****

**Table of Contents**

[.NET Framework and Its Components 1](#_gjdgxs)

[Compilation and Execution Process 3](#_30j0zll)

[Arrays - Compute Sum and Average 3](#_1fob9te)

[Type Conversion 4](#_3znysh7)

[Classes - Defining a Constructor 4](#_2et92p0)

[MSIL (Microsoft Intermediate Language) and JIT (Just-In-Time) Compilation 5](#_tyjcwt)

[Events in C# 6](#_3dy6vkm)

[Why C# is Type-Safe and Modern 7](#_1t3h5sf)

[LINQ (Language Integrated Query) and SQL 7](#_4d34og8)

[Implicitly Typed Variables 8](#_2s8eyo1)

[Exception Handling Example 8](#_17dp8vu)

[Events in C# 9](#_3rdcrjn)

[Why C# is Type-Safe and Modern 10](#_26in1rg)

[LINQ (Language Integrated Query) vs. SQL 11](#_lnxbz9)

[Implicitly Typed Variables 12](#_35nkun2)

[Exception Handling Example 12](#_1ksv4uv)

[Instance Method vs. Class Method (Static Method) 13](#_2jxsxqh)

[MS SQL Database Example + DataGridView 14](#_z337ya)

[Connection-Oriented vs. Disconnected-Oriented Approach 15](#_3j2qqm3)

[Rectangular vs. Ragged Arrays 15](#_1y810tw)

[C# as a Type-Safe, Modern, and Interoperable Language 16](#_4i7ojhp)

[Database Programming in C# 17](#_2xcytpi)

[QUESTION 4 ASSIGNMENT 1 24](#_1ci93xb)

[ASSIGNMENT TWO 27](#_3whwml4)

[QUESTION 6 ASSIGNMENT TWO 28](#_qsh70q)

# .NET Framework and Its Components

The .NET Framework is a comprehensive software development platform from Microsoft. It provides a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but distributed over the internet, or executed remotely.

**Components of the .NET Framework:**

1. **Common Language Runtime (CLR)**
   * **Description**: The CLR is the core runtime engine for executing applications. It handles various services such as memory management, security enforcement, and exception handling.
   * **Benefits**:
     + **Simplifies Development**: By managing low-level programming tasks, developers can focus more on business logic rather than memory management.
     + **Improves Performance**: Optimized memory management and efficient garbage collection enhance application performance.
     + **Enhances Security**: Provides a robust security model, including code access security (CAS) and validation for type safety, preventing harmful operations.
2. **Base Class Library (BCL)**
   * **Description**: The BCL is a large library of classes, interfaces, and value types that provide access to system functionality, including file I/O, database interaction, and web application development.
   * **Benefits**:
     + **Code Reusability**: Reduces the amount of code developers need to write by providing reusable functions for common tasks.
     + **Consistency**: Offers a consistent programming model across various types of applications (web, desktop, mobile).
     + **Extensibility**: Developers can extend the existing BCL classes to add custom functionalities.
3. **Just-In-Time (JIT) Compilation**
   * **Description**: JIT compilation converts Intermediate Language (IL) code to native machine code at runtime.
   * **Benefits**:
     + **Performance Optimization**: JIT optimizes the code for the current machine, taking advantage of specific hardware features.
     + **Portability**: The same IL code can run on different hardware architectures, making applications more versatile.
4. **Common Language Specification (CLS)**
   * **Description**: CLS defines a set of rules and conventions that ensure interoperability between .NET languages.
   * **Benefits**:
     + **Language Interoperability**: Ensures that code written in one .NET language can be used by another, fostering a collaborative development environment.
     + **Standardization**: Provides a standardized set of features that all .NET languages can support, making cross-language integration seamless.

