCS SCHEME

USN | 1 | C | R | 2 | 1 | M | C | 0 | 4 | 5 |

20MCA42

Fourth Semester MCA Degree Examination, June/July 2023
Programming using C#

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. With neat diagram, explain the work flow that takes place between .NET Source code, .NET Compiler and the execution engine. **(10 Marks)**

b. Discuss between managed code and unmanaged code. **(06 Marks)**

c. Explain the role of Common Type System. **(04 Marks)**

OR

2 a. What are namespaces? List and explain the purpose of any three namespaces. **(10 Marks)**

b. Write short notes on:
(i) JIT compiler
(ii) Windows Communication Foundation **(10 Marks)**

Module-2

3 a. Explain how to create an array of objects of the class with the help of a C# program. **(10 Marks)**

b. Explain the concept of static methods and static data members with suitable examples. **(10 Marks)**

OR

4 a. Write a C# program to explain accessor and mutator properties used in encapsulation. **(10 Marks)**

b. Explain the following with example:
(i) Abstract class and abstract methods
(ii) Compile time and runtime polymorphism **(10 Marks)**

Module-3

5 a. What are delegates? Explain the concept of multicast delegate with an example. **(10 Marks)**

b. Write a C# program to calculate square of numbers using delegates. **(10 Marks)**

OR

6 a. Write a C# program using try, catch, finally to explain any predefined exceptions. **(10 Marks)**

b. Explain the properties and methods of Data Reader and Data Adapter Class. **(10 Marks)**

Module-4

7 a. Explain various keyboard events in C# windows applications. **(10 Marks)**

b. Discuss the architecture of WPF with a neat diagram. **(10 Marks)**

OR

8 a. Explain the following:
(i) XAML definition and elements (10 Marks)
(ii) WPF Core Types (10 Marks)
b. What is GUI? List and explain basic controls of GUI.

Module-5

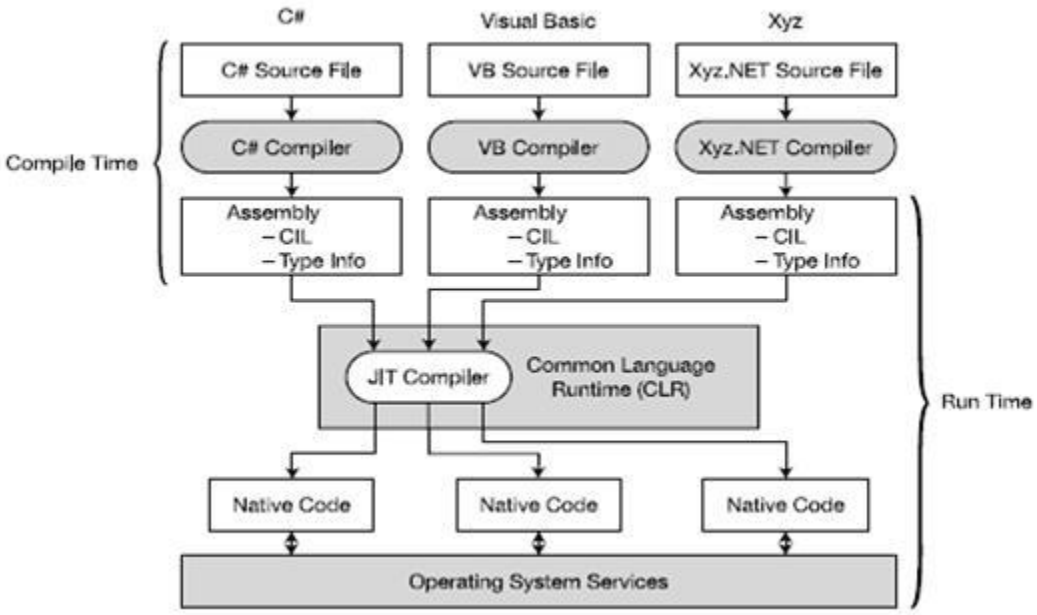
9 a. Explain the architecture of a three tier web based application with a neat diagram. (10 Marks)
b. Write steps in session tracking with http session state using cookies. (10 Marks)

OR

10 a. Explain the controls from AJAX control toolkit. (10 Marks)
b. Explain different validation controls with suitable example supported by ASP.NET. (10 Marks)

* * * * *

1a) With Neat diagram ,Explain the work flow takes place between .NET source coe,.NET compiler and execution engine



1. **Source Code**: The process begins with the .NET source code, which is written by developers. This source code contains instructions and logic written in a .NET-supported programming language like C#, VB.NET, or F#.
2. **Compilation**: The next step is compilation, where the .NET source code is transformed into an intermediate language (IL) code. This is done by the .NET compiler, such as the C# compiler (csc.exe) or the VB.NET compiler (vbc.exe). The IL code is also known as Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL). This step ensures that the code is platform-independent and can be executed on any system with the .NET runtime.
3. **Assembly**: The compiled IL code is stored in an assembly. An assembly is a unit of deployment in .NET and can be a dynamic link library (DLL) or an executable file (EXE). It contains metadata about types, methods, and resources used in the code.
4. **Just-In-Time Compilation (JIT)**: When you run a .NET application, the assembly is loaded into memory, and the execution engine (Common Language Runtime or CLR) comes into play. Before execution, the IL code is compiled into machine code specific to the host system. This compilation is known as Just-In-Time Compilation (JIT), and it's performed by the CLR. The result is native machine code that can be executed directly by the CPU.
5. **Execution**: With the compiled machine code, the .NET application can now be executed on the host system. The execution engine manages memory, handles exceptions, and ensures type safety during runtime. It also provides various services like garbage collection.

