**Visual Studio 2022 Overview**

**Visual Studio 2022** is a powerful Integrated Development Environment (IDE) developed by Microsoft. It is widely used for building modern applications across platforms such as Windows, web, mobile, and cloud. Here's what makes Visual Studio 2022 unique:

**Key Features:**

1. **64-bit Architecture**:

* Enhanced performance for larger projects.
* Smooth handling of complex solutions without memory limitations.

1. **Improved Code Editor**:

* **IntelliCode**: AI-assisted code suggestions.

1. **Enhanced Debugging Tools**:

* Real-time debugging with better visualizations.

1. **Built-in Git Integration**:

* Seamless management of Git repositories.
* Features for version control, pull requests, and conflict resolution.

1. **Cross-platform Development**:

* Develop applications for Windows, macOS, Android, iOS, and cloud platforms.
* Integrated tools for frameworks like .NET 6, .NET Core.

1. **Collaboration Features**:

* **Live Share**: Real-time code collaboration.
* Tools for remote teamwork and pair programming.

**System Requirements:**

* **Operating System**: Windows 10/11.
* **RAM**: Minimum 4 GB (8 GB or more recommended).
* **Processor**: 1.8 GHz or faster.
* **Disk Space**: Minimum 20 GB free.

**Use Cases:**

* Web and desktop applications using **.NET**.
* Mobile app development.
* Cloud-native apps for Azure.

**Why Visual Studio 2022?**

* Optimized for developer productivity with modern tools and features.
* Ideal for beginners and professionals working on .NET, C#, or other Microsoft technologies.

**Visual Studio and VS Code**

Visual Studio and Visual Studio Code (VS Code) are different tools for software development, with Visual Studio being a full-featured IDE and VS Code being a lightweight code editor:

* **Purpose**

Visual Studio is designed for large-scale software development, while VS Code is better for smaller projects and quick edits.

* **Features**

Visual Studio is a comprehensive IDE with a full set of tools for software development. VS Code is a code editor that's more flexible and customizable, and relies on extensions to support other languages.

* **Language support**

Visual Studio has native support for many programming languages, including C#, .NET, C, C++, Python, HTML, CSS, and JavaScript. VS Code Supports almost all programming languages via extensions.

* **System resources**

VS Code is lighter on system resources than Visual Studio, so it can run on a machine with as little as 300 MB of RAM

**Introduction to C#**

**C#** (pronounced "C-sharp") is a modern, object-oriented programming language developed by Microsoft. It is part of the .NET ecosystem and is widely used for developing a variety of applications, including web, desktop, mobile, gaming, and cloud-based solutions.

**Key Features of C#:**

1. **Object-Oriented**:

* Emphasizes objects and classes for modular, reusable code.

1. **Type-Safe**:

* Prevents operations that could lead to runtime errors by enforcing strict type-checking.

1. **Rich Library Support**:

* Access to extensive libraries within the .NET framework for faster development.

1. **Cross-Platform Development**:

* With .NET Core, C# enables development for Windows, macOS, and Linux.

1. **Automatic Garbage Collection**:

* Frees up unused memory to improve performance.

1. **Modern Language Features**:

* Supports async/await, LINQ (Language Integrated Query), pattern matching, and more.

**Why Use C#?**

* **Developer-Friendly**: Syntax is clean and easy to learn for beginners and professionals alike.
* **Microsoft Ecosystem**: Seamlessly integrates with Microsoft tools like Visual Studio and Azure.
* **Widely Used**: Ideal for building robust web applications and games (via Unity engine).

**Common Use Cases:**

1. Web Applications
2. Desktop Applications
3. Mobile Applications
4. Game Development
5. Cloud-Based Solutions

**Create first C# program ‘Hello world’**

Code :

using System;

namespace Basic

{

class Program

{

static void Main()

{

Console.WriteLine("HelloWorld");

}

}

}

**Understanding C# Program Structure**

A basic C# program is made up of several components that work together to execute the program. Below is an explanation of the common structure of a C# program using an example:

using System; // Importing the System namespace

namespace Basic // Declaring a namespace called 'Basic'

{

class Program // Defining a class named 'Program'

{

static void Main() // The Main method, the program's entry point

{

Console.WriteLine("HelloWorld"); // Printing "HelloWorld" to the console

}

}

}

**Components :**

1. **using System;**

* This directive imports the System namespace.
* It gives access to fundamental classes like Console, String, etc.

**Example:** Without using System, you would need to write System.Console.WriteLine.

1. **namespace Basic**

* A namespace groups related classes and prevents name conflicts.
* Here, the namespace is named Basic.

1. **class Program**

* Defines a class named Program.
* Classes are containers for methods, properties, and fields.

1. **static void Main()**

* The Main method is the entry point for any C# program.
* static: The method belongs to the class, not an instance.
* void: It doesn't return any value.
* Empty parentheses (): Indicates no arguments are passed.

1. **Console.WriteLine("HelloWorld");**

* This line prints "HelloWorld" to the console.
* Console: A class from the System namespace.
* WriteLine(): A method used to display output and move to the next line.

**Program Execution Flow**

* The compiler looks for the Main method to start execution.
* The statements inside the Main method are executed sequentially.
* Outputs or errors are displayed in the console.

**Working with Code Files, Projects, and Solutions**

In **C# development with Visual Studio**, code is organized into files, projects, and solutions. Understanding their relationship is crucial for managing and building applications.

**1. Code Files**

* A **code file** contains the actual C# source code that defines the logic of your application.
* File extension: **.cs** (e.g., Program.cs).
* Each code file usually contains:
  + Namespace declaration.
  + Class definition.
  + Methods and other members.

**2. Projects**

* A **project** is a collection of code files, resources, and settings needed to build a single application or component.
* File extension: **.csproj** (C# Project file).
* Types of projects:
  + **Console Application**: A command-line app.
  + **Class Library**: A reusable library of code.
  + **Web Application**: For websites or web APIs.

Key Features of a Project:

1. **Project File**:
   * Defines the structure and dependencies of the project.
2. **Build Output**:
   * Generates an executable file (.exe) or a library (.dll).

**3. Solutions**

* A **solution** is a container for one or more projects.
* File extension: **.sln** (Solution file).
* Solutions help organize multiple projects into a single entity.

Example: A web application with a backend project and a frontend project in the same solution.

Key Benefits:

1. **Multiple Projects**:

* Manage projects that share resources or interact (e.g., APIs and databases).

1. **Dependency Management**:

* Control how projects depend on each other.

**Working in Visual Studio**

**Creating a Solution and Project**

1. Open Visual Studio and select **Create a new project**.
2. Choose a project type (e.g., Console App, Web App).
3. Name your project and solution.
4. Visual Studio will generate a solution file containing the project.

**Adding Code Files**

1. Right-click on the project in **Solution Explorer**.
2. Select **Add > New Item**.
3. Choose file type.

**Running and Building Projects**

* Press **F5** to run the project. (Visual Studio compiles the project into an executable or library.

