Visual Studio 2022 Overview

Visual Studio 2022 is a modern, 64-bit IDE designed for efficient and powerful development of applications across platforms.

Key Highlights

- **64-bit Support**: Handles larger projects and solutions without performance issues.
- **Hot Reload**: Modify .NET or C++ apps while debugging without restarting.
- **Git Integration**: Built-in tools for branching, committing, and collaboration.
- **AI-Assisted Development**: IntelliCode provides smart code suggestions.
- **Cross-Platform Development**: Build apps for Windows, macOS, Android, iOS, and Linux.

Essential Shortcuts

Action	Shortcut
Open Solution Explorer	**Ctrl + Alt + L**
Build Solution	**Ctrl + Shift + B**
Start Debugging	**F5**
Stop Debugging	**Shift + F5**
Step Into (Debugging)	**F11**
Step Over (Debugging)	**F10**
Find and Replace	**Ctrl + F** / **Ctrl + H**
Go to Definition	**F12**
Quick Actions (Refactor)	**Ctrl + .**
Open Terminal	**Ctrl +** (`)
Toggle Comment	**Ctrl + K, Ctrl + C**
Uncomment Code	**Ctrl + K, Ctrl + U**
Format Document	**Ctrl + K, Ctrl + D**
Navigate to File/Type/Symbol	**Ctrl + T**
Show IntelliSense Suggestions	**Ctrl + Space**
Close Active Tab	**Ctrl + F4**

Important Features

- **IntelliSense**: Smart code completion and hints for faster development.
- **Live Share**: Real-time collaboration with teammates for editing and debugging.
- **Performance Profiler**: Analyze and optimize application performance.
- **Azure Integration**: Seamless deployment to Azure cloud.

Supported Workloads

- **Desktop Development**: .NET, C++, Python.
- **Web Development**: ASP.NET, JavaScript, Node.js.
- **Mobile Development**: Xamarin, .NET MAUI.
- **Game Development**: Unity, Unreal Engine.

Introduction to C#

What is C#?

- C# (pronounced “C-Sharp”) is a modern, object-oriented, and type-safe programming language developed by Microsoft.
- It is part of the .NET framework ecosystem and is widely used for building:
  - Desktop applications
  - Web applications
  - Mobile apps
  - Game development (via Unity)
  - Cloud-based services and APIs

Features of C#

1. **Object-Oriented**: Supports concepts like inheritance, polymorphism, and encapsulation.
2. **Type-Safe**: Prevents unintended type conversions, ensuring code reliability.
3. **Rich Libraries**: Access to a vast set of libraries in the .NET framework for various functionalities.
4. **Cross-Platform**: Develop applications that run on Windows, macOS, and Linux using .NET Core/6+.
5. **Automatic Memory Management**: Managed by the .NET runtime using garbage collection.
6. **Strong Community Support**: Regular updates, extensive documentation, and community resources.

Why Learn C#?

- **Versatility**: From Windows apps to cross-platform web and mobile apps.
- **Ease of Use**: Simple syntax inspired by C++ and Java.
- **Career Opportunities**: High demand for C# developers, especially in enterprise-level software development.
- **Powerful Tools**: Supported by Visual Studio, a feature-rich IDE.

Create Your First C# Program: 'Hello, World!'

Step 1: Set Up Your Environment

Install Required Tools:

- **Visual Studio**: Download from visualstudio.microsoft.com. Choose the **.NET Desktop Development** workload during installation.

Alternatives:

- **Visual Studio Code** with the C# extension and .NET SDK.
- Online editors like dotnetfiddle.net.

Step 2: Write Your First Program

Code Example:

```
using System; // Importing the System namespace

class Program // Class definition
{
    static void Main(string[] args) // Entry point of the program
    {
        Console.WriteLine("Hello, World!"); // Output text to the console
    }
}
```

Steps to Run:

1. Open Visual Studio and create a new project:
   - Go to **File > New > Project**.

   - Select **Console App (.NET)**.
2. Name your project (e.g., **HelloWorld**) and click **Create**.
3. Replace the default code in **Program.cs** with the above example.
4. Press **Ctrl + F5** or click **Start Without Debugging** to run the program.
5. The output **Hello, World!** will appear in the console.

Step 3: Explanation of Code

using System;

- Imports the **System** namespace, which includes basic classes like **Console**.

class Program

- Defines a class named **Program**.
- In C#, everything is encapsulated within classes.

static void Main(string[] args)

- Entry point of the application.
- **static**: No instance of the class is needed to execute this method.
- **void**: The method does not return a value.
- **args**: An array for command-line arguments.

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

- **Console**: A class in the **System** namespace.
- **WriteLine()**: Outputs text followed by a new line.

