

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 + (backtick)
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.
-

Customization Tips

- **Change Theme:** Tools > Options > Environment > General > Color Theme.
 - **Extensions:** Install tools like ReSharper or Prettier from the Extensions Marketplace.
 - **Keyboard Mapping:** Customize shortcuts via Tools > Options > Keyboard.
-

Supported Workloads

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

Getting Started

- **Download:** [Visual Studio 2022](#)
 - **System Requirements:**
 - OS: Windows 10/11
 - RAM: 4 GB (8 GB or more recommended)
 - Disk Space: 20-50 GB depending on workloads.
-

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.
-

Example with Comments

```
using System; // Import System namespace

// Define a namespace for the program
namespace ExampleNamespace
{
    // Define the Program class
    class Program
    {
        // Entry point of the application
        static void Main(string[] args)
        {
            // Print a message to the console
            Console.WriteLine("Welcome to C#!");
        }
    }
}
```

Common Errors

1. **Missing Semicolon:**
 - Error: ; expected.
 - Fix: Ensure every statement ends with a ;.
 2. **Case Sensitivity:**
 - Error: Console or Main spelled incorrectly.
 - Fix: Ensure correct capitalization.
 3. **Missing Main Method:**
 - Error: Program has no entry point.
 - Fix: Define a Main method as the entry point.
-

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

There are many other as shown in the demo

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;
```

Namespace & .NET Library

What is a Namespace?

- A **namespace** is a container for classes, structs, enums, interfaces, and delegates 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
    }
}
```

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**) 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
```

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);  
}
```