Q1b) Discuss between managed code and unmanaged code

1. Managed code and unmanaged code are two different programming paradigms, and they have significant differences in terms of memory management, performance, and security. Here's a discussion of the key distinctions between them:
2. **Managed Code:**
3. **Memory Management:** In managed code, memory management is automated. The runtime environment (e.g., .NET Common Language Runtime or Java Virtual Machine) takes care of memory allocation and deallocation. Developers don't need to manually allocate and release memory, which helps prevent common memory-related errors like buffer overflows and memory leaks.
4. **Safety:** Managed code is designed to be safe and secure. The runtime environment enforces type safety, access controls, and exception handling. This reduces the likelihood of crashes and security vulnerabilities caused by unsafe code practices.
5. **Portability:** Managed code is often platform-independent. It can run on different operating systems and architectures as long as there's a compatible runtime environment. This promotes cross-platform compatibility and reduces the need to write platform-specific code.
6. **Performance:** Managed code tends to have a performance overhead compared to unmanaged code. The runtime environment adds a layer of abstraction and introduces tasks like Just-In-Time (JIT) compilation, which can impact execution speed.
7. **Garbage Collection:** Managed code uses garbage collection to automatically reclaim memory occupied by objects that are no longer in use. This simplifies memory management for developers but can introduce occasional pauses in application execution when garbage collection occurs.
8. **Unmanaged Code:**
9. **Memory Management:** In unmanaged code, memory management is manual. Developers are responsible for allocating and releasing memory explicitly. While this provides more control, it also increases the risk of memory-related bugs and security vulnerabilities.
10. **Safety:** Unmanaged code can be less safe since there are no built-in checks for array bounds, null references, or type safety. This can lead to buffer overflows, crashes, and security vulnerabilities if not handled carefully.
11. **Portability:** Unmanaged code is often platform-specific. Code written for one operating system or architecture may not work on another without significant modification.
12. **Performance:** Unmanaged code can offer better performance compared to managed code because there is no runtime layer introducing overhead. It allows for fine-grained control over memory and system resources.
13. **Garbage Collection:** Unmanaged code does not use garbage collection. Developers must manage memory explicitly, which can be error-prone but also gives them control over memory usage.

Q1c) Explain the role of Common Type System

1. **Type Safety**: CTS enforces strong type safety within the .NET Framework. It defines a set of data types and rules for how those types can be used. This ensures that operations on data are performed only with compatible types, reducing the risk of runtime errors such as type mismatches or memory corruption.
2. **Interoperability**: One of the primary purposes of CTS is to facilitate interoperability between programming languages. In a typical .NET application, you might have components written in

different languages (e.g., C#, VB.NET, F#). CTS defines a common set of data types that are understood and usable by all of these languages. This allows objects created in one language to be seamlessly used and manipulated by code written in another language.

3. **Assembly Loading**: CTS helps define how assemblies (the building blocks of .NET applications) are loaded and used. Assemblies can contain code written in different languages, but because CTS defines a common type system, these assemblies can be used together without issues. This enables the development of large, modular applications.
4. **Garbage Collection**: The CTS includes rules for how memory is managed, particularly concerning garbage collection. Managed objects, which are instances of types defined in CTS, are automatically tracked by the garbage collector for memory management. This ensures that memory is efficiently reclaimed when objects are no longer in use, preventing memory leaks.
5. **Data Type Definitions**: CTS defines a wide range of data types, including integers, floating-point numbers, characters, strings, and more. These data types are common across all .NET languages, ensuring consistency in data representation.
6. **Value Types and Reference Types**: CTS distinguishes between value types (e.g., integers, structs) and reference types (e.g., classes, interfaces). This distinction is essential for understanding how objects are stored in memory and how they behave when passed as method arguments or returned from methods.
7. **Metadata**: CTS also defines how type metadata is represented in assemblies. This metadata includes information about types, methods, properties, and other members of classes. Metadata is crucial for reflection, which allows code to examine and interact with types at runtime.
8. **Compatibility**: CTS ensures that code written in one .NET language can be used alongside code written in another .NET language without compatibility issues. This is vital for multi-language development and reusability of code libraries.

Q2a) What are namespace? List and explain the purpose of any three namespaces

- In .NET, namespaces are a way to organize and logically group related types (classes, structs, enums, interfaces, etc.) within a code library or assembly. They help prevent naming conflicts and make it easier to manage and locate types within a project. Namespaces are hierarchical, allowing you to create a structured organization of types. Here are three common namespaces in .NET along with their purposes:

System Namespace:

- **Purpose**: The **System** namespace is one of the most fundamental namespaces in .NET. It contains core types and classes that are essential for working with the runtime environment, fundamental data types, and basic system functionality. Some of its key components include:
  - **Object**: The base class for all types in C# and .NET.
  - **String**: Represents text as a sequence of characters.
  - **Console**: Provides input and output functionality for console-based applications.
  - **DateTime**: Represents date and time values.
  - **Math**: Contains mathematical functions and constants.
  - **Environment**: Provides information about the current environment and system.
  - **Exception**: The base class for all exceptions in .NET.

System.Collections Namespace:

- **Purpose**: The **System.Collections** namespace contains classes and interfaces for working with collections of objects. Collections are data structures that allow you to store and manipulate multiple items. It includes various types of collections like lists, dictionaries, queues, and stacks. Some key components include:
    - **ArrayList**: A dynamically resizable array.
    - **List<T>**: A strongly typed generic list.
    - **Dictionary<TKey, TValue>**: A collection of key-value pairs.
    - **Queue<T>**: Represents a first-in, first-out (FIFO) collection.
    - **Stack<T>**: Represents a last-in, first-out (LIFO) collection.

System.IO Namespace:

- **Purpose**: The **System.IO** namespace provides classes and methods for performing input and output operations, including file and directory manipulation. It allows you to work with files, streams, directories, and paths. Some key components include:
    - **File**: Provides static methods for working with files.
    - **Directory**: Provides static methods for working with directories.
    - **FileStream**: Represents a stream of bytes to read from or write to a file.
    - **Path**: Provides methods and properties for working with file and directory paths.

Q2b) Write short notes on:

- JIT Compiler
- Windows Communication Foundation

JIT Compiler

A Just-In-Time (JIT) compiler is a crucial component of many modern programming languages and runtime environments, including the .NET Framework, Java Virtual Machine (JVM), and others. It plays a significant role in optimizing the execution of code written in these languages. Here are some key points about JIT compilers:

- **Dynamic Compilation**: A JIT compiler is responsible for converting intermediate code, such as Common Intermediate Language (CIL) in .NET or bytecode in Java, into native machine code at runtime. Unlike traditional compilers that produce machine code ahead of time, JIT compilation occurs dynamically, just before a method or function is executed.
- **Performance Optimization**: JIT compilation aims to improve the performance of an application. By translating code into native machine instructions tailored for the specific CPU and platform, it can lead to faster execution compared to interpreting the intermediate code directly. This optimization is especially beneficial for long-running or frequently used applications.
- **Cross-Platform Execution**: JIT compilation allows for cross-platform compatibility. Since the same intermediate code can be compiled into machine code for different architectures, it enables developers to write code that runs on multiple platforms without modification, as long as there is a compatible JIT compiler for each platform.