**Datatypes, Variables, and Conversion in C#**

**1. Variables in C#**

* A **variable** is a named storage location in memory used to store data.
* **Syntax** to declare a variable : datatype variableName = value;
* **Example**: int age = 20;

**2. Datatypes in C#**

Datatypes define the type of data a variable can hold. They are broadly categorized into:

**Value Types (Stores data directly)**

* **Numeric**:
  + int (4 bytes): Whole numbers. Example: int x = 10;
  + float (4 bytes): Single-precision decimal. Example: float f = 3.14f;
  + double (8 bytes): Double-precision decimal. Example: double d = 3.14159;
  + decimal (16 bytes): High precision for financial data. Example: decimal money = 99.99m;
  + byte (1 byte): 0 to 255. Example: byte b = 255;
  + long (8 bytes): Larger integers. Example: long bigNum = 9000000000;
* **Boolean**:
  + bool: Holds true or false. Example: bool isValid = true;
* **Character**:
  + char: A single Unicode character. Example: char letter = 'A';

**Reference Types (Stores references to data)**

* **String**:
  + string: Sequence of characters. Example: string name = "Keyur";
* **Object**:
  + object: Base type for all data types. Example: object obj = 42;

**3. Type Conversion**

Conversion is needed when assigning a value of one datatype to another. C# supports **implicit** and **explicit** conversions.

**Implicit Conversion**

* Automatically performed when there's no data loss. converting a smaller type to a larger type size : char -> int -> long -> float -> double

Example :

int num = 10;

double result = num; // int to double

**Explicit Conversion (Casting)**

* Requires casting when there is a possibility of data loss.

Example :

double price = 99.99;

int discountedPrice = (int)price; // Converts double to int

**Using Built-In Methods**

**1 Convert Class:**

* Provides methods to convert between types.

Example**:**

string str = "123";

int num = Convert.ToInt32(str); // Converts string to int

**2 Parsing**:

* Converts strings to specific types.

Example:

string str = "3.14";

float num = float.Parse(str); // Converts string to float

**3 TryParse**:

* Safer alternative to Parse. Avoids exceptions on invalid input.

Example:

string str = "456";

if (int.TryParse(str, out int result))

{

Console.WriteLine(result); // Outputs 456

}

**Operators & Expressions**

**Operators**

**1. Arithmetic Operators**

* Perform mathematical operations.
* Examples:
  + + (Addition): int result = a + b;
  + - (Subtraction): int result = a - b;
  + \* (Multiplication): int result = a \* b;
  + / (Division): int result = a / b;
  + % (Modulus): int remainder = a % b;

**2. Relational Operators**

* Compare values and return a boolean result.
* Examples:
  + == (Equal): a == b
  + != (Not Equal): a != b
  + > (Greater Than): a > b
  + < (Less Than): a < b

**3. Logical Operators**

* Combine conditional expressions.
* Examples:
  + && (AND): a > b && c > d
  + || (OR): a > b || c > d
  + ! (NOT): !isTrue

**4. Assignment Operators**

* Assign values to variables.
* Examples:
  + =: a = 10;
  + +=: a += 5; (Equivalent to a = a + 5).

**5. Increment and Decrement Operators**

* Increase or decrease a value by 1.
* Examples:
  + ++a (Pre-Increment)
  + a++ (Post-Increment)
  + --a (Pre-Decrement)
  + a-- (Post-Decrement)

**6. Bitwise Operators**

* Operate at the bit level.
* Examples:
  + & (AND): a & b
  + | (OR): a | b
  + ^ (XOR): a ^ b

**Expressions**

* **Definition**: A combination of variables, operators, and values that produce a result.
* Examples:
  + Arithmetic Expression:

int result = (a + b) \* c;

* + Logical Expression:

bool isValid = (a > b) && (c < d);

**Operator Precedence**

Operators have different levels of precedence, determining the order in which they are evaluated. For example, multiplication (\*) has higher precedence than addition (+).

1. Parentheses ()
2. Multiplication, Division, Modulus \* / %
3. Addition, Subtraction + -
4. Relational & Logical Operators == != > < && ||

You can control the order of evaluation by using parentheses ().

**Statements**

A **statement** in C# is a single action or command that the program executes. They are the building blocks of C# programs.

**1. Types of Statements**

**A. Declaration Statement**

* Declares variables or constants.

int age = 25;

**B. Expression Statement**

* Performs an operation.

int sum = 5 + 3;

**C. Assignment Statement**

* Assigns values to variables.

int result = 10;

result = result + 5; // result = 15

**2. Control Flow Statements**

**A. Conditional Statements**

* **if**: Executes if the condition is true.

if (age >= 18) { Console.WriteLine("Adult"); }

* **if-else**: Executes one block if true, another if false.

if (age >= 18) { Console.WriteLine("Adult"); }

else { Console.WriteLine("Minor"); }

* **switch**: Checks multiple conditions.

switch (day) { case 1: Console.WriteLine("Monday"); break; }

**3. Looping Statements**

* **for**: Repeats a block for a set number of iterations.

for (int i = 0; i < 5; i++) { Console.WriteLine(i); }

* **while**: Repeats a block while a condition is true.

while (i < 5) { Console.WriteLine(i); i++; }

* **do-while**: Executes a block once, then repeats while true.

do { Console.WriteLine(i); i++; } while (i < 5);

**4. Jump Statements**

* **break**: Exits a loop or switch.

for (int i = 0; i < 5; i++) { if (i == 3) break; }

* **continue**: Skips to the next loop iteration.

for (int i = 0; i < 5; i++) { if (i == 2) continue; }

* **return**: Exits a method and returns a value.

return 10;

**Understanding Arrays**

An **array** in C# is a collection of items, all of the same data type, stored in contiguous memory locations. It allows you to manage multiple values efficiently under one variable.

**Key Points about Arrays**

* Arrays are **fixed in size** and their size is defined during initialization.
* Array elements are accessed using an **index**, which starts from 0.
* Arrays can store **primitive types** (like int, float) and **objects** (like string, classes).

**Declaring and Initializing Arrays**

**A. Declaration**

* Specify the data type, followed by square brackets [] and the array name.

int[] numbers;

**B. Initialization**

1. **With Size:**

int[] numbers = new int[5]; // Array of 5 integers

1. **With Values:**

int[] numbers = {1, 2, 3, 4, 5}; // Predefined values

1. **Combination of Declaration and Initialization:**

string[] names = new string[] {"Keyur", "Hit", "Raj"};

**Accessing Array Elements**

* Access elements using their index (0-based).

Console.WriteLine(numbers[0]); // Access the first element

* Modify elements by assigning new values.

numbers[1] = 10; // Assign 10 to the second element

**Looping Through an Array**

* Use loops to iterate over array elements.

1. **Using a for Loop:**

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

{

Console.WriteLine(numbers[i]);

}

1. **Using a foreach Loop:**

foreach (int num in numbers)

{

Console.WriteLine(num);

}

**Types of Arrays**

1. **Single-Dimensional Array:**
   * A basic array with one row of elements.

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

1. **Multi-Dimensional Array:**
   * An array with rows and columns.

int[,] matrix = { {1, 2}, {3, 4} };

Access elements:

Console.WriteLine(matrix[0, 1]); // Outputs 2

1. **Jagged Array:**
   * An array of arrays with varying lengths.

int[][] jaggedArray = new int[2][];

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

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

Access elements:

Console.WriteLine(jaggedArray[1][2]); // Outputs 5

**Array Methods**

1. **Length**:
   * Gets the total number of elements.
   * Example:

int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers.Length); // Outputs: 3

1. **Sort**:
   * Sorts the array in ascending order.
   * Example:

int[] numbers = { 3, 1, 2 };

Array.Sort(numbers);

1. **Reverse**:
   * Reverses the order of elements.
   * Example:

Array.Reverse(numbers);

**Defining and Calling Methods**

A **method** in C# is a block of code that performs a specific task. Methods help organize code into reusable, modular units, improving readability and reducing redundancy.