Understanding C# Program Structure

Basic Structure

```
using System;          // Namespace declaration

namespace MyNamespace  // Optional: Defines the namespace for the program
{
    class Program      // Class declaration
    {
        static void Main(string[] args) // Main method: Entry point
        {
            // Statements
        }
    }
}
```

Key Components

1. Namespace

- Organizes classes and avoids naming conflicts.
- Example:

```
namespace MyApp
{
    class Example { }
}
```

2. Class

- A blueprint for creating objects and encapsulating methods and variables.
- Example:

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

3. Main Method

- The starting point of the program.
- Can take optional parameters like **string[] args** for command-line arguments.
- Example:

```
static void Main(string[] args)
{
    Console.WriteLine("Program Starts Here");
}
```

4. Statements

- The logical instructions that the program executes.
- Example:

```
Console.WriteLine("This is a statement.");
```

Program Execution Flow

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

Working with Code Files, Projects & Solutions

Understanding Code Files, Projects, and Solutions

1. Code Files

- Files containing C# code, typically with the extension **.cs**.
- Each file can contain classes, interfaces, enums, or methods.
- Example:

```
// File: Program.cs
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

2. Projects

- A project represents a single application, library, or service.
- Contains all code files, dependencies, and settings required to build and run the application.
- Types:
  - **Console App**: Command-line applications.
  - **Windows App**: Desktop GUI applications.
  - **Class Library**: Reusable code libraries.

3. Solutions

- A solution is a container for one or more projects.
- Used to manage large applications with multiple components (e.g., frontend, backend).
- Solution files have the extension **.sln**.

How They Work Together

1. **Solution**:
   - Contains multiple **projects**.
2. **Project**:
   - Contains multiple **code files**.
3. **Code File**:
   - Contains C# code that defines classes, methods, etc.

Using Visual Studio

- **Create a Solution**:
  - Go to **File > New > Project**, then choose a template.
- **Add a New Code File**:
  - Right-click the project, select **Add > New Item**, and choose **Class**, **Interface**, etc.
- **Build and Run**:
  - Press **Ctrl + F5** or click **Start Without Debugging**.

Datatypes & Variables with Conversion

Datatypes in C#

Value Types

- Store data directly in memory.
- Examples:

  - **int** (Integer): 32-bit signed integer.
  - **float** (Floating Point): Single precision (32-bit).
  - **bool** (Boolean): **true** or **false**.
  - **char** (Character): Single Unicode character.
  - **struct** (Structure): User-defined value type.

Reference Types

- Store references to memory locations.
- Examples:
  - **string**: Sequence of characters.
  - **object**: Base type of all types in C#.
  - **class**: User-defined reference type.

Nullable Types

- Allow value types to represent **null**.
- Example:

```
int? age = null;
```

Variables in C#

- **Definition**: A variable is a named memory location used to store data.
- **Declaration**:

```
int number = 10; // Variable declaration with initialization
```

Variable Types

1. **Local Variables**:
   - Declared inside a method or block.
   - Example:

```
void Example()
{
    int count = 5; // Local variable
}
```

2. **Instance Variables**:
   - Declared in a class but outside methods.
   - Example:

```
class Example
{
    private string name; // Instance variable
}
```

3. **Static Variables**:
   - Shared across all instances of a class.
   - Example:

```
static int count = 0; // Static variable
```

Type Conversion in C#

1. **Implicit Conversion**:
   - Automatically done by the compiler when no data loss occurs.
   - Example:

```
int num = 10;
double result = num; // Implicit conversion
```

2. **Explicit Conversion (Casting)**:
   - Requires a cast operator.
   - Example:

```
double value = 10.5;
int result = (int)value; // Explicit conversion
```

3. **Using Convert Class**:

- Converts data between types.
- Example:

```
string str = "123";
int num = Convert.ToInt32(str); // Conversion using Convert class
```

4. **Parsing**:
    - Converts strings to specific types.
    - Example:

```
string str = "123";
int num = int.Parse(str); // Parsing
```

5. **TryParse Method**:
    - Safe way to parse without throwing exceptions.
    - Example:

```
string str = "123";
int result;
if (int.TryParse(str, out result))
{
    Console.WriteLine("Parsed successfully.");
}
```

Operators & Expressions

Operators in C#

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

- Defines the order of operations in an expression.
- Example:
  - Multiplication (*) and Division (/) are evaluated before Addition (+) and Subtraction (-).
  - Use parentheses () to override precedence.

Statements

What Are Statements?

- Statements are individual instructions executed by the C# compiler.
- They can perform actions like variable declarations, assignments, method calls, or loops.
- Each statement ends with a **semicolon (;)**.

Types of Statements

1. **Declaration Statements**:

   - Declare and initialize variables.

   - Example:

     ```
     int number = 10; // Variable declaration and initialization
     ```

2. **Expression Statements**:

   - Perform actions like assignments, method calls, or operations.

   - Example:

     ```
     Console.WriteLine("Hello, World!"); // Method call
     number += 5; // Assignment expression
     ```

3. **Control Flow Statements**:

   - Alter the flow of execution.
   - Examples:
     - **Conditional**: **if**, **else**, **switch**.
     - **Loops**: **for**, **while**, **do-while**, **foreach**.

4. **Jump Statements**:

   - Transfer control to other parts of the program.

   - Examples:

     ```
     break;   // Exit loops or switch cases
     continue; // Skip the current iteration
     return;  // Exit from a method
     ```

5. **Block Statements**:

   - Group multiple statements in curly braces **{ }**.

   - Example:

     ```
     if (number > 0)
     {
        Console.WriteLine("Positive number");
        Console.WriteLine("End of check");
     }
     ```

Understanding Arrays

What Are Arrays?

- Arrays are a collection of elements of the same type, stored in contiguous memory locations.

- They allow multiple values to be stored in a single variable.

Syntax for Declaring Arrays

