**C#**

* **Introduction to C#**

C# (pronounced "C-sharp") is a **modern**, **object-oriented**, and **type-safe programming language** developed by **Microsoft**. It runs on the **.NET platform** and is used to build a wide variety of applications like web apps, desktop apps, mobile apps, games and much more.

* **Basic C# Program Structure**

using System;

namespace HelloWorld

{

class Program

{

static void Main(string[] args)

{

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

}

}

}

1. **using** directives: This tells the compiler to include the **System** namespace, which contains basic classes like Console. You can include multiple namespaces.
2. **namespace** Declaration: A **namespace** is used to **organize code** and **avoid name conflicts**. It groups related classes, interfaces, and methods.
3. **class** Declaration: C# is an **object-oriented language**, so all code must be inside a **class** (or struct).
4. **main** Method: This is the **entry point** of a C# program. When the program starts, execution begins here. It is compulsory that the Main method be inside a class or struct.
5. **statements**: A **statement** is an instruction to perform an action. Console.WriteLine() prints text to the console. Every statement ends with a semicolon (;).

* **What Are Top-Level Statements in C#?**

Top-level statements are a feature introduced in C# 9.0 (along with .NET 5) that allow you to write C# code without explicitly defining a Main method or a class in simple programs. Top-level statements are designed to Simplify small programs, such as scripts, tutorials, or console apps. If you define your own Main, top-level statements are not allowed in that file.

* **Files, projects and solution in C#**

1. **Code files(.cs):** A code file in C# is a plain text file with the **.cs extension** that contains C# source code. These files are the building blocks of your application - they define the logic, structure, and behaviour of your program. **A single .cs file can include namespace, class, struct, interface, methods, etc.** There are multiple code files in a project.
2. **Project:** A project is a fundamental unit of organization for building and managing an application, library, or service. A C# project includes **code files (.cs)**, Project settings (like **build configuration**), References to **libraries** or other projects, Resources like images, config files, etc. Projects are defined by a project file with the **.csproj extension**, which controls how the application is built.
   * **Types of C# Projects**

| **Project Type** | **Description** |
| --- | --- |
| **Console App** | Command-line program (e.g., tools, scripts) |
| **Web App** | ASP.NET-based website or API |
| **Class Library** | Reusable code (DLL) for other projects |
| **Windows Forms/WPF** | GUI desktop applications |
| **Blazor App** | Web app using C# instead of JavaScript |
| **Worker Service** | Background service (like daemons) |

1. **Solution:** A Solution in C# is a container that organizes **one or more related projects**. It helps developers manage, build, and debug multiple components of a larger system in a unified way. The solution file has a **.sln extension**.
   * A .sln (solution) file is a plain text file that defines:
2. The list of projects in the solution
3. The paths to each project’s .csproj file
4. The build configuration (e.g., Debug or Release)
5. Other metadata like project GUIDs, solution folders, and dependencies

* **Data types in C#**
* **What Are Variables?**

Variables are named storage locations in memory that hold data. Each variable has a data type, which defines what kind of data it can store, how much space it occupies, what operations can be performed on it.

* **What is Data Type?**

A **data type** defines the **kind of data** a variable can hold. It tells the compiler:

* What type of value the variable stores (numbers, text, true/false, etc.)
* How much memory to allocate
* What operations are allowed on that data

The most common data types are:

| **Data Type** | **Size** | **Description** | **Example Values** |
| --- | --- | --- | --- |
| int | 4 bytes | Integer numbers | 123, -456 |
| long | 8 bytes | Large integers | 1234567890123 |
| float | 4 bytes | Single-precision floating point | 3.14f |
| double | 8 bytes | Double-precision floating point | 3.14159 |
| string | 2 bytes per char | Stores a sequence of characters | "Hello World! " |
| char | 2 bytes | Single Unicode character | 'A', 'z' |
| bool | 1 byte | Boolean true/false | true, false |

* **What is Type casting?**

Type casting is when you assign a value of one data type to another type. In C#, there are two types of casting:

* **Implicit Casting** (automatically) - Automatically performed by the compiler when converting a smaller or compatible type to a larger or more general type. No data loss.  
  char -> int -> long -> float -> double
* **Explicit Casting** (manually) - You must explicitly specify the conversion using a cast operator when converting from a larger or more complex type to a smaller or more specific type. It can lead data loss.  
  double -> float -> long -> int -> char

Explicit Type Conversion with Helper Methods:

1. Using Convert class: The Convert class provides many static methods to convert from one type to another safely.

1. ToInt32(...): Converts to int (32-bit)
2. ToString(...): Converts any value to a string
3. ToBoolean(...): Converts to bool
4. ToChar(...): Converts to char

2. Using Parse() Methods: Many data types have Parse methods to convert strings to their types.

int.Parse(string)

3. Using TryParse() Methods: Safe version of Parse(). Returns true if successful; false otherwise.

int.TryParse(string, out int result)

* **Operators in C#**

Operators in C# are special symbols or keywords used to perform operations on one or more operands. These operations can include mathematical calculations, comparisons, logical evaluations, bitwise manipulations, and more. C# categorizes its operators into several groups based on their functionality:

1. Arithmetic Operators: Used for mathematical calculations

Ex: + (Addition), - (Subtraction), \* (Multiplication), / (Division), and % (Modulus - returns the remainder of a division).