**Defining a Method**

A method is defined using the following syntax:

accessModifier returnType MethodName(parameters)

{

// Method body (code to execute)

}

**Parts of a Method**

* **Access Modifier**: Defines the visibility (e.g., public, private).
* **Return Type**: Specifies the type of value the method returns.
* **Method Name**: Identifies the method (e.g., PrintMessage).
* **Parameters**: Inputs to the method (optional).

**Example: Defining Methods**

1. **Method Without Parameters**  
   A method with no input values.

public void PrintMessage()

{

Console.WriteLine("Hello, World!");

}

1. **Method With Parameters**  
   A method that accepts input values.

public void PrintName(string name)

{

Console.WriteLine("Hello, " + name);

}

1. **Method With Return Type**  
   A method that returns a value.

public int AddNumbers(int a, int b)

{

return a + b;

}

**Calling a Method**

To call a method, use the method name followed by parentheses. If the method has parameters, pass the required values.

**Example: Calling Methods**

// Method without parameters

PrintMessage();

// Method with parameters

PrintName("Keyur");

// Method with return value

int result = AddNumbers(5, 10);

Console.WriteLine("Sum: " + result);

**Understanding Classes and OOP Concepts**

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects** and **classes**. C# is a fully OOP-compliant language, making it ideal for building modular, reusable, and maintainable applications.

**What is a Class?**

A **class** is a blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) of the objects.

**Syntax for a Class**

class ClassName

{

// Fields (variables)

// Methods (functions)

}

Example: Defining a Class

class Person

{

// Fields

public string Name;

public int Age;

// Method

public void Greet()

{

Console.WriteLine("Hello, my name is " + Name);

}

}

**What is an Object?**

An **object** is an instance of a class. It represents a real-world entity and is created using the new keyword.

**Example: Creating and Using an Object**

Person person1 = new Person(); // Create an object

person1.Name = "Keyur"; // Assign values to fields

person1.Age = 20;

person1.Greet(); // Call the method

**Key OOP Concepts**

**A. Encapsulation**

* Bundling data fields and methods into a single class.
* Restricting direct access to fields using **access modifiers**.

Example: Encapsulation with Properties

class BankAccount

{

private double balance;

// Public property to access private field

public double Balance

{

get { return balance; }

set { if (value >= 0) balance = value; }

}

}

**B. Inheritance**

* Enables one class to inherit fields and methods from another.
* Promotes **code reusability**.

Example: Inheritance

class Animal

{

public void Eat() { Console.WriteLine("This animal eats food."); }

}

class Dog : Animal

{

public void Bark() { Console.WriteLine("The dog barks."); }

}

**C. Polymorphism**

* Allows methods to have **different forms** (overloading and overriding).

**Example: Method Overloading**

class Calculator

{

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

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

}

**Example: Method Overriding**

class Animal

{

public virtual void Speak() { Console.WriteLine("Animal sound"); }

}

class Dog : Animal

{

public override void Speak() { Console.WriteLine("Dog barks"); }

}

**D. Abstraction**

* Hides implementation details while exposing essential features.
* Achieved using **abstract classes** or **interfaces**.

**Example: Abstraction with Abstract Class**

abstract class Shape

{

public abstract void Draw(); // Abstract method

}

class Circle : Shape

{

public override void Draw() { Console.WriteLine("Drawing a Circle"); }

}

**Interface and Multiple Inheritance**

**What is an Interface?**

An **interface** in C# is a blueprint for a class. It defines **what a class should do** but not **how it should do it**. It contains **method declarations** but no implementation.

* Declared using the interface keyword.
* All methods are implicitly **public** and **abstract**.
* A class or struct implements an interface using the : interfaceName syntax.

**Defining an Interface**

interface IShape

{

void Draw(); // Method declaration (no body)

}

**Implementing an Interface**

A class implements an interface by providing the body for all its methods.

class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing a Circle");

}

}

**Multiple Inheritance Using Interfaces**

C# **does not support multiple inheritance** with classes but allows it with interfaces. A class can implement multiple interfaces, combining their functionalities.

**Example: Multiple Interfaces**

interface IShape

{

void Draw();

}

interface IColor

{

void FillColor(string color);

}

class Rectangle : IShape, IColor

{

public void Draw()

{

Console.WriteLine("Drawing a Rectangle");

}

public void FillColor(string color)

{

Console.WriteLine($"Filling Rectangle with {color} color");

}

}

**Key Differences: Interface vs Class**

| **Aspect** | **Class** | **Interface** |
| --- | --- | --- |
| **Implementation** | Can have method implementations. | No implementation, only declarations. |
| **Inheritance** | Supports single inheritance. | Supports multiple inheritance. |
| **Access Modifiers** | Methods can have various access modifiers. | All methods are public by default. |

**Additional OOP Topics**

**Constructors**

* Special methods used to initialize objects.
* **Types**:
* Default Constructor:

public ClassName() { }

* Parameterized Constructor:

public ClassName(int value) { }

* Copy Constructor:

public ClassName(ClassName obj) { }

**Destructors**

* Used to clean up resources when an object is destroyed.

Defined using  ~ClassName.

Example:

~MyClass()

{

Console.WriteLine("Destructor called");

}

**Static Members**

* Belong to the class rather than any object.

Example:

class Counter

{

public static int Count = 0;

}

**Sealed Classes and Methods**

* Prevent inheritance or method overriding.

Example:

sealed class FinalClass { }

**Scope and Accessibility Modifiers**

**Scope**

**Scope** refers to the visibility or lifetime of variables, methods, or objects within a program. It determines **where in the code a variable or method can be accessed**.

**Types of Scope**

1. **Local Scope**
   * Variables declared inside a method or block (e.g., if, for) are accessible only within that block.
   * They are created when the block is executed and destroyed when it ends.

**Example:**

void ShowMessage()

{

int number = 10; // Local scope

Console.WriteLine(number); // Accessible here

}

// Console.WriteLine(number); // Not accessible here

1. **Class/Instance Scope**
   * Variables (fields) declared inside a class but outside any method or block.
   * Accessible across all methods of the class, and their lifetime equals the lifetime of the object or program.

**Example:**

class Example

{

int age; // Instance scope

void DisplayAge()

{

age = 25;

Console.WriteLine(age);

}

}

**Accessibility Modifiers**

* **Accessibility Modifiers** control the visibility of types and their members. They define where a class, field, method, or property can be accessed.

**Types of Accessibility Modifiers**

1. **public**:
   * The member is accessible from anywhere, both inside and outside the class.

Example:

public int Age { get; set; }

1. **private**:
   * The member is only accessible within the class where it is declared.
   * Default for class members.

Example:

private int number;

1. **protected**:
   * The member is accessible within the class and by derived (child) classes.

Example:

protected int Id;

1. **internal**:
   * The member is accessible within the same assembly (project) but not outside it.

Example:

internal void Display()

{

Console.WriteLine("Inside the assembly");

}

1. **protected internal**:
   * The member is accessible within the same assembly or in derived classes from other assemblies.