```
datatype[] arrayName = new datatype[size];
```

- **datatype**: Type of elements in the array.
- **arrayName**: Name of the array.
- **size**: Number of elements in the array.

Examples

1. **Declaration and Initialization**:

```
int[] numbers = new int[5]; // Array with 5 integers
numbers[0] = 10;           // Assign value to the first element
```

2. **Inline Initialization**:

```
string[] fruits = { "Apple", "Banana", "Cherry" }; // Array with predefined values
```

3. **Accessing Elements**:

```
Console.WriteLine(fruits[1]); // Outputs: Banana
```

Types of Arrays

1. **Single-Dimensional Array**:
   - A simple list of elements.
   - Example:

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

2. **Multi-Dimensional Array**:
   - A table-like structure with rows and columns.
   - Example:

```
int[,] matrix = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
Console.WriteLine(matrix[1, 2]); // Outputs: 6
```

3. **Jagged Array**:

   - An array of arrays with varying lengths.
   - Example:

```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[] { 1, 2, 3 };
jaggedArray[1] = new int[] { 4, 5 };
Console.WriteLine(jaggedArray[1][1]); // Outputs: 5
```

Array Methods

1. **Length**:

   - Gets the total number of elements.
   - Example:

```
int[] numbers = { 1, 2, 3 };
Console.WriteLine(numbers.Length); // Outputs: 3
```

2. **Sort**:

   - Sorts the array in ascending order.
   - Example:

```
int[] numbers = { 3, 1, 2 };
Array.Sort(numbers);
```

3. **Reverse**:

   - Reverses the order of elements.
   - Example:

```
Array.Reverse(numbers);
```

Define & Calling Methods

What Are Methods?

- Methods are blocks of code designed to perform specific tasks.
- They promote code reuse and modular programming.

Defining a Method

Syntax

```
accessModifier returnType MethodName(parameters)
{
   // Method body
}
```

- **accessModifier**: Determines method visibility (**public**, **private**, etc.).
- **returnType**: Data type returned by the method (**void** if no value is returned).
- **MethodName**: Name of the method (Pascal Case is standard).
- **parameters**: Input values for the method (optional).

Example

```
public int Add(int a, int b)
{
   return a + b; // Returns the sum of two numbers
}
```

Calling a Method

Syntax

```
MethodName(arguments);
```

Example

```
class Program
{
   static void Main(string[] args)
   {
       Program obj = new Program();
       int result = obj.Add(10, 20); // Call the Add method
       Console.WriteLine(result);   // Outputs: 30
   }

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

Types of Methods

1. Parameterless Methods

- Do not take any input arguments.
- Example:

```
public void Greet()
{
    Console.WriteLine("Hello!");
}
```

2. Parameterized Methods

- Accept input arguments.
- Example:

```
public void Display(string message)
{
    Console.WriteLine(message);
}
```

3. Static Methods

- Called without creating an object of the class.
- Example:

```
public static void ShowMessage()
{
   Console.WriteLine("Static Method");
}
```

4. Method Overloading

- Multiple methods with the same name but different parameters.
- Example:

```
public int Add(int a, int b) => a + b;
public double Add(double a, double b) => a + b;
```

Returning Values

Syntax

```
return value;
```

Example:

```
public int Multiply(int a, int b)
{
   return a * b;
}
```

Using void Methods

- Methods that do not return any value.
- Example:

```
public void PrintMessage(string message)
{
   Console.WriteLine(message);
}
```

Recursion

- A method calling itself to solve a problem.
- Example:

```
public int Factorial(int n)
{
    if (n == 1) return 1;
    return n * Factorial(n - 1);
}
```

Object-Oriented Programming (OOP) Concepts in C#

Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of “objects,” which contain data and methods to operate on that data. The four fundamental principles of OOP are:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

---

1. Encapsulation

Definition:

- Encapsulation is the bundling of data (fields) and methods (functions) into a single unit (class) while restricting direct access to the internal state.

Key Features:

1. **Access Modifiers**:
   – Control visibility of class members.
   – Types:
     - **public**: Accessible from anywhere.
     - **private**: Accessible only within the class.

    - **protected**: Accessible within the class and its derived classes.
    - **internal**: Accessible within the same assembly.

  - Example:

```
class Employee
{
    private int _id; // Private field

    public int ID   // Public property
    {
        get { return _id; }
        set { _id = value; }
    }
}
```

2. **Properties**:

   - Provide controlled access to private fields.

   - Example:

```
public class Product
{
    private double price;

    public double Price
    {
        get { return price; }
        set
        {
            if (value > 0)
                price = value;
        }
    }
}
```

3. **Benefits**:

   - Protects data integrity.
   - Hides implementation details.

2. Abstraction

Definition:

- Abstraction is the process of hiding the implementation details while exposing only the essential features of an object.

Implementation in C#:

1. **Abstract Classes**:

   – Cannot be instantiated.

   – Contain both abstract (no implementation) and non-abstract (with implementation) methods.

   – Example:

```
abstract class Shape
{
   public abstract void Draw(); // Abstract method
   public void Display()       // Non-abstract method
   {
      Console.WriteLine("Displaying Shape");
   }
}

class Circle : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing Circle");
   }
}
```

2. **Interfaces**:

   – Define a contract that implementing classes must follow.

   – All methods are abstract by default.

