**Chapter 1**

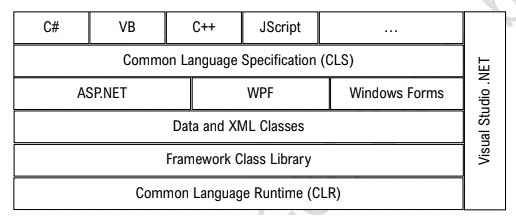
**Introduction to Visual Programming**

**Microsoft .NET Framework**

The Microsoft .NET Framework is a comprehensive platform for building, deploying, and running applications. It provides a robust and versatile environment that supports various programming languages, allowing developers to create a wide range of applications, from desktop software to web services.

**Components**

The .NET Framework is a comprehensive platform that consists of several key components, each playing a crucial role in supporting and facilitating the development, deployment, and execution of applications. Here are the main components of the .NET Framework:



**Common Language Runtime (CLR)**

The CLR is the runtime environment that manages the execution of .NET applications. It provides essential services such as memory management, garbage collection, and exception handling. The CLR also enables interoperability between different programming languages.

**Framework Class Library**

The Class Library, also known as the Framework Class Library (FCL), is a collection of pre-built code that developers can use to perform common programming tasks. It includes classes, interfaces, and namespaces that cover a wide range of functionalities, making it easier for developers to write code without starting from scratch.

**ASP.NET**

ASP.NET is a web application framework that facilitates the development of dynamic, data-driven web

applications and services. It supports various programming models, including Web Forms for rapid application development and MVC (Model-View-Controller) for more structured and scalable web applications.

**Windows Forms (WinForms)**

WinForms is a graphical user interface (GUI) framework for creating Windows desktop applications.

**Windows Presentation Foundation (WPF)**

WPF is a more modern and flexible GUI framework that allows developers to create visually stunning Windows desktop applications. It supports advanced graphics, multimedia, and data-binding capabilities, providing a more immersive user experience.

**ADO.NET**

ADO.NET is a set of components that facilitate data access and manipulation. It includes classes for connecting to databases, executing queries, and managing data, making it an integral part of building database-driven applications.

**Language Compilers**

NET supports multiple programming languages, and language compilers are essential components that

translate code written in languages like C#, VB.NET, F#, and others into Common Intermediate Language (CIL) code, which is then executed by the CLR.

**Common Language Specification**

The Common Language Specification (CLS) defines a set of rules within the Common Language Infrastructure (CLI) to ensure interoperability between different .NET languages. It establishes conventions for data types, naming, method signatures, exception handling, and other aspects to facilitate seamless integration of components written in diverse languages.

The ***.NET Framework*** has evolved over the years, with each version introducing new features, improvements, and enhancements to the platform. Here are the major versions of the .NET Framework, along with their significance compared to earlier versions:

**Note:**

.NET Framework 1.0 (2002)

• Initial release of the .NET Framework.

• Introduced Common Language Runtime (CLR) and Base Class Library (BCL).

• Marked the shift towards a unified development platform for Windows applications.

.NET Framework 1.1 (2003)

• Minor updates and bug fixes.

• Improved ASP.NET performance and added support for mobile devices.

.NET Framework 2.0 (2005)

• Introduced Generics, providing increased type safety and performance.

• Added ASP.NET 2.0 with new controls and features.

• Introduced ClickOnce deployment for easier application deployment.

.NET Framework 3.0 (2006)

• Included Windows Presentation Foundation (WPF) for building rich user interfaces.

• Introduced Windows Communication Foundation (WCF) for building distributed and service-oriented

applications.

• Featured Windows Workflow Foundation (WF) for workflow-enabled applications.

**Benefits of Using .NET Framework**

**Language Interoperability**

.NET supports multiple programming languages, enabling developers to choose the language that best suits their expertise while still collaborating within the same application.

**Rich Class Library (FCL)**

The extensive Framework Class Library (FCL) provides pre-built, reusable code for common tasks, reducing development time and effort.

**Common Language Runtime (CLR)**

CLR offers features like automatic memory management, exception handling, and support for cross-language integration, enhancing application reliability and performance.

**Unified Development Platform**

.NET provides a unified development platform for building diverse applications, including desktop, web, mobile, and cloud-based solutions.

**Integrated Development Environment (IDE)**

Visual Studio, the primary IDE for .NET, offers powerful tools for coding, debugging, testing, and deployment, enhancing developer productivity.

**Security Features**

.NET incorporates robust security features, including Code Access Security (CAS), to control and restrict the actions of code, reducing the risk of malicious activities.

**Scalability and Performance**

.NET applications are known for their scalability and performance. Features like Just-In-Time (JIT) compilation and performance optimizations contribute to efficient execution.

**Cross-Platform Development**

With the introduction of .NET Core and subsequent versions, developers can build cross-platform applications that run on Windows, Linux, and macOS.

**Community and Support**

.NET has a large and active developer community, and Microsoft provides extensive documentation, support, and regular updates to the framework.

**Common Language Runtime (CLR)**

The Common Language Runtime (CLR) is the dynamic execution environment at the heart of the .NET Framework. It provides essential services for running managed code, ensuring memory management, security, and seamless interoperability across different programming languages. This detailed exploration sheds light on the inner workings of the CLR.

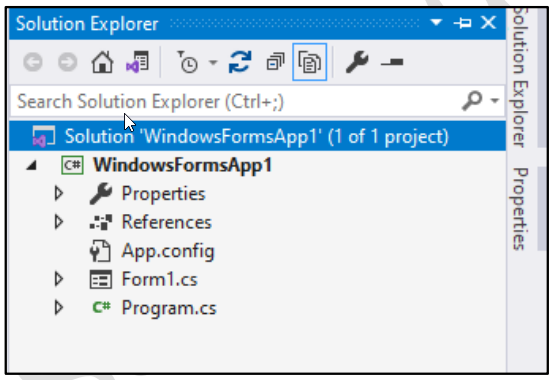
**Visual Studio Integrated Development Environment (IDE)**

Visual Studio is a robust integrated development environment (IDE) designed to streamline and enhance the software development process. Developed by Microsoft, Visual Studio has become a cornerstone for developers working on a wide array of applications, ranging from desktop and web solutions to mobile and cloud-based projects.

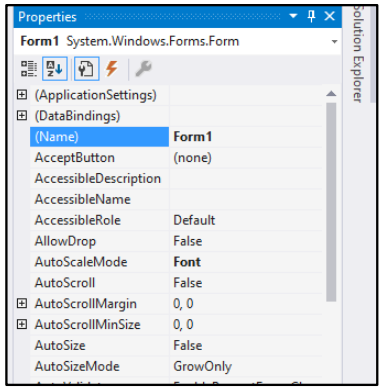
**Following are some important components of Visual Studio IDE:**

Toolbox The 'Toolbox' in Visual Studio is a central hub of pre-built components and controls, streamlining the development process by offering a visual palette for easy integration into applications. Accessible within the IDE, developers can efficiently drag and drop elements onto design surfaces, accelerating the creation of user interfaces.

**Solution Explorer** :

Solution Explorer stands as the organizational backbone within the Visual Studio IDE, providing developers with a structured and hierarchical view of their project's files, dependencies, and components. This indispensable tool not only presents a comprehensive overview of the project's architecture but also enables effortless navigation and management of various elements. From source code files to references, Solution Explorer acts as a centralized hub for developers to efficiently organize, locate, and modify project assets.

**Properties Window**:

 The 'Properties Window' in Visual Studio serves as a dynamic interface for inspecting and modifying properties of selected elements within a project. This essential tool provides developers with a centralized location to customize settings, styles, and configurations, offering a real-time view of the attributes associated with the currently selected item. Accessed through the View menu or by using the F4 shortcut key, the 'Properties Window' ensures a seamless and intuitive means of fine-tuning project elements, enhancing the efficiency and precision of the development process.

**Code View and Designer View**

In Visual Studio, 'Code View' and 'Designer View' represent two complementary perspectives through which developers can interact with their project. 'Code View' offers a text-based environment where developers can write, edit, and review source code directly. This is the domain for precise coding, allowing developers to delve into the intricacies of algorithms and logic.

On the other hand, 'Designer View' provides a graphical representation of the user interface, enabling developers to design and visualize the layout of their application components visually.

**Code View Menu Path: View > Code**

**Code View Shortcut Key: F7**

**using Statement**

The 'using' statement in C# is also used to import namespaces, making types within those namespaces accessible in the code.

**Namespace** In the 'Code View' of a form in Visual Studio, the term 'namespace' is a fundamental organizational construct that encapsulates a set of related code elements, such as classes, interfaces, and other types. Acting as a container for these elements, a 'namespace' helps prevent naming conflicts and organizes code in a modular and structured manner. It allows developers to group logically related components, fostering maintainability and enhancing the overall structure of the application.

**Partial Class**

In the 'Code View' of a form in Visual Studio, a 'partial class' is a feature that allows a single class to be defined across multiple files. Each file contributes to the overall class definition, enabling developers to organize and separate concerns within a class while maintaining a unified, cohesive structure. This concept is particularly useful for large projects, promoting code readability, and easing collaboration among developers working on different aspects of the same class.

**Class Constructor**

A constructor in C# is a special method within a class that is automatically called when an instance of the class is created. It is primarily used for initializing object state and executing setup tasks, providing a way to ensure that an object is in a valid state upon creation.

**Event and Event Handlers**

They constitute a crucial mechanism for handling user interactions and system notifications. An event represents a specific occurrence, such as a button click or a form load, and an event handler is a method that responds to and processes that event. For instance, in a Windows Forms application, a button click event may be handled by a corresponding event handler method that executes specific actions when the button is clicked.

**Graphical User Interface (GUI)**

Application A Graphical User Interface (GUI) application is a type of software that presents a visual interface to users, allowing them to interact with the application through graphical elements such as windows, buttons, icons, and menus. Unlike Console Applications, GUI applications provide a more intuitive and user-friendly experience, enabling users to perform tasks by interacting with visual components rather than entering commands in a console.

**Console Application**

A Console Application in the context of software development refers to a program that interacts with the user through a text-based console or command-line interface. Unlike graphical user interface (GUI) applications, which have visual elements like windows and buttons, a console application relies on text input and output. In C# and many other programming languages, console applications are often used for tasks that require straightforward user input and output, automation, or running processes in the background.

**Keyboard Events**

Keyboard events refer to interactions generated by a computer keyboard, triggering specific actions or responses within a software application. In the context of programming, keyboard events are often handled by event handlers, which are functions or methods designed to execute when a particular key or combination of keys is pressed, released, or held down.

**KeyDown Event**

The KeyDown event occurs when a key on the keyboard is pressed down. It is commonly used to capture the initial moment when a key is pressed before it is released. Developers often use this event to trigger specific actions or functions based on the key pressed, allowing for responsive and interactive user interfaces.

**KeyUp Event** The KeyUp event occurs when a key that was previously pressed is released. It provides developers with the ability to capture the moment when a key is released after being pressed down. This event is useful for implementing functionality that responds to the release of specific keys, such as confirming an action or navigating through a document.

**KeyPress Event** The KeyPress event is triggered when a character key is pressed and results in a character input. It is distinct from KeyDown and KeyUp events, as it specifically focuses on alphanumeric and symbol keys that produce a visible character.

**Mouse Events**

Mouse events in software development pertain to interactions initiated by the movement and clicks of a computer mouse. These events are crucial for capturing user input in graphical user interfaces, enabling developers to create interactive and responsive applications.

**Click Event**

The Click event occurs when a mouse button is pressed and released on a specific user interface element, such as a button or a graphic. It is a fundamental mouse event used to trigger actions or functions when the user clicks on a designated area. Click events are commonly employed in user interfaces for interactive elements like buttons, checkboxes, or hyperlinks**.**

**MouseMove Event**

The MouseMove event is triggered when the mouse pointer moves over a user interface element. This event allows developers to capture and respond to the mouse's movement, enabling features such as dynamic tooltips, highlighting elements on hover, or tracking the mouse position for interactive graphics. **MouseDown and MouseUp Events**

The MouseDown event occurs when a mouse button is pressed, while the MouseUp event occurs when the mouse button is released. These events are essential for capturing the start and end of a mouse click or drag operation. Developers often use these events to implement functionalities like drag-and-drop, drawing, or resizing elements based on mouse interactions. The combination of MouseDown and MouseUp events allows for the detection and handling of mouse button actions. MouseEnter and **MouseLeave Events**

The 'MouseEnter' event is triggered when the mouse pointer enters the boundaries of a specified user interface element. This event is valuable for capturing the moment when the cursor starts hovering over an element, allowing developers to initiate actions or visual changes. Common use cases include displaying tooltips, highlighting elements, or triggering animations to provide immediate feedback to the user upon entering a specific area.