Example: protected internal int Counter;

1. **private protected**:
   * The member is accessible only within the same class and derived classes within the same assembly.

Example: private protected int score;

**Namespace and .NET Library**

**Namespace**

A **namespace** is a container that organizes classes, interfaces, enums, and other types into a logical structure. It helps avoid naming conflicts and makes code more manageable.

**Why Use Namespaces?**

* Prevents **name collisions** when multiple classes have the same name.
* Provides a **hierarchical structure** for organizing code.
* Helps in **logical grouping** of related classes and methods.

**Defining a Namespace**

namespace MyNamespace

{

class Program

{

static void Main()

{

Console.WriteLine("Hello, Namespace!");

}

}

}

**Using Namespaces**

To use classes from a namespace, either:

1. Fully qualify the class name:

System.Console.WriteLine("Hello, World!");

1. Add a using directive at the top:

using System;

class Program

{

static void Main()

{

Console.WriteLine("Hello, World!");

}

}

**Nested Namespaces**

Namespaces can be nested for better organization.

namespace OuterNamespace

{

namespace InnerNamespace

{

class MyClass

{

public void Display()

{

Console.WriteLine("Inside InnerNamespace");

}

}

}

}

Usage:

OuterNamespace.InnerNamespace.MyClass obj = new OuterNamespace.InnerNamespace.MyClass();

obj.Display();

**.NET Library**

The **.NET Library**, also known as the **Base Class Library (BCL)**, is a collection of reusable types provided by the .NET framework. It provides essential functionality required for building applications.

**Common Features of .NET Library**

1. **Core Functionality**: Basic data types, math operations, collections, and file I/O.
2. **Data Access**: Access to databases through ADO.NET.
3. **Networking**: Classes for handling HTTP, FTP, sockets, etc.
4. **Security**: Authentication and encryption utilities.
5. **UI Frameworks**: Support for building graphical or web-based user interfaces.
6. **Multithreading**: Classes for managing threads and parallel programming.

|  |  |  |
| --- | --- | --- |
| **Namespace** | **Purpose** | **Example Class/Method** |
| System | Core types and operations. | Console, String, Math |
| System.Collections | Classes for data structures like lists, queues, stacks. | ArrayList, Hashtable |
| System.IO | File input/output operations. | File, StreamReader |
| System.Net | Networking functionality. | WebClient, HttpWebRequest |
| System.Threading | Multithreading support. | Thread, Task |
| System.Linq | LINQ for querying collections. | Where, Select |
| System.Data | Working with databases and datasets. | DataTable, SqlConnection |

**Creating and Adding Reference to Assemblies**

**What is an Assembly?**

* An **assembly** is a compiled code library in .NET, such as a .dll (Dynamic Link Library) or .exe (Executable).
* It contains the intermediate language (IL) code, metadata, and resources needed for execution.
* Assemblies are the building blocks of .NET applications.

**Creating an Assembly**

1. **Create a Class Library Project:**

* Open Visual Studio.
* Go to File > New > Project.
* Select **Class Library** (e.g., .NET Framework Class Library or .NET Core Class Library).
* Name the project (e.g., MathLibrary) and click **Create**.

1. **Write Code in the Class Library:**

namespace MathLibrary

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

}

}

1. **Build the Project:**

* Go to Build > Build Solution or press Ctrl+Shift+B.
* This generates a .dll file in the bin directory of the project.

**Adding Reference to an Assembly**

To use the created assembly in another project:

1. **Create a New Project:**

* Go to File > New > Project.
* Select a Console Application or any other project type.

1. **Add a Reference to the Assembly:**

* Right-click on the project in **Solution Explorer**.
* Select **Add > Project Reference** (if the assembly is in the same solution) or **Add > Reference** (for external assemblies).
* Browse to the .dll file of the Class Library and add it.

1. **Use the Assembly in the Code:**

* Add the using directive for the namespace of the assembly.
* Use the class and methods from the referenced assembly.

**Example:**

using MathLibrary;

class Program

{

static void Main()

{

Calculator calc = new Calculator();

int result = calc.Add(10, 20);

Console.WriteLine($"Result: {result}");

}

}

**Working with Collections**

**What are Collections?**

* **Collections** in C# are specialized data structures for storing and managing groups of objects.
* Unlike arrays, collections provide flexibility in:
  + Dynamic resizing.
  + Managing different types of data.
  + Advanced operations like sorting, searching, and filtering.

**1. ArrayList (Non-Generic Collection)**

* **Description**:
  + Part of the System.Collections namespace.
  + A dynamically resizable collection that can hold elements of any type (object).
  + **Type-safety** is not enforced, so runtime errors may occur if type mismatches happen during retrieval.
* **When to Use**:
  + When you need a collection but don't know the element types at compile time.
  + Legacy support for older .NET frameworks.
* **Example**:

using System.Collections;

void ArrayListExample()

{

ArrayList list = new ArrayList();

list.Add(1); // Adding an integer

list.Add("Two"); // Adding a string

list.Add(3.5); // Adding a double

foreach (var item in list)

{

Console.WriteLine(item); // Output: 1, Two, 3.5

}

}

**2. List<T> (Generic Collection)**

* **Description**:
  + Part of the System.Collections.Generic namespace.
  + A strongly-typed dynamic array, ensuring type-safety at compile time.
  + The T represents the type of elements the list will hold (e.g., List<int> or List<string>).
* **When to Use**:
  + When you need a dynamic, type-safe collection.
* **Example**:

using System.Collections.Generic;

void ListExample()

{

List<int> numbers = new List<int>();

numbers.Add(1);

numbers.Add(2);

numbers.Add(3);

foreach (int number in numbers)

{

Console.WriteLine(number); // Output: 1, 2, 3

}

}

**3. Dictionary<TKey, TValue> (Key-Value Pair Collection)**

* **Description**:
  + Part of the System.Collections.Generic namespace.
  + Stores key-value pairs, where keys must be unique and are used to retrieve values.
  + Both TKey and TValue are type-safe.
* **When to Use**:
  + When you need to associate a value with a unique key.
* **Example**:

using System.Collections.Generic;

void DictionaryExample()

{

Dictionary<int, string> employees = new Dictionary<int, string>();

employees.Add(1, "Keyur");

employees.Add(2, "Hit");

foreach (var employee in employees)

{

Console.WriteLine($"ID: {employee.Key}, Name: {employee.Value}");

}

}

**4. Queue<T> (FIFO Collection)**

* **Description**:
  + Part of the System.Collections.Generic namespace.
  + First-In-First-Out (FIFO) collection: elements are added to the end and removed from the front.
* **When to Use**:
  + When order of processing is important, such as task scheduling or print queues.
* **Example**:

using System.Collections.Generic;

void QueueExample()

{

Queue<string> queue = new Queue<string>();

queue.Enqueue("Task1"); // Add to the queue

queue.Enqueue("Task2");

queue.Enqueue("Task3");

while (queue.Count > 0)

{

Console.WriteLine(queue.Dequeue()); // Output: Task1, Task2, Task3

}

}

**5. Stack<T> (LIFO Collection)**

* **Description**:
  + Part of the System.Collections.Generic namespace.
  + Last-In-First-Out (LIFO) collection: elements are added to the top and removed from the top.
* **When to Use**:
  + When reverse order processing is required, such as backtracking or undo operations.
* **Example**:

using System.Collections.Generic;

void StackExample()

{

Stack<string> stack = new Stack<string>();

stack.Push("Task1"); // Add to the stack

stack.Push("Task2");

stack.Push("Task3");

while (stack.Count > 0)

{

Console.WriteLine(stack.Pop()); // Output: Task3, Task2, Task1

}

}

**Enumerations**

**What is an Enumeration?**

* An **Enumeration (enum)** is a user-defined data type in C# that consists of a set of named constant values.
* It improves code readability by allowing you to work with meaningful names instead of numeric values.
* Defined using the enum keyword.