   – Example:

```
interface IAnimal
{
    void Speak(); // Abstract method
}

class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Bark");
    }
}
```

3. **Benefits**:

   – Simplifies code by focusing on what an object does rather than how it does it.
   – Promotes flexibility and scalability.

---

3. Inheritance

Definition:

- Inheritance is a mechanism where one class (child/derived) inherits the properties and methods of another class (parent/base).

Implementation in C#:

1. **Syntax**:

```
class BaseClass
{
    public void Display()
    {
        Console.WriteLine("Base Class Method");
    }
}

class DerivedClass : BaseClass
{
    public void Show()
    {
        Console.WriteLine("Derived Class Method");
    }
}
```

2. **Types of Inheritance**:

   - **Single Inheritance**: A class inherits from one base class.
   - **Multilevel Inheritance**: A class inherits from a derived class.
   - **Hierarchical Inheritance**: Multiple classes inherit from one base class.
   - **C# Limitation**: C# does **not** support multiple inheritance but allows implementing multiple interfaces.

3. **Accessing Parent Members**:

   - Use the **base** keyword to access parent class methods or constructors.

   - Example:

     ```
     class BaseClass
     {
         public void Greet() => Console.WriteLine("Hello from Base Class");
     }

     class DerivedClass : BaseClass
     {
         public void GreetDerived()
         {
             base.Greet(); // Call parent method
             Console.WriteLine("Hello from Derived Class");
         }
     }
     ```

4. **Benefits**:

   - Promotes code reuse.
   - Establishes a parent-child relationship.

---

4. Polymorphism

Definition:

- Polymorphism allows a single method, property, or operator to have multiple forms.

Types of Polymorphism:

1. **Compile-Time (Static) Polymorphism**:
   - Achieved through **method overloading** or **operator overloading**.
   - Example:

```
class Calculator
{
    public int Add(int a, int b) => a + b;
    public double Add(double a, double b) => a + b;
}
```

2. **Run-Time (Dynamic) Polymorphism**:
   - Achieved through **method overriding**.
   - Example:

```
class Animal
{
    public virtual void Speak() => Console.WriteLine("Animal speaks");
}

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

Benefits:

- Enhances flexibility and code readability.
- Supports dynamic behavior.

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 & Accessibility Modifiers

Scope in C#

- **Scope** refers to the region of the program where a variable, method, class, or any identifier is accessible.
- C# defines different types of scopes based on where and how variables and methods are declared.

Types of Scopes

1. **Local Scope**:
   - Variables declared inside a method or block.
   - Accessible only within that method/block.
   - Example:

```
void MyMethod()
{
    int x = 10; // Local variable
    Console.WriteLine(x); // Accessible within MyMethod
}
```

2. **Method Scope**:
   - Variables are declared inside a method and can only be accessed within that method.
   - Example:

```
void Display()
{
    string message = "Hello";
    Console.WriteLine(message); // Accessible within Display method
}
```

3. **Class Scope**:

   - Variables declared inside a class but outside of any method.
   - Can be accessed by all methods within the class.
   - Example:

```
class MyClass
{
    int count = 5; // Class scope

    public void ShowCount()
    {
        Console.WriteLine(count); // Accessible within the class
    }
}
```

4. **Global Scope**:

   - Variables or methods declared at the class level and can be accessed from anywhere in the class or program (if public).

Accessibility Modifiers in C#

- **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; }
```

2. **private**:

- The member is only accessible within the class where it is declared.
- Default for class members.
- Example:

```
private int number;
```

3. **protected**:

- The member is accessible within the class and by derived (child) classes.
- Example:

```
protected int Id;
```

4. **internal**:

- The member is accessible within the same assembly (project) but not outside it.
- Example:

```
internal void Display()
{
   Console.WriteLine("Inside the assembly");
}
```

5. **protected internal**:

- The member is accessible from within the same assembly and by derived classes.
- Example:

```
protected internal int Counter;
```

6. **private protected**:

- The member is accessible only within the same class or derived classes within the same assembly.
- Example:

```
private protected int score;
```

What is a Namespace?

- A **namespace** is a container for classes, structs, enums & interfaces in C#. It helps organize the code into logical groups to avoid name conflicts.

Syntax for Declaring a Namespace:

```
namespace MyApplication
{
   class MyClass
   {
      // Class code here
   }
}
```

Using Namespaces:

- To access a class or method from a different namespace, you can either use a fully qualified name or the **using** directive.

Example:

```
using MyApplication;

class Program
{
   static void Main()
   {
      MyClass obj = new MyClass();  // Access MyClass from MyApplication namespace
   }
}
```

System Namespace:

- The **System** namespace is a predefined namespace that contains basic classes used by many programs, such as **Console**, **String**, **Int32**, etc.

  ```
  using System;

  class Program
  {
  ```

```
    static void Main()
    {
        Console.WriteLine("Hello, World!");  // Access System.Console
    }
}
```

The .NET Library

- The **.NET Library** (also called the **.NET Framework Class Library** or **BCL** for Base Class Library) is a collection of reusable classes and functions that are available to C# developers.