1. Assignment Operators: Used to assign values to variables

Ex: = (Simple assignment), +=, -=, \*=, /=, %=

1. Comparison Operators (Relational Operators): Used to compare two values and return a boolean result (true or false)

Ex: == (Equality), != (Inequality), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to)

1. Logical Operators: Used to combine or manipulate boolean expressions

Ex: && (Logical AND), || (Logical OR), ! (Logical NOT)

1. Bitwise and Shift Operators: Used to perform operations at the bit level on integral types

Ex: & (Bitwise AND), | (Bitwise OR), ^ (Bitwise XOR), ~ (Bitwise NOT/Complement), << (Left shift), >> (Right shift)

1. Increment and Decrement Operators: Used to increase or decrease the value of a variable by one: ++ (increment) and -- (decrement).
2. Ternary Operator (Conditional Operator): A shorthand for an if-else statement, taking three operands: condition ? expression\_if\_true : expression\_if\_false;
3. Null-Coalescing Operator: ?? (Returns the left-hand operand if it's not null; otherwise, returns the right-hand operand)
4. Type-Testing and Conversion Operators: Used for type checking and safe casting.

Ex:

* is (Checks if an object is compatible with a given type)
* as (Performs a safe type conversion, returning null on failure)
* typeof (Returns the System.Type object for a given type)
* Cast operator () (Explicit type conversion)

1. Member Access Operator: Used to access members (fields, methods, properties) of a class or object.

* **Array in C#**

Arrays in C# are data structures that hold multiple variables of the same type accessed by indices. They are reference types whose elements are stored contiguously.

An array has a **rank** that determines the number of indices associated with each array element. The rank of an array is also referred to as the **dimensions of the array**. An array with a rank of one is called a single-dimensional array. An array with a rank greater than one is called a multi-dimensional array.

* **Types of Arrays in C#**

1. One dimensional array: It's the simplest form of array that stores elements in a linear list (like a row).

int[] numbers = new int[5]; // Declaration and memory allocation

int[] values = { 1, 2, 3, 4, 5 }; // Initialization with values

string[] fruits = { "Apple", "Banana", "Mango" };

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

Console.WriteLine(fruits[i]);

1. Multi-Dimensional Array: These arrays have more than one dimension (like a matrix or grid).

int[,] matrix = new int[2, 3]; // 2 rows, 3 columns

int[,] grid = { { 1, 2, 3 }, { 4, 5, 6 } }; // initialization

int[,] table = { {10, 20}, {30, 40} };

for (int i = 0; i < table.GetLength(0); i++)

for (int j = 0; j < table.GetLength(1); j++)

Console.WriteLine($"Element at [{i},{j}] = {table[i,j]}");

1. Jagged Array (Array of Arrays): A jagged array is an array where each element is itself an array. Unlike multi-dimensional arrays, jagged arrays can have **rows of different lengths**.

int[][] jaggedArray = new int[3][]; // 3 elements, each is an array

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

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

jaggedArray[2] = new int[] { 6 };

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

{

for (int j = 0; j < jaggedArray[i].Length; j++)

{

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

}

Console.WriteLine();

}

* **Array property in C#**

1. Length: Returns the total number of elements

Console.WriteLine(arrayName.Length);

1. Rank: Returns the number of dimensions

Console.WriteLine(arrayName.Rank);

1. GetLength(dim): Gets the size of a specific dimension

Console.WriteLine(arrayName.GetLength(0)); // Size of the first dimension

* **Array methods in C#**

1. Array.Sort(): Sorts elements in ascending order.

Array.Sort(new int[] { 3, 1, 2 }); // Result: {1, 2, 3}

1. Array.Reverse(): Reverses the order of elements.

Array.Reverse(new int[] { 1, 2, 3 }); // Result: {3, 2, 1}

1. Array.IndexOf(array, value): Returns the index of the **first occurrence** of the value.

Console.WriteLine(Array.IndexOf(new int[] { 10, 20, 30 }, 20)); // Output: 1

1. Array.LastIndexOf(array, value): Returns the index of the **last occurrence** of the value.

Console.WriteLine(Array.LastIndexOf(new int[] { 1, 2, 1 }, 1)); // Output: 2

1. Array.Clear(array, startIndex, count): Resets the range of elements to the default value.

var a = new int[] { 1, 2, 3 };

Array.Clear(a, 1, 2); // a becomes {1, 0, 0}

1. Array.Copy(source, destination, length): Copies a range of elements.

var src = new int[] { 1, 2, 3 };

var dest = new int[3];

Array.Copy(src, dest, 3); // dest = {1, 2, 3}

* **Method in C#**

A method is a block of code that performs a specific task. You define it once and can reuse it multiple times. It provides better organization of code, easier debugging and maintenance. And also gives encapsulation of logic.

Basic structure of method:

[access\_modifier] [return\_type] MethodName([parameters]) {

// Method body

}

* **Access modifiers of method**

| **Modifier** | **Description** |
| --- | --- |
| public | Accessible from anywhere |
| private | Accessible only within the same class |
| protected | Accessible within the class and its derived classes |
| internal | Accessible within the same assembly |
| protected internal | Combination of protected + internal |
| private protected | Accessible in containing class or derived class within same assembly |

* **Parameters of method**

1. Value Parameters: Passed by value (default)

void PrintName(string name) {

Console.WriteLine(name);

}

1. ref Parameter: Passes by reference (original value can change). The variable must be initialized before it is passed to the method.

void Square(ref int x) {

x = x \* x;

}

1. out Parameter: Passes by reference, must be assigned in method. The variable does not need to be initialized before passing.

void GetData(out int number) {

number = 100;

}

1. Default/optional Parameter: You can give default values

void Greet(string name = "Guest") {

Console.WriteLine("Hello " + name);

}

1. Named Arguments: Call method using parameter names

PrintDetails(name: "John", age: 30);

* **OOP in C#**

OOP is a programming paradigm that organizes software design around **objects**, which are instances of **classes**. It allows for modelling real-world entities and behaviours.