--------------------------------------------------------------------------------------------------------------------------

**Chapter 2**

***C# Language Core***

**Basic Terminologies**

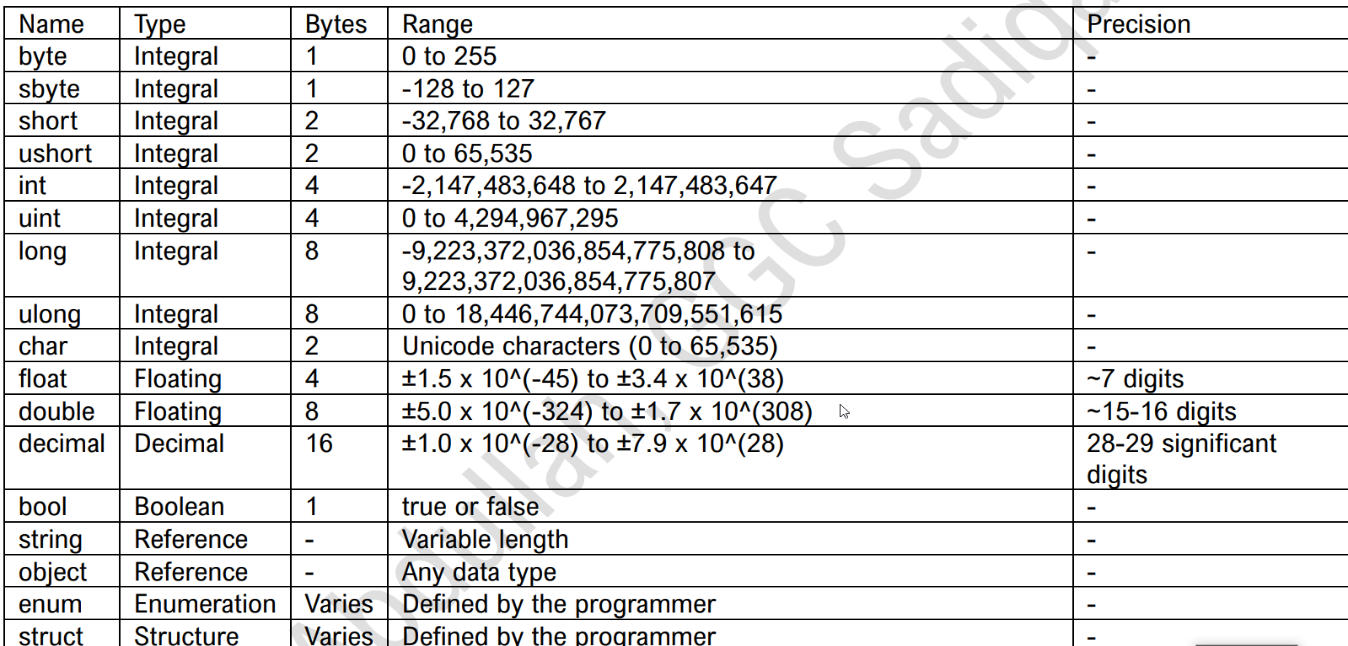
We will discuss basic terminologies of the C# language.

**Variable**

A variable is a named storage location in a computer's memory that holds a data value, allowing a program to store, retrieve, and manipulate information during its execution. Variables are essential elements in programming languages, representing dynamic entities whose values can change during the course of a program.

**Data Type**

A data type in programming specifies the type of data that a variable can hold, defining the characteristics and operations applicable to the stored values. C# supports following data types: -



**Variable Declaration**

Variable Declaration is the process of specifying the data type and name of a variable before it is used in a program. This informs the compiler about the type of data the variable will store. For example,

int age;

string message = "Hello, World!";

**Variable Initialization**

Variable Initialization is the process of assigning an initial value to a variable at the time of declaration or later in the program. For example,

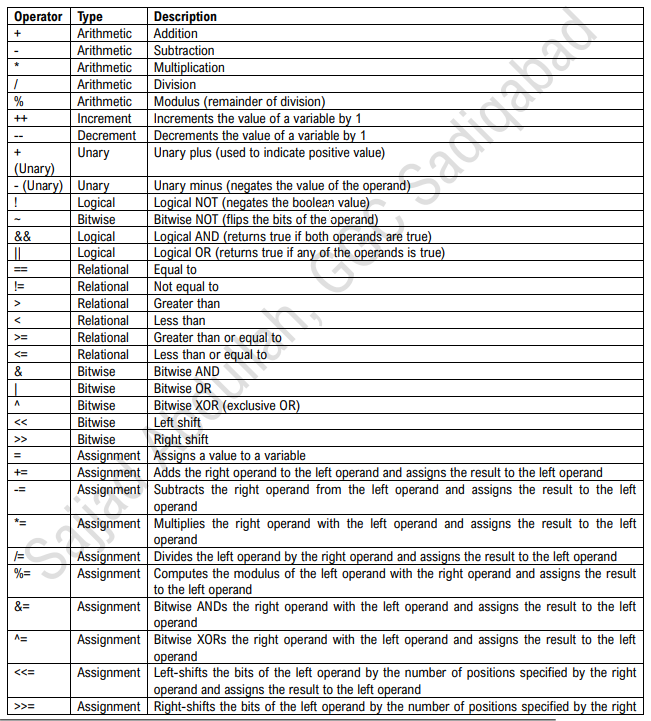
int count = 10;

string name;

name = "John Doe";

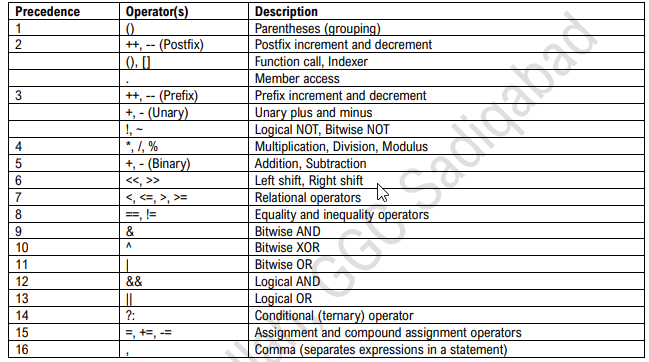
**Operators**

An operator in C# is a symbol that represents a specific operation on one or more operands, producing a result. These operators serve various purposes, including arithmetic computations, logical operations, bitwise manipulation, and assignment of values. Following operators are supported in C#: -



**Operator Precedence**

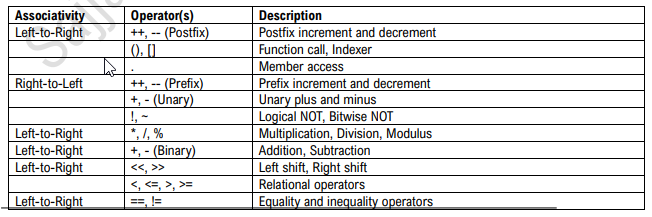
Operator precedence defines the order in which different operators are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. Following table shows Operator Precedence in C#: -



**Operator Associativity**

Operator associativity determines the order in which operators of the same precedence are evaluated in the absence of parentheses. Operators with left-to-right associativity are evaluated from left to right, and operators with right-to-left associativity are evaluated from right to left. For example, in the expression a + b + c, the + operator has left-to-right associativity, so it is evaluated as (a + b) + c.

Following table shows the Operator Associativity in C#: -



**Expression**

An expression is a combination of values, variables, operators, and function calls that results in a single value.

Following are different types of expressions in C#: -

Arithmetic Expressions Involving mathematical operations like addition, subtraction, multiplication, and division. Relational Expressions Evaluating relationships between values, such as equality (==), inequality (!=), greater than (>), etc. Logical Expressions Combining boolean values using logical operators like AND (&&), OR (||), and NOT (!). Conditional (Ternary) Expressions A shorthand way of writing if-else statements in a single line.

Assignment Expressions Assigning a value to a variable using the assignment operator (=). Bitwise Expressions Manipulating individual bits of binary numbers using operators like AND (&), OR (|), and XOR (^). String Concatenation Expressions Combining strings using the concatenation operator (+). Function Call Expressions Invoking a function or method to perform a specific operation.

**Examples Here are two examples of arithmetic expressions in C#:**

Example 1: Simple Addition

int num1 = 10;

int num2 = 5;

int result = num1 + num2;

Console.WriteLine($"The sum of {num1} and {num2} is: {result}");

**Scope and Lifetime of Variables**

**Scope:** The scope of a variable refers to the region in a program where the variable can be accessed or modified. Variables can have local scope, meaning they are accessible only within a specific block of code, or global scope, allowing access throughout the entire program. Local Variables Local variables are variables declared within a specific block or method in a program and are only accessible within that scope**.**

**Global Variables** Global variables are variables declared at the outermost level of a program, accessible from any part of the code, and have a scope extending throughout the entire program.

**Static Variables** Static variables are variables in a program that retain their values across multiple calls to a function, maintaining their state between invocations, and they are declared with the `static` keyword.

**Lifetime:** The lifetime of a variable is the duration during which the variable exists in the computer's memory. It begins when the variable is declared or initialized and ends when it goes out of scope or the program terminates.

**Class and Object**

**Class:** A class is a blueprint or template that defines the structure and behavior of objects. It serves as a prototype for creating objects by specifying attributes (fields) and methods that the objects will have. **Object:** An object is an instance of a class, representing a real-world entity or concept. Objects are created from classes and have their own unique state (attributes) and behavior (methods).

using System;

// Define a simple class class Car

{

// Fields (attributes)

public string Model; public int Year;

// Method (behavior)

public void StartEngine()

{

Console.WriteLine($"{Model} is starting the engine.");

}

}

class Program

{

static void Main()

{

// Create an object of the Car class

Car myCar = new Car();

// Set object properties

myCar.Model = "Toyota Camry"; myCar.Year = 2022;

// Call object method

myCar.StartEngine();

// Output object properties Console.WriteLine($"Car Details: Model = {myCar.Model}, Year = {myCar.Year}");

}

}

**Conditional Structures**

Conditional structures in programming are constructs that allow the execution of different code blocks based on specified conditions, enabling decision-making in the flow of a program. These conditional structures allow developers to control the flow of program execution based on different conditions and make decisions accordingly. In C#, the types of conditional structures include:

**1. if Statement:** Executes a block of code if a specified condition is true.

**2. if-else Statement**: Executes one block of code if a specified condition is true and another block if it's false.

**3. if-else if-else Statement:**

Allows for multiple conditions to be checked in sequence, executing the block associated with the first true condition.

**4. switch Statement:**

Provides a way to handle multiple possible conditions by evaluating the value of an expression against constant case values.

**5. Conditional Operator (`? :`):**

The conditional operator in C# (`? :`) is a ternary operator that provides a concise way to express conditional statements in a single line, evaluating one of two expressions based on a specified condition. **if Statement**

The `if` statement is a fundamental conditional structure in C# that allows the execution of a block of code only if a specified condition is true. If the condition evaluates to false, the block is skipped. The `if` statement is crucial for implementing decision-making in a program, allowing different code paths based on specified conditions.

**Syntax**

if (condition)

{

// Code to be executed if the condition is true

}

**if-else Statement**

The `if-else` statement in C# extends the basic `if` statement by providing an alternative block of code to execute when the specified condition is false. It allows for two different paths of execution based on the outcome of the condition. The `if-else` statement is useful when there are two possible outcomes based on a condition, providing a way to handle both scenarios in a clear and organized manner.

**Syntax**

if (condition)

{

// Code to be executed if the condition is true

}

else

{

// Code to be executed if the condition is false

}

**if-else if-else Statement**

The `if-else if-else` statement in C# allows for checking multiple conditions in sequence. It provides a series of conditions, each with its own block of code to be executed if the condition is true. If none of the conditions is true, an optional `else` block can be used to specify the code to be executed in that case. The `if-else if-else` statement is helpful when there are multiple possible conditions, and you want to execute the code associated with the first true condition, or the code in the `else` block if none of the conditions is true.

**Syntax**

if (condition1)

{

// Code to be executed if condition1 is true

}

else if (condition2)

{

// Code to be executed if condition2 is true

}

// Additional else if blocks can be added as needed else

{

// Code to be executed if none of the conditions is true

}

**switch Statement**

The `switch` statement in C# provides a way to handle multiple possible conditions by evaluating the value of an expression against constant case values. It allows for a more organized and efficient way to write code for multiple branching conditions. The `switch` statement is particularly useful when dealing with scenarios where a single variable can take on multiple discrete values, and different actions need to be taken for each value.

**Syntax**

switch (expression)

{

case constantValue1:

// Code to be executed if expression equals constantValue1 break;

case constantValue2:

// Code to be executed if expression equals constantValue2 break;

// Additional case blocks can be added as needed default:

// Code to be executed if none of the case values match expression break;

}

**Conditional Operator (`? :`)**

The conditional operator, often referred to as the ternary operator, is a concise way to express conditional statements in a single line. It evaluates a boolean expression and returns one of two values based on whether the expression is true or false. The conditional operator is useful for simplifying simple conditional assignments in a single line of code. It can improve code readability when the logic is straightforward.

**Syntax**

result = (condition) ? expressionIfTrue : expressionIfFalse;

**Loop Structures**

Loop structures in programming, such as `for`, `while`, and `do-while` loops, allow the repetitive execution of a block of code as long as a specified condition is true, providing a mechanism for efficient iteration over data or tasks. In C#, there are three main types of loops:

**1. for Loop**: Executes a block of code a specified number of times, iterating over a range of values.

**2. while Loop:** Repeatedly executes a block of code as long as a specified condition is true, with the condition checked before each iteration

**3. do-while Loop:** Similar to the `while` loop, but the condition is checked after each iteration, ensuring that the block of code is executed at least once.

**for Loop**

The `for` loop in C# is used for iterating over a range of values or elements. It consists of three parts: initialization, condition, and increment/decrement, allowing precise control over the loop execution. The `for` loop is particularly useful when the number of iterations is known, providing a compact and structured way to express looping constructs.

**Syntax**

for (initialization; condition; increment/decrement)

{

// Code to be executed in each iteration

}

**while Loop**

The `while` loop in C# is used to repeatedly execute a block of code as long as a specified condition is true. The condition is checked before each iteration, and the loop continues until the condition becomes false. `while` loop is useful when the number of iterations is not known beforehand, and the loop should continue until a specific condition is met.

**Syntax**

while (condition)

{

// Code to be executed in each iteration

}

**do-while Loop**

The `do-while` loop in C# is similar to the `while` loop, but the condition is checked after each iteration. This ensures that the block of code is executed at least once, even if the condition is initially false. The `do-while` loop is useful when you want to ensure that the loop body is executed at least once, regardless of the initial condition. It is suitable for scenarios where the loop should run until a certain condition is met.

**Syntax**

do

{

// Code to be executed in each iteration

} while (condition);

**break Statement**

The `break` statement in C# is used to exit a loop prematurely, whether it's a `for`, `while`, or `do-while` loop. It is often used within conditional statements to terminate the loop based on a certain condition. The `break` statement is valuable when you need to terminate a loop based on a certain condition, providing a way to exit the loop before it completes all iterations.