# Compilation and Execution Process

1. **Source Code Writing**
   * You write C# code using an Integrated Development Environment (IDE) like Visual Studio, which provides tools for coding, debugging, and compiling applications.
2. **Compilation**
   * The C# compiler (csc.exe) compiles the source code into an Intermediate Language (IL), stored in an assembly file (.exe or .dll).
3. **JIT Compilation**
   * At runtime, the CLR’s JIT compiler translates the IL code into native machine code specific to the processor architecture, optimizing the code during execution.
4. **Execution**
   * The CLR executes the native machine code, managing memory, security, and other aspects of execution, ensuring that the application runs smoothly and securely.

# Arrays - Compute Sum and Average

In C#, you can compute the sum and average of elements in an array using a loop:

| csharp **int**[] numbers = { 1, 2, 3, 4, 5 }; **int** **sum** = 0;  **for** (**int** i = 0; i < numbers.Length; i++) {  **sum** += numbers[i]; }  **double** average = (**double**)**sum** / numbers.Length;  Console.WriteLine($"Sum: {sum}"); Console.WriteLine($"Average: {average}"); |
| --- |

* **Explanation**:
  + **Sum Calculation**: Iterate through the array and accumulate the sum of its elements.
  + **Average Calculation**: Divide the sum by the number of elements to get the average. Casting the sum to double ensures a floating-point division, providing an accurate average.

# Type Conversion

**Type Conversion in C#** involves converting a variable from one data type to another. There are two main types of conversions:

1. **Widening (Implicit Conversion)**
   * **Definition**: Automatically converting a smaller data type to a larger data type without data loss.
   * **Example**:

| csharp  int num = 123; **double** doubleNum = num; // **Implicit** conversion |
| --- |

2. **Narrowing (Explicit Conversion - Casting)**

* **Definition**: Converting a larger data type to a smaller data type, which requires explicit casting and may result in data loss.
* **Example**:

| csharp  double doubleNum = 123.45; int num = (int)doubleNum; *// Explicit conversion* |
| --- |

# Classes - Defining a Constructor

A constructor in C# is a special method used to initialize objects. Here’s an example:

| csharp **public** **class** **Person** {  **public** **string** Name { **get**; **set**; }  **public** **int** Age { **get**; **set**; }   **public** **Person**(**string** name, **int** age) *// Constructor*  {  Name = name;  Age = age;  } }  *// Usage* Person john = **new** Person("John", 30); Console.WriteLine($"Name: {john.Name}, Age: {john.Age}"); |
| --- |

* **Explanation**:
  + **Constructor Definition**: The constructor has the same name as the class and is used to initialize the class's fields (Name and Age).
  + **Object Creation**: When a Person object is created, the constructor is called with the specified parameters ("John" and 30), setting the object's properties.

# MSIL (Microsoft Intermediate Language) and JIT (Just-In-Time) Compilation

1. **MSIL**
   * **Description**: MSIL is an intermediate language used by the .NET Framework. It is platform-independent, allowing code to be run on any system that supports the .NET Framework.
   * **Benefits**:
     + **Portability**: Ensures that applications can run on different hardware and operating systems that support .NET.
     + **Interoperability**: Facilitates interaction between different .NET languages.
2. **JIT Compilation**
   * **Description**: At runtime, the JIT compiler translates MSIL into native machine code for the specific architecture on which the application is running, optimizing the code for performance.
   * **Benefits**:
     + **Performance**: JIT compilation allows for real-time optimizations based on the current execution environment.
     + **Flexibility**: Applications can be optimized for the specific hardware they run on, enhancing performance.

# Events in C#

Events in C# are used to provide notifications. Common examples include button clicks and form closing events.

1. **Button Click Event**:

| csharp button.Click += **new** EventHandler(button\_Click);  **void** **button\_Click**(**object** sender, EventArgs e) {  MessageBox.Show("Button clicked!"); } |
| --- |

2. **Form Closing Event**:

| csharp form.FormClosing += **new** FormClosingEventHandler(form\_Closing);  **void** **form\_Closing**(**object** sender, FormClosingEventArgs e) {  **var** result = MessageBox.Show("Are you sure you want to close?", "Confirm", MessageBoxButtons.YesNo);  **if** (result == DialogResult.No)  {  e.Cancel = true; *// Cancel the close event*  } } |
| --- |

* **Explanation**:
  + **Button Click**: Attaches an event handler (button\_Click) that displays a message when the button is clicked.
  + **Form Closing**: Attaches an event handler (form\_Closing) that asks for confirmation before closing the form, allowing the operation to be canceled if necessary.