The four fundamental principles of OOP are:

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

* **Class and object in C#**

A **class** is a **blueprint** or **template** for creating objects. It defines the **properties** (data) and **methods** (behaviour) that the objects created from the class will have.

An **object** is an **instance** of a class. When you create an object, you are using the blueprint (class) to make a real-world usable item.

public class Car

{

// Properties (fields)

public string color;

public string model;

// Method

public void Drive()

{

Console.WriteLine("The car is driving.");

}

}

Car myCar = new Car();

myCar.color = "Red";

myCar.model = "Toyota";

myCar.Drive();

**Types of classes:**

1. Regular (Concrete) Classes

**Definition**: A regular or concrete class is a standard class that can be instantiated directly using the new keyword. It provides a complete implementation of its members and can be inherited from unless marked otherwise.

**Use**: Used to create objects that represent entities with state and behaviour. They form the basis for most application logic, such as models in MVC applications or business objects.

**Importance**: Concrete classes enable direct object creation, making them essential for runtime instances. They support inheritance hierarchies and polymorphism, allowing flexible designs. Without them, you couldn't create reusable, instantiable types.

1. Abstract Classes

**Definition**: An abstract class is declared with the abstract keyword and cannot be instantiated directly. It may contain abstract members (methods without implementation) that must be implemented by derived classes, along with concrete members.

**Use**: Serves as a base class for inheritance, providing common functionality while forcing subclasses to implement specific behaviours. Common in frameworks (e.g., base classes for UI controls or shapes in graphics).

**Importance**: Abstract classes enforce a contract for derived classes, promoting consistency and polymorphism. They allow partial implementation, reducing code duplication in hierarchies. This is crucial for designing extensible systems where subclasses specialize behaviour.

1. Sealed Classes

**Definition**: A sealed class is declared with the sealed keyword and cannot be inherited from. It can be instantiated like a regular class but prevents further derivation.

**Use**: Used for classes that are not designed for extension, such as utility classes or those with sensitive implementations (e.g., security-related classes). Often seen in libraries to control inheritance.

**Importance**: Sealing prevents unintended modifications or extensions, improving security and performance (e.g., enabling compiler optimizations). It's important for maintaining design integrity in large codebases or when distributing code as assemblies.

1. Static Classes

**Definition**: A static class is declared with the static keyword. It cannot be instantiated, and all its members must be static. It acts as a container for static members.

**Use**: Ideal for utility functions or global helpers that don't require instance state, like math operations (Math class in .NET) or configuration managers.

**Importance**: Static classes provide a way to group related static methods without allowing instantiation, preventing misuse. They are thread-safe by design (no instance state) and optimize memory usage since no objects are created. Essential for stateless, shared functionality across an application.

1. Partial Classes

**Definition**: A partial class is a class whose definition is split across multiple source files, each declared with the partial keyword. All parts are combined into a single class at compile-time. Each part must have the same name and be in the same namespace.

**Use**: Ideal for large classes, code organization, or when multiple developers work on the same class. Commonly used in frameworks like .NET for auto-generated code (e.g., Entity Framework or Windows Forms), where one file holds generated code and another holds custom logic.

**Importance**: Partial classes enhance code maintainability by allowing separation of concerns, such as isolating generated code from custom code. They simplify collaboration, reduce merge conflicts, and improve readability for complex classes without affecting functionality, as the compiler merges all parts into a single cohesive class.

1. Nested Classes

**Definition**: A nested class is defined inside another class (the outer class). It can be public, private, etc., and has access to the outer class's members. Nested classes can be static or non-static (inner classes).

**Use**: Used for encapsulation when a class is only relevant to the outer class, like helper classes (e.g., a Node class inside a LinkedList). Non-static nested classes can access instance members of the outer class.

**Importance**: Nested classes enhance encapsulation by hiding implementation details. They reduce namespace pollution and improve readability in tightly coupled designs. Crucial for patterns like iterators or event handlers.

* **Abstraction in C#**

**Abstraction** is the process of **hiding the complex implementation details** and **showing only the essential features** of an object.

In C#, abstraction is achieved in two main ways:

1. Using Abstract Classes

abstract class Animal

{

public abstract void MakeSound(); // Abstract method

public void Eat() // Non-abstract method

{

Console.WriteLine("Animal is eating.");

}

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Dog barks.");

}

}

Animal myDog = new Dog();

myDog.MakeSound();

myDog.Eat();

1. Using Interfaces

interface Bird

{

public void Fly();

}

class Parrot : Bird

{

public void Fly()

{

Console.WriteLine("Bird id flying");

}

}

Bird myBird = new Parrot();

myBird.Fly();

* **Encapsulation in C#**

Encapsulation is the **bundling of data and methods** that operate on the data within one unit (a class). It also involves **hiding the internal state** of an object and only exposing a controlled interface.

public class Person

{

private string name;

private int age;

public string Name

{

get { return name; }

set { name = value; }

}

public int Age

{

get { return age; }

set

{

if(value > 0) age = value;

else age = 5;

}

}

}

Person Student = new Person();

Student.Name = "Kiran";

Student.Age = 20;

Console.WriteLine("Name of student is " + Student.Name + " and age is " + Student.Age);

* **Polymorphism in C#**

**Polymorphism** means **“many forms”** - the ability of different classes to be treated through the same interface or base class, even if they behave differently.