Example

using System;

class BreakStatementExample

{

static void Main()

{

// Example: Print numbers from 1 to 5, but break if the value is 3

for (int i = 1; i <= 5; i++)

{

if (i == 3)

{

Console.WriteLine("Breaking the loop at 3."); break;

// Exit the loop when i equals 3

}

Console.WriteLine(i);

}

}

}

**continue Statement**

The `continue` statement in C# is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration. It is typically used within conditional statements to control the flow of the loop. The `continue` statement is useful when you want to skip specific iterations of a loop based on a certain condition, allowing more control over the loop's behavior.

Example

using System;

class ContinueStatementExample

{

static void Main()

{

// Example: Skip printing the number 3 in a loop for (int i = 1; i <= 5; i++)

{ if (i == 3)

{

Console.WriteLine("Skipping the number 3.");

continue;

// Skip the rest of the code and move to the next iteration

}

Console.WriteLine(i);

}

}

**Functions**

A function, also known as a method, is a block of code that performs a specific task or set of tasks. Functions are defined within classes and can be called (invoked) from other parts of the program to execute their functionality. Functions provide a way to modularize code, making it more organized, reusable, and easier to maintain.

**Syntax**

access\_modifier return\_type function\_name(parameters)

{

// Code to be executed return result;

// Optional: Return a value based on the return type

}

**Where:**

• Access Modifier: Specifies the visibility or accessibility of the function (e.g., public, private, etc.).

• Return Type: Specifies the data type of the value that the function returns. If the function doesn't return a value, use void.

• Function Name: A unique identifier for the function.

• Parameters: Input values passed to the function (optional).

• Code Block: The block of code executed when the function is called.

• Return Statement: Optional statement to return a value from the function

**Function Declaration and Definition**

**In** C#, function declaration and definition involve specifying the function's signature, including its return type, name, parameters, and the code block that represents its functionality**.**

**Function Parameters and Arguments**

In C#, function parameters allow you to pass values to a function, and function arguments are the actual values passed during a function call. Parameters define the input data required by a function, and arguments provide the specific values for those parameters. Understanding function parameters and arguments is essential for creating flexible and reusable functions in C#.

**Example 1: Basic Parameters and Arguments**

using System;

class ParametersAndArgumentsExample

{

// Function with parameters public static void GreetUser(string name)

{

Console.WriteLine($"Hello, {name}!");

}

static void Main()

{

// Calling the function with arguments

GreetUser("Alice");

GreetUser("Bob");

}

}

**Return Statement** In C#, the return statement is used to exit a function and optionally provide a result back to the calling code. The return statement is often used to convey the output or result of a function's operation. Example Let's consider an example where the function calculates the area of a rectangle: using System;

class RectangleExample

{

// Function to calculate the area of a rectangle public static double CalculateRectangleArea(double length, double width)

{

double area = length \* width; return area;

// Return the calculated area to the calling code

}

static void Main()

{

// Calling the function to calculate the area of a rectangle double length = 10.5; double width = 5.2; double area = CalculateRectangleArea(length, width); Console.WriteLine($"The area of the rectangle is: {area} square units");

}

}

**Access Modifiers**

Access modifiers in C# are keywords that define the visibility or accessibility level of types and type members (such as fields, methods, properties, etc.) within a program. They control which parts of the code can access or interact with certain elements. Access modifiers help in enforcing encapsulation and controlling the visibility of code elements. They contribute to the design and maintainability of code by specifying who can use or extend certain parts of the program.

Here are the main access modifiers in C#:

**1. Public (`public`)** Public members are accessible from any other code within the same assembly or from other assemblies. There is no restriction on accessibility.

**2. Private (`private`)** Private members are only accessible within the same type or class. They cannot be accessed from outside the class or other types

**. 3. Protected (`protected`)** Protected members are accessible within the same type and by derived types. They are not accessible from outside the class.

**4. Internal (`internal`)** Internal members are accessible within the same assembly (project). They are not accessible from outside the assembly.

**5. Protected Internal (`protected internal`)** Protected internal members are accessible within the same assembly and by derived types. They can also be accessed from outside the assembly, but only by types that are derived from the declaring class.

**6. Private Protected (`private protected`)** Private protected members are accessible within the same assembly and by derived types, but only if they are in the same assembly. Static and Instance Functions Static functions are useful when the operation does not depend on instance-specific data, while instance functions are used when the operation is related to the state of a particular instance of the class.

In C#, functions can be classified into two main types: static functions and instance functions.

1. **Static Functions Definition:** Static functions are associated with the class itself rather than an instance of the class. They are called using the class name, and they do not require an instance of the class to be created.

**Example**

using System;

class StaticExample

{

// Static function public static void PrintMessage()

{

Console.WriteLine("This is a static message.");

}

static void Main()

{

// Calling the static function using the class name StaticExample.PrintMessage();

}

}

1. **Instance Functions Definition:** Instance functions, also known as non-static functions, are associated with instances (objects) of the class. They are called using an instance of the class, and they can access and modify instance-specific data.

**Example**

using System;

class InstanceExample

{

// Instance function public void DisplayMessage()

{

Console.WriteLine("This is an instance message.");

}

static void Main()

{

// Creating an instance of the class InstanceExample instance = new InstanceExample();

// Calling the instance function using the instance instance.DisplayMessage();

}

}

**Arrays**

Arrays in C# are collections of elements of the same data type, stored in contiguous memory locations. They provide a convenient way to group and access multiple values under a single variable name. The elements in an array are accessed using an index, with the index starting from 0 for the first element. Arrays can be of a fixed size (static) or resizable (dynamic) using classes like `List` or `ArrayList`

**Declaration**

In C#, array declaration involves specifying the data type of the elements and the array name, along with an optional size.

For example, int[] numbers;

It declares an integer array named `numbers`. The size can be specified later during initialization or left unspecified for dynamic sizing. Initialization

**Array initialization** in C# involves assigning values to the elements of the array.

For example, int[] numbers = {1, 2, 3, 4, 5};

It initializes an integer array named `numbers` with values 1, 2, 3, 4, and 5. Alternatively, you can initialize an array with a specified size using `new` keyword, such as

int[] scores = new int[3]

**Two Dimensional Arrays Declaration**

int[,] matrix;

**Initialization**

int[,] matrix = new int[3, 3];

**Example**

using System;

class TwoDArrayExample

{

static void Main()

{

// Declaration and Initialization of a 2-D Array int[,] matrix = new int[3, 3]

{

{1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

// Accessing and printing elements of the 2-D array

Console.WriteLine("2-D Array Elements:"); for (int i = 0;

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

{

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

}

Console.WriteLine(); }

}

}

**Three Dimensional Arrays Declaration**

int[,,] cube;

**Initialization**

int[,,] cube = new int[3, 3, 3];

**Example**

using System;

class ThreeDArrayExample { static void Main()

{

// Declaration and Initialization of a 3-D Array

int[,,] cube = new int[2, 2, 2] { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };

// Accessing and printing elements of the 3-D array

Console.WriteLine("3-D Array Elements:");

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

{

for (int j = 0; j < 2; j++)

{

for (int k = 0; k < 2; k++)

{

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

}

Console.WriteLine(); } Console.WriteLine();

}

}

**List<T>**

List is a generic class in C# that provides a dynamic, resizable array-like structure. It is part of the `System.Collections.Generic` namespace and is widely used for storing and manipulating collections of elements. The `T` in `List` represents the type of elements the list can store, making it flexible and typesafe.

`List` is a versatile and commonly used collection in C# due to its dynamic resizing capabilities and extensive set of methods for manipulating collections of elements. Key Features of List

**1. Dynamic Sizing:** Lists automatically resize themselves to accommodate the number of elements added to them.

**2. Type Safety:** Lists are strongly typed, ensuring that you can only add elements of the specified type. **3. Versatility:** Lists offer a variety of methods and properties for efficient element manipulation

**Example**

using System;

using System.Collections.Generic;

class ListExample { static void Main()

{

// Creating a List of integers List numbers = new List();

// Adding elements to the list numbers.Add(10);

numbers.Add(20);

numbers.Add(30);

// Accessing elements using index

Console.WriteLine("Element at index 1: " + numbers[1]);

// Iterating through the list

Console.WriteLine("List Elements:");

foreach (int number in numbers)

{

Console.WriteLine(number);

}

// Removing an element numbers.Remove(20);

// Count of elements in the list

Console.WriteLine("Number of elements in the list: " + numbers.Count);

// Check if a specific element is present

bool containsElement = numbers.Contains(30);

Console.WriteLine("List contains 30: " + containsElement);

}

}

**Structures**

A structure is a value type that allows the grouping of related data members under a single name. Unlike classes, structures are typically used for lightweight objects that do not require inheritance or have behavior associated with them. They are defined using the `struct` keyword and can contain fields, properties, and methods. Structures are well-suited for representing simple data structures, such as coordinates, points, or small sets of related values.

**The syntax for declaring a structure in C# is as follows:**

struct StructureName

{

// Fields (data members) of the structure

DataType1 Field1;

DataType2 Field2;

// ... // Properties, methods, and other members can also be included

}

**Strings**

In C#, a string is a sequence of characters represented using the `string` data type. Strings are used to store and manipulate text data. C# provides a rich set of methods for working with strings, and strings are immutable, meaning that once a string is created, its value cannot be changed. Strings are widely used in C# for tasks involving text manipulation, formatting, and representation.

**Enums**

Enums, short for enumerations, are a distinct value type in C# used to define a set of named integral constants. Enums make code more readable and maintainable by providing a way to represent a set of predefined values with meaningful names. Each named constant within an enum is assigned an integral value by default, starting from 0, and subsequent values increment by 1.

Exception Handling

Exception handling in C# is a mechanism that allows you to gracefully handle runtime errors, preventing the application from crashing. Exceptions are unexpected events that occur during the execution of a program and disrupt its normal flow.

In C#, the `try`, `catch`, `finally`, and `throw` keywords are used for exception handling. Here's an example program demonstrating exception handling:

using System;

class ExceptionHandlingExample { static void Main()

{ try

{

// Code that might throw an exception Console.Write("Enter a number: ");

int userInput = int.Parse(Console.ReadLine());

// Division by zero example (will throw an exception for 1/0)

int result = 10 / userInput;

// Array index out of bounds example (will throw an exception for an array with less than 5 elements) int[] numbers = { 1, 2, 3, 4 };

Console.WriteLine("Value at index 5: " + numbers[5]);

} catch (FormatException ex

{

Console.WriteLine($"FormatException: {ex.Message}");

} catch (DivideByZeroException ex)

{

Console.WriteLine($"DivideByZeroException: {ex.Message}");

} catch (IndexOutOfRangeException ex)

{

Console.WriteLine($"IndexOutOfRangeException:

{ex.Message}");

} catch (Exception ex)

{

// Catch-all block for other exceptions

Console.WriteLine($"An unexpected error occurred: {ex.Message}");

} finally

{

// Code in this block will always be executed, regardless of whether an exception occurred Console.WriteLine("Finally block executed.");

}

Console.WriteLine("Program continues after exception handling.");

}

}

--------------------------------------------------------------------------------------------------------------------------

**Chapter 3**

**Object Oriented Programming in C#**

**Namespace**

A namespace in C# provides a way to organize related classes and other types. It helps prevent naming conflicts and provides a hierarchical organization of code elements.

Let's define a namespace called `MyNamespace`:

namespace MyNamespace

{

// Classes, interfaces, and other types will be defined here

}

**Properties**

Properties of a class are member variables encapsulated within the class, often representing attributes or characteristics of objects, with associated getter and setter methods to access and modify their values. They provide controlled access to the internal state of objects and facilitate data abstraction and encapsulation in object-oriented programming.

Let's add properties to the `Person` class to represent attributes like name and age:

namespace MyNamespace { public class Person

{

// Properties public string Name { get; set; } public int Age { get; set;

}

}

}

**Getter and Setter Functions**

Getter and setter functions are methods used to access (get) and modify (set) the values of properties respectively. They provide controlled access to the internal state of an object, allowing validation and manipulation of data before it is assigned to or retrieved from a property. In C#, properties with both getter and setter methods are known as auto-implemented properties. They are declared using the `get` and `set` accessors, and the compiler automatically generates the underlying field to store the property value.

Here's how getter and setter functions work in the context of the provided `Person` class:

public class Person

{

// Auto-implemented properties with getter and setter

public string Name { get; set;

}

public int Age

{

get; set;

}

// Custom getter and setter methods for the Age property

public int GetAge()

{

return Age;

}

public void SetAge(int newAge)

{

// Add validation logic if needed if (newAge >= 0 && newAge <= 150)

// Assuming reasonable age range

{ Age = newAge;

}

else {

// Throw an exception or handle invalid age input accordingly throw new ArgumentException("Invalid age input.");

}

}

**Methods**

Methods of a class are functions or procedures encapsulated within the class definition that define the behavior or actions that objects of the class can perform. They allow manipulation of the object's state (properties) and facilitate interaction with other objects, enabling the execution of specific tasks or operations as defined by the class blueprint. Methods provide a means of abstraction and encapsulation, promoting modularity, reusability, and maintainability in object-oriented programming.

**Events**

Events of a class are mechanisms for communication between objects, allowing one object to notify other objects when a specific action or condition occurs. They enable the implementation of the publisher-subscriber pattern, where objects (subscribers) register to receive notifications (events) from another object (publisher) and respond accordingly. Events facilitate decoupling and modularity in object-oriented programming by allowing objects to react to changes or actions without requiring explicit knowledge of each other's implementation details.

**Constructors**

Constructors are special methods in a class used to initialize objects of that class by assigning initial values to their properties or performing other setup tasks.