# Why C# is Type-Safe and Modern

* **Type-Safe**: C# enforces strict type rules, preventing operations on incompatible types. Variables must be declared with a specific data type, ensuring type safety.
* **Modern**: C# includes features like generics, lambda expressions, asynchronous programming (async/await), and Language Integrated Query (LINQ), making it a powerful and expressive language.

# LINQ (Language Integrated Query) and SQL

**LINQ**:

* Allows querying of collections, databases, XML, and more using a syntax similar to SQL.
* LINQ is a set of extensions to the C# language that allows developers to write SQL-like code in C#
* to query and manipulate data from various sources, such as databases, arrays, and collections.
* LINQ provides a concise and expressive way to filter, sort, and aggregate data.

**SQL**:

* A language for querying and manipulating relational databases.
* SQL is a standard language for managing relational databases. It is used to create, modify, and
* query databases. SQL commands include SELECT, INSERT, UPDATE, and DELETE, which
* allow developers to interact with data stored in a database.

**Example of LINQ**:

| csharp **var** numbers = **new** List<int> { 1, 2, 3, 4, 5 }; **var** evenNumbers = from num **in** numbers  where num % 2 == 0  select num;  foreach (**var** num **in** evenNumbers) {  Console.WriteLine(num); } |
| --- |

* **Explanation**:
  + **LINQ Query**: Selects even numbers from a list using LINQ's query syntax.
  + **Iteration**: Iterates over the result and prints each even number.

# Implicitly Typed Variables

* **Definition**: Implicitly typed variables use the var keyword, allowing the compiler to infer the variable's type.
* **Example**:

| csharp **var** name = "John"; // **Compiler** infers the **type** **as** string **Console**.**WriteLine**(name); |
| --- |

# Exception Handling Example

Exception handling in C# uses try, catch, and finally blocks to manage runtime errors.

**Example**:

| csharp try {  **int** result = 10 / 0; *// This line will throw a DivideByZeroException* } catch (DivideByZeroException ex) {  Console.WriteLine("Error: Division by zero occurred.");  Console.WriteLine(ex.Message); } finally {  Console.WriteLine("This block is always executed."); } |
| --- |

* **Explanation**:
  + **try**: Contains code that may throw an exception.
  + **catch**: Catches and handles specific exceptions (DivideByZeroException in this case).
  + **finally**: Contains code that executes regardless of whether an exception was thrown.

# Events in C#

Events in C# are used to provide notifications when something happens. They are often used in graphical user interfaces (GUIs) to handle user actions such as clicks and key presses.

**Example 1: Button Click Event**

| csharp **using** System; **using** System.Windows.Forms;  **public** **class** **MyForm** : **Form** {  **private** Button myButton;   **public** **MyForm**()  {  myButton = **new** Button();  myButton.Text = "Click Me";  myButton.Click += **new** EventHandler(MyButton\_Click);  Controls.Add(myButton);  }   **private** **void** **MyButton\_Click**(**object** sender, EventArgs e)  {  MessageBox.Show("Button clicked!");  }    [**STAThread**]  **public** **static** **void** **Main**()  {  Application.Run(**new** MyForm());  } } |
| --- |

* **Explanation**:
  + **Button Initialization**: A Button control is created and added to the form.
  + **Event Subscription**: The Click event of the button is associated with the MyButton\_Click event handler.
  + **Event Handler**: When the button is clicked, the MyButton\_Click method is called, displaying a message box.