Common .NET Libraries:

1. **System Namespace**:
   - Contains fundamental types like **Console**, **String**, **Collections**, etc.
   - Example:

   ```
   using System;
   ```

2. **System.Collections Namespace**:
   - Contains classes for data collections such as **List<T>**, **Dictionary<K,V>**, **Queue<T>**.
   - Example:

   ```
   using System.Collections.Generic;
   ```

3. **System.IO Namespace**:
   - Contains classes for reading from and writing to files and data streams.
   - Example:

   ```
   using System.IO;
   ```

4. **System.Linq Namespace**:
   - Contains classes for LINQ (Language Integrated Query) operations.
   - Example:

   ```
   using System.Linq;
   ```

5. **System.Threading Namespace**:
   - Provides classes and methods for multithreading and parallel programming.
   - Example:

```
using System.Threading;
```

Creating & Adding Reference to Assemblies

What is an Assembly?

- An **assembly** is a compiled code library used by the .NET runtime. Assemblies can be in the form of **.exe** or **.dll** files.
- Assemblies contain one or more namespaces and types like classes, interfaces, structs, etc.

Creating an Assembly in C#

- When you compile a C# program, the output file (either **.exe** or **.dll**) is the assembly.
- Example of creating an assembly:

```
// File: MyLibrary.cs
public class MyLibraryClass
{
    public void PrintMessage()
    {
        Console.WriteLine("Hello from MyLibrary!");
    }
}
```

Compiling the Assembly

1. In **Visual Studio**: Press **Ctrl+Shift+B** to build the project and generate the assembly (**.dll** or **.exe**).
2. Using **Command Line**: You can compile a C# file into an assembly using the C# compiler **csc**:

```
csc /target:library MyLibrary.cs
```

Adding References to Assemblies

- **References** allow you to use classes, methods, and other members from external assemblies.

Adding Reference in Visual Studio:

1. Right-click on the project in **Solution Explorer**.
2. Click **Add** → **Reference**.
3. In the Reference Manager, choose **Assemblies** or **Browse** to add a custom assembly.

Adding a Reference Programmatically:

- You can add references to assemblies using **using** directives, which enable you to use types from referenced assemblies.

```
using MyLibrary;

class Program
{
   static void Main()
   {
      MyLibraryClass obj = new MyLibraryClass();
      obj.PrintMessage();
   }
}
```

Adding External DLL References:

- If you want to reference external **.dll** files:
  1. Right-click **References** in the Solution Explorer.
  2. Choose **Add Reference** and browse to the **.dll** file.
  3. You can now use the types defined in that DLL.

Assembly Versioning:

- Assemblies can have versions, which helps in managing updates and compatibility.

- Example of versioning:

```
MyLibrary.dll -> Version 1.0.0.0
MyLibrary.dll -> Version 1.1.0.0
```

Working with Collections

What are Collections in C#?

- Collections in C# are classes that provide a way to store and manage a group of related objects.
- Collections are used to handle objects that are logically related, such as lists, queues, or dictionaries.
- C# provides several built-in collection classes under the **System.Collections** and **System.Collections.Generic** namespaces.

Types of Collections

1. **Array**:

   – Fixed-size collection of elements of the same type.

   – Syntax for declaring an array:

   ```
   int[] numbers = new int[5];
   numbers[0] = 10;
   numbers[1] = 20;
   ```

2. **List**:

   – A generic collection that can grow or shrink dynamically.

   – Provides methods for adding, removing, and accessing elements.

   – Syntax:

   ```
   List<int> list = new List<int>();
   list.Add(10);
   list.Add(20);
   ```

3. **Dictionary<TKey, TValue>**:

   – A collection of key-value pairs.

   – Allows fast lookups by key.

   – Syntax:

   ```
   Dictionary<int, string> dict = new Dictionary<int, string>();
   dict.Add(1, "One");
   dict.Add(2, "Two");
   ```

4. **Queue**:

   – A collection representing a first-in, first-out (FIFO) list of objects.

   – Syntax:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("First");
queue.Enqueue("Second");
```

5. **Stack**:

   – A collection representing a last-in, first-out (LIFO) list of objects.

   – Syntax:

```
Stack<string> stack = new Stack<string>();
stack.Push("First");
stack.Push("Second");
```

Collection Methods

- Common methods used with collections include:
  - **Add()**: Adds an element.
  - **Remove()**: Removes an element.
  - **Contains()**: Checks if an element exists.
  - **Clear()**: Removes all elements.
  - **Count**: Returns the number of elements.

Enumerations

What is an Enumeration?

- An **enumeration (enum)** is a special value type that defines a set of named constants.
- Enums are used when you need a predefined set of values, like days of the week or directions.

Declaring an Enum

- Enums are declared using the **enum** keyword.

- Syntax:

```
enum Days
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```

Working with Enums

- By default, the first value of an enum has a value of **0**, and each subsequent value is incremented by 1.
- You can change the default values by explicitly assigning them:

```
enum Days
{
   Sunday = 1,
   Monday = 2,
   Tuesday = 3
}
```

- **Using Enums**:
   - You can use enums in switch statements, comparisons, and as variables.
   - Example:

```
Days today = Days.Monday;

switch (today)
{
   case Days.Monday:
      Console.WriteLine("Start of the work week.");
      break;
   case Days.Sunday:
      Console.WriteLine("It's the weekend!");
```

```
            break;
        }
```

- **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);
}
```

Data Table

What is a Data Table?

- A **DataTable** is an in-memory representation of a single table of data.
- It is part of the **System.Data** namespace and is used in ADO.NET to store data retrieved from a database.

Creating a Data Table

- You can create a DataTable by defining columns and adding rows.
- Syntax:

```
DataTable dt = new DataTable();
dt.Columns.Add("ID", typeof(int));
dt.Columns.Add("Name", typeof(string));

dt.Rows.Add(1, "John");
dt.Rows.Add(2, "Jane");
```

Working with DataTable

- You can perform various operations on a DataTable, like filtering, sorting, and accessing individual rows.

- Example:

```
foreach (DataRow row in dt.Rows)
{
   Console.WriteLine($"ID: {row["ID"]}, Name: {row["Name"]}");
}
```

Using DataTable with DataAdapter

- A **DataAdapter** is used to fill a DataTable with data from a database.