- **Execution Profiling**: JIT compilers often incorporate profiling information gathered during program execution. This information helps the compiler make optimization decisions, such as inlining methods, eliminating dead code, and optimizing loops. As a result, the compiled code is tailored to the actual execution patterns of the program.
- **Trade-Offs**: JIT compilation introduces a trade-off between startup time and execution speed. While it may cause a slight delay at the beginning of program execution as code is compiled, the optimized native code typically leads to faster execution over the long run.
- **Memory Usage**: JIT compilers generate machine code at runtime, which means that memory must be allocated to store the compiled code. This can increase the memory footprint of the application compared to interpreted or precompiled code.
- **Managed Environments**: JIT compilers are common in managed runtime environments like .NET and Java, where they work in conjunction with garbage collectors and other runtime services to provide a safe and efficient execution environment.

Windows Communication Foundation (WCF) is a framework provided by Microsoft for building service-oriented applications. It is a part of the .NET Framework and is designed to facilitate the creation, deployment, and management of distributed and interoperable applications, especially web services. Here are some key aspects and features of Windows Communication Foundation:

1. **Service-Oriented Architecture (SOA):** WCF promotes the development of applications based on the principles of Service-Oriented Architecture (SOA). It allows you to create services that can be accessed by clients over various communication protocols, making it suitable for building distributed and loosely coupled systems.
2. **Interoperability:** One of WCF's strengths is its support for interoperability with different platforms and technologies. It can communicate with services and clients implemented in other languages and using various protocols, such as HTTP, TCP, and more.
3. **Unified Programming Model:** WCF provides a unified programming model for building services, regardless of the communication protocol or transport layer used. This means you can use a consistent set of APIs and tools to develop and manage services.
4. **Bindings:** WCF allows you to define communication bindings, which specify the communication protocol, message encoding, and other communication settings. You can create custom bindings to meet specific requirements.
5. **Endpoints:** WCF services are exposed through endpoints, which define the address, binding, and contract. Each endpoint represents a specific way to access the service. You can configure multiple endpoints for a single service to support different communication protocols or transport mechanisms.
6. **Contracts:** Contracts define the operations that a service provides, including the input and output message types. WCF supports several types of contracts, such as service contracts, data contracts, and message contracts.
7. **Security:** WCF offers robust security features, including message encryption, authentication, authorization, and support for various security standards and protocols. This ensures the confidentiality and integrity of data during communication.

8. **Extensibility:** WCF is highly extensible, allowing you to customize various aspects of its behavior through extensions. You can create custom behaviors, message inspectors, and other extensions to tailor WCF to your specific needs.
9. **Hosting Options:** WCF services can be hosted in various environments, including Windows services, IIS (Internet Information Services), self-hosting in a custom application, and more. This flexibility allows you to choose the hosting model that best suits your requirements.
10. **Integration with Other Microsoft Technologies:** WCF seamlessly integrates with other Microsoft technologies, such as Windows Workflow Foundation (WF), Windows Identity Foundation (WIF), and Entity Framework, making it suitable for building comprehensive enterprise applications.

Q3a) Explain how to create an array of objects of the class with the help of a c# program

```
using System;

class Program
{
    static void Main()
    {
        // Create an array of Person objects
        Person[] peopleArray = new Person[3];

        // Populate the array with Person objects
        peopleArray[0] = new Person { Name = "Alice", Age = 30 };
        peopleArray[1] = new Person { Name = "Bob", Age = 25 };
        peopleArray[2] = new Person { Name = "Charlie", Age = 35 };

        // Access and display the elements of the array
        Console.WriteLine("Name: " + peopleArray[0].Name + ", Age: " + peopleArray[0].Age);
        Console.WriteLine("Name: " + peopleArray[1].Name + ", Age: " + peopleArray[1].Age);
        Console.WriteLine("Name: " + peopleArray[2].Name + ", Age: " + peopleArray[2].Age);
    }
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Q3b) Explain the concept of static methods and static data members with suitable examples

In object-oriented programming, a static method is a method that belongs to a class rather than an instance of that class. It is a method that can be called on the class itself, rather than on an object created from the class. Static methods are associated with the class as a whole and do not have access to instance-specific data. Here's an explanation with an example:

```
using System;

class MyClass
{
    // Static method
    public static void StaticMethod()
    {
        Console.WriteLine("This is a static method.");
    }
}

class Program
{
    static void Main()
    {
        // Call the static method using the class name
        MyClass.StaticMethod();
    }
}
```

Static Data Members:

Static data members, also known as class-level variables or static fields, are variables that belong to a class rather than an instance of the class. They are shared among all instances of the class and are not tied to any particular object. Static data members are declared using the static keyword. Here's an example:

```
using System;

class MyClass
{
    // Static data member
    public static int StaticField = 0;
}

class Program
{
    static void Main()
    {
```

```
        // Access the static data member using the class name
        MyClass.StaticField = 42;

        // Create instances of MyClass
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        // Access the static data member through instances
        // But it's the same value for all instances
        Console.WriteLine(obj1.StaticField); // Outputs: 42
        Console.WriteLine(obj2.StaticField); // Outputs: 42
    }
}
```

Q4a) Write a C# program to explain accessory and mutator properties used in encapsulation

```
using System;
class Person
{
    // Private fields
    private string name;
    private int age;
    // Accessor (getter) property for the 'name' field
    public string Name
    {
        get { return name; }
    }
    // Mutator (setter) property for the 'name' field
    public void SetName(string newName)
    {
        // You can add validation or logic here
        if (!string.IsNullOrWhiteSpace(newName))
        {
            name = newName;
        }
        else
        {
            Console.WriteLine("Invalid name.");
        }
    }
    // Accessor (getter) property for the 'age' field
```