**Example 2: Form Closing Event**

| csharp **using** System; **using** System.Windows.Forms;  **public** **class** **MyForm** : **Form** {  **public** **MyForm**()  {  **this**.FormClosing += **new** FormClosingEventHandler(MyForm\_Closing);  }   **private** **void** **MyForm\_Closing**(**object** sender, FormClosingEventArgs e)  {  **var** result = MessageBox.Show("Are you sure you want to close?", "Confirm", MessageBoxButtons.YesNo);  **if** (result == DialogResult.No)  {  e.Cancel = true; *// Cancel the close event*  }  }   [**STAThread**]  **public** **static** **void** **Main**()  {  Application.Run(**new** MyForm());  } } |
| --- |

* **Explanation**:
  + **Event Subscription**: The FormClosing event of the form is associated with the MyForm\_Closing event handler.
  + **Event Handler**: Before the form closes, a confirmation message box is shown. If the user selects "No," the closing event is canceled.

# Why C# is Type-Safe and Modern

**Type-Safe**:

* C# enforces strict type checking at compile-time, preventing operations on incompatible types.
* Variables must be declared with a specific data type, ensuring that the data remains consistent and preventing unexpected behaviors.

**Modern**:

* C# includes advanced features such as:
  + **Generics**: Allow you to define classes, methods, and data structures with a placeholder for the type of data they store or use.
  + **Lambda Expressions**: Provide a concise way to write anonymous methods.
  + **Asynchronous Programming** (async and await): Simplifies writing asynchronous code.
  + **LINQ (Language Integrated Query)**: Enables querying of collections, databases, XML, and more using a SQL-like syntax.

# LINQ (Language Integrated Query) vs. SQL

**LINQ**:

* **Description**: LINQ is a feature in C# that provides a consistent way to query various data sources using a syntax similar to SQL.
* **Usage**: Can be used to query arrays, collections, databases, XML, and other data sources.
* **Example**:

| csharp **var** numbers = **new** List<int> { 1, 2, 3, 4, 5 }; **var** evenNumbers = from num **in** numbers  where num % 2 == 0  select num;  foreach (**var** num **in** evenNumbers) {  Console.WriteLine(num); } |
| --- |

**SQL**:

* **Description**: SQL (Structured Query Language) is used for managing and querying relational databases.
* **Usage**: Specifically designed for database operations like selecting, inserting, updating, and deleting data.
* **Example**:

| sql **SELECT** \* **FROM** Employees **WHERE** Department = 'Sales'; |
| --- |

# Implicitly Typed Variables

**Implicitly Typed Variables**:

* **Definition**: In C#, you can use the var keyword to declare variables without explicitly specifying the type. The compiler infers the type based on the assigned value.
* **Example**:

| csharp **var** name = "John"; *// Compiler infers the type as string* **var** age = 30; *// Compiler infers the type as int*  Console.WriteLine($"Name: {name}, Age: {age}"); |
| --- |

# Exception Handling Example

Exception handling in C# involves using try, catch, and finally blocks to manage runtime errors gracefully.

**Example**:

| csharp **try** {  **int**[] numbers = { 1, 2, 3 };  Console.**WriteLine**(numbers[5]); *// This line will throw an IndexOutOfRangeException* } **catch** (IndexOutOfRangeException ex) {  Console.**WriteLine**("Error: Index out of range.");  Console.**WriteLine**(ex.Message); } **finally** {  Console.**WriteLine**("This block is always executed."); } |
| --- |

* **Explanation**:
  + **try**: Contains code that may throw an exception.
  + **catch**: Catches and handles specific exceptions (IndexOutOfRangeException in this case).
  + **finally**: Contains code that executes regardless of whether an exception was thrown, typically used for cleanup activities.

This concludes the detailed overview of the .NET Framework components, compilation and execution process, arrays, type conversion, classes, MSIL, JIT, events, C#'s type-safety, LINQ, implicitly typed variables, and exception handling. If you have any further questions or need more examples, feel free to ask!