**Types of Constructors**

Here are the different types of constructors along with their definitions. Each type of constructor serves a specific purpose and provides flexibility in object initialization and customization in object-oriented programming.

**Default Constructor**

A default constructor is a constructor with no parameters defined explicitly. It is automatically generated by the compiler if no constructors are explicitly defined in the class. • It initializes the object with default values, typically zero for numeric types, null for reference types, and false for boolean types. **Parameterless Constructor**

• A parameterless constructor is a constructor that takes no parameters explicitly.

• It initializes the object with predefined default values or performs any necessary setup tasks without requiring any external input. Parameterized Constructor

• A parameterized constructor is a constructor that accepts one or more parameters explicitly. • It allows customization of object initialization by accepting arguments that specify initial values for object properties.

**Copy Constructor**

• A copy constructor is a constructor that creates a new object by copying the state of an existing object of the same class.

• It initializes the new object with the same values as the existing object, providing a way to create a deep copy of the object.

**Inheritance**

Inheritance is a fundamental concept in object-oriented programming that allows classes to inherit properties and behavior from other classes.

In C#, inheritance enables code reuse, promotes modularity, and facilitates the creation of hierarchical relationships between classes.

**Benefits of Inheritance**

**Code Reusability:** Inheritance promotes reuse of code by allowing classes to inherit behavior from existing classes.

**Extensibility:** Derived classes can add new functionality or modify existing behavior inherited from the base class.

**Hierarchical Organization:** Inheritance facilitates the creation of hierarchical relationships between classes, making code more organized and easier to manage. Inheritance is a powerful mechanism in C# that enables code reuse, extensibility, and hierarchical organization of classes, leading to more maintainable and scalable software solutions. By understanding and leveraging inheritance, developers can design robust and flexible object-oriented systems in C#.

**Key Concepts Base Class and Derived Class**

**•** Inheritance involves two types of classes: a base class (also known as a parent class or superclass) and a derived class (also known as a child class or subclass).

• The derived class inherits properties, methods, and other members from the base class, allowing it to extend or specialize the functionality of the base class. Syntax of Inheritance

• In C#, inheritance is implemented using the colon (`:`) syntax, followed by the name of the base class.

• Example: `class DerivedClass : BaseClass` Access Modifiers and Inheritance

• Inherited members retain their access modifiers from the base class unless overridden in the derived class.

• Public and protected members are accessible in derived classes, while private members are not. Single Inheritance

• C# supports single inheritance, where a derived class can inherit from only one base class

. • This ensures simplicity and avoids issues such as the diamond problem encountered in multiple inheritance

**Encapsulation**

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) that promotes data hiding and abstraction by bundling data (attributes or properties) and methods (behaviors or operations) that operate on the data within a single unit, known as a class.

In C#, encapsulation ensures that the internal state of an object is protected from direct external access, allowing controlled and secure manipulation of object state through well-defined interfaces. Key Concepts

**Access Modifiers**

• Access modifiers such as `public`, `private`, `protected`, and `internal` are used to control the visibility and accessibility of members (properties and methods) within a class.

**Data Hiding**

• Encapsulation hides the internal state of an object from external code, preventing direct manipulation of data and ensuring the integrity and consistency of object state

• Private member variables encapsulated within a class are inaccessible from outside the class, limiting access to authorized methods.

**Abstraction**

• Encapsulation facilitates abstraction by exposing a simplified interface for interacting with objects, hiding complex implementation details and internal workings.

• Users of encapsulated objects interact with them through well-defined methods and properties, without needing to understand the underlying implementation.

**Polymorphism**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different types to be treated as objects of a common base type, enabling flexibility, extensibility, and code reuse.

In C#, polymorphism is achieved through method overriding and method overloading, along with interfaces and inheritance, allowing for dynamic binding and runtime resolution of method calls. Key Concepts Method Overriding

• Method overriding is a mechanism in which a derived class provides a specific implementation of a method that is already defined in its base class.

• It allows derived classes to replace or extend the behavior of inherited methods, providing customization and specialization. Method Overloading

• Method overloading is a feature that allows a class to define multiple methods with the same name but with different parameter lists.

• It enables the creation of methods with the same name that perform different tasks or accept different types of arguments, enhancing code readability and flexibility. Dynamic Polymorphism • Dynamic polymorphism, also known as runtime polymorphism, occurs when the method call is resolved at runtime based on the actual type of the object.

• It allows for the selection of the appropriate method implementation based on the runtime type of the object, enabling flexibility and extensibility in code execution.

**Example**

using System;

// Base class public class Shape { public virtual void Draw()

{

Console.WriteLine("Drawing a generic shape.");

}

}

// Derived class (Circle) public class Circle : Shape { public override void Draw()

{

Console.WriteLine("Drawing a circle.");

}

}

// Derived class (Rectangle) public class Rectangle : Shape { public override void Draw()

{

Console.WriteLine("Drawing a rectangle.");

}

}

class Program { static void Main(string[] args)

{

// Create objects of different shapes Shape shape1 = new Shape(); Shape shape2 = new Circle();

Shape shape3 = new Rectangle();

// Call the Draw() method for each shape shape1.Draw();

// Calls Draw() of Shape class shape2.Draw();

// Calls Draw() of Circle class shape3.Draw();

// Calls Draw() of Rectangle class

}

}

**Interfaces**

Interfaces are a key feature of object-oriented programming (OOP) in C# that define contracts for classes to implement certain behaviors or capabilities. They provide a way to achieve abstraction and polymorphism by specifying a set of method signatures without providing implementation details.

In C#, interfaces enable loose coupling and promote code reuse by allowing classes to adhere to common contracts while maintaining their own unique implementations. Key Concepts Interface Definition

• An interface in C# is a reference type that defines a contract containing method signatures, properties, events, or indexers.

• It serves as a blueprint for classes that implement it, specifying the methods and properties that the implementing classes must provide. Interface Implementation

• A class implements an interface by providing concrete implementations for all the members declared in the interface.

• A single class can implement multiple interfaces, enabling it to fulfill multiple contracts and exhibit polymorphic behavior. Interface Inheritance

• Interfaces support inheritance, allowing one interface to inherit from one or more other interfaces.

• Inherited interfaces can extend the contract by adding additional members, further promoting code reuse and modularity.

**Example**

Consider a scenario where we have an interface `IDrawable` representing the ability to draw shapes, and two classes `Circle` and `Rectangle` that implement this interface to provide specific drawing functionality.

using System;

// Interface representing drawable objects public interface IDrawable { void Draw();

}

// Class representing a circle public class Circle : IDrawable { public void Draw()

{

Console.WriteLine("Drawing a circle.");

}

}

// Class representing a rectangle public class Rectangle : IDrawable

{

public void Draw()

{

Console.WriteLine("Drawing a rectangle.");

}

}

class Program

{

static void Main(string[] args)

{

// Create objects of drawable shapes IDrawable circle = new Circle();

IDrawable rectangle = new Rectangle();

// Call the Draw() method for each shape circle.Draw();

// Draws a circle rectangle.Draw();

// Draws a rectangle

}

}

-------------------------------------------------------------------------------------------------------------------------------

**Chapter 4**

**Graphical User Interface Applications**

**Introduction**

In C#, Graphical User Interface (GUI) development is commonly done using frameworks like Windows Forms (WinForms) or Windows Presentation Foundation (WPF). These frameworks provide developers with a set of tools and components for creating interactive and visually appealing desktop applications. GUI applications in C# allow users to interact with the software through graphical elements such as buttons, textboxes, menus, and dialogs, providing a more intuitive and user-friendly experience compared to command-line interfaces.

**Windows Forms**

Windows Forms (WinForms) is a graphical user interface (GUI) framework for building desktop applications in the Microsoft .NET Framework. It provides a set of pre-built controls and components that developers can use to create rich and interactive user interfaces for Windows-based applications. With WinForms, developers can design and customize forms, dialogs, menus, and other visual elements using a drag-and-drop interface in development environments like Visual Studio.

**Windows Controls**

Windows Controls, also known as WinControls, are pre-built user interface elements provided by the Windows Forms framework for building desktop applications in C#. These controls include buttons, textboxes, labels, listboxes, dropdown lists, and many others, allowing developers to create interactive and visually appealing user interfaces without writing custom code for each element

. WinControls provide functionality for user input, data display, navigation, and interaction, enabling developers to design intuitive and feature-rich desktop applications quickly and efficiently.

**Types of Windows Controls Following are some types of Windows Controls: -**

• Common Controls

• Containers

• Menu and Toolbars

• Components

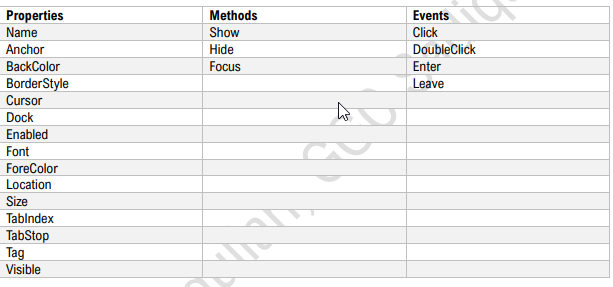
• Dialog

**Common Controls**

Common Controls refer to a set of pre-built user interface elements provided by the Windows Forms framework in C#, which are commonly used in desktop application development. These controls include essential elements such as buttons, textboxes, labels, checkboxes, radio buttons, listboxes, dropdown lists, and more.

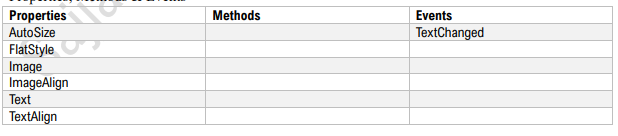
Common Controls provide the building blocks for creating intuitive and interactive user interfaces, offering functionalities for user input, data display, selection, and navigation. With their familiarity and versatility, Common Controls streamline the development process by allowing developers to incorporate standard UI elements into their applications, ensuring consistency and enhancing user experience across different Windows applications. Common Properties, Methods & Events

**There are some properties, methods and events that are common to many controls. These are listed as follows: -**



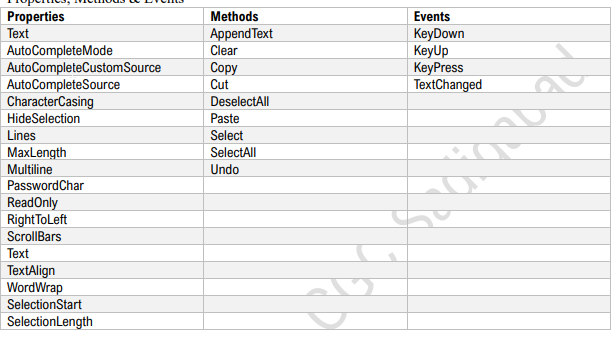
**Label Control**

A Label Control in Windows Forms is a user interface element used to display text or images that provide information or instructions to the user. It is typically used for static content and does not allow user input. Properties, Methods & Events



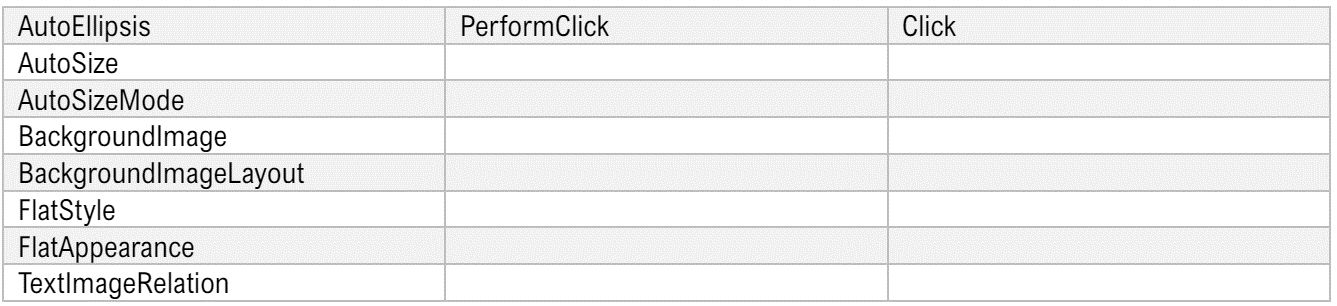
**TextBox Control**

A TextBox Control in Windows Forms is a user interface element used to accept and display single-line or multiline text input from the user. It allows users to enter text, numbers, or other characters and supports features like text selection, copying, and pasting.



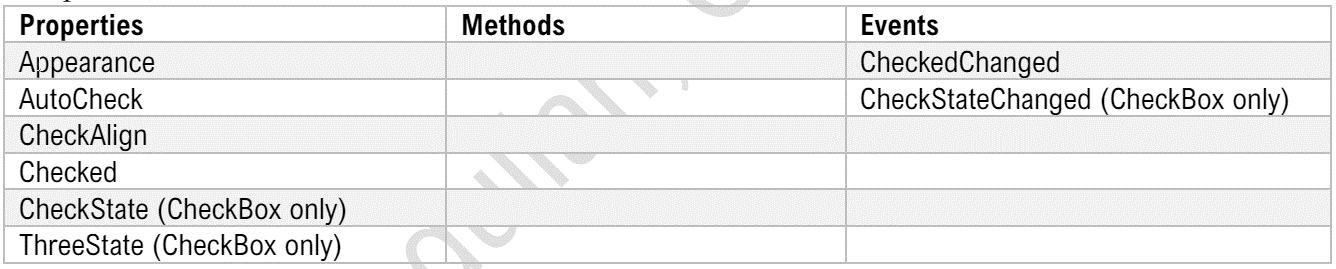
**Button Control**

A Button Control in Windows Forms is a user interface element used to trigger an action or event when clicked by the user. It typically displays text or an image and responds to user interaction by invoking a predefined event handler.