```
    public int Age
    {
        get { return age; }
    }
    // Mutator (setter) property for the 'age' field
    public void SetAge(int newAge)
    {
        // You can add validation or logic here
        if (newAge >= 0 && newAge <= 120)
        {
            age = newAge;
        }
        else
        {
            Console.WriteLine("Invalid age.");
        }
    }
}
class Program
{
    static void Main()
    {
        // Create a Person object
        Person person = new Person();
        // Use accessor property to get the name
        Console.WriteLine("Name: " + person.Name); // Outputs: Name:
        // Use mutator property to set the name
        person.SetName("Alice");
        Console.WriteLine("Name: " + person.Name); // Outputs: Name: Alice
        // Use mutator property to set an invalid name
        person.SetName("");
        Console.WriteLine("Name: " + person.Name); // Outputs: Name: Alice (unchanged)
        // Use accessor property to get the age
        Console.WriteLine("Age: " + person.Age); // Outputs: Age: 0
        // Use mutator property to set the age
        person.SetAge(30);
        Console.WriteLine("Age: " + person.Age); // Outputs: Age: 30
        // Use mutator property to set an invalid age
        person.SetAge(150);
        Console.WriteLine("Age: " + person.Age); // Outputs: Age: 30 (unchanged)
```

```
    }
}
```

4b) Explain the following with example

- Abstract class and abstract methods
- Compile time and run time polymorphism

In C#, an abstract class is a class that cannot be instantiated directly and is typically used as a base class for other classes. Abstract classes can contain abstract methods, which are methods that do not have an implementation in the abstract class itself. Instead, the responsibility of providing an implementation for abstract methods lies with the derived classes. Here's an explanation with an example:

```
using System;
// Abstract class
abstract class Shape
{
    // Abstract method (no implementation)
    public abstract double CalculateArea();

    // Regular (concrete) method
    public void DisplayShape()
    {
        Console.WriteLine("This is a shape.");
    }
}
// Derived class
class Circle : Shape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    // Implementing the abstract method
    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(radius, 2);
    }
}
```

```
class Program
{
    static void Main()
    {
        // Cannot create an instance of an abstract class
        // Shape shape = new Shape(); // Error!

        // Create an instance of a derived class
        Circle circle = new Circle(5.0);

        // Call the abstract method
        double area = circle.CalculateArea();
        Console.WriteLine("Area of the circle: " + area);

        // Call a regular method from the base class
        circle.DisplayShape();
    }
}
```

Compile Time Polymorphism (Static Polymorphism):

Compile time polymorphism occurs when the decision about which method or function to call is made at compile time (i.e., during code compilation). This type of polymorphism is also known as "static polymorphism" or "early binding." The key mechanism for achieving compile-time polymorphism is method overloading and operator overloading.

Method Overloading: In method overloading, multiple methods in the same class have the same name but different parameters (either a different number of parameters or parameters of different types). The appropriate method to be called is determined by the number and types of arguments at compile time.

```
class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

Run Time Polymorphism (Dynamic Polymorphism):

Run time polymorphism occurs when the decision about which method or function to call is made at runtime. This type of polymorphism is also known as "dynamic polymorphism" or "late binding." The key mechanism for achieving runtime polymorphism is method overriding, typically with the use of inheritance and virtual/override keywords.

Method Overriding: In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass (base class). The method in the subclass should have the same method signature (name, return type, and parameters) as the one in the base class.

```
class Animal
{
   public virtual void MakeSound()
   {
      Console.WriteLine("Animal makes a sound.");
   }
}

class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Dog barks.");
   }
}
```

Q5a) What are delegates? Explain the concept of multicast delegate with an example

In C#, a **delegate** is a type that represents references to methods with a particular parameter list and return type. Delegates are used to define callback methods or to create references to functions, making it possible to treat functions as first-class objects. They are especially useful for implementing events and callbacks in event-driven programming.

One important feature of delegates in C# is the ability to create **multicast delegates**, which can reference and invoke multiple methods simultaneously. Multicast delegates combine the functionality of several delegate instances into a single delegate, allowing you to call multiple methods in a sequence. This is useful in scenarios where you want to notify multiple subscribers when an event occurs.

Here's an example that illustrates the concept of multicast delegates:

```
using System;

// Declare a delegate type with the same signature as the methods it can reference

delegate void MyDelegate(string message);

class Program{
```

```
    static void Main()
    {
        // Create delegate instances
        MyDelegate delegate1 = DisplayMessage;
        MyDelegate delegate2 = PrintMessage;
        // Combine the delegates into a multicast delegate
        MyDelegate multicastDelegate = delegate1 + delegate2;
        // Invoke the multicast delegate (calls both DisplayMessage and PrintMessage)
        multicastDelegate("Hello, Multicast Delegates!");
        Console.WriteLine("\nRemoving delegate2 from the multicast delegate.\n");
        // Remove delegate2 from the multicast delegate
        multicastDelegate -= delegate2;
        // Invoke the multicast delegate again (calls only DisplayMessage)
        multicastDelegate("Hello, Multicast Delegates!");
        Console.ReadKey();
    }
    static void DisplayMessage(string message)
    {
        Console.WriteLine("Display: " + message);
    }
    static void PrintMessage(string message)
    {
        Console.WriteLine("Print: " + message);
    }
}
```

Q5 b) Write a C# program to calculate square of numbers using delegates

```
using System;
// Define a delegate type for a method that takes an integer and returns an integer
delegate int SquareDelegate(int x);
class Program
{
    static void Main()
    {
        // Create a delegate instance and associate it with the Square method
        SquareDelegate square = Square;
        // Calculate and display the square of a number
        int number = 5;
        int result = square(number);
        Console.WriteLine($"The square of {number} is {result}");

        // You can use the delegate to calculate the square of other numbers
        number = 8;
        result = square(number);
        Console.WriteLine($"The square of {number} is {result}");

        Console.ReadKey();
    }
    // A method that calculates the square of a number
    static int Square(int x)
    {
```

```
        return x * x;
    }
}
```