C# supports two main types of polymorphism:

1. Compile-time Polymorphism (Static Polymorphism): Compile-time polymorphism refers to the ability of a class to have multiple methods with the **same name but different signatures**, allowing method calls to be resolved at **compile time** based on the method parameters. This is typically achieved through **method overloading** or **operator overloading**.
2. Run-Time Polymorphism (Method Overriding): Run-time polymorphism refers to the ability of a derived class to override a method defined in a base class, such that the correct method is called based on the object type at **runtime**, not the reference type. This is typically achieved through **method overriding** using **virtual** and **override** **keywords**.

| Use **abstract** when… |
| --- |
| * You want to enforce all derived classes to implement a method. |
| * There is **no meaningful default** implementation. |

| Use **virtual** when… |
| --- |
| * You want to provide a **default behaviour**, but let child classes override it **if needed**. |
| * Not all derived classes need to change the behaviour.   In **method overriding,** if method called through a **base class reference pointing to a child object**, the **child class method** is invoked (runtime polymorphism).  In **method hiding,** if method called through a **base class reference**, the **base class method** is invoked (method hiding depends on reference type). |

* **Inheritance in C#**

Inheritance enables one class to inherit fields, properties, and methods from another class, promoting **code reuse**, **extensibility**, and **polymorphism**.

Types of Inheritance in C# (Supported):

| **Type** | **Example** | **Supported in C#?** |
| --- | --- | --- |
| **Single** | One base class, one derived class | Yes |
| **Multilevel** | Class A → Class B → Class C | Yes |
| **Hierarchical** | One base class → many derived classes | Yes |
| **Multiple** | A class inherits from more than 1 class | Not directly (Use Interfaces) |

* **Interface in C#**

An interface is a **completely abstract class** that contains only declarations (no implementation) for members. Classes or structs that implement the interface must provide the actual implementation.

Some important points about interface:

1. Like **abstract classes**, interfaces **cannot** be used to create objects
2. Interface methods do not have a body - the body is provided by the "implement" class
3. On implementation of an interface, you must override all of its methods
4. Interfaces can contain properties and methods, but not fields/variables
5. Interface members are by default abstract and public
6. An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When to Use Interfaces?

1. To achieve security - hide certain details and only show the important details of an object (interface).
2. C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.

public interface IPayment

{

void Pay(float amount);

}

public class Cash : IPayment

{

public void Pay(float amount)

{

Console.WriteLine($"Rs. {amount} paid with cash");

}

}

public class UPI : IPayment

{

public void Pay(float amount)

{

Console.WriteLine($"Rs. {amount} paid via UPI");

}

}

public class Interface

{

public static void InterfaceDemo()

{

new Cash().Pay(500);

new UPI().Pay(1000);

}

}

* **Access modifiers in C#**

Access modifiers in C# control **where** a class or class member (methods, properties, fields) can be **accessed** from. They are applied to classes, methods, properties, fields, etc.

List of Access Modifiers:

| **Modifier** | **Access Level** |
| --- | --- |
| public | Accessible from **anywhere** |
| private | Accessible **only within the same class** (default for class members) |
| protected | Accessible in the **same class and derived classes** |
| internal | Accessible **within the same assembly/project** |
| protected internal | Accessible from **derived classes or same assembly** |
| private protected | Accessible from **derived classes in the same assembly** |

* **Namespace in C#**

A **namespace** in C# is a **container** that holds **classes, interfaces, structs, enums, delegates, and other namespaces**. It is used to **organize code** and prevent **name conflicts** between types that may have the same name.

Types of Namespaces in C#:

1. User-defined namespaces: Created by developers to organize their own code.

namespace HelloWorld

{

class Program

{

static void Main(string[] args)

{

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

//Console.WriteLine("This is my C# program");

}

}

}

1. .NET built-in namespaces: Provided by the .NET Framework and .NET Core.

using System;

using System.Collections.Generic;

* **Collections in C#**

**Collections** are **classes** provided by the .NET Framework that are used to **store and manage groups of related objects**. Unlike arrays, collections are **dynamic**, which means you don't need to know the number of elements in advance.

Types of Collections in C#:

1. **Non-generic Collections** (in System.Collections): Non-generic collections can store any type of object, but they lack type safety and require casting.

| **Collection** | **Description** |
| --- | --- |
| ArrayList | A resizable array that stores objects. |
| Hashtable | Stores key-value pairs using hashing. |
| SortedList | Stores key-value pairs sorted by the key. |
| Stack | Last-In-First-Out (LIFO) collection. |
| Queue | First-In-First-Out (FIFO) collection. |
| BitArray | A collection of bit values, represented as Booleans. |
| DictionaryBase | An abstract base class for creating custom dictionary-type collections. |
| CollectionBase | An abstract base class for strongly typed custom collection classes. |
| ReadOnlyCollectionBase | Base class for creating read-only custom collections. |

1. **Generic Collections** (in System.Collections.Generic): Generic collections are type-safe, efficient, and more widely used in modern C# applications.

| **Collection Type** | **Description** |
| --- | --- |
| List | A dynamic array (resizable), most commonly used generic collection. |
| Dictionary<TKey, TValue> | A collection of key-value pairs, optimized for lookup. |
| HashSet | A collection that contains unique elements, similar to mathematical sets. |
| SortedList<TKey, TValue> | A sorted collection of key-value pairs (sorted by key). |
| SortedDictionary<TKey, TValue> | Similar to SortedList, but better performance for insertions/removals. |
| LinkedList | A doubly linked list, useful for frequent insertions/removals. |
| Queue | A First-In-First-Out (FIFO) collection. |
| Stack | A Last-In-First-Out (LIFO) collection. |
| KeyValuePair<TKey, TValue> | Represents a single key-value pair, often used with dictionaries. |
| IEnumerable | Base interface for all generic collections. |
| ICollection | Base interface for all collections that can be modified. |
| IList | An interface for collections that are indexed like arrays/lists. |
| IDictionary<TKey, TValue> | Interface for key-value collections. |
| IReadOnlyList | A read-only list interface. |
| IReadOnlyDictionary<TKey, TValue> | A read-only dictionary interface. |