**Syntax of an Enum**

enum EnumName

{

Value1,

Value2,

Value3

}

By default:

* The underlying type of the enum is int.
* The first value has an implicit value of 0, the second is 1, and so on.
* You can explicitly assign values.

**Explicit Value Assignment**

You can assign specific values to enum members.

enum Status

{

Pending = 1,

InProgress = 2,

Completed = 3

}

**Enum Methods**:

* + Enum.GetValues(): Returns an array of all values in an enum.
  + Enum.GetName(): Gets the name of a specific enum value.

Example:

foreach (Days day in Enum.GetValues(typeof(Days)))

{

Console.WriteLine(day);

}

**DataTable**

**What is a DataTable?**

* A DataTable is a part of the **ADO.NET** library that represents an in-memory, tabular data structure.
* It is similar to a database table, consisting of **rows** and **columns**, and is defined in the System.Data namespace.
* It is commonly used for managing and manipulating tabular data in memory.

**Namespace**

using System.Data;

**Creating a DataTable**

A DataTable consists of the following components:

1. **Columns**: Define the schema of the table.
2. **Rows**: Hold the data for the table.

**Steps to Create and Use a DataTable:**

1. Create a DataTable object.
2. Define the columns.
3. Add rows of data.
4. Access and manipulate data.

EX :

// Step 1: Create a DataTable object

DataTable table = new DataTable("Employees");

// Step 2: Define columns

table.Columns.Add("ID", typeof(int));

table.Columns.Add("Name", typeof(string));

// Step 3: Add rows

table.Rows.Add(1, "Keyur");

**Key Features**

1. **Dynamic Structure**:
   * You can add or remove columns and rows at runtime.
2. **Data Types**:
   * Each column can be strongly typed (e.g., int, string, decimal).
3. **Querying Data**:
   * You can filter or sort rows using LINQ or the Select method.

**Common Operations**

**1. Filtering Data**

DataRow[] filteredRows = table.Select("Department = 'IT'");

foreach (DataRow row in filteredRows)

{

Console.WriteLine($"ID: {row["ID"]}, Name: {row["Name"]}");

}

**2. Sorting Data**

DataTable sortedTable = table.Clone(); // Clone structure

foreach (DataRow row in table.Select("", "Salary DESC")) // Sort by Salary

{

sortedTable.ImportRow(row);

}

**3. Adding a New Column**

table.Columns.Add("Bonus", typeof(decimal));

foreach (DataRow row in table.Rows)

{

row["Bonus"] = (decimal)row["Salary"] \* 0.10; // 10% bonus

}

**4. Updating a Row**

table.Rows[1]["Salary"] = 62000; // Update Bob's Salary

**Exception Handling**

**What is Exception Handling?**

* Exception handling is a mechanism in C# to handle runtime errors gracefully without crashing the program.
* It allows developers to manage errors and provide meaningful responses or recovery mechanisms.

**Key Terminology**

1. **Exception**:
   * An exception is a runtime error or unexpected condition that disrupts the normal flow of the program.
   * Examples: Division by zero, file not found, null reference, etc.
2. **Try-Catch Block**:
   * Used to detect and handle exceptions.
3. **Finally Block**:
   * Used to execute code regardless of whether an exception occurs or not.

**Syntax**

try

{

// Code that may cause an exception

}

catch (ExceptionType ex)

{

// Code to handle the exception

}

finally

{

// Code that will always execute (optional)

}

**Example:**

try{

int x = 10, y = 0;

int result = x / y; // Throws DivideByZeroException

}

catch (DivideByZeroException ex)

{

Console.WriteLine("Cannot divide by zero.");

}

**Catching Specific Exceptions**

* Always catch specific exceptions before the general Exception class to handle errors appropriately.

**Example:**

try

{

string input = null;

Console.WriteLine(input.Length); // NullReferenceException

}

catch (NullReferenceException ex)

{

Console.WriteLine("Error: Null value encountered.");

}

catch (Exception ex)

{

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

}

**Finally Block**

* **Purpose**: Clean up resources like closing files, releasing memory, or database connections.

**Example:**

try{

}

catch (Exception ex){

}

finally

{

Console.WriteLine("Cleanup code executed.");

}

**Throwing Exceptions**

* You can throw exceptions explicitly using the throw keyword.

**Example:**

void ValidateAge(int age)

{

if (age < 18)

throw new ArgumentException("Age must be 18 or older.");

}

try

{

ValidateAge(15);

}

catch (ArgumentException ex)

{

Console.WriteLine(ex.Message); // Output: Age must be 18 or older.

}

**Different Project Types**

Visual Studio provides several project templates for building different types of applications in C#. Below is a summary of common project types and their purposes:

**1. Console Application**

* **Purpose**: Develop applications with a command-line interface.
* **Use Cases**:
  + Simple tools.
  + Testing code snippets.
  + Batch processing scripts.

**2. Windows Forms Application**

* **Purpose**: Create desktop applications with a graphical user interface (GUI).
* **Features**:
  + Drag-and-drop UI components like buttons, textboxes, and labels.
  + Event-driven programming.
* **Use Cases**:
  + Simple utilities.
  + Business desktop tools.

**3. ASP.NET Web Application**

* **Purpose**: Develop web applications and services.
* **Variants**:
  + **ASP.NET MVC**: Model-View-Controller architecture for web apps.
  + **ASP.NET Web Forms**: Event-driven model for simpler web apps.
  + **ASP.NET Core**: Cross-platform and modern web development framework.
* **Use Cases**:
  + E-commerce websites.
  + Enterprise web portals.
  + RESTful APIs.

**4. Class Library**

* **Purpose**: Create reusable code libraries that can be used by other applications.
* **Use Cases**:
  + Shared functionality like logging or utility methods.
  + Custom frameworks or SDKs.

**5. Blazor App**

* **Purpose**: Build interactive web applications using C# instead of JavaScript.
* **Types**:
  + **Blazor Server**: Runs on the server, updates UI in real-time.
  + **Blazor WebAssembly**: Runs entirely in the browser.
* **Use Cases**:
  + Single-page applications (SPAs).
  + Web apps with heavy reliance on C#.

**6. Azure Cloud Service**

* **Purpose**: Develop and deploy applications in the cloud using Microsoft Azure.
* **Types**:
  + **Azure Functions**: Event-driven microservices.
  + **Azure Web Apps**: Web applications hosted on Azure.
* **Use Cases**:
  + Scalable cloud solutions.
  + Serverless computing.

**7. Game Development**

* **Framework**: Unity or MonoGame.
* **Purpose**: Develop 2D and 3D games using C#.
* **Use Cases**:
  + Mobile games.
  + Console/PC games.

**Working with the String Class**

The String class in C# is used to represent and manipulate text. It provides a variety of methods and properties to work with strings efficiently.