Q6 a) Write a C# program using try, catch, finally to explain any predefined exceptions

```
using System;
class Program
{
    static void Main()
    {
        try
        {
            Console.Write("Enter a number: ");
            string userInput = Console.ReadLine();
            int number = int.Parse(userInput);
            // Attempt to divide by zero
            int result = 10 / number;
            Console.WriteLine($"Result: {result}");
        }
        catch (FormatException ex)
        {
            Console.WriteLine($"FormatException: {ex.Message}");
            Console.WriteLine("Please enter a valid number.");
        }
        catch (DivideByZeroException ex)
        {
```

```
            Console.WriteLine($"DivideByZeroException: {ex.Message}");
            Console.WriteLine("You cannot divide by zero.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception: {ex.Message}");
            Console.WriteLine("An error occurred.");
        }
        finally
        {
            Console.WriteLine("Finally block executed.");
        }
        Console.WriteLine("Program continues after the try-catch-finally block.");
        Console.ReadKey();
    }
}
```

Q6b) Explain the properties and methods of Data Reader and Data Adapter Class

In ADO.NET, the DataReader and DataAdapter classes are used to interact with databases. They are part of the .NET Framework's data access technology and are essential for reading and manipulating data from relational databases. Here's an overview of the properties and methods of these classes:

DataReader Class:

The DataReader class is used for forward-only, read-only access to data retrieved from a database. It is highly efficient and well-suited for scenarios where you need to quickly read and process large volumes of data without the need for updating the data source. Some of the key properties and methods of the DataReader class are:

Properties:

Item[Int32]: Allows you to access the column values by specifying the column index.

Item[String]: Allows you to access the column values by specifying the column name.

FieldCount: Returns the number of columns in the result set.

IsClosed: Indicates whether the DataReader is closed.

HasRows: Indicates whether the result set contains one or more rows.

Methods:

Read: Advances the DataReader to the next record, returning true if there are more rows and false if there are no more rows.

Close: Closes the DataReader and the associated database connection.

Dispose: Releases the resources used by the DataReader.

GetDataTypeName: Returns the data type name of a specified column.

GetName: Returns the name of a specified column.

GetOrdinal: Returns the index of a column given its name.

DataAdapter Class:

The DataAdapter class is used for retrieving and updating data in a database. It acts as a bridge between a dataset and a database, allowing you to fill a dataset with data from a data source and update the data source with changes made to the dataset. Some of the key properties and methods of the DataAdapter class are:

Properties:

SelectCommand: Gets or sets the SQL command used to retrieve data from the data source.

InsertCommand: Gets or sets the SQL command used to insert data into the data source.

UpdateCommand: Gets or sets the SQL command used to update data in the data source.

DeleteCommand: Gets or sets the SQL command used to delete data from the data source.

Methods:

Fill: Populates a dataset with data from the data source using the SelectCommand.

Update: Updates the data source with changes made in the dataset using the InsertCommand, UpdateCommand, and DeleteCommand.

FillSchema: Populates a dataset with schema information (table structure) without data.

Q7a) Explain various keyboard events in C# windows applications

KeyDown Event:

Occurs when a key is pressed while the control has focus.

Provides information about the key that was pressed, including the key code.

Commonly used for implementing actions that should occur as soon as a key is pressed.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        // Handle Enter key press
    }
}
```

KeyUp Event:

Occurs when a key is released after being pressed.

Useful for implementing actions that should occur when a key is released.

```
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape)
    {
        // Handle Escape key release
    }
}
```

KeyPress Event:

Occurs when a character key is pressed.

Provides the character that was typed as a KeyPressEventArgs parameter.

Useful for handling text input.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!char.IsDigit(e.KeyChar))
    {
        e.Handled = true; // Prevent non-digit characters
    }
}
```

PreviewKeyDown Event:

Occurs before the KeyDown event and provides more information about the key press.

Allows you to handle certain key combinations that may not be detected by KeyDown alone.

```
private void textBox1_PreviewKeyDown(object sender, PreviewKeyDownEventArgs e)
{
    if (e.KeyCode == Keys.Tab && e.Control)
    {
        // Handle Ctrl+Tab key combination
    }
}
```

LostFocus and GotFocus Events:

These events occur when a control loses or gains focus, respectively.

Useful for tracking when the user switches between controls using the keyboard.

```
private void textBox1_LostFocus(object sender, EventArgs e)
{
    // Control lost focus
```

```
}

private void textBox1_GotFocus(object sender, EventArgs e)
{
    // Control gained focus
}
```

KeyPreview Property:

This is not an event but a property of a form that, when set to true, allows the form to receive keyboard events before they are sent to individual controls.

Useful for handling global keyboard shortcuts.

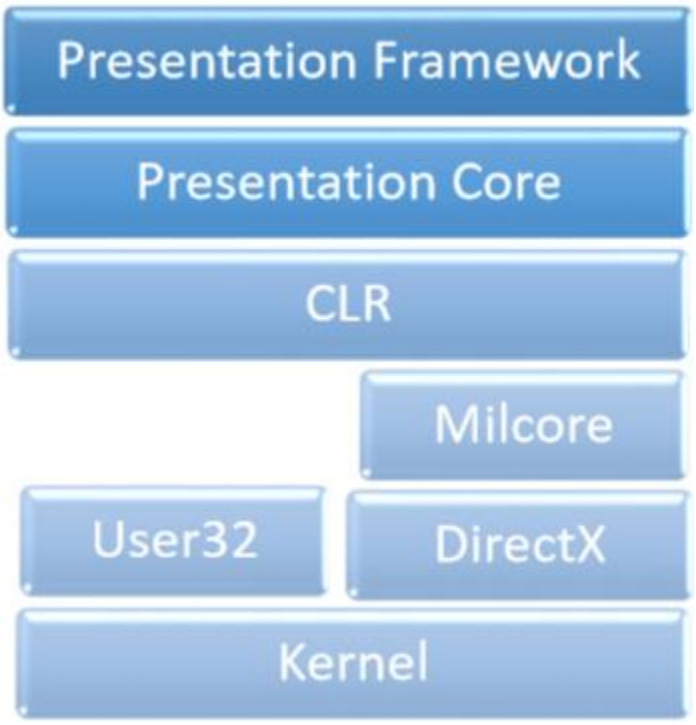
```
public Form1()
{
    InitializeComponent();
    this.KeyPreview = true;
}

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Control && e.KeyCode == Keys.S)
    {
        // Handle Ctrl+S global shortcut
    }
}
```

Q7b) Discuss the architecture of WPF with a neat diagram

Windows Presentation Foundation (WPF) is a framework for building Windows desktop applications with rich user interfaces. It provides a powerful architecture for creating interactive

and visually appealing applications. While it's challenging to provide a "neat diagram" in text format, I can describe the key architectural components and how they interact in a typical WPF application:



1. Presentation Layer:

The top layer of a WPF application is the presentation layer.