1. **Concurrent & Specialized Collections** (in System.Collections.Concurrent and others): For multi-threaded applications, .NET provides thread-safe collections in the System.Collections.Concurrent namespace.

| Collection | Description |
| --- | --- |
| ConcurrentDictionary<TKey, TValue> | Thread-safe dictionary; supports atomic operations like GetOrAdd. |
| ConcurrentQueue | Thread-safe FIFO queue. |
| ConcurrentStack | Thread-safe LIFO stack. |
| ConcurrentBag | Thread-safe unordered collection; good for scenarios like work stealing. |
| BlockingCollection | Thread-safe collection that supports bounding, blocking, and producer-consumer patterns. Internally uses other collections like ConcurrentQueue<T>. |

* **List in C#**

A generic collection that **represents a strongly typed list of objects** that can be accessed by index. It provides **dynamic resizing**, unlike arrays.

* **Methods in List Collection**

1. Add(T item): Adds an object to the end of the List.

List<string> names = new List<string>();

names.Add("Alice");

1. AddRange(IEnumerable<T> collection): Adds the elements of the specified collection to the end of the List.

List<int> numbers = new List<int>() { 1, 2 };

numbers.AddRange(new int[] { 3, 4, 5 });

Console.WriteLine(string.Join(", ", numbers)); // 1, 2, 3, 4, 5

1. Insert(int index, T item): Inserts an element at the specified index.

List<string> fruits = new List<string>() { "Apple", "Banana" };

fruits.Insert(1, "Mango");

Console.WriteLine(string.Join(", ", fruits)); // Apple, Mango, Banana

1. InsertRange(int index, IEnumerable<T> collection): Inserts a collection of elements starting at a specific index.

List<int> numbers = new List<int>() { 1, 4 };

numbers.InsertRange(1, new int[] { 2, 3 });

Console.WriteLine(string.Join(", ", numbers)); // 1, 2, 3, 4

1. Remove(T item): Removes the first occurrence of a specific object.

List<string> animals = new List<string>() { "Cat", "Dog", "Cat" };

animals.Remove("Cat");

Console.WriteLine(string.Join(", ", animals)); // Dog, Cat

1. RemoveAt(int index): Removes the element at the specified index.

List<string> cities = new List<string>() { "NY", "LA", "Chicago" };

cities.RemoveAt(1);

Console.WriteLine(string.Join(", ", cities)); // NY, Chicago

1. RemoveRange(int index, int count): Removes a range of elements starting at the specified index.

List<int> numbers = new List<int>() { 0, 1, 2, 3, 4, 5 };

numbers.RemoveRange(2, 3);

Console.WriteLine(string.Join(", ", numbers)); // 0, 1, 5

1. RemoveAll(Predicate match): Removes all elements matching the predicate; returns count removed.

List<int> numbers = new List<int> { 10, 20, 30, 40 };

numbers.RemoveAll(n => n > 25);

Console.WriteLine(string.Join(", ", numbers)); // 10, 20

1. Clear(): Removes all elements from the list.

List<string> items = new List<string>() { "A", "B" };

items.Clear();

Console.WriteLine(items.Count); // 0

1. Contains(T item): Checks if the list contains a specific item.

List<int> nums = new List<int>() { 1, 2, 3 };

Console.WriteLine(nums.Contains(2)); // True

1. IndexOf(T item): Returns the index of the first occurrence of an item.

List<string> colors = new List<string>() { "Red", "Blue", "Green" };

int index = colors.IndexOf("Blue");

Console.WriteLine(index); // 1

1. LastIndexOf(T item): Returns the index of the last occurrence of an item.

List<string> colors = new List<string>() { "Red", "Blue", "Red" };

int index = colors.LastIndexOf("Red");

Console.WriteLine(index); // 2

1. Count : It is property which gets the number of elements in the list.

List<int> numbers = new List<int>() { 10, 20, 30 };

Console.WriteLine(numbers.Count); // 3

1. Capacity: Gets or sets the total number of elements the internal data structure can hold without resizing.

Console.WriteLine(nums.Capacity); // Default capacity

1. EnsureCapacity(int): Ensures the list can hold a specified number of items.

numbers.EnsureCapacity(100);

1. Sort(): Sorts the elements in ascending order.

List<int> nums = new List<int>() { 3, 1, 2 };

nums.Sort();

Console.WriteLine(string.Join(", ", nums)); // 1, 2, 3

1. Reverse(): Reverses the order of the list.

List<int> nums = new List<int>() { 1, 2, 3 };

nums.Reverse();

Console.WriteLine(string.Join(", ", nums)); // 3, 2, 1

1. ToArray(): Converts the list to an array.

List<int> nums = new List<int>() { 1, 2, 3 };

int[] arr = nums.ToArray();

Console.WriteLine(string.Join(", ", arr)); // 1, 2, 3

1. Find(Predicate<T> match): Finds the first item that matches a condition.

List<int> nums = new List<int>() { 10, 20, 30 };

int result = nums.Find(x => x > 15);

Console.WriteLine(result); // 20

1. FindAll(Predicate<T> match): Returns all items that match a condition.

List<int> nums = new List<int>() { 10, 20, 30 };

List<int> result = nums.FindAll(x => x > 15);

Console.WriteLine(string.Join(", ", result)); // 20, 30

1. Exists(Predicate<T> match): Checks if any item matches the condition.

List<int> nums = new List<int>() { 10, 20, 30 };

bool exists = nums.Exists(x => x == 20);