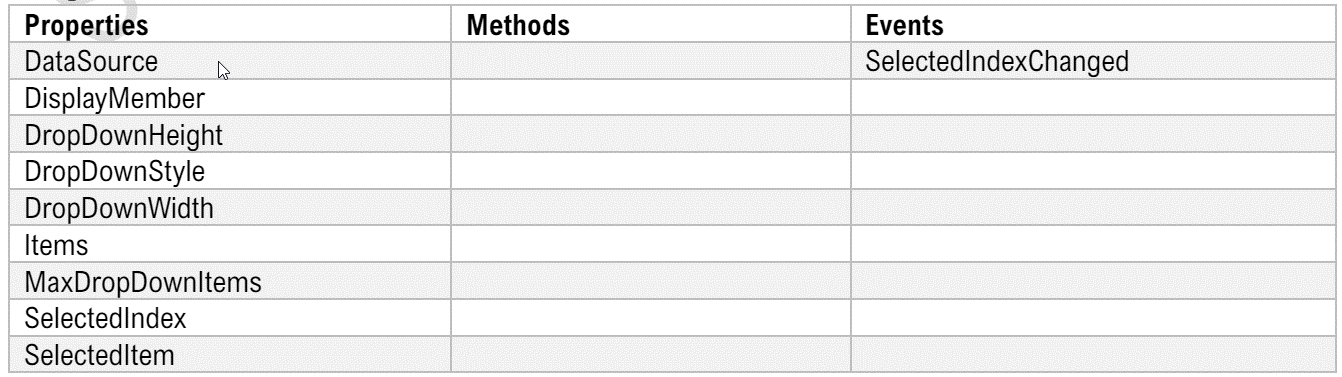
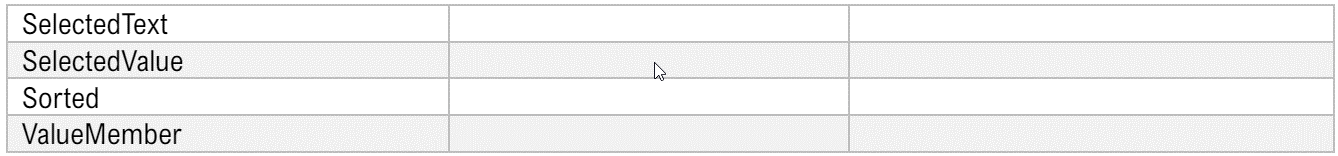
**CheckBox and RadioButton Controls**

A **CheckBox Control** in Windows Forms is a user interface element used to represent a binary state, typically denoting an option that can be selected or deselected by the user. It allows users to toggle between checked and unchecked states, providing a visual indication of the selection status. A **RadioButton Control** in Windows Forms is a user interface element used to present a set of mutually exclusive options from which the user can select only one. It allows users to make a single selection from a group of options by clicking on the radio button associated with their choice.



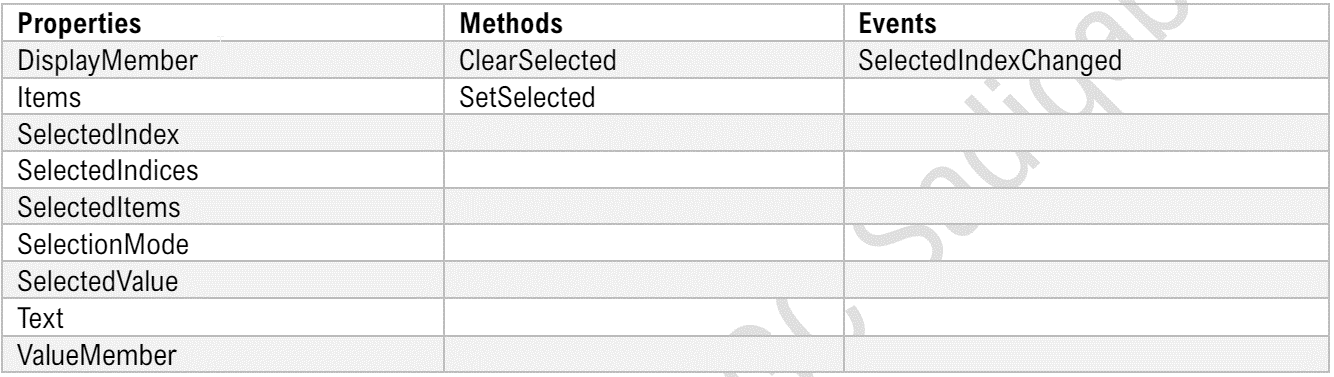
**ComboBox Control**

A ComboBox Control in Windows Forms is a user interface element used to present a list of items to the user in a dropdown menu, allowing them to select one option from the list. It combines the functionality of a TextBox with a ListBox, providing users with a convenient way to choose from a predefined set of options or enter custom values.



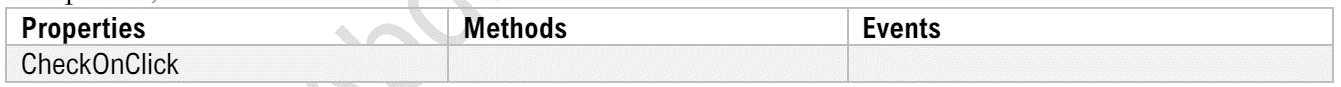
**ListBox Control**

A ListBox Control in Windows Forms is a user interface element used to display a list of items to the user, allowing them to select one or more items from the list. It presents items in a vertical list format and supports scrolling for viewing large sets of items, making it suitable for scenarios where users need to choose from a predefined set of options or view a list of data entries.



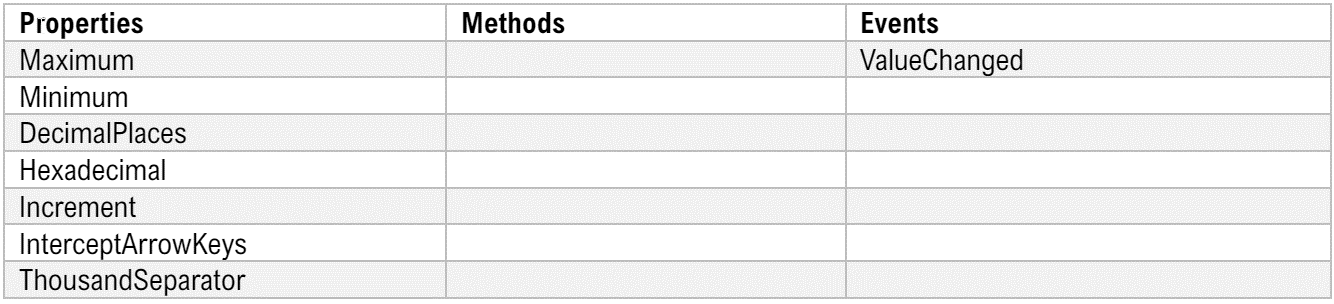
**CheckedListBox Control**

A CheckedListBox Control in Windows Forms is a user interface element that displays a list of items, each with a checkbox next to it, allowing users to select multiple items from the list. It combines the features of a ListBox and CheckBox, enabling users to make multiple selections easily by checking or unchecking the corresponding checkboxes.

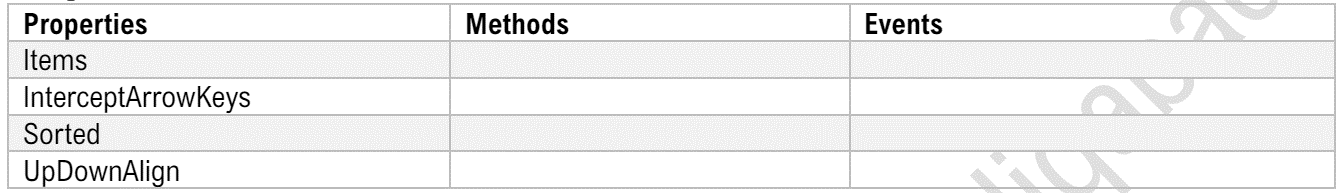


**NumericUpDown Control**

A NumericUpDown Control in Windows Forms is a user interface element used to input numeric values by allowing users to increase or decrease the value using up and down arrows or by typing directly into the control. It provides a convenient way to select numeric values within a predefined range, making it suitable for applications that require input of numerical data with specific constraints.

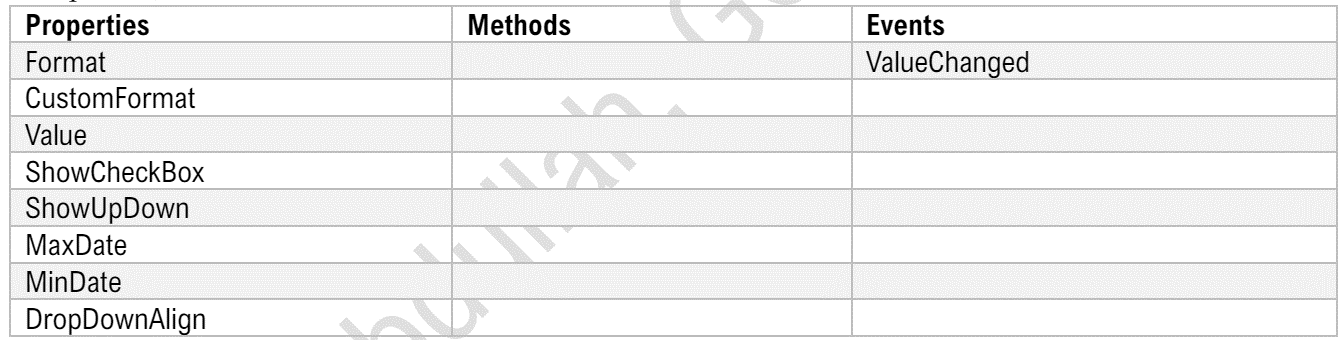


**DomainUpDown Control**

A DomainUpDown Control in Windows Forms is a user interface element used to select items from a predefined list by allowing users to scroll through the items in a cyclic manner. It combines the functionality of a TextBox and a NumericUpDown control, providing users with a way to input and select values from a finite set of options.

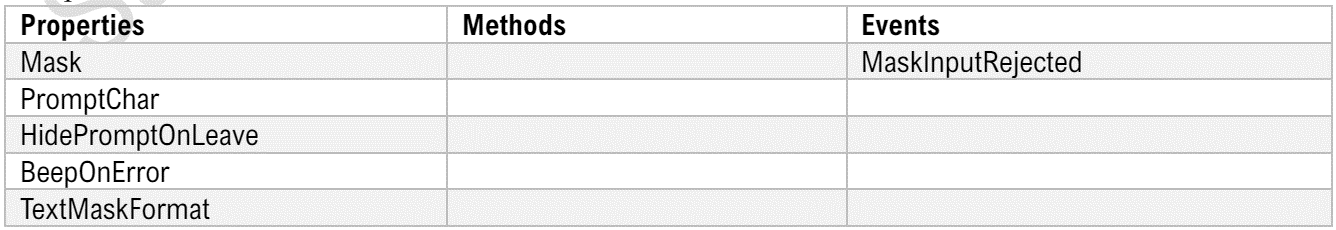
**DateTimePicker Control**

A DateTimePicker Control in Windows Forms is a user interface element used to select dates and times from a graphical calendar and clock interface. It allows users to easily input and manipulate date and time values, providing functionalities for selecting dates, times, or both, depending on the configuration.



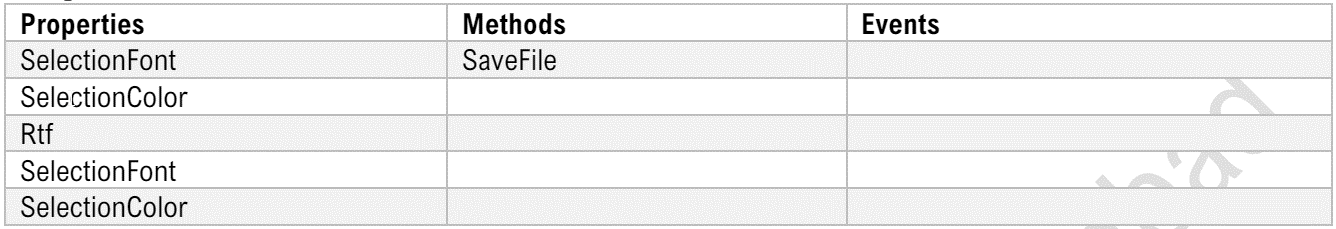
**MaskedTextBox Control**

A MaskedTextBox Control in Windows Forms is a user interface element that allows users to input text in a specific format by applying a mask, such as dates, phone numbers, or postal codes. This control ensures that the entered data conforms to the predefined format, enhancing data validation and consistency in user input.