**Declaring Strings**

Strings can be declared using the string keyword or the System.String class.

string greeting = "Hello, World!";

String message = "Welcome to C#.";

**Common String Operations**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| Length | Gets the number of characters in the string. | int length = greeting.Length; (Output: 13) |
| Concatenation | Combines two or more strings. | string fullName = firstName + " " + lastName; |
| Substring | Extracts part of a string. | string sub = greeting.Substring(0, 5); (Output: "Hello") |
| Replace | Replaces part of a string with another. | string replaced = greeting.Replace("World", "C#"); (Output: "Hello, C#!") |
| ToUpper/ToLower | Converts string to uppercase/lowercase. | string upper = greeting.ToUpper(); (Output: "HELLO, WORLD!") |
| Trim | Removes whitespace from start/end of a string. | string trimmed = " Hello ".Trim(); (Output: "Hello") |
| Split | Splits the string into an array based on a delimiter. | string[] words = greeting.Split(','); |
| Join | Combines an array into a single string. | string joined = string.Join("-", words); |
| Contains | Checks if the string contains a substring. | bool hasHello = greeting.Contains("Hello"); |
| StartsWith/EndsWith | Checks if the string starts/ends with a substring. | bool starts = greeting.StartsWith("Hello"); |
| Equals | Compares two strings for equality. | bool isEqual = greeting.Equals("Hello, World!"); |

**String Interpolation**

String interpolation simplifies formatting strings using $.

**Example:**

string name = "Keyur";

int age = 20;

string intro = $"My name is {name}, and I am {age} years old.";

Console.WriteLine(intro); // Output: My name is Keyur, and I am 20 years old.

**Working with the DateTime Class**

The DateTime class in C# is used to work with dates and times. It provides properties and methods for creating, manipulating, and formatting date and time values.

**Creating a DateTime Object**

1. **Current Date and Time**

DateTime now = DateTime.Now;

Console.WriteLine(now); // Output: Current date and time

1. **Specific Date and Time**

DateTime specificDate = new DateTime(2024, 12, 18, 14, 30, 0);

Console.WriteLine(specificDate); // Output: 12/18/2024 2:30:00 PM

1. **Only Date or Time**
   * Date only: DateTime.Today
   * Time only: DateTime.Now.TimeOfDay

**Common Properties**

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Example** |
| Now | Current date and time. | DateTime.Now |
| Today | Current date (time = 00:00:00). | DateTime.Today |
| Year, Month, Day | Extracts year, month, or day. | DateTime.Now.Year |
| Hour, Minute, Second | Extracts time components. | DateTime.Now.Hour |
| DayOfWeek | Day of the week. | DateTime.Now.DayOfWeek |
| DayOfYear | Day of the year (1–365/366). | DateTime.Now.DayOfYear |

**Formatting DateTime**

You can format DateTime using the ToString method.

**Common Formats**

DateTime now = DateTime.Now;

Console.WriteLine(now.ToString("MM/dd/yyyy")); // Output: 12/18/2024

Console.WriteLine(now.ToString("dddd, dd MMMM yyyy")); // Output: Wednesday, 18 December 2024

Console.WriteLine(now.ToString("hh:mm tt")); // Output: 02:30 PM

Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm:ss")); // Output: 2024-12-18 14:30:00

**Manipulating Dates and Times**

1. **Add/Subtract Days, Months, or Years**

DateTime today = DateTime.Now;

DateTime nextWeek = today.AddDays(7); // Adds 7 days

DateTime lastMonth = today.AddMonths(-1); // Subtracts 1 month

Console.WriteLine(nextWeek);

Console.WriteLine(lastMonth);

**Parsing Strings into DateTime**

The DateTime.Parse and DateTime.TryParse methods are used to convert strings into DateTime.

**Example: Parse**

string dateString = "12/18/2024";

DateTime date = DateTime.Parse(dateString);

Console.WriteLine(date); // Output: 12/18/2024 12:00:00 AM

**Example: TryParse**

string invalidDate = "invalid date";

bool success = DateTime.TryParse(invalidDate, out DateTime parsedDate);

if (success)

Console.WriteLine(parsedDate);

else

Console.WriteLine("Invalid date format.");

**Basic File Operations**

The System.IO namespace provides classes for working with files, directories, and streams. Common operations include creating, reading, writing, and deleting files.

**Namespace Required**

using System.IO; // Required for file operations

**Common File Operations**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Method** | **Description** |
| Create a file | File.Create() | Creates a new file. |
| Write to a file | File.WriteAllText() | Writes text to a file (overwrites if file exists). |
| Append to a file | File.AppendAllText() | Adds text to the end of an existing file. |
| Read a file | File.ReadAllText() | Reads all text from a file. |
| Delete a file | File.Delete() | Deletes a specified file. |
| Check if file exists | File.Exists() | Checks if a file exists at a given path. |

Example :

**1. Create and Write to a File**

File.WriteAllText("example.txt", "Hello, C#!");

Console.WriteLine("File created and text written.");

**2. Append Text to a File**

File.AppendAllText("example.txt", "\nAppending more text.");

Console.WriteLine("Text appended.");

**3. Read from a File**

string content = File.ReadAllText("example.txt");

Console.WriteLine(content);

**4. Check if File Exists**

if (File.Exists("example.txt"))

Console.WriteLine("File exists.");

else

Console.WriteLine("File does not exist.");

**5. Delete a File**

if (File.Exists("example.txt"))

{

File.Delete("example.txt");

Console.WriteLine("File deleted.");

}

**ASP.NET Web Application (.NET Framework) – 5 Types**

**1. Empty Web Application**

**Overview:**

The Empty Web Application is a minimal project template, providing a basic structure without predefined components. It’s ideal when you want complete control over which components you add to your project.

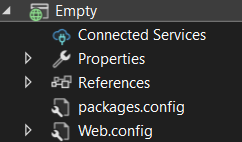
**Use Cases**:

* Custom projects with unique requirements.
* Learning and experimentation.

**Key Features**:

* Fully customizable.
* No unnecessary components or dependencies.

**Project Structure:**

****

**Key Files:**

**Web.config**:

The Web.config file is an XML-based configuration file for web applications built on the .NET Framework. It provides settings and configurations for the application, such as connection strings, authentication, authorization, custom error pages, and more.

<?xml version="1.0" encoding="utf-8"?>

<!--

For more information on how to configure your ASP.NET application, please visit

https://go.microsoft.com/fwlink/?LinkId=169433

-->

<configuration>

<system.web>

<compilation debug="true" targetFramework="4.7.2" />

<httpRuntime targetFramework="4.7.2" />

</system.web>

<system.codedom>

<compilers>

<compiler language="c#;cs;csharp" extension=".cs" type="Microsoft.CodeDom.Providers.DotNetCompilerPlatform.CSharpCodeProvider, Microsoft.CodeDom.Providers.DotNetCompilerPlatform, Version=2.0.1.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" warningLevel="4" compilerOptions="/langversion:default /nowarn:1659;1699;1701" />

<compiler language="vb;vbs;visualbasic;vbscript" extension=".vb" type="Microsoft.CodeDom.Providers.DotNetCompilerPlatform.VBCodeProvider, Microsoft.CodeDom.Providers.DotNetCompilerPlatform, Version=2.0.1.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" warningLevel="4" compilerOptions="/langversion:default /nowarn:41008 /define:\_MYTYPE=\&quot;Web\&quot; /optionInfer+" />

</compilers>

</system.codedom>

</configuration>

**packages.config:**

The packages.config file is used to manage NuGet package dependencies in older .NET Framework projects. It lists the NuGet packages required by the project and their specific versions.