# 

# Instance Method vs. Class Method (Static Method)

**Instance Method**:

* **Definition**: Instance methods are defined within a class and operate on specific object instances of that class. They can access both instance-specific data (fields, properties) and class-level data (static members).
* **Invocation**: Called using an object instance.
* **Purpose**: Encapsulates behavior specific to an object, allowing manipulation or interaction with the object's data.
* **Example**:

| csharp **public** **class** **Person** {  **public** **string** Name { **get**; **set**; }   **public** **void** **Greet**()  {  Console.WriteLine($"Hello, my name is {Name}!");  } }  Person john = **new** Person { Name = "John" }; john.Greet(); *// Outputs: Hello, my name is John!* |
| --- |

**Class Method (Static Method)**:

* **Definition**: Class methods, also known as static methods, operate on the class itself rather than on instances. They can only access static members of the class.
* **Invocation**: Called using the class name.
* **Purpose**: Useful for utility functions that don't depend on instance-specific data.
* **Example**:

| csharp **public** **class** **MathUtils** {  **public** **static** **int** **Add**(**int** x, **int** y)  {  **return** x + y;  } }  **int** sum = MathUtils.Add(5, 3); *// Outputs: 8* |
| --- |

# MS SQL Database Example + DataGridView

**Connecting to Database**:

| csharp **string** connectionString = "Data Source=localhost;Initial Catalog=MyDatabase;Integrated Security=True"; **using** (SqlConnection connection = **new** SqlConnection(connectionString)) {  try  {  connection.Open();  *// Execute queries and display data in DataGridView here*  }  catch (Exception ex)  {  Console.WriteLine("Error connecting to database: " + ex.Message);  }  finally  {  connection.Close();  } } |
| --- |

**Populating DataGridView**:

| csharp DataTable dataTable = new DataTable(); using (SqlCommand **cmd** = new SqlCommand("SELECT \* FROM MyTable", connection)) {  using (SqlDataReader reader = **cmd**.ExecuteReader())  {  dataTable.Load(reader);  } } dataGridView.DataSource = dataTable; |
| --- |

# Connection-Oriented vs. Disconnected-Oriented Approach

**Connection-Oriented**:

* **Description**: Maintains a continuous connection to the database during operations.
* **Use Cases**: Suitable for real-time data updates or frequent interactions.
* **Example**: Using SqlConnection and SqlDataReader for reading data.

**Disconnected-Oriented**:

* **Description**: Establishes a connection, retrieves data, and then closes the connection. Uses DataSet and DataAdapter to manage data in memory.
* **Use Cases**: Suitable for scenarios with less frequent updates, improving performance.
* **Example**:

| csharp string query = "SELECT \* FROM MyTable"; SqlDataAdapter adapter = **new** **SqlDataAdapter**(query, connection); DataSet dataSet = **new** **DataSet**(); adapter.Fill(dataSet); |
| --- |

# Rectangular vs. Ragged Arrays

**Rectangular Array**:

* **Description**: All rows have the same number of columns.
* **Example**:

| csharp int[,] rectangularArray = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } }; |
| --- |

**Ragged Array**:

* **Description**: Each row can have a different number of columns.
* **Example**:

| csharp int[][] raggedArray = new int[3][] {  new int[] { 1 },  new int[] { 2, 3 },  new int[] { 4, 5, 6 } }; |
| --- |

# C# as a Type-Safe, Modern, and Interoperable Language

**Type-Safe**:

* **Description**: C# enforces type safety by ensuring variables are declared with specific types, preventing operations on incompatible types.
* **Example**: Variables cannot change type once declared.

| csharp int x = 10*;* // x = "Hello"*; // This would cause a compile-time error.* |
| --- |

**Modern**:

* **Description**: Supports advanced programming constructs like generics, async/await, LINQ, and more.
* **Example**: Using async/await for asynchronous programming.

| csharp public async Task<string> FetchDataAsync() {  using (HttpClient client = new HttpClient())  {  string response = await client.GetStringAsync("http://example.com");  return response;  } } |
| --- |

**Interoperable**:

* **Description**: Can interact with other .NET languages and COM components, facilitating code reuse and integration with existing systems.
* **Example**: Using a COM component in C#.

| csharp **using** System.Runtime.InteropServices;  [**ComImport, Guid("000209FF-0000-0000-C000-000000000046")**] **public** **class** **ApplicationClass** {  *// COM methods and properties* } |
| --- |

# Database Programming in C#

Database programming in C# involves interacting with databases to perform operations such as creating, reading, updating, and deleting records. ADO.NET and Entity Framework are the primary technologies used.