Console.WriteLine(exists); // True

1. ForEach(Action<T> action): Performs an action on each element.

List<string> names = new List<string>() { "Alice", "Bob" };

names.ForEach(name => Console.WriteLine("Hello, " + name));

1. GetRange(int index, int count): Returns a shallow copy (sub-list) of the specified range.

List<int> numbers = new List<int>() { 10, 20, 30, 40 };

List<int> sublist = numbers.GetRange(1, 2);

Console.WriteLine(string.Join(", ", sublist)); // 20, 30

1. TrimExcess(): Reduces capacity to match Count if underutilized.

List<int> list = new List<int>(100) { 10 };

list.TrimExcess();

Console.WriteLine(list.Capacity); // Output: 1

1. AsReadOnly(): Returns a read-only wrapper around the List.

var readOnly = list.AsReadOnly();

1. ConvertAll(Converter<T, TOutput> converter): Converts all elements to another type and returns a new List.

List<int> list = new List<int> { 10, 20 };

var strings = list.ConvertAll(x => x.ToString());

Console.WriteLine(string.Join(", ", strings)); // Output: 10, 20

* **Enumeration in C#**

An enumeration is a **distinct value type** that defines a set of **named constants** called the **enumerator list**. It’s defined using the enum keyword and is **typically used to represent a group of related constant values** in a type-safe way.

public enum WeekDay

{

Monday, Tuesday, Wednesday, Thursday = 10, Friday, Saturday, Sunday

}

public class Enumeration

{

public static void EnumerationDemo()

{

foreach(WeekDay d in Enum.GetValues(typeof(WeekDay)))

{

Console.WriteLine($"{(int)d} - {d}");

}

}

}

* **DataTable in C#**

DataTable is a class in C# that represents **one in-memory table** of data. It is part of the **System.Data** namespace.

It is commonly used when working with **disconnected data architecture**, i.e., data not directly tied to a database connection.

* **Structure of DataTable**

| **Component** | **Description** |
| --- | --- |
| Columns | Defines the schema (name, data type, constraints) |
| Rows | Actual data |
| Constraints | Rules for data validation (like primary keys) |
| DataTypes | Each column can have its own data type (string, int, etc.) |

* **Properties & Methods of DataTable**

| **Property/Method** | **Description** |
| --- | --- |
| TableName | Name of the table |
| Rows.Count | Number of rows |
| Columns.Count | Number of columns |
| Clear() | Removes all rows |
| AcceptChanges() | Commits changes |
| RejectChanges() | Cancels changes |

1. How to Create a DataTable

DataTable table = new DataTable("Students");

1. Adding Columns to a DataTable

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

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

1. Adding Rows to a DataTable

table.Rows.Add(2201, "AA");

table.Rows.Add(2202, "AK");

1. Accessing Data in a DataTable

foreach (DataRow row in table.Rows)

{

Console.WriteLine("Id : {0} & Name : {1}", row[0], row[1]);

}

1. Updating Data in a DataTable

table.Rows[0]["Name"] = "AD";

1. Deleting Rows

copiedTable.Rows[1].Delete();

1. Setting Primary Keys

table.PrimaryKey = new DataColumn[] { table.Columns["Id"] };

1. Cloning and Copying

DataTable copiedTable = table.Copy();

DataTable clonedTable = table.Clone();

* **Exception Handling in C#**

Exception Handling in C# is a mechanism to handle **runtime errors** (called exceptions) so that the program can continue executing or exit gracefully without crashing.

|  |  |
| --- | --- |
| **Keyword** | **Definition** |
| try | Used to define a try block. This block holds the code that may throw an exception. |
| catch | Used to define a catch block. This block catches the exception thrown by the try block. |
| finally | Used to define the finally block. This block holds the default code. |
| throw | Used to throw an exception manually. |

* **Types of Exceptions**

| **Exception Class** | **Description** |
| --- | --- |
| System.Exception | Base class for all exceptions |
| DivideByZeroException | Thrown when dividing by zero |
| NullReferenceException | Accessing a null object |
| IndexOutOfRangeException | Invalid index in an array or list |
| FileNotFoundException | File not found |
| FormatException | Invalid string-to-number conversion |

**Sample Code:**

try

{

int a = 10, b = 0;

Console.WriteLine(a / b);

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

//Console.WriteLine(numbers[5]);

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

if(i == 0)

{

throw new Exception("Number must be greater than zero");

}

}

catch (DivideByZeroException e)

{

Console.WriteLine(e);

}

catch (IndexOutOfRangeException e)

{

Console.WriteLine(e);

}

catch (Exception e)

{

Console.WriteLine(e);

}

finally

{

Console.WriteLine("\nFinally block\n");

}

* **Different Project Types**

| **Project Type** | **Main Use** |
| --- | --- |
| Console App | Simple utilities, Testing algorithms, experiments |
| Windows Forms / WPF | Desktop apps |
| ASP.NET Core | Web and APIs |
| MAUI | Mobile/Desktop cross-platform |
| Class Library | Reusable logic, Shared code |
| Test Project | Unit/integration testing |
| Azure Functions | Serverless cloud code |
| ML.NET | Machine Learning |
| SQL Project | Database development |
| Unity | Game development |

* **String in C#**

In C#, a string is an **object of the System.String class** used to represent **a sequence of characters**. Strings are **immutable** (cannot be changed after creation).