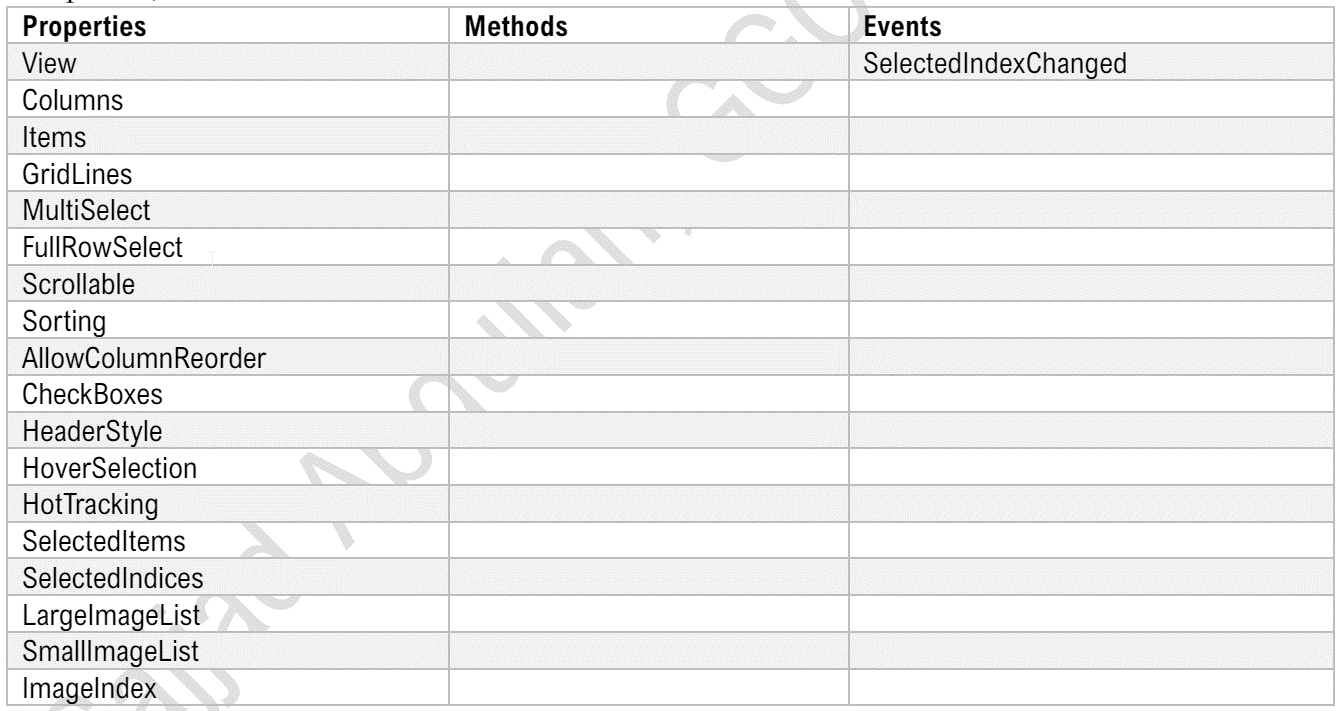
**RichTextBox Control**

A RichTextBox Control in Windows Forms is a user interface element that allows users to input and display formatted text with various fonts, colors, and styles. It provides advanced text editing capabilities, including support for bullets, numbering, hyperlinks, and embedded images, making it suitable for creating rich and visually appealing document-based interfaces.



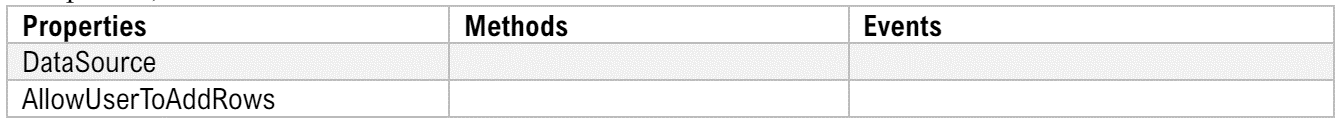
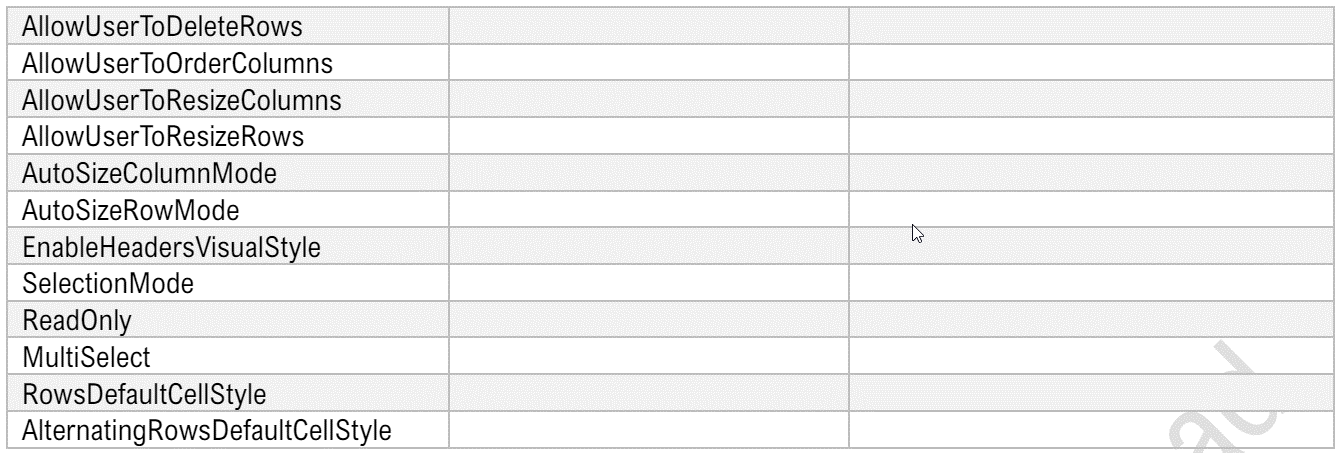
**ListView Control**

A ListView Control in Windows Forms is a user interface element used to display a collection of items in various views, such as details, icons, or lists. It supports features like sorting, grouping, and editing, making it suitable for presenting tabular data or file lists with interactive functionalities.



**DataGridView Control**

A DataGridView Control in Windows Forms is a user interface element used to display and edit tabular data in a grid format with rows and columns. It supports features like sorting, filtering, and cell formatting, making it suitable for presenting and manipulating large datasets in desktop applications.



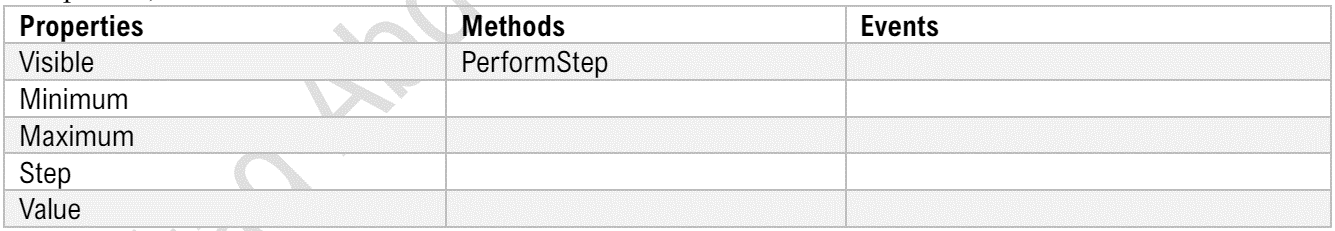
**TreeView Control**

A TreeView Control in Windows Forms is a user interface element used to display hierarchical data in a tree-like structure with parent and child nodes. It allows users to expand and collapse nodes to navigate through the hierarchy, making it suitable for representing folder structures, organization charts, or nested categories in desktop applications.



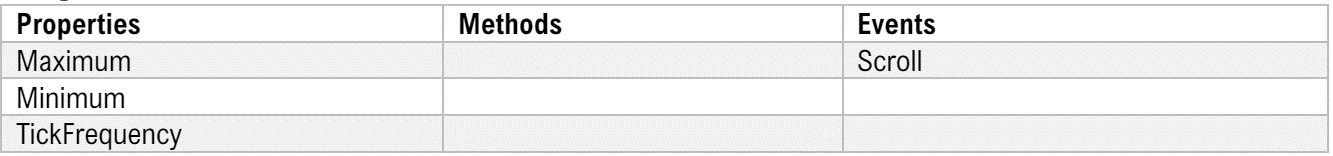
**ProgressBar Control**

A ProgressBar Control in Windows Forms is a visual indicator used to display the progress of a task, typically in terms of a percentage completed. It provides feedback to users about the status of an ongoing operation, such as file downloads, data processing, or installation processes, enhancing user experience and interaction in desktop applications.



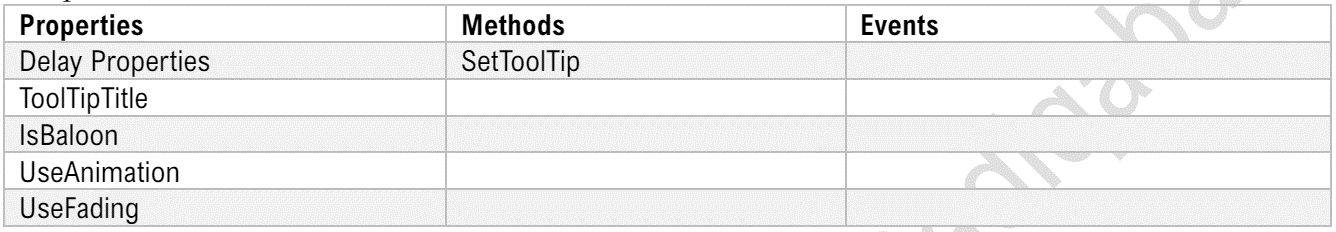
**TrackBar Control**

A TrackBar Control in Windows Forms is a user interface element used to allow users to select a value from a continuous range by sliding a thumb along a track. It provides a visual representation of the selected value and is commonly used for settings such as volume control, brightness adjustment, or scrolling through a timeline in desktop applications.



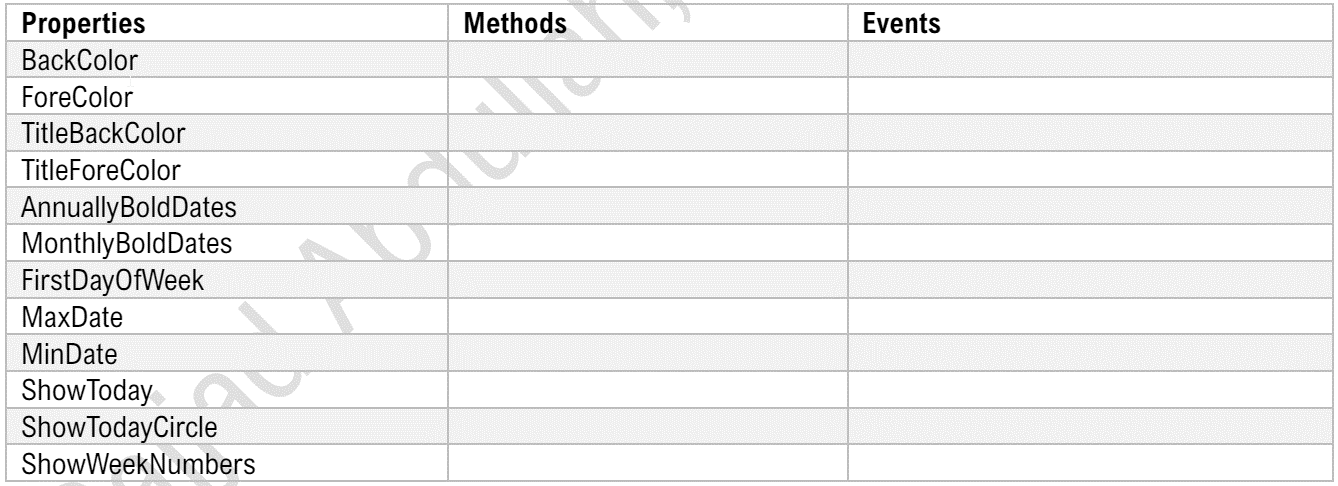
**ToolTip Control**

A ToolTip Control in Windows Forms is a user interface element used to display helpful information or hints when the user hovers over another control, such as a button or textbox. It provides contextual guidance and enhances the user experience by providing additional information about the purpose or functionality of the control.



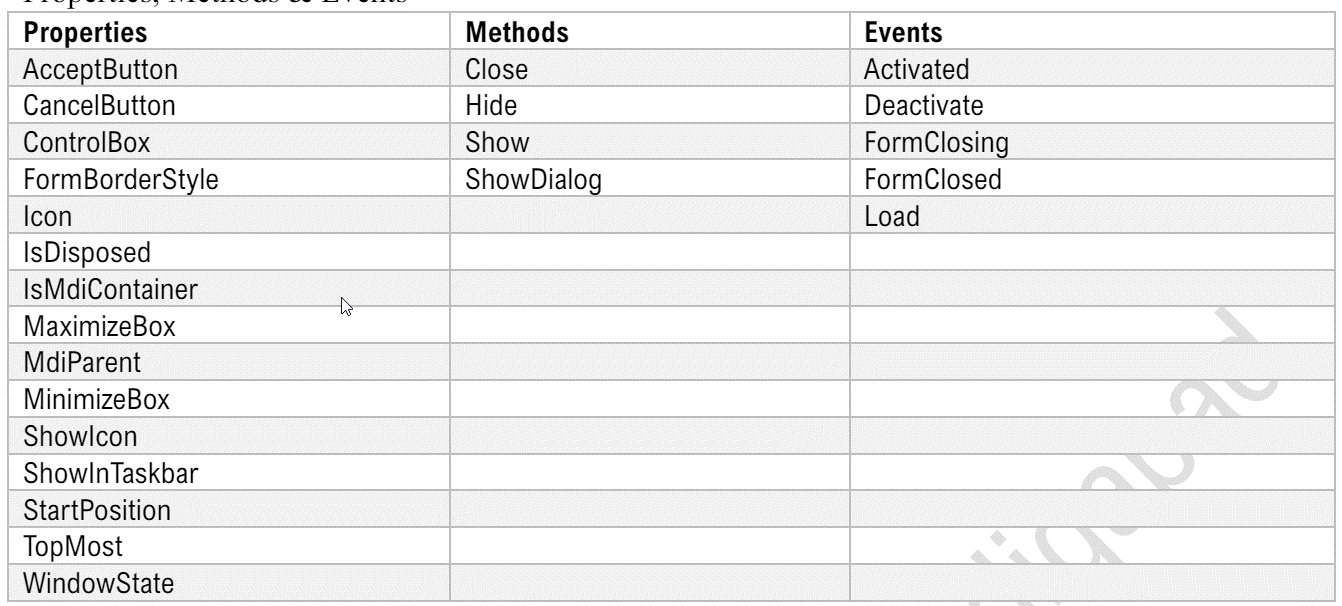
**MonthCalendar Control**

A MonthCalendar Control in Windows Forms is a user interface element used to display a graphical calendar that allows users to select dates or navigate through different months and years. It provides an intuitive way for users to interact with dates and schedule events, making it suitable for applications that require date selection or scheduling functionalities.