**Write Your Own C# Database Connection Code**:

| csharp **using** System; **using** System.Data.SqlClient;  **class** **Program** {  **static** **void** **Main**()  {  **string** connectionString = "your\_connection\_string\_here";    **using** (SqlConnection connection = **new** SqlConnection(connectionString))  {  **try**  {  connection.Open();  Console.WriteLine("Database connection opened successfully.");  *// Perform database operations*  }  **catch** (Exception ex)  {  Console.WriteLine($"An error occurred: {ex.Message}");  }  }  } } |
| --- |

**Connect to Any Data Source**:

* **Example: Connecting to MySQL**:

| csharp **using** MySql.Data.MySqlClient;  **class** **Program** {  **static** **void** **Main**()  {  **string** connectionString = "server=your\_server;user=your\_username;database=your\_database;port=3306;password=your\_password;";    **using** (MySqlConnection connection = **new** MySqlConnection(connectionString))  {  **try**  {  connection.Open();  Console.WriteLine("MySQL Database connection opened successfully.");  *// Perform database operations*  }  **catch** (Exception ex)  {  Console.WriteLine($"An error occurred: {ex.Message}");  }  }  } } |
| --- |

**DataSets and DataAdapters**

**DataSets**:

* **Description**: In-memory representations of data.
* **Example**:

| csharp **using** System; **using** System.Data; **using** System.Data.SqlClient;  **class** **Program** {  **static** **void** **Main**()  {  **string** connectionString = "your\_connection\_string\_here";  **string** query = "SELECT \* FROM your\_table";   **using** (SqlConnection connection = **new** SqlConnection(connectionString))  {  SqlDataAdapter adapter = **new** SqlDataAdapter(query, connection);  DataSet dataSet = **new** DataSet();  adapter.Fill(dataSet);   Console.WriteLine("Data retrieved successfully.");  *// Display data (optional)*  **foreach** (DataRow row **in** dataSet.Tables[0].Rows)  {  Console.WriteLine(**string**.Join(", ", row.ItemArray));  }  }  } } |
| --- |

**Data Adapters**:

* **Description**: Used to fill DataSets and update the data source.
* **Example**:

| csharp string query = "SELECT \* FROM your\_table"; SqlDataAdapter adapter = **new** **SqlDataAdapter**(query, connection); DataSet dataSet = **new** **DataSet**(); adapter.Fill(dataSet); |
| --- |

**Display the Data in the DataSet**

**Console Application Example**:

| csharp **foreach** (DataTable **table** **in** dataSet.Tables) {  **foreach** (DataRow row **in** **table**.Rows)  {  **foreach** (**var** item **in** row.ItemArray)  {  Console.Write(item + " ");  }  Console.WriteLine();  } } |
| --- |

**Navigate a Database with C#**

**Using DataReader**:

* **Description**: Provides a forward-only read-only access to data from the database.
* **Example**:

| csharp **using** (SqlCommand command = **new** SqlCommand(query, connection)) {  **using** (SqlDataReader reader = command.ExecuteReader())  {  while (reader.Read())  {  Console.WriteLine(reader[0] + ", " + reader[1]); *// Example to read first two columns*  }  } } |
| --- |

**Add, Update, and Delete Records**

**Add Record**:

* **Example**:

| csharp string **insertQuery** = "INSERT INTO your\_table (Column1, Column2) VALUES (@Value1, @Value2)"*;* using (SqlCommand command = new SqlCommand(**insertQuery,** connection)) {  command.Parameters.**AddWithValue("@Value1",** value1)*;*  command.Parameters.**AddWithValue("@Value2",** value2)*;*  command.ExecuteNonQuery()*;* } |
| --- |