* **Properties of String**

| **Property** | **Description** | **Example** |
| --- | --- | --- |
| Length | Number of characters in string | "Hello".Length → 5 |
| Chars[] | Access character by index | "Hello"[1] → 'e' |

* **Methods of String**

1. Contains(): Checks if a string contains a specified substring.

Console.WriteLine("hello world".Contains("world")); // True

1. StartsWith(): Checks if a string starts with a specified substring.

Console.WriteLine("hello world".StartsWith("hello")); // True

1. EndsWith(): Checks if a string ends with a specified substring.

Console.WriteLine("hello world".EndsWith("world")); // True

1. IndexOf(): Finds the index of the first occurrence of a specified character or substring.

Console.WriteLine("hello".IndexOf('l')); // 2

1. LastIndexOf(): Finds the index of the last occurrence of a specified character or substring.

Console.WriteLine("hello".LastIndexOf('l')); // 3

1. Equals(): Compares strings for equality.

Console.WriteLine("test".Equals("test")); // True

1. IsNullOrEmpty(): Static method that checks if a string is null or empty.

Console.WriteLine(string.IsNullOrEmpty("")); // True

1. IsNullOrWhiteSpace(): Static method that checks if a string is null, empty, or whitespace.

Console.WriteLine(string.IsNullOrWhiteSpace(" ")); // True

1. Replace(): Replaces occurrences of a substring with another substring.

Console.WriteLine("apple".Replace("a", "A")); // Apple

1. Remove(): Removes characters from a string.

Console.WriteLine("abcdef".Remove(2, 2)); // abef

1. Insert(): Inserts a string at the specified index.

Console.WriteLine("abcdef".Insert(3, "XYZ")); // abcXYZdef

1. ToUpper(): Converts a string to uppercase.

Console.WriteLine("hello".ToUpper()); // HELLO

1. ToLower(): Converts a string to lowercase.

Console.WriteLine("HELLO".ToLower()); // hello

1. Trim(): Removes leading and trailing whitespaces.

Console.WriteLine(" hello ".Trim()); // "hello"

1. PadLeft(): Pads a string on the left with spaces or a specified character to a certain length.

Console.WriteLine("42".PadLeft(5, '0')); // 00042

1. PadRight(): Pads a string on the right with spaces or a specified character to a certain length.

Console.WriteLine("42".PadRight(5, '\*')); // 42\*\*\*

1. Split(): Splits a string into an array based on a specified delimiter.

foreach(var item in "a,b,c".Split(',')) Console.WriteLine(item);

1. Join(): Combines an array of strings into a single string with a specified delimiter.

Console.WriteLine(string.Join("-", new[] { "a", "b", "c" })); // a-b-c

1. Substring(start, len): Returns a substring from the index with the given length.

Console.WriteLine("abcdef".Substring(2, 3)); // cde

1. ToCharArray(): Converts a string to a character array.

foreach (char c in "abc".ToCharArray()) Console.WriteLine(c);

* **StringBuilder in C#**

StringBuilder is a **mutable** (modifiable) string class provided in the System.Text namespace in C#. Unlike regular strings (string), which are **immutable** (cannot be changed after creation), StringBuilder allows you to modify the content without creating a new object each time.

Syntax:

StringBuilder sb = new StringBuilder("Hello");

* **Methods of StringBuilder**

1. Append: Adds text at the end

StringBuilder sb = new StringBuilder("Hello");

sb.Append(" World");

Console.WriteLine(sb); // Output: Hello World

1. Insert: Inserts text at specified index

sb.Insert(5, ",");

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

1. Replace: Replaces text

sb.Replace("World", "C#");

Console.WriteLine(sb); // Output: Hello, C#

1. Remove: Removes text from specified index

sb.Remove(5, 2); // removes ", "

Console.WriteLine(sb); // Output: HelloC#

1. Length: Gets or sets the number of characters

Console.WriteLine("Length: " + sb.Length);

* **DateTime in C#**

The DateTime structure in C# represents **dates and times**. It includes **date**, **time**, or **both**, and provides methods and properties to perform **date/time operations**, like comparing, formatting, parsing, and arithmetic.

* **Properties of DateTime**

1. DateTime.Now: Gets the current local date and time

Console.WriteLine(DateTime.Now); // e.g., 8/18/2025 2:23:45 PM

1. DateTime.Today: Returns the current date with time set to 00:00:00

Console.WriteLine(DateTime.Today); // e.g., 8/18/2025 12:00:00 AM

1. DateTime.Year: Gets the year component of the DateTime instance.

Console.WriteLine(DateTime.Now.Year); // 2025

1. DateTime.Month: Gets the month component (1 through 12) of the DateTime instance.

Console.WriteLine(DateTime.Now.Month); // 8

1. DateTime.Day: Gets the day of the month (1 through 31) from the DateTime instance.

Console.WriteLine(DateTime.Now.Day); // 18

1. DateTime.Hour: Gets the hour component (0 through 23) of the DateTime instance.

Console.WriteLine(DateTime.Now.Hour); // e.g., 14

1. DateTime.Minute: Gets the minute component (0 through 59) of the DateTime instance.

Console.WriteLine(DateTime.Now.Minute); // e.g., 23

1. DateTime.Second: Gets the second component (0 through 59) of the DateTime instance.

Console.WriteLine(DateTime.Now.Second); // e.g., 45

1. DateTime.Millisecond: Gets the milliseconds component (0 through 999) of the DateTime instance.

Console.WriteLine(DateTime.Now.Millisecond); // e.g., 123

1. DateTime.DayOfWeek: Gets the day of the week as a DayOfWeek enum (Sunday through Saturday) from the DateTime instance.

Console.WriteLine(DateTime.Now.DayOfWeek); // Monday

1. DateTime.DayOfYear: Gets the day number within the year (1 through 366, accounting for leap years) from the DateTime instance.