**Form Control**

A 'Form' control represents a window or dialog box in a graphical user interface (GUI) application. It serves as the primary container for other controls like buttons, textboxes, and labels, allowing users to interact with the application. Forms provide a visual representation of the application's interface, enabling developers to design and arrange various elements to create a functional and user-friendly experience. Through properties and events, developers can customize the appearance and behavior of forms to suit their application's requirements.



**Modal versus Non-Modal Forms**

Modal forms in C# are windows that require user interaction before allowing interaction with other windows in the application. They demand immediate attention and must be addressed before proceeding further.

In contrast, non-modal forms allow users to interact with other windows in the application while they are open, enabling multitasking within the application environment. They offer more flexibility and do not halt the user's workflow, unlike modal forms.

**MDI (Multiple Document Interface) Forms**

MDI (Multiple Document Interface) forms in C# are a type of form arrangement where a parent form acts as a container for multiple child forms. The parent form typically holds a menu and toolbar for controlling the child forms, which can be opened, closed, and arranged within the parent window. Child forms reside within the boundaries of the parent form and allow for efficient management and navigation of multiple documents or views within a single application window. Parent forms serve as the main frame of the application, while child forms represent individual documents or subviews contained within the parent window.

**MenuStrip Control**

The 'MenuStrip' control in C# is a graphical control element that provides a horizontal menu bar for organizing and accessing various commands and functions within a Windows Forms application. It typically contains menu items and submenus, which can be customized to suit the application's functionality. Developers can easily add, remove, or modify menu items through the control's properties and events.

The **'MenuStrip' control** enhances the user interface by offering a structured and intuitive way for users to navigate and interact with the application's features and commands.

**ContextMenuStrip Control**

The 'ContextMenuStrip' control in C# is a pop-up menu that appears in response to a right-click or contextclick event within a Windows Forms application. It provides a convenient way for developers to offer contextsensitive options or commands based on the user's interaction with specific controls or areas of the application. Developers can dynamically populate the menu with items tailored to the current context, such as actions relevant to selected items or the current application state.

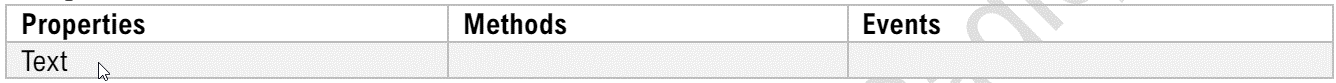
The **'ContextMenuStrip**' control enhances user experience by offering quick access to relevant commands at the point of interaction, streamlining workflow and improving usability.

**ContextMenuStrip Property**

The 'ContextMenuStrip' property in C# associates a 'ContextMenuStrip' control with another control, such as a form or a specific UI element like a textbox or a button. It enables the control to display a context menu when triggered by a right-click or context-click event. By assigning a 'ContextMenuStrip' to a control's 'ContextMenuStrip' property, developers can easily define and customize the options presented to users based on the context of their interaction with the control, enhancing the application's functionality and user experience.

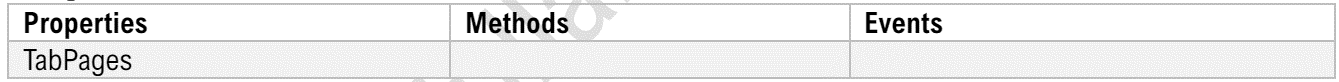
**GroupBox Control**

The 'GroupBox' control in C# is a visual container that groups related controls together within a Windows Forms application. It provides a visual boundary around the grouped controls, enhancing the organization and clarity of the user interface. Developers use the 'GroupBox' to logically group controls that share a common purpose or functionality, such as settings or options within a form. It helps improve the visual hierarchy of the interface and makes it easier for users to understand and interact with the grouped elements.



**TabControl Control**

The 'TabControl' control in C# organizes content into multiple tab pages, allowing users to switch between them within a single container. Each tab page can hold distinct content, such as forms, controls, or information, making it ideal for presenting different views or sections within an application. Developers can easily add, remove, or rearrange tab pages dynamically, providing a flexible and intuitive way to organize and navigate content. The 'TabControl' enhances the user experience by conserving screen space and simplifying the presentation of complex information or functionalities.



**Panel Control**

The 'Panel' control in C# is a container that holds other controls within a Windows Forms application. It acts as a grouping mechanism, allowing developers to organize and manage related controls together. Panels are versatile and can be used to create custom layouts, implement scrolling functionality, or group controls for organizational purposes. They provide a convenient way to manage the visual arrangement and behavior of controls, enhancing the overall structure and usability of the user interface.

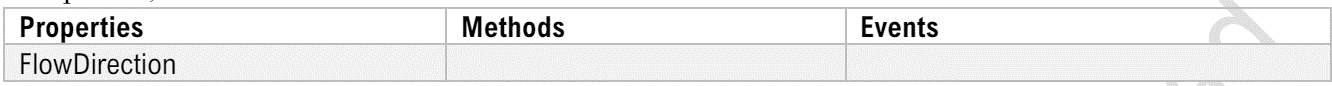


**TableLayoutPanel Control**

The 'TableLayoutPanel' control in C# is a layout control that arranges its child controls in a grid-like structure of rows and columns. It provides a flexible and efficient way to create complex layouts with precise control over the positioning and sizing of controls. Developers can easily add, remove, or resize rows and columns dynamically, making it ideal for designing responsive and adaptable user interfaces. The 'TableLayoutPanel' simplifies the management of controls within a form, improving code readability and maintenance.

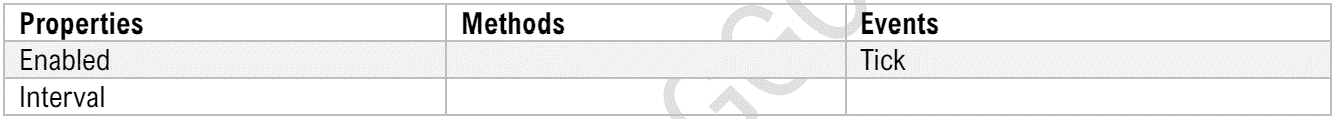
**FlowLayoutPanel Control**

The 'FlowLayoutPanel' control in C# arranges its child controls in a flowing layout, either horizontally or vertically, based on specified settings. It dynamically adjusts the positioning of controls as the container resizes, ensuring optimal use of available space. Developers can easily add, remove, or reorder controls within the 'FlowLayoutPanel,' making it ideal for creating resizable and flexible user interfaces. This control simplifies the design process by automatically managing the layout of controls, improving the application's responsiveness and usability.



**Timer Control**

The 'Timer' control in C# enables developers to execute code at regular intervals within a Windows Forms application. It provides a mechanism for performing tasks such as updating the UI, refreshing data, or triggering events periodically. Developers can set the interval for the timer and define actions to be performed each time it elapses. The 'Timer' control enhances application functionality by automating repetitive tasks and ensuring timely updates, improving user experience and efficiency.



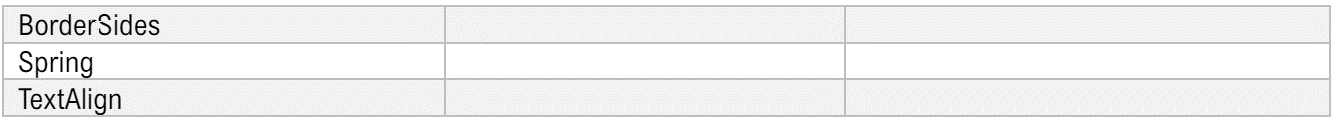
**ToolStrip Control**

The 'ToolStrip' control in C# is a customizable toolbar that provides quick access to frequently used commands and actions within a Windows Forms application. It typically contains buttons, dropdowns, and other controls for invoking various functionalities. Developers can easily add, remove, or rearrange items in the toolbar, making it adaptable to different application needs. The 'ToolStrip' enhances user productivity by offering a convenient and organized way to access application features, improving overall usability and efficiency.



**StatusStrip Control**

The 'StatusStrip' control in C# is a horizontal bar typically located at the bottom of a window, displaying information about the application's status or providing context-specific feedback. It commonly contains labels, progress bars, and other controls to convey information to the user. Developers can dynamically update the content of the 'StatusStrip' to reflect changes in the application state or provide feedback on ongoing operations. The 'StatusStrip' enhances user experience by offering real-time updates and contextual information, improving usability and user engagement.

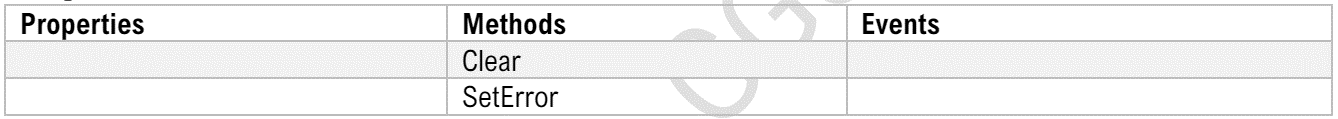


**ToolStripContainer Control**

The 'ToolStripContainer' control in C# is a container that combines a ToolStrip, ToolStripPanel, and optionally, a ContentPanel, providing a customizable framework for building complex user interfaces. It enables developers to organize and manage toolbars and other controls effectively within a Windows Forms application. The 'ToolStripContainer' simplifies the arrangement of toolbars and content panels, allowing for flexible customization of the user interface layout. It enhances user productivity by offering a cohesive and organized workspace for accessing application features and content.

**ErrorProvider Control**

The 'ErrorProvider' control in C# is a component that helps developers manage and display validation errors within Windows Forms applications. It visually alerts users to input errors by displaying an icon next to the control with invalid data and a tooltip providing details about the error. Developers can easily associate the 'ErrorProvider' with input controls and dynamically set error messages, simplifying the validation process. This control enhances the user experience by providing immediate feedback on erroneous input, improving data accuracy and usability.



---------------------------------------------------------------------------------------------------------------------------

**Chapter 5**

**Database Programming**

**Introduction**

ADO.NET ADO.NET (ActiveX Data Objects for .NET) is a set of classes in the .NET Framework designed for data access. It provides a bridge between the front-end user interface of an application and the back-end database. ADO.NET is an integral part of the .NET Framework that enables developers to interact with data sources, such as databases and XML files, in a consistent and efficient manner.

**Key Components of ADO.NET**

1. Data Providers:

• SQL Server Data Provider: `System.Data.SqlClient` for SQL Server.

• OLE DB Data Provider: `System.Data.OleDb` for OLE DB-enabled data sources.

• ODBC Data Provider: `System.Data.Odbc` for ODBC-enabled data sources.

• Oracle Data Provider: `System.Data.OracleClient` for Oracle databases (deprecated in later versions of .NET).

1. Connection Object:

• Establishes a connection to the data source

. • Example: `SqlConnection` for SQL Server.

1. Command Object:

• Executes SQL commands against the database.

• Example: `SqlCommand` for SQL Server.

1. DataReader Object:

• Provides a forward-only, read-only stream of data from the database.

• Example: `SqlDataReader` for SQL Server.

1. DataAdapter Object

: • Serves as a bridge between a DataSet and the data source for retrieving and saving data.

• Example: `SqlDataAdapter` for SQL Server.

DataSet Object:

1. • An in-memory representation of data that can hold multiple tables and relationships.

• It can be used to manipulate data offline and synchronize changes with the database.

1. DataTable Object:

• Represents a single table of in-memory data

• It can be used independently or as part of a DataSet.

1. DataView Object:

• Provides a customized view of a DataTable for sorting, filtering, and searching.

**Introduction to LINQ**

LINQ (Language Integrated Query) is a powerful feature in C# that allows developers to write concise, readable, and expressive code to query and manipulate data. LINQ integrates query capabilities directly into the C# language, enabling queries to be written against various data sources such as arrays, collections, XML, databases, and more. This integration brings the querying capabilities into the language syntax, making data manipulation easier and more intuitive.

**Key Benefits of LINQ**

1. **Readability:** LINQ queries are often easier to read and understand compared to traditional loops and conditionals
2. . **Type Safety:** LINQ provides compile-time checking of query expressions, reducing runtime errors.
3. **Flexibility:** LINQ can query different data sources using a consistent syntax.

**Key Components of LINQ**

**Standard Query Operators**

• A set of methods that provide query capabilities similar to SQL (e.g., `Select`, `Where`, `OrderBy`, `GroupBy`, `Join`).

• These methods can be called as extension methods on any `IEnumerable` or `IQueryable` collection**.**

**Query Expressions**

A more readable syntax for writing LINQ queries, resembling SQL-like syntax but integrated into C#.

var results = from item in collection

where item.

Property == value select item;

**Examples LINQ Query Syntax**

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

static void Main()

{

List numbers = new List

{ 1, 2, 3, 4, 5 };

// LINQ Query Syntax var evenNumbers = from n in numbers where n % 2 == 0 select n; Console.WriteLine("Even numbers using LINQ Query Syntax:");

foreach (var number in evenNumbers)

{ Console.WriteLine(number);

}

}

}

**Lambda Expressions**

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

static void Main()

{

List numbers = new List { 1, 2, 3, 4, 5

};

// Lambda Expressions var evenNumbers = numbers.Where(n => n % 2 == 0); Console.WriteLine("Even numbers using Lambda Expressions:");

foreach (var number in evenNumbers) { Console.WriteLine(number);

}

}

}

**Standard Query Operators**

LINQ (Language Integrated Query) in C# provides a comprehensive set of standard query operators that enable data manipulation and querying. These operators are implemented as extension methods on the `IEnumerable` and `IQueryable` interfaces. Here’s a categorized list of these operators: Filtering Operators

• Where: Filters elements based on a predicate.

• OfType: Filters elements based on a specified type.