```
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Users", connection);
DataTable dt = new DataTable();
adapter.Fill(dt);
```

Accessing and Modifying Data in DataTable

- You can access a specific row or column in a DataTable using indexers.

```
DataRow row = dt.Rows[0]; // Access the first row
Console.WriteLine(row["Name"]); // Access the "Name" column of the first row
```

Exception Handling

What is Exception Handling?

- Exception handling in C# provides a way to handle runtime errors and ensure that the program can continue to execute after an error occurs.
- It uses **try**, **catch**, **finally** blocks to manage exceptions.

Syntax of Exception Handling

```
try
{
   // Code that might throw an exception
}
catch (ExceptionType ex)
{
   // Code to handle the exception
}
```

```
finally
{
   // Code that runs regardless of whether an exception was thrown
}
```

Exception Types

- **Exception**: The base class for all exceptions.
- Common derived classes include:
  - **System.NullReferenceException**: Thrown when you try to access a null object.
  - **System.IO.IOException**: Thrown when an I/O error occurs (file not found, etc.).
  - **System.DivideByZeroException**: Thrown when attempting to divide by zero.

Throwing Exceptions

- You can manually throw exceptions using the **throw** keyword:

```
if (age < 0)
{
   throw new ArgumentOutOfRangeException("Age cannot be negative.");
}
```

Handling Multiple Exceptions

- You can catch different types of exceptions using multiple **catch** blocks:

```
try
{
   int result = 10 / 0;
}
catch (DivideByZeroException ex)
{
   Console.WriteLine("Cannot divide by zero.");
}
catch (Exception ex)
{
   Console.WriteLine("An error occurred: " + ex.Message);
}
```

Finally Block

- The **finally** block is optional and runs after the **try** and **catch** blocks, regardless of whether an exception was thrown.

```
try
{
   // Code
}
catch (Exception ex)
{
   // Handle exception
}
finally
{
   // Code that always runs (e.g., cleanup code)
}
```

Custom Exceptions

- You can create custom exceptions by inheriting from the **Exception** class.

```
public class InvalidAgeException : Exception
{
   public InvalidAgeException(string message) : base(message) { }
}
```

- Example usage:

```
throw new InvalidAgeException("Age must be between 1 and 100.");
```

Best Practices for Exception Handling

- Use exceptions to handle exceptional, unforeseen errors, not for regular control flow.
- Catch specific exceptions rather than a general **Exception** class.
- Avoid empty **catch** blocks; log the exception or rethrow it.
- Always clean up resources in the **finally** block.

Different Project Types in C#

In C#, there are several types of projects that you can create depending on the application you are developing. These projects vary in functionality and target environments. Here are some common types:

1. Console Application

- **Description**: A console application is a simple application that runs in a command-line environment. It's a text-based interface where the user interacts with the application through the console window.
- **Uses**: Suitable for utilities, learning programming basics, or backend processing.
- **Example**: Simple calculators, command-line tools.

2. Windows Forms Application

- **Description**: Windows Forms applications are used to create graphical user interfaces (GUIs) on Windows operating systems. It uses controls like buttons, textboxes, and labels to build the interface.
- **Uses**: Desktop applications like media players, text editors.
- **Example**: A simple text editor or a calculator with GUI.

3. WPF (Windows Presentation Foundation) Application

- **Description**: WPF is used for building modern Windows desktop applications with rich graphical interfaces. It supports more advanced graphics, animations, and data binding.
- **Uses**: Desktop applications with complex UIs, advanced graphics.
- **Example**: Complex desktop applications like accounting software or graphical design tools.

4. ASP.NET Core Application

- **Description**: ASP.NET Core is used for creating web applications. It is a modern, cross-platform framework for building web applications and APIs.
- **Uses**: Websites, web services, and web APIs.
- **Example**: E-commerce sites, RESTful APIs.

5. Class Library

- **Description**: A class library project is a collection of classes and functions that can be used by other applications.
- **Uses**: Creating reusable libraries that can be shared across different applications.
- **Example**: Utility libraries, frameworks, or custom class libraries for an application.

6. Xamarin Application

- **Description**: Xamarin is used for building mobile applications for Android, iOS, and Windows using a single C# codebase.
- **Uses**: Cross-platform mobile applications.
- **Example**: Mobile apps like social media clients, task management apps.

7. Azure Functions

- **Description**: Azure Functions allows you to run small pieces of code (functions) in the cloud without having to manage the underlying infrastructure.
- **Uses**: Serverless applications, cloud-triggered functions.
- **Example**: Event-driven applications that respond to cloud events.

8. Blazor Application

- **Description**: Blazor is a framework for building interactive web UIs using C# instead of JavaScript. It can run on the client-side via WebAssembly or server-side.
- **Uses**: Interactive web applications with C# on both server and client sides.
- **Example**: Web-based dashboards, e-commerce platforms.

Working with String Class in C#

Strings are one of the most commonly used data types in C#. The **String** class in C# is part of the **System** namespace and provides various methods to manipulate strings.

Common String Methods:

- **Length**: Returns the number of characters in a string.