It consists of XAML (eXtensible Application Markup Language) files that define the user interface (UI) elements, such as windows, controls, and their layout.

XAML files are declarative and define the visual structure and behavior of the UI.

2. Application Logic:

The application logic layer contains the code-behind files for XAML files.

These files are written in C# or VB.NET and handle the application's logic, event handling, data binding, and interaction with the UI.

Developers write code in these files to respond to user actions and update the UI accordingly.

3. WPF Libraries:

WPF applications rely on a set of libraries provided by the .NET Framework for rendering and managing UI components.

These libraries handle rendering, layout, data binding, animation, and other UI-related tasks.

Key libraries include PresentationCore.dll, PresentationFramework.dll, and WindowsBase.dll.

4. Common Language Runtime (CLR):

The CLR is the execution environment for .NET applications, including WPF applications.

It manages memory, provides security, and executes managed code.

WPF applications are managed code and run within the CLR.

5. DirectX and MIL (Media Integration Layer):

DirectX is a set of APIs for multimedia and game development.

MIL (Media Integration Layer) is a part of WPF that uses DirectX for rendering.

It enables hardware acceleration and provides advanced graphics capabilities for WPF applications.

MIL handles rendering, animation, and visual effects.

6. Windows Operating System:

WPF applications run on the Windows operating system.

They leverage the Windows graphical subsystem for rendering and hardware interaction.

WPF applications have a native look and feel on Windows.

7. Data Access and Business Logic:

WPF applications often interact with data sources and business logic components.

Data access may involve databases, web services, or other data providers.

Business logic manages application behavior, data processing, and business rules.

8. MVVM (Model-View-ViewModel):

While not depicted in the diagram, the MVVM design pattern is commonly used in WPF applications.

MVVM separates the UI (View) from the application logic (ViewModel) and the data (Model).

ViewModel classes bind data to the View and handle user interactions.

Q8a) Explain the following:

- XAML definition and elements
- WPF core types

XAML (eXtensible Application Markup Language) is a markup language used in the development of Windows Presentation Foundation (WPF), Universal Windows Platform (UWP), and other XAML-based applications. XAML is used to define and create the user interface (UI) and the structure of an application's visual elements. It separates the UI design from the application logic, allowing designers and developers to collaborate more efficiently. Here's an explanation of XAML and its key elements:

1. XAML Definition:

XAML is a declarative markup language that defines the structure, appearance, and behavior of UI elements in a XAML-based application.

It is similar in concept to HTML in web development but is designed specifically for creating rich Windows desktop and UWP applications.

XAML files typically have the .xaml extension and are used alongside code-behind files written in C# or VB.NET.

2. XAML Elements:

XAML defines UI elements and their properties using a hierarchical structure. Here are some key XAML elements and concepts:

Elements: In XAML, UI elements are represented as XML elements.

Attributes: Elements have attributes that define their properties. In the example above, the Content attribute sets the text displayed on the button.

Nesting: Elements can be nested within each other to create complex UI hierarchies.

Namespaces: XAML files include XML namespaces that map to .NET namespaces and assemblies. These namespaces provide access to predefined controls and types. The most common namespaces in WPF XAML include xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" and xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml".

Content Property: Some controls in XAML have a designated property called the "content property." This property allows you to specify the content of the control without explicitly naming it. For example, the Label control's content property is Content,

Windows Presentation Foundation (WPF) is a framework for building Windows desktop applications with rich user interfaces. WPF provides a wide range of core types (classes and structures) that are fundamental to building graphical user interfaces. These core types are part of the System.Windows namespace and are essential building blocks for creating WPF applications. Here are some of the key WPF core types:

UIElement:
- The base class for all WPF user interface elements.
- Provides core functionality for layout, input, and event handling.
- Derived classes include controls like Button, TextBox, and Label.

FrameworkElement:
- Extends UIElement and adds features for layout, data binding, and styles.
- Serves as the base class for most visual elements in WPF.
- Includes properties like Width, Height, Margin, and Style.

Visual:
- Represents an abstract base class for all visual objects.
- Used for rendering graphics and other visual content.
- Derived classes include DrawingVisual and UIElement.

DependencyObject:
- The base class for all classes that support dependency properties.
- Allows properties to be defined and managed as dependencies.
- Essential for data binding and property value inheritance.

DependencyProperty:
- Represents a property that is backed by the WPF dependency property system.
- Allows properties to support features like data binding, animation, and property value inheritance.

Control:
- A base class for controls that provide a consistent look and feel.
- Includes properties like Content, Background, and Foreground.
- Derived classes include Button, TextBox, ListBox, and many others.

ContentControl:
- A control that can contain a single piece of content.
- Used for displaying content like text, images, or other controls.
- Includes properties like Content and ContentTemplate.

Panel:
- The base class for layout containers in WPF.
- Provides layout logic for arranging child elements.
- Derived classes include Grid, StackPanel, WrapPanel, and Canvas.

Window:
- Represents a top-level window in a WPF application.
- Includes properties and methods for window management.
- Used as the main container for UI elements.

Application:
- Represents the entry point and lifetime management of a WPF application.
- Provides events like Startup, Exit, and DispatcherUnhandledException.
- Used to initialize and start the application.

ResourceDictionary:

- Represents a collection of resources, such as styles, templates, and brushes.
- Allows for centralizing and reusing resources in XAML.
- Used to define resources at the application or window level.

Brush:

- The base class for defining color and pattern brushes.
- Used for filling shapes, backgrounds, and other visual elements.
- Derived classes include SolidColorBrush, LinearGradientBrush, and ImageBrush.

Style:

- Defines a set of property values that can be applied to controls and elements.
- Allows for consistent styling of UI elements.
- Includes setters for various properties.