Console.WriteLine(DateTime.Now.DayOfYear); // 230

1. DateTime.Kind: Gets a DateTimeKind enum value indicating whether the time is Local, UTC, or Unspecified.

Console.WriteLine(DateTime.Now.Kind); // Local

1. DateTime.Ticks:Gets the total number of 100-nanosecond intervals that have elapsed since 0001-01-01 00:00:00.000.

Console.WriteLine(DateTime.Now.Ticks); // e.g., 638283504451231230

* **Methods of DateTime**

1. DaysInMonth(int year, int month): Returns the number of days in the specified month and year.

Console.WriteLine(DateTime.DaysInMonth(2024, 2)); // 29

1. IsLeapYear(int year): Returns true if the specified year is a leap year, otherwise false.

Console.WriteLine(DateTime.IsLeapYear(2024)); // True

1. Parse(string s): Converts a date/time string to a DateTime object. Throws exception if invalid.

Console.WriteLine(DateTime.Parse("2025-08-18")); // 8/18/2025 12:00:00 AM

1. AddYears(int value): Returns a new DateTime with years added/subtracted.

Console.WriteLine(DateTime.Now.AddYears(1)); // e.g., 8/18/2026 2:23:45 PM

1. AddMonths(int value): Returns a new DateTime with months added/subtracted.

Console.WriteLine(DateTime.Now.AddMonths(2)); // e.g., 10/18/2025 2:23:45 PM

1. AddDays(double value): Returns a new DateTime with days added/subtracted (fractional days allowed).

Console.WriteLine(DateTime.Now.AddDays(1.5)); // e.g., 8/19/2025 2:23:45 AM

1. AddHours(double value): Returns a new DateTime with hours added/subtracted (fractional hours allowed).

Console.WriteLine(DateTime.Now.AddHours(3.5)); // e.g., 8/18/2025 5:53:45 PM

1. CompareTo(DateTime other): Compares this instance with another DateTime. Returns negative, zero, or positive.

Console.WriteLine(DateTime.Now.CompareTo(DateTime.Today)); // 1 (if Now is later than Today)

* **Basic file operations in C#**

1. File.Exists(string path): Checks whether a file exists at a given path.
2. File.Create(string path): Creates a new file at the specified path.
3. File.WriteAllText(string path, string contents): Writes content to a file. Overwrites existing content.
4. File.AppendAllText(string path, string contents): Adds content to the end of an existing file without overwriting.
5. File.ReadAllText(string path): Reads the entire content or individual lines of a file.
6. File.Delete(string path): Removes a file from the filesystem.
7. File.Copy(string sourceFileName, string destFileName, bool overwrite): Creates a copy of a file.
8. File.Move(string sourceFileName, string destFileName): Moves a file to a new location or renames it.

* **Delegates in C#**

A **delegate** is a **type-safe function pointer**. In simple terms, it's a reference type that **holds a reference to a method with a specific signature and return type**.

Delegates can **point to any method** that matches their signature. They can be **passed as parameters**, stored in **collections**, or assigned dynamically.

* **Why to use Delegates?**

1. Decoupling Behavior: Allows you to separate what is done from how it’s done.
2. Flexibility: Easily plug in different behavior at runtime without modifying the core logic.
3. Reusability: Write general-purpose methods that accept behavior as a delegate, making them reusable with different logic.
4. Callback Mechanism: Useful for event handling or notifying when an action is completed.
5. Supports Anonymous Methods & Lambdas: Delegates make functional-style programming possible in C# with lambda expressions and anonymous methods.

Code:

public delegate bool StudentFilter(Student s);

public class StudentManager

{

public void FilterStudents(List<Student> students, StudentFilter filter)

{

foreach (var student in students)

{

if (filter(student))

{

Console.WriteLine(student.Name);

}

}

}

}

manager.FilterStudents(students, s => s.Grade > 85);

* **Advantages of Delegates**

| **Feature** | **Description** |
| --- | --- |
| **Type Safety** | Only methods with a matching signature can be assigned. |
| **Runtime Flexibility** | Methods can be assigned or changed at runtime. |
| **Multicast Support** | Can invoke multiple methods through one delegate instance. |
| **Event Handling** | Delegates are the basis of the **event** model in C#. |
| **Cleaner Code** | Reduces tight coupling and makes code easier to test and extend. |

* **Difference Between Locals Window and Watch Window in Visual Studio**

| **Feature** | **Locals Window** | **Watch Window** |
| --- | --- | --- |
| **Purpose** | Displays all local variables and method parameters in the current scope | Allows you to manually monitor specific variables or expressions during debugging |
| **Scope** | Limited to the current method or block (stack frame) | Can observe variables across different scopes and methods |
| **User Interaction** | Read-only — variables are displayed automatically | Interactive — you manually add, edit, and remove expressions |
| **Custom Expressions** | Not supported | Supported (e.g., myObj.Property + 10, arr[2], etc.) |
| **Automatic Display** | Yes — variables appear automatically when in scope | No — you must manually add what you want to watch |
| **Persistence After Debugging** | Cleared when debugging ends | Watch list persists across debugging sessions (until manually removed) |
| **Usage Scenario** | Quickly view all current local values and parameters | Focused tracking of selected variables or complex expressions over time |
| **Number of Windows** | One | Multiple (Watch 1, Watch 2, Watch 3, Watch 4) available |

* **Lambda expression in C#**

A **lambda expression** is a **concise way to represent an anonymous method** (a method without a name) using a special syntax:

(parameters) => expression or statement block

Lambda expressions is **shortcuts for writing methods**, especially useful when passing functionality as arguments (e.g., to delegates, Func, Action, LINQ).