```
string str = "Hello";
int length = str.Length; // 5
```

- **Substring()**: Extracts a substring from a given string.

```
string str = "Hello World";
string sub = str.Substring(6, 5); // "World"
```

- **Replace()**: Replaces all occurrences of a substring with another substring.

```
string str = "Hello World";
string newStr = str.Replace("World", "C#"); // "Hello C#"
```

- **ToUpper() / ToLower()**: Converts all characters of a string to uppercase or lowercase.

```
string str = "Hello";
string upperStr = str.ToUpper(); // "HELLO"
string lowerStr = str.ToLower(); // "hello"
```

- **Trim()**: Removes whitespace from both ends of a string.

```
string str = "  Hello  ";
string trimmed = str.Trim(); // "Hello"
```

- **Split()**: Splits a string into an array of substrings based on a delimiter.

```
string str = "apple,banana,grape";
string[] fruits = str.Split(','); // ["apple", "banana", "grape"]
```

- **IndexOf()**: Returns the index of the first occurrence of a specified substring.

```
string str = "Hello World";
int index = str.IndexOf("World"); // 6
```

- **Contains()**: Checks if a string contains a specific substring.

```
string str = "Hello World";
bool contains = str.Contains("World"); // true
```

- **Concat()**: Concatenates multiple strings into one.

```
string str1 = "Hello";
string str2 = "World";
string result = string.Concat(str1, " ", str2); // "Hello World"
```

Working with DateTime Class in C#

The **DateTime** class is part of the **System** namespace and provides functionality for working with dates and times.

Common DateTime Methods:

- **Now**: Gets the current date and time.

```
DateTime now = DateTime.Now;
Console.WriteLine(now); // Prints the current date and time
```

- **UtcNow**: Gets the current date and time in UTC (Coordinated Universal Time).

```
DateTime utcNow = DateTime.UtcNow;
```

- **Today**: Gets the current date with the time set to midnight.

```
DateTime today = DateTime.Today;
```

- **AddDays()**: Adds a specified number of days to a **DateTime**.

```
DateTime date = DateTime.Now;
DateTime newDate = date.AddDays(10); // Adds 10 days
```

- **AddMonths()**: Adds a specified number of months to a **DateTime**.

```
DateTime date = DateTime.Now;
DateTime newDate = date.AddMonths(3); // Adds 3 months
```

- **ToString()**: Converts a **DateTime** to a string with a specified format.

```
DateTime date = DateTime.Now;
string formattedDate = date.ToString("MM/dd/yyyy");
```

- **Parse()**: Converts a string representation of a date and time to a **DateTime** object.

```
DateTime date = DateTime.Parse("2024-11-27");
```

- **Compare()**: Compares two **DateTime** values and returns an integer indicating whether the first is earlier, the same, or later than the second.

```
DateTime date1 = DateTime.Now;
DateTime date2 = DateTime.Now.AddHours(1);
int result = DateTime.Compare(date1, date2); // -1 if date1 < date2
```

- **Subtract()**: Subtracts one **DateTime** from another, returning a **TimeSpan** object.

```
DateTime date1 = DateTime.Now;
DateTime date2 = DateTime.Now.AddDays(2);
TimeSpan difference = date2.Subtract(date1);
```

Basic File Operations in C#

The **System.IO** namespace provides various classes to work with files, such as **File**, **FileInfo**, **StreamReader**, and **StreamWriter**.

Common File Operations:

1. Reading Files:

- **StreamReader**: Used to read text from a file.

```
using (StreamReader reader = new StreamReader("file.txt"))
{
   string content = reader.ReadToEnd();
   Console.WriteLine(content);
}
```

- **File.ReadAllText()**: Reads the entire content of a file.

```
string content = File.ReadAllText("file.txt");
```

2. Writing Files:

- **StreamWriter**: Used to write text to a file.

```
using (StreamWriter writer = new StreamWriter("file.txt"))
{
   writer.WriteLine("Hello, World!");
}
```

- **File.WriteAllText()**: Writes text to a file, creating the file if it doesn’t exist.

```
File.WriteAllText("file.txt", "Hello, World!");
```

3. File Existence:

- **File.Exists()**: Checks if a file exists.

```
bool exists = File.Exists("file.txt");
```

4. Copying Files:

- **File.Copy()**: Copies a file to a new location.

```
File.Copy("source.txt", "destination.txt");
```

5. Deleting Files:

- **File.Delete()**: Deletes a specified file.

```
File.Delete("file.txt");
```

6. Appending Text to a File:

- **File.AppendAllText()**: Appends text to a file.