Q8b) what is GUI? List and explain basic controls of GUI

GUI (Graphical User Interface) is a type of user interface that allows users to interact with electronic devices or software through graphical elements, such as icons, buttons, windows, and menus, rather than using only text-based commands. GUIs are designed to make software more user-friendly and visually intuitive. Here are some basic GUI controls commonly found in graphical user interfaces:

Button:

A button is a control that the user can click or press to trigger an action.
It often displays text or an icon to indicate its purpose.
Buttons are used for actions like submitting forms, confirming choices, or triggering specific functions.

Label:

A label is a static text element used to provide information or context to the user.
Labels do not allow user input and are typically used to display text or descriptions.

Text Box:

A text box is an input control that allows the user to enter text or numeric data.
It can be single-line or multi-line, depending on the application's requirements.
Text boxes are commonly used for data entry, search, and text-based input.

Check Box:

A check box is a control that represents a binary choice, such as enabling or disabling an option.
It has two states: checked (selected) and unchecked (deselected).
Check boxes are used in preference settings, form submissions, and more.

Radio Button:

A radio button is a control that represents a list of mutually exclusive options.
Only one radio button in a group can be selected at a time.
Radio buttons are used when the user needs to choose a single option from a list.

Combo Box (or Drop-Down List):

A combo box is a control that combines a text box with a drop-down list of selectable items.
Users can either type text or select an item from the list.
Combo boxes are used for selecting items from a predefined list.
List Box:

A list box is a control that displays a list of items from which users can select one or more.
Users can typically select multiple items by holding down the Ctrl key (for non-contiguous selection) or Shift key (for contiguous selection).

Slider (or Trackbar):

A slider is a control that allows users to set a value within a predefined range by dragging a slider thumb.
It's used for selecting values like volume levels, brightness, or any numeric range.

Progress Bar:

A progress bar is a visual indicator that displays the progress of a task or operation.
It's used to show the status of a process and often fills gradually as the task completes.

Menu:

A menu is a list of commands or options that are organized in a hierarchical structure.
Menus are used to access various functions and features of an application.

Toolbar:

A toolbar is a row or column of buttons that provide quick access to frequently used commands or functions.
Toolbars are often found at the top of windows or below the menu bar.

Dialog Box:

A dialog box is a separate window that appears to request user input or provide information.
It is often used for settings, file operations, and user interactions that require specific input.

Q9a) Explain the architecture of a three tier web based application with a neat diagram

A three-tier web-based application architecture is a popular way to design and structure web applications for scalability, maintainability, and separation of concerns. It divides the application into three main tiers or layers, each responsible for specific functionalities. Here's an explanation of the architecture along with a simplified diagram:

1. Presentation Tier (User Interface):

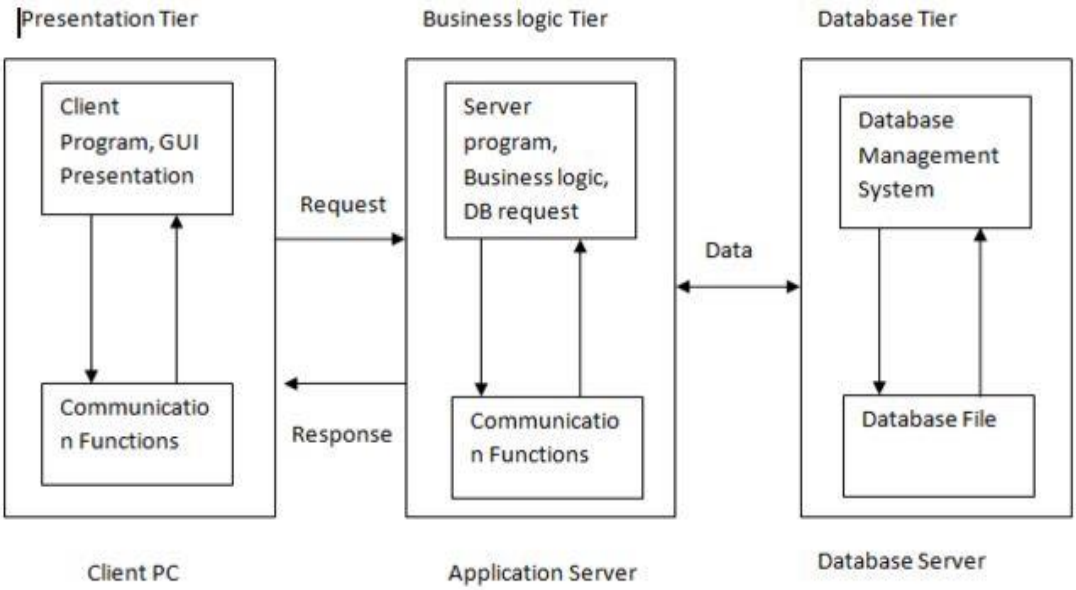
The presentation tier is the topmost layer and is responsible for interacting with users.
It includes the user interface (UI) components, such as web pages, forms, and client-side scripts.
The primary goal is to present information to users and capture user input.
In web applications, this tier typically consists of web browsers or mobile app UI.

2. Application Tier (Business Logic):

The application tier, also known as the business logic layer, contains the core logic of the application.
It processes user requests, enforces business rules, and coordinates interactions between the presentation and data tiers.
This tier often includes server-side scripts, application servers, and middleware.
It's responsible for making decisions and orchestrating the flow of data and operations.

3. Data Tier (Data Storage and Management):

The data tier is the bottommost layer and handles data storage, retrieval, and management.
It includes databases, data storage systems, and data access components.
The primary role is to store and manage the application's data, including user information, content, and configuration settings.
It communicates with the application tier to provide data for processing.



Q9b) Write steps in session tracking with http session state using cookies

Session tracking is a crucial aspect of web development, allowing you to maintain user-specific data across multiple web requests. One common method for session tracking is using HTTP Session State with cookies in web applications. Here are the steps to implement session tracking using this approach:

Configure Session State in Web.config:

Open your web application's Web.config file.

Add or modify the <sessionState> element to configure session state mode to use cookies.

The mode attribute should be set to "InProc" to use in-process session state management.

cookieless="false" ensures that cookies are used for session tracking.

timeout specifies the session timeout duration in minutes.