<?xml version="1.0" encoding="utf-8"?>

<packages>

<package id="Microsoft.CodeDom.Providers.DotNetCompilerPlatform" version="2.0.1" targetFramework="net472" />

</packages>

**2. Web Forms Application**

**Overview:**

A traditional web application model for building dynamic, data-driven websites using server-side controls and events. It's based on the event-driven programming model.

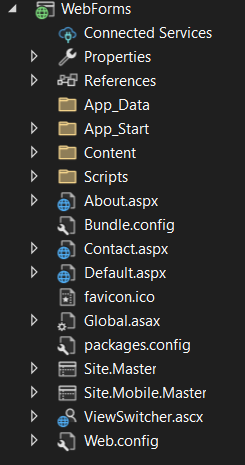
**Use Cases**:

* Small-to-medium-scale web applications.
* Rapid prototyping of web interfaces.

**Key Features**:

* Drag-and-drop design support.
* Code-behind files to separate UI and logic.

**Project Structure:**



**Key Files:**

**App\_Data :** Stores application data like databases, XML files, or other data-related files.

**App\_Start :** Contains configuration files for setting up the application, such as routing (RouteConfig.cs) or bundles (BundleConfig.cs).

**Content :** Holds static files like CSS, images, or other design-related resources.

**Scripts** : Stores JavaScript and other client-side libraries like jQuery or AngularJS.

**Default.aspx :** The default web page that serves as the starting point of the application. (Active Server Pages Extended)

**Global.asax :** A file for handling application-level events such as Application\_Start, Application\_End, or Session\_Start. (ASP.NET Server Application File)

**3. MVC (Model-View-Controller)**

**Overview:**

This template is based on the MVC architecture, which separates the application into three components: Model (data), View (UI), and Controller (logic). It promotes a clean separation of concerns.

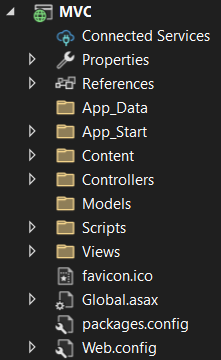
**Use Cases**:

* Enterprise-level web applications.
* Applications with complex workflows or custom UI needs.

**Key Features**:

* Clean architecture with testability.
* URL routing for SEO-friendly URLs.

**Project Structure:**



**Key Files :**

**Models :** Defines the data structure, typically matching the database tables or API responses. It can also include validation rules and methods related to data manipulation.

**Views :** Each view corresponds to a specific action in the controller and is used to render dynamic content.

**Controllers :** Each controller action corresponds to a user interaction (e.g., submitting a form) and returns a result, typically a view or a redirect.

**4. Web API**

**Overview:**

A project template for building RESTful services using ASP.NET Web API. It allows you to create HTTP-based APIs for communication between systems.

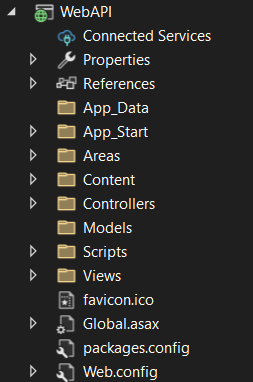
**Use Cases**:

* Backend services for mobile and web applications.
* Exposing APIs for third-party integrations.

**Key Features**:

* Lightweight and stateless communication.
* Supports multiple data formats like JSON and XML.

**Project Structure:**



**5. SPA (Single Page Application)**

**Overview:**

This project type is designed to build Single Page Applications using frameworks like Angular, React.The application dynamically updates the user interface without reloading the entire page.

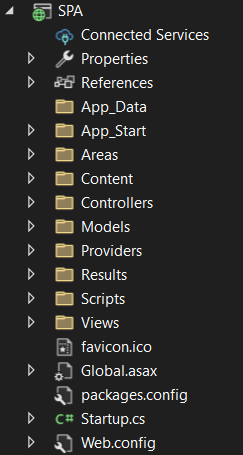
**Use Cases**:

* Modern web applications with rich user experiences.
* Real-time applications like chat platforms or dashboards.

**Key Features**:

* Minimal server-side page loading.
* Heavy reliance on client-side frameworks.
* Seamless and responsive user experiences.

**Project Structure :**

****

**Revise**

**Build vs Rebuild**  
  
**Build**

* **Purpose**: The Build operation compiles only the files that have changed since the last build or the files that depend on the changed files.
* **What it does**:
  + It checks for changes in your project files.
  + If no files have been changed since the last build, it will not recompile them.
  + This makes it faster than a rebuild, as it only compiles modified or dependent files.
* **When to use**:
  + When you have made small changes and want to quickly compile the project.

**Rebuild**

* **Purpose**: The Rebuild operation performs a complete clean of the project and then rebuilds it from scratch.
* **What it does**:
  + It first deletes all the compiled files (e.g., DLLs, EXEs, PDBs, etc.) in the output folder.
  + It then compiles all project files, regardless of whether they have changed or not.
* **When to use**:
  + When you want to ensure that your project is compiled from scratch, clearing any potential issues caused by old binaries.
  + When encountering issues that might be due to leftover compiled files (e.g., when changes aren't reflected after a **Build**).

**Watch Window**

The **Watch** window allows you to add specific variables or expressions to monitor their values while debugging. You can use it to track variables, properties, or custom expressions, regardless of where they are in the code.

**Features of the Watch Window:**

* Add any variable or expression to track its value during runtime.
* Allows evaluating complex expressions.
* Shows the value of variables as you step through the code.

**How to Use Watch Window:**

1. Start debugging by setting a breakpoint.
2. When the breakpoint is hit, go to **Debug > Windows > Watch > Watch 1** (or press Ctrl + Alt + W).
3. In the **Watch 1** window, type the name of a variable or an expression you want to track.
4. As you step through the code, you can see the real-time value of the variable/expressions you added.

**Locals Window**

The **Locals** window displays all local variables within the current scope (method or function) while debugging. It's useful for quickly viewing and inspecting all variables within the current context.

**Features of the Locals Window:**

* Automatically lists all local variables and their values for the current scope.
* Updates as you step through the code.
* Can expand objects and arrays to explore their properties and elements.

**How to Use Locals Window:**

1. Set a breakpoint and start debugging.
2. Go to **Debug > Windows > Locals** (or press Ctrl + Alt + V, L).
3. Once the debugger hits the breakpoint, the **Locals** window will display all local variables and their values in the current method.

**Operators :**

**Index from end operator ^ (circumflex)**

The ^ operator indicates the element position from the end of a sequence. For a sequence of length length, ^n points to the element with offset length - n from the start of a sequence. For example, ^1 points to the last element of a sequence and ^length points to the first element of a sequence.

EX :

int[] xs = [0, 10, 20, 30, 40];

int last = xs[^1];

Console.WriteLine(last); // output: 40

List<string> lines = ["one", "two", "three", "four"];

string prelast = lines[^2];

Console.WriteLine(prelast); // output: three

**Range operator ..**

The .. operator specifies the start and end of a range of indices as its operands. The left-hand operand is an *inclusive* start of a range. The right-hand operand is an *exclusive* end of a range. Either of the operands can be an index from the start or from the end of a sequence.