```
File.AppendAllText("file.txt", "Appended Text");
```

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.

Project Structure:

```
/EmptyWebApp
 ├── /App_Data
 ├── /Content
 ├── /Scripts
 ├── /Views
 ├── Global.asax
 ├── Web.config
```

- **App_Data**: Directory for database files, data, or other data-related resources.
- **Content**: Stores static files like CSS and images.
- **Scripts**: Stores JavaScript files.
- **Views**: The folder where your Razor views reside (if you add MVC).

Key Files:

1. **Global.asax**:
   - Handles application-level events, such as Application_Start, Application_End, etc.

```
<%@ Application Language="C#" Inherits="System.Web.HttpApplication" %>

<script runat="server">
void Application_Start(object sender, EventArgs e) {
  // Code that runs on application startup
}
</script>
```

2. **Web.config**:

   – Configuration file for the web application, like database connection strings, routing, and security settings.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <!-- Your app settings -->
  </appSettings>
  <connectionStrings>
    <!-- Your connection strings -->
  </connectionStrings>
</configuration>
```

---

2. Web Forms Application

Overview:

Web Forms is a traditional framework for building web pages in ASP.NET. It uses a drag-and-drop approach with controls like TextBoxes, Buttons, and Grids. This template is commonly used for enterprise-level web apps and forms-based sites.

Project Structure:

```
/WebFormsApp
├── /App_Data
├── /Content
├── /Scripts
├── /Pages
│   └── Default.aspx
├── Global.asax
├── Web.config
```

Key Files:

1. **Default.aspx**:

   – A typical Web Forms page containing HTML markup and server controls.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebFormsApp._Default" %>

<html>
  <body>
    <form id="form1" runat="server">
      <div>
        <asp:Label runat="server" ID="Label1" Text="Hello, Web Forms!" />
        <asp:Button runat="server" Text="Click Me" OnClick="Button1_Click" />
      </div>
    </form>
  </body>
</html>
```

2. **Default.aspx.cs**:

   – The code-behind file where you handle server-side logic, such as button clicks.

```
using System;

namespace WebFormsApp
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Button1_Click(object sender, EventArgs e)
        {
            Label1.Text = "Button clicked!";
        }
    }
}
```

3. MVC Web Application

Overview:

MVC (Model-View-Controller) is a design pattern that separates application logic into three components: Model (data), View (UI), and Controller (business logic). It's suitable for applications that require more complex and maintainable code.

Project Structure:

```
/MvcApp
  ├── /Controllers
  │    └── HomeController.cs
  ├── /Models
  │    └── WeatherForecast.cs
  ├── /Views
  │    └── /Home
  │         └── Index.cshtml
  ├── Global.asax
  ├── Web.config
```

Key Files:

1. **HomeController.cs**:

   – The controller that handles HTTP requests and returns appropriate views.

```
using System.Web.Mvc;

namespace MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            var model = new WeatherForecast { Date = "2024-11-27", Summary = "Sunny",
TemperatureC = 22 };
            return View(model);
        }
    }
}
```

2. **Index.cshtml**:

   – The Razor view that represents the HTML page for the Index action.

```
@model MvcApp.Models.WeatherForecast

<h1>Weather Forecast</h1>
```

```
<p>Date: @Model.Date</p>
<p>Temperature: @Model.TemperatureC °C</p>
<p>Summary: @Model.Summary</p>
```

3. **Web.config**:

   – Contains configuration settings for routing, security, etc.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.7.2" />
  </system.web>
</configuration>
```

---

4. Web API Application

Overview:

Web API applications allow you to create RESTful APIs that can be consumed by various clients. It’s commonly used for building back-end services.

Project Structure:

```
/WebAPIApp
  ├── /Controllers
  │     └── WeatherController.cs
  ├── /Models
  │     └── WeatherForecast.cs
  ├── Global.asax
  ├── Web.config
```

Key Files:

1. **WeatherController.cs**:

   – The API controller that handles HTTP requests and returns data in JSON format.

```
using System.Collections.Generic;
using System.Web.Http;
```

```
namespace WebAPIApp.Controllers
{
    public class WeatherController : ApiController
    {
        public IEnumerable<WeatherForecast> Get()
        {
            return new List<WeatherForecast>
            {
                new WeatherForecast { Date = "2024-11-27", TemperatureC = 20, Summary =
"Sunny" },
                new WeatherForecast { Date = "2024-11-28", TemperatureC = 15, Summary =
"Cloudy" }
            };
        }
    }
}
```

2. **WeatherForecast.cs** (Model):

```
namespace WebAPIApp.Models
{
    public class WeatherForecast
    {
        public string Date { get; set; }
        public int TemperatureC { get; set; }
        public string Summary { get; set; }
    }
}
```

---

5. Single Page Application (SPA)

Overview:

SPA projects use client-side technologies like Angular, React, or Vue.js to create dynamic, single-page applications. The back-end is usually an API server that handles HTTP requests.

Project Structure:

```
/SPAApp
├── /ClientApp (Angular/React/Vue)
├── /Controllers
│   └── ApiController.cs
├── Web.config
```

Key Files:

1. **ApiController.cs** (Web API Controller):

```
using System.Collections.Generic;
using System.Web.Http;

namespace SPAApp.Controllers
{
    public class ApiController : ApiController
    {
        public IEnumerable<string> Get()
        {
            return new string[] { "Value1", "Value2" };
        }
    }
}
```

2. **Web.config**:

   – Configuration settings, including API routes and security for the back-end.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.7.2" />
  </system.web>
</configuration>
```