Create or Retrieve Session Data:

In your web application code, you can create or retrieve session data. Session data is stored in the HttpSessionState object.

You can store any serializable object in the session state.

Use Cookies for Session Tracking:

Behind the scenes, ASP.NET will use cookies to store a unique session identifier on the client's browser.

This identifier is used to associate the client with their specific session data on the server.

Ensure Cookies Are Enabled:

Inform users that your website requires cookies for session tracking and functionality.

Include a message or instructions in your UI to enable cookies in their browser settings if needed.

Handle Session Timeout:

When a session times out (as specified in the timeout attribute in Web.config), the session data will be cleared.

You can handle this scenario by redirecting the user to a login page or displaying a message.

Secure Sensitive Session Data:
If your session data contains sensitive information, consider using HTTPS to encrypt data in transit between the client and server.
Additionally, ensure that the data stored in session variables is appropriately protected from unauthorized access.
Clean Up Expired Sessions:
ASP.NET automatically manages session cleanup, but you can implement custom logic to clean up session data if needed.
This can help avoid session data buildup in the server's memory.
Test Session Handling:
Thoroughly test your session handling to ensure that session data is stored and retrieved correctly.
Test scenarios such as session timeouts, multiple users, and concurrent access to session data.

Q10a) Explain the controls from AJAX control toolkit

The AJAX Control Toolkit is a set of open-source controls and components for building rich, interactive web applications using ASP.NET and AJAX (Asynchronous JavaScript and XML) technologies. These controls extend the capabilities of standard ASP.NET controls by adding client-side functionality and improved user experiences. Here are some of the commonly used controls from the AJAX Control Toolkit:

Accordion:
- The Accordion control allows you to create collapsible panels that conserve screen space.
- Users can expand or collapse panels to view or hide content.
- It's commonly used for FAQs, navigation menus, and organizing content.

AutoCompleteExtender:
- Provides auto-complete suggestions as users type into a text box.
- Ideal for search boxes and data entry forms where you want to assist users in finding relevant information quickly.

CalendarExtender:
- Enhances a standard text box with a pop-up calendar for date selection.
- Makes date selection easier and more user-friendly.

CascadingDropDown:
- Allows you to create dependent drop-down lists (or combo boxes) where the options in one list depend on the selection made in another.
- Useful for filtering data based on user choices.

CollapsiblePanel:
- Similar to the Accordion control but allows individual panels to expand or collapse independently.
- Useful for displaying collapsible sections of content on a web page.

ModalPopupExtender:
- Displays a modal (popup) dialog box that overlays the main content.
- Great for displaying messages, forms, or additional information without navigating away from the current page.

SlideShowExtender:

- Creates image slideshows with various transition effects.
- Useful for showcasing images in a visually appealing way.

TabContainer and TabPanel:

- The TabContainer control allows you to organize content into tabbed panels.
- Each TabPanel represents a single tab with its content.
- Tabbed navigation is a common way to organize and present information on a page.

AsyncFileUpload:

- Provides an asynchronous file upload control that doesn't require a postback.
- Allows users to upload files without leaving or refreshing the current page.

Rating:

- Allows users to rate items by clicking on stars or other rating symbols.
- Useful for user-generated content websites, product reviews, and rating-based feedback systems.

HTMLEditor:

- A rich text editor control that provides a familiar, word-processor-like interface for creating and editing HTML content.
- Useful for content management systems and other applications where rich text editing is required.

FilteredTextBox:

- Allows you to specify a regular expression pattern to validate and restrict the input in a text box.
- Useful for ensuring that user input matches specific criteria.

Q10b) Explain different validation controls with suitable example supported by ASP.NET

ASP.NET provides a set of validation controls that make it easy to perform client-side and server-side validation of user input in web forms. These controls help ensure that data entered by users is accurate and meets specified criteria. Here are some of the key validation controls in ASP.NET with suitable examples:

RequiredFieldValidator:

Ensures that a user provides input in a specified form field.
It validates that the associated input control is not empty.

```
<asp:TextBox ID="txtUsername" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfvUsername" runat="server" ControlToValidate="txtUsername"
    ErrorMessage="Username is required" ForeColor="Red"></asp:RequiredFieldValidator>
```

RegularExpressionValidator:

Validates that user input matches a specified regular expression pattern.
Useful for enforcing specific formats like email addresses or phone numbers.

```
<asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
<asp:RegularExpressionValidator ID="revEmail" runat="server" ControlToValidate="txtEmail"
    ErrorMessage="Invalid email format" ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"></asp:RegularExpressionValidator>
```

CompareValidator:

Compares the value in one input control to the value in another input control or a constant value.
Useful for password confirmation and comparing values.

<asp:TextBox ID="txtPassword" runat="server" TextMode="Password"></asp:TextBox>
<asp:TextBox ID="txtConfirmPassword" runat="server" TextMode="Password"></asp:TextBox>
<asp:CompareValidator ID="cvPassword" runat="server" ControlToValidate="txtConfirmPassword"
ControlToCompare="txtPassword" Operator="Equal" Type="String" ErrorMessage="Passwords do not match" ForeColor="Red"></asp:CompareValidator>
RangeValidator:
Validates that the input falls within a specified numeric or date range.
Useful for age restrictions, date range validation, or numeric value constraints.
<asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
<asp:RangeValidator ID="rvAge" runat="server" ControlToValidate="txtAge" Type="Integer" MinimumValue="18" MaximumValue="100"
ErrorMessage="Age must be between 18 and 100" ForeColor="Red"></asp:RangeValidator>
CustomValidator:
Allows you to define custom validation logic using server-side or client-side code.
Useful for complex validation scenarios.
<asp:TextBox ID="txtCustom" runat="server"></asp:TextBox>
<asp:CustomValidator ID="cvCustom" runat="server" ControlToValidate="txtCustom"
OnServerValidate="ValidateCustomInput" ErrorMessage="Invalid input" ForeColor="Red"></asp:CustomValidator>
ValidationSummary:
Displays a summary of all validation errors on the form.
Useful for presenting a list of validation errors to the user.
<asp:ValidationSummary ID="vsSummary" runat="server" HeaderText="Validation Errors:" ForeColor="Red" DisplayMode="BulletList" />
html