**Update Record**:

* **Example**:

| csharp string updateQuery = "UPDATE your\_table SET Column1 = @Value1 WHERE Column2 = @Value2"; **using** (SqlCommand command = **new** **SqlCommand**(updateQuery, connection)) {  command.Parameters.AddWithValue("@Value1", **newValue1**);  command.Parameters.AddWithValue("@Value2", targetValue2);  command.ExecuteNonQuery(); } |
| --- |

**Delete Record**:

* **Example**:

| csharp string deleteQuery = "**DELETE** FROM your\_table WHERE Column1 = @Value1"; using (SqlCommand command = new SqlCommand(deleteQuery, connection)) {  command.Parameters.AddWithValue("@Value1", targetValue1);  command.ExecuteNonQuery(); } |
| --- |

**Example with ASP.NET**

**ASP.NET Web Form with Database Interaction**:

1. **ASP.NET Web Form**:

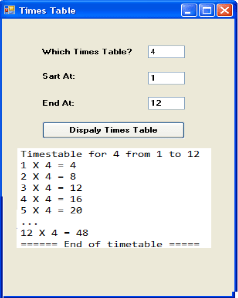
| html <%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="\_Default" %> **<!DOCTYPE html>** <html> <head>  <title>Database Example</title> </head> <body>  <form id="form1" runat="server">  <div>  <asp:GridView ID="GridView1" runat="server"></asp:GridView>  </div>  </form> </body> </html> |
| --- |

2. **Code-Behind (Default.aspx.cs)**:

| csharp using System; using System.Data; using System.Data.SqlClient;  public partial class \_Default : System.Web.UI.Page {  protected void Page\_Load(object sender, EventArgs e)  {  **if** (!IsPostBack)  {  BindGrid();  }  }   private void BindGrid()  {  string connectionString = "your\_connection\_string\_here";  string query = "SELECT \* FROM your\_table";   using (SqlConnection connection = new SqlConnection(connectionString))  {  SqlDataAdapter adapter = new SqlDataAdapter(query, connection);  DataTable dataTable = new DataTable();  adapter.Fill(dataTable);  GridView1.DataSource = dataTable;  GridView1.DataBind();  } } |
| --- |

# QUESTION 4 ASSIGNMENT 1

Design a windows forms application that performs the following



i. Allow the user to input the following variables through text boxes [3]

 any number num

 starting number start\_num

 end number end\_num

ii. Use any appropriate loop to multiply num with numbers from start\_num to

end\_num and display the computations in a list box as shown on the fig above.

[6]

iii. The program should display:

 The opening statement as shown in the fig above [3]

 The closing statement

| **using** System; **using** System.Windows.Forms;  **namespace** **WindowsFormsApp1** {  **public** **partial** **class** **Form1** : **Form**  {  **public** **Form1**()  {  InitializeComponent();  }   **private** **void** **button1\_Click**(**object** sender, EventArgs e)  {  **try**  {  *// Convert input from text boxes to integers*  **int** num = Convert.ToInt32(txtnum.Text);  **int** start\_num = Convert.ToInt32(txtstart.Text);  **int** end\_num = Convert.ToInt32(txtend.Text);   *// Clear previous items in the list box*  listBox1.Items.Clear();   *// Add header to the list box*  listBox1.Items.Add($"Times Table for {num} from {start\_num} to {end\_num}");   *// Generate and add times table entries to the list box*  **for** (**int** i = start\_num; i <= end\_num; i++)  {  listBox1.Items.Add($"{i} X {num} = {i \* num}");  }   *// Add footer to indicate the end of the times table*  listBox1.Items.Add("====== End of timetable ====");  }  **catch** (FormatException)  {  MessageBox.Show("Please enter valid numbers.", "Input Error", MessageBoxButtons.OK, MessageBoxIcon.Error);  }  }  } } |
| --- |

# ASSIGNMENT TWO

**1. C# as an Event-Driven Programming Language**

* **Event-Driven Nature**:
  + C# allows applications to respond to user actions or other events.
  + Events are actions or occurrences such as button clicks, mouse movements, or system-generated notifications.
* **Event Handlers**:
  + Methods that handle or respond to specific events. They are registered with events to define what should happen when an event occurs.
* **Delegates**:
  + Special types that define a method signature. Delegates are used to declare event handlers.