• Take:

• TakeWhile:

• Skip:

• SkipWhile:

**Projection Operators**

• Select: Projects each element of a sequence into a new form.

• SelectMany: Projects each element of a sequence to an `IEnumerable` and flattens the resulting sequences into one sequence.

**Sorting Operators**

• OrderBy: Sorts elements in ascending order.

• OrderByDescending: Sorts elements in descending order.

• ThenBy: Performs a subsequent ordering of elements in ascending order

. • ThenByDescending: Performs a subsequent ordering of elements in descending order.

• Reverse: Reverses the order of elements in a sequence.

**Grouping Operators**

• GroupBy: Groups elements that share a common attribute.

**Join Operators**

• Join: Joins two sequences based on a key selector function.

• GroupJoin: Performs a grouped join on two sequences based on keys.

**Set Operators**

• Distinct: Removes duplicate elements from a sequence.

• Union: Produces the set union of two sequences.

• Intersect: Produces the set intersection of two sequences.

• Except: Produces the set difference of two sequences. Quantifiers

• All: Determines whether all elements of a sequence satisfy a condition

. • Any: Determines whether any element of a sequence satisfies a condition

. • Contains: Determines whether a sequence contains a specified element.

**Aggregate Operators**

• Aggregate: Applies an accumulator function over a sequence

. • Average: Computes the average of a sequence of numeric values.

• Count: Counts the elements in a sequence.

• LongCount: Counts the elements in a sequence, returning an `Int64`

. • Max: Finds the maximum value in a sequence

. • Min: Finds the minimum value in a sequence.

• Sum: Computes the sum of a sequence of numeric values.

**Element Operators**

• ElementAt: Returns the element at a specified index in a sequence

. • ElementAtOrDefault: Returns the element at a specified index or a default value if the index is out of range.

• First: Returns the first element in a sequence.

• FirstOrDefault: Returns the first element in a sequence, or a default value if no element is found.

• Last: Returns the last element in a sequence.

• LastOrDefault: Returns the last element in a sequence, or a default value if no element is found.

• Single: Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.

• SingleOrDefault: Returns the only element of a sequence, or a default value if the sequence is empty; throws an exception if there is more than one element in the sequence.

**Generation Operators**

• DefaultIfEmpty: Returns the elements of a sequence, or a default value in a singleton collection if the sequence is empty.

• Empty: Returns an empty `IEnumerable`.

• Range: Generates a sequence of integral numbers within a specified range.

• Repeat: Generates a sequence that contains one repeated value**.**

**Conversion Operators**

• AsEnumerable: Returns the input typed as `IEnumerable`.

• AsQueryable: Converts a generic `IEnumerable` to an `IQueryable`.

• ToArray: Converts a sequence to an array

. • ToDictionary: Converts a sequence to a dictionary

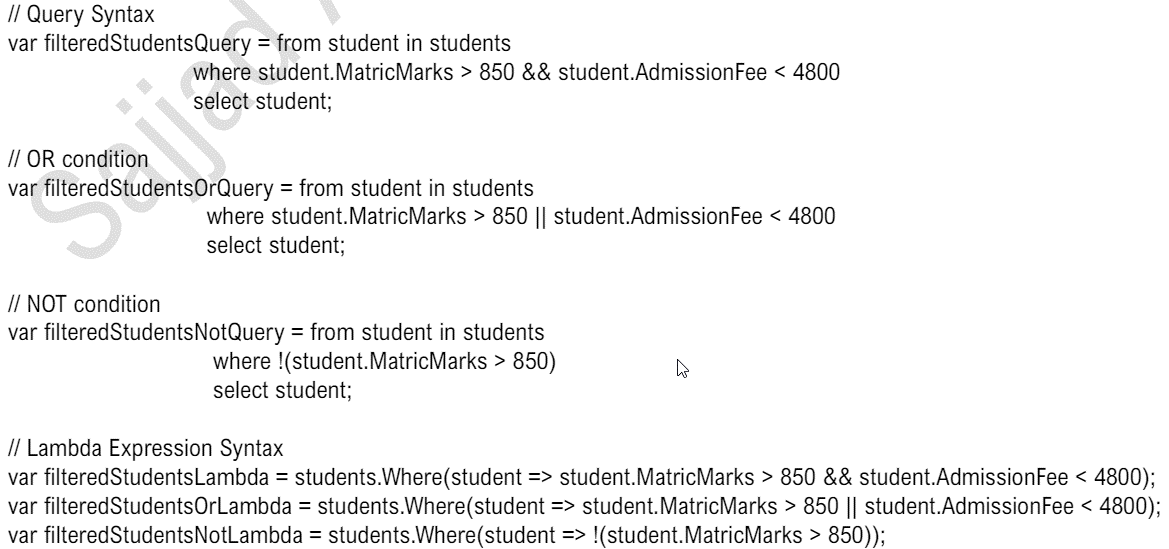
. • ToList: Converts a sequence to a `List`.

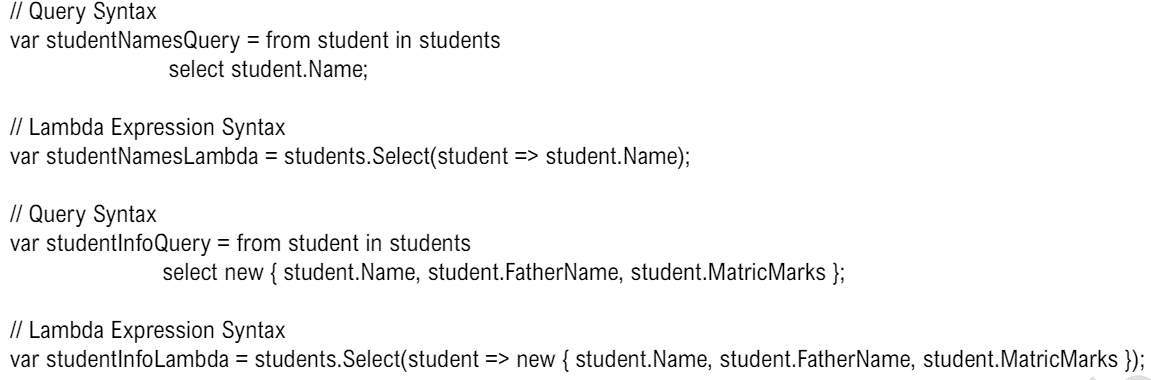
• ToLookup: Converts a sequence to a `Lookup`. Zip Operator

• Zip: Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results. Equality Operators

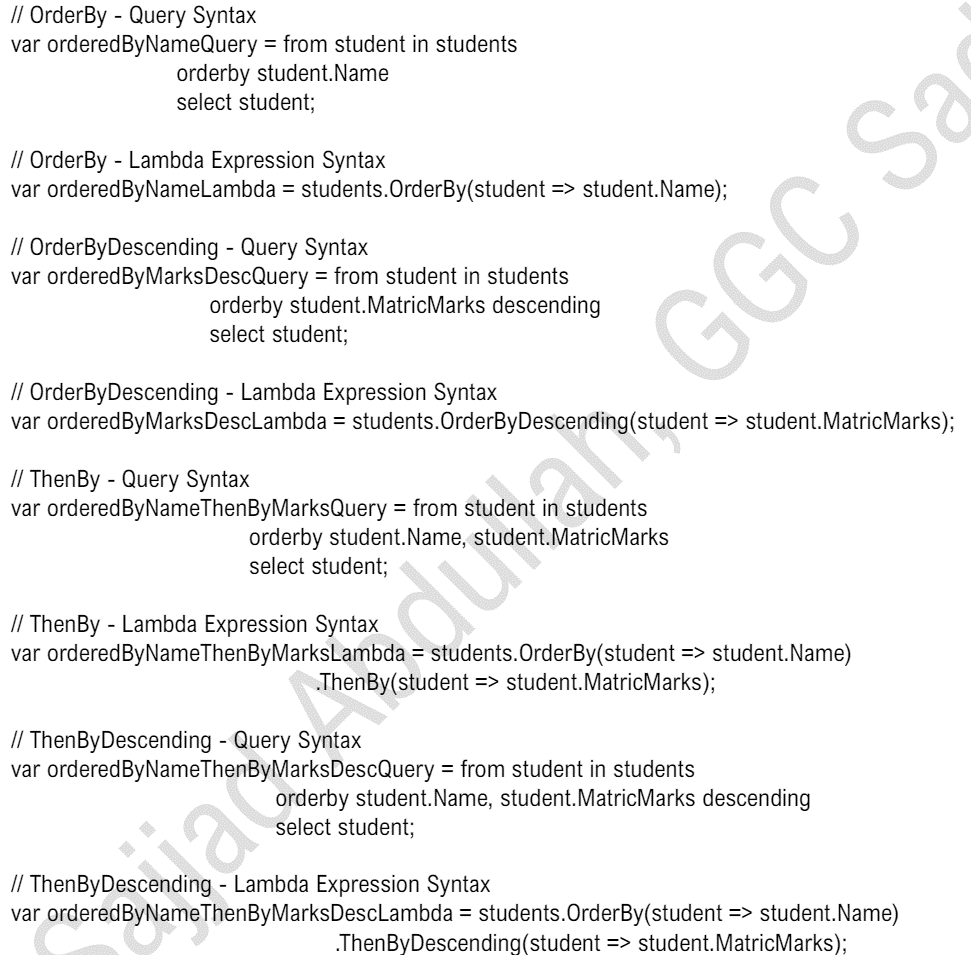
• SequenceEqual: Determines whether two sequences are equal by comparing their elements.

**Some Basic Examples of LINQ Query and Lambda Expression Use of Where Operator**

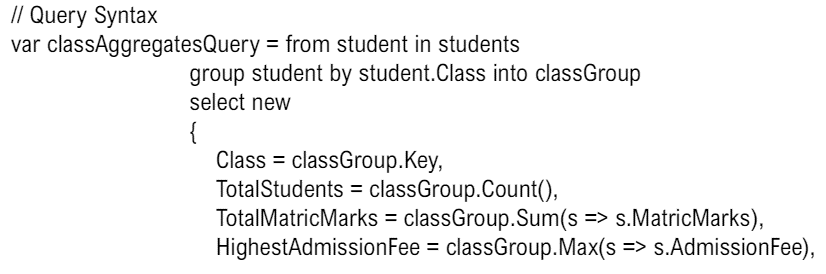


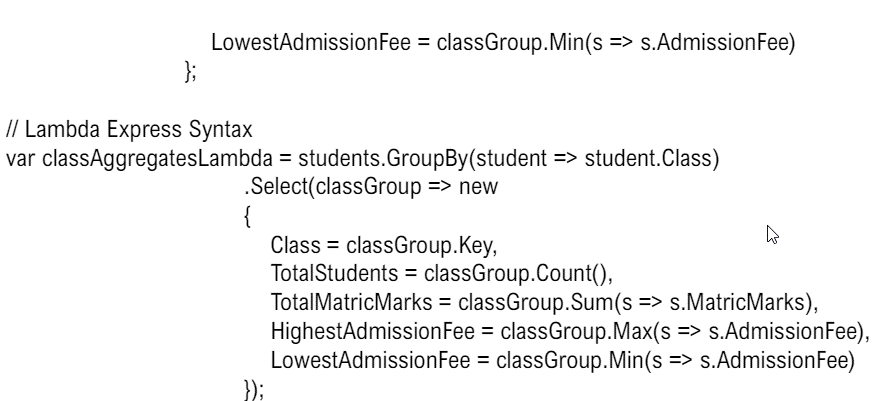
**Use of Select Operator**

**Use of OrderBy and ThenBy Operators**



**Use of ‘GroupBy’ Operator**





**XML Files**

XML (eXtensible Markup Language) files are a type of file format that is both human-readable and machinereadable. They are used to store and transport data in a structured format through a series of nested tags. XML files are often used for configuration files, data exchange between systems, and storage of structured data.

**Structure of an XML File**

An XML (eXtensible Markup Language) file is a structured format for storing data. It consists of elements and attributes organized in a hierarchical manner. Here is a breakdown of the structure of a typical XML file:

**XML Declaration: -**

1. The XML declaration is optional but recommended.
2. - It specifies the version of XML and the encoding used.
3. - It appears at the very top of the file.

**Root Element:**

• The root element is the top-level element that contains all other elements in the XML file

• There must be one and only one root element in an XML document.

**Child Elements:**

Child elements are nested within the root element. They can contain data, other child elements, and attributes.

**Attributes:**

• Attributes provide additional information about elements.

• They are defined within the start tag of an element.

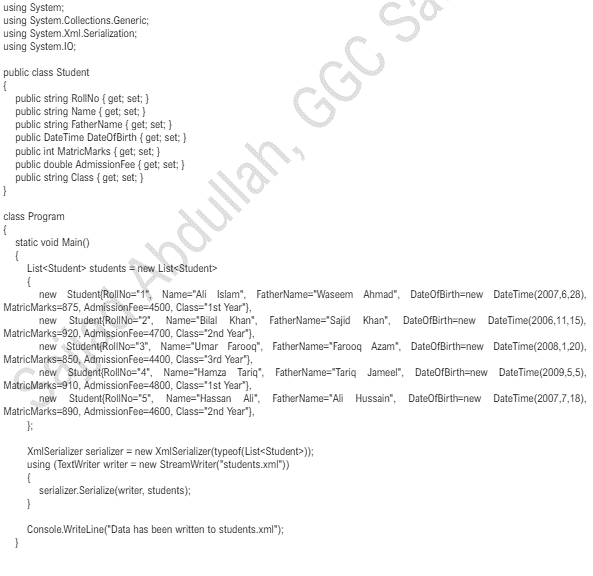
**Element Content:**

• Elements can contain text, other elements, or be empty.

**Program to Write and Read from XML Files in C#**

Here is a complete example program in C# that demonstrates how to write data to an XML file and then read data from an XML file using the `System.Xml` namespace.

Writing to an XML File



-------------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------