Ex :

int[] numbers = [0, 10, 20, 30, 40, 50];

int start = 1;

int amountToTake = 3;

int[] subset = numbers[start..(start + amountToTake)];

Display(subset); // output: 10 20 30

string line = "one two three";

int amountToTakeFromEnd = 5;

Range endIndices = ^amountToTakeFromEnd..^0;

string end = line[endIndices];

Console.WriteLine(end); // output: three

You can omit any of the operands of the .. operator to obtain an open-ended range:

* a.. is equivalent to a..^0
* ..b is equivalent to 0..b
* .. is equivalent to 0..^0

**is operator**

The is operator checks if the run-time type of an expression result is compatible with a given type. The is operator also tests an expression result against a pattern.

Ex :

object obj = 123;

if (obj is int)

{

Console.WriteLine("obj is an integer.");

}

// Output: obj is an integer.

**as operator**

The as operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion isn't possible, the as operator returns null.

Ex :

object obj = "Hello, World!";

string str = obj as string; // Safe cast to string

if (str != null)

{

Console.WriteLine(str); // Output: Hello, World!

}

else

{

Console.WriteLine("Conversion failed.");

}

**typeof operator**

The typeof operator obtains the System.Type instance for a type. The argument to the typeof operator must be the name of a type or a type parameter.

Ex:

void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>)); // System.Collections.Generic.List`1[System.String]

PrintType<int>(); // System.Int32

PrintType<System.Int32>(); // System.Int32

PrintType<Dictionary<int, char>>(); //System.Collections.Generic.Dictionary`2[System.Int32,System.Char]

**Null Conditional Operator (?.)**

The ?. operator is used to safely access members (properties, methods, or fields) of an object that might be null. It avoids null reference exceptions by short-circuiting if the object is null.

**Syntax**

object?.Member

**Example**

string name = null;

int? length = name?.Length; // If name is null, length is assigned null.

Console.WriteLine(length); // Output: (blank, as it's null)

* If name is not null, it evaluates name.Length.
* If name is null, it doesn't throw an exception but returns null

**Null Coalescing Operator (??)**

The ?? operator works with the ? concept and provides a default value if the left-hand operand is null.

**Syntax**

value\_if\_not\_null ?? default\_value;

**Example**

string name = null;

string displayName = name ?? "Unknown";

Console.WriteLine(displayName); // Output: Unknown

**sizeof operator**

The sizeof operator returns the number of bytes occupied by a variable of a given type.

EX :

Console.WriteLine(sizeof(byte)); // output: 1

Console.WriteLine(sizeof(double)); // output: 8

**Private vs sealed class**

|  |  |  |
| --- | --- | --- |
| Aspect | Private Class | Sealed Class |
| Purpose | Controls visibility (accessibility). | Prevents inheritance (behavior restriction). |
| Declaration | Can only be nested inside another class. | Declared at the namespace or class level. |
| Accessibility | Accessible only within the containing class. | Accessible from anywhere it is visible. |
| Inheritance | Can be inherited within its scope. | Cannot be inherited at all. |
| Use Case | Encapsulation of helper or utility classes. | Securing functionality or preventing misuse. |

**Partial Class**

A **partial class** allows a class definition to be split across multiple files. At compile time, these separate parts are combined into a single class by the compiler.

**Syntax**

**File 1 (Part1.cs):**

namespace MyNamespace

{

public partial class MyClass

{

public void Method1()

{

Console.WriteLine("Method1 in Part 1.");

}

}

}

**File 2 (Part2.cs):**

namespace MyNamespace

{

public partial class MyClass

{

public void Method2()

{

Console.WriteLine("Method2 in Part 2.");

}

}

}

**Scope & Accessibility modifier**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | In derived class | In whole assembly | Derived but diff assembly | Whole solution |
| public | true | true | true | True |
| private | false | false | false | false |
| protected | true | false | true | false |
| internal | true | true | false | false |
| protected internal | true | true | true | false |
| private protected | true | false | false | false |

**Use of <T> (type parameter)**

using System;

using System.Collections.Generic;

public class HelloWorld

{

public static void createList<T>(T input){

List<T> k = new List<T>();

k.Add(input);

Console.WriteLine(k[0].GetType());

}

public static void Main(string[] args)

{

createList<int>(1);

createList<string>("keyur");

}

}

**AddRange() to concat two list**

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

List<int> extraNumbers = [8, 9, 10 ,11];

numbers.AddRange(extraNumbers);

**Using Statement**

When the control leaves the block of the using statement, an acquired IDisposable instance is disposed. In particular, the using statement ensures that a disposable instance is disposed even if an exception occurs within the block of the using statement.

EX :

var numbers = new List<int>();

using (StreamReader reader = File.OpenText("numbers.txt"))

{

string line;

while ((line = reader.ReadLine()) is not null)

{

if (int.TryParse(line, out int number))

{

numbers.Add(number);

}

}

}

**StreamReader**

StreamReader is used to read characters from a text file.

Namespace : System.IO

**Key Features**

* Reads text files line-by-line or character-by-character.
* Handles text encoding automatically.

Common Methods

|  |  |
| --- | --- |
| Method | Description |
| Read() | Reads the next character and returns it as an integer. |
| ReadLine() | Reads the next line from the file. |
| ReadToEnd() | Reads all characters from the current position to the end of the file. |
| Close() | Closes the StreamReader and releases resources. |

EX:

string filePath = "example.txt";

using (StreamReader reader = new StreamReader(filePath))

{

string content = reader.ReadToEnd();

Console.WriteLine($"File Content: {content}");

}

**Peek()** :

Returns the next available character (in int) but does not consume it.

Returns : An integer representing the next character to be read, or -1 if there are no characters to be read or if the stream does not support seeking.

Use Case :

using (StreamReader sr = new StreamReader(path))

{

while (sr.Peek() > -1)

{

Console.WriteLine(sr.ReadLine());

}

}

**StreamWriter**

StreamWriter is used to write characters to a text file.

**Key Features**

* Writes text data line-by-line or character-by-character.
* Can append or overwrite existing files.

Common Methods

|  |  |
| --- | --- |
| Method | Description |
| Write() | Writes a string or character to the file. |
| WriteLine() | Writes a string followed by a newline. |
| Flush() | Clears all buffers and writes data to the file. |
| Close() | Closes the StreamWriter and releases resources. |

EX :

string filePath = "example.txt";

string content = "Hello, StreamWriter!\nThis is an example of writing text.";

using (StreamWriter writer = new StreamWriter(filePath))

{

writer.WriteLine(content);

Console.WriteLine("Data written to the file.");

}

**Flush() :**

Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.

**Seek() :**

Sets the current position of this stream to the given value.

Syntax : FileStreamObj.Seek(long offset, System.IO.SeekOrigin origin)

Offset : The point relative to origin from which to begin seeking.

Origin :

Specifies the beginning, the end, or the current position as a reference point for offset, using a value of type SeekOrigin (Enum) .

SeekOrigin.Bigin =0 SeekOrigin.Current = 1, SeekOrigin.End = 2

EX :

using (FileStream fs = new FileStream(path, FileMode.Open))

{

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

{

fs.Seek(-i, SeekOrigin.End);

Console.Write((char)fs.ReadByte());

}

}