**2. Exception Handling in C#**

* **Definition**:
  + Exception handling in C# is a mechanism to handle runtime errors, ensuring the program can continue or fail gracefully.
* **Example**:

| csharp **try** {  *// Code that might throw an exception*  **int**[] numbers = { 1, 2, 3 };  Console.**WriteLine**(numbers[5]); } **catch** (IndexOutOfRangeException ex) {  *// Code to handle the exception*  Console.**WriteLine**("Index out of range: " + ex.Message); } **finally** {  *// Code that runs regardless of an exception*  Console.**WriteLine**("This will always execute."); } |
| --- |

**3. DataReader vs. DataSet**

* **DataReader**:
  + Forward-only and read-only access to data from a database.
  + Uses fewer resources, providing fast data retrieval.
  + Connected architecture; maintains a connection to the database while reading data.
* **DataSet**:
  + In-memory representation of data; can hold multiple tables and relationships.
  + Supports disconnected architecture; data can be manipulated without a continuous database connection.
  + Allows data manipulation (update, delete, insert) and has a richer set of functionalities like data relations and constraints.

**5. Interface vs. Abstract Class**

* **Interface**:
  + Contains only method signatures and properties, no implementation.
  + A class can implement multiple interfaces.
  + Cannot contain any code or data; purely a contract for what methods and properties a class must implement.
* **Abstract Class**:
  + Can contain both abstract methods (without implementation) and concrete methods (with implementation).
  + A class can inherit only one abstract class.
  + Can contain fields, properties, and methods with implementations, providing a common base for derived classes.

# 

# QUESTION 6 ASSIGNMENT TWO

Create a Windows form application that allows user to log in using username and

password. The application should connect to a MS SQL Server database named

LADS. If username and password is correct show a message box that displays a

message that reads “Welcome [username]!”

Assume the database has a table named projects. Your program should retrieve all the

project records and populate a dataGridView . [25]

Take note of the following

i. Make use of the try catch block

ii. Database connection timeout is 5 seconds.

iii. Replace the username in the message box with the username

that was used to log in.

iv. Print out the source code and screen shorts of the login form

and the message box

| **using** System; **using** System.Data; **using** System.Data.SqlClient; **using** System.Windows.Forms;  **namespace** **LoginApp** {  **public** **partial** **class** **Form1** : **Form**  {  **public** **Form1**()  {  InitializeComponent();  }   **private** **void** **btnLogin\_Click**(**object** sender, EventArgs e)  {  **string** username = txtUsername.Text;  **string** password = txtPassword.Text;   **string** connectionString = "Server=your\_server\_name;Database=LADS;User Id=your\_db\_username;Password=your\_db\_password;Connection Timeout=5;";   **using** (SqlConnection connection = **new** SqlConnection(connectionString))  {  **try**  {  connection.Open();  **string** query = "SELECT COUNT(1) FROM Users WHERE Username=@Username AND Password=@Password";  SqlCommand cmd = **new** SqlCommand(query, connection);  cmd.Parameters.AddWithValue("@Username", username);  cmd.Parameters.AddWithValue("@Password", password);   **int** count = Convert.ToInt32(cmd.ExecuteScalar());   **if** (count == 1)  {  MessageBox.Show($"Welcome {username}!");   *// Retrieve project records*  **string** projectQuery = "SELECT \* FROM projects";  SqlDataAdapter dataAdapter = **new** SqlDataAdapter(projectQuery, connection);  DataTable dataTable = **new** DataTable();  dataAdapter.Fill(dataTable);   *// Populate DataGridView*  dataGridView1.DataSource = dataTable;  }  **else**  {  MessageBox.Show("Username or Password is incorrect");  }  }  **catch** (Exception ex)  {  MessageBox.Show($"Error: {ex.Message}");  }  }  }  } } |
| --- |