Basic Api Part 1

Introduction to C#:

1. C# (C-Sharp) is a modern, object-oriented programming language developed by Microsoft.
2. If you have basic knowledge of Java or C++, understanding C# will be relatively easy because its syntax is quite similar.

Use of c#:

- Windows Applications
- Web Development
- Game Development
- Mobile Apps
- Enterprise Applications

Why learn c#:

- Ease of Learning
- Integration with Microsoft Technologies
- Easy to Learn for Beginners
- Cross-Platform Development
- High Performance

Key features of c#:

- Object-Oriented Programming (OOP)
- Type Safety
- Automatic Memory Management
- Exception Handling
- Security
- Scalability and Performance

Hello world Program:

```
using System;

namespace BasicCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hello world");
        }


    }
}
```

Explanation of Code:

1. **using System;**
   This line includes the `System` namespace, which provides basic functionalities such as input-output operations (e.g., `Console.WriteLine`).

2. **namespace Basic_C_Sharp**
   A namespace is a logical grouping that organizes your code. It helps manage multiple classes and methods by avoiding naming conflicts.

3. **class Program**
   A class is a blueprint that defines the behavior of your program. Every C# program consists of one or more classes.

4. **static void Main(string[] args)**
   The `Main` method is the entry point of the program. When the program is executed, it starts from this method.

   `static`: Indicates that the method can be called without creating an instance of the class.

   `void`: The return type, meaning the method does not return any value

   `Main`: The name of the method, which is fixed in C# for the program's entry point `string[] args`: Accepts command-line arguments as a string array, allowing you to pass inputs at runtime.

5. `{` and `}`

Curly braces group code blocks in C#.

- `{` opens a block.
- `}` closes a block

6. `Console.WriteLine("hello world");`

`Console`: A predefined class from the `System` namespace for handling input-output.

`WriteLine`: A method of the `Console` class that writes text to the console and automatically adds a newline.

UnderStanding C# Program structure:

```
using System;           // Namespace inclusion

namespace MyNamespace    // Namespace
declaration
{
    class Program        // Class declaration
    {
        static void Main(string[] args)
               // Main method (Entry point)
        {
            // Code logic here
            Console.WriteLine("Hello, World!");
                              // Statement
        }
    }
}
```

1.using system:

- The `System` namespace includes common functionalities, such as the `Console` class, `Math` class, and more.

Example:

- To use `Console.WriteLine("Hello, World!")`, the `System` namespace must be included.

2.namespace MyNamespace

- A namespace is a logical container that groups related classes and code.
- It helps to avoid naming conflicts

Why Namespaces?

If your project has two classes named `Program`, namespaces allow you to differentiate between them

```
namespace Namespace1
{
    class Program { }
}

namespace Namespace2
{
    class Program { }
}
```

Access:

```
Namespace1.Program prog1=new
Namespace1.Program();
```

```
Namespace2.Program prog2 = new
Namespace2.Program();
```

3.class Program

- A class is a blueprint in which you write your code. Every C# program uses one or more classes.
- C# is an Object-Oriented Programming (OOP) language, and classes are core to OOP.
- Inside classes, methods and properties are defined.

```
class Car
{
    public string Brand { get; set; }  //
Property

    public void Drive()                //
Method
    {
        Console.WriteLine("Car is driving!");
    }
}
```

4.static void Main(string[] args)

- This is the entry point of the program. When the program is run, execution starts from this method.

static: Means the method can be called directly without creating an instance of the class

void: Indicates the method doesn't return any value

Main: The name of the method, which is predefined in C# as the entry point.

string[] args: An array that accepts command-line arguments.

5.statements:

The logical instructions that the program executes.

```
Console.WriteLine("Hello");
```

C# Program Execution Flow:

- Namespace Declaration
- Class Declaration
- Main Method
- Statements Execution

Types of comments:

1. Single-line Comments (//)
2. Multi-line Comments (/* */)
3. XML Documentation Comments (///)

Data types and conversion:

Value type:

Store data directly in memory

Ex.

Int - 4 bytes

Float - 4 bytes

Bool - true-false 1 bit

Char - 2 byte

Reference type:

String : Sequence of characters

Object: Base type of all types

Class: user defined

Null : no reference

Nullable type:

Allow value types to represent null.

Ex.

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

Variables:

1).Local variable:

Declared inside a method or block.

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

2)instance variable

Declared in a class but outside methods.

Ex .

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

3) static variable

Shared across all instances of a class.

```
static int count = 0;
```

Type conversion in C#

Implicit Conversion (Type-Safe Conversion)

When a smaller data type is converted to a larger data type automatically.

here is no data loss, and the conversion is handled by the compiler.

This is also called type-safe conversion.

ex.

```
int num = 10;          // int is 4 bytes

double result = num; // Implicit conversion (double is 8 bytes)

Console.WriteLine(result); // Output: 10.0
```

Explicit Conversion (Type Casting)

When a larger data type is converted to a smaller data type.

Data loss may occur, and you must use the cast operator ((type)).

```
Ex. double pi = 3.14;         // double is 8 bytes
```

```
int integerPi = (int)pi; // Explicit conversion

Console.WriteLine(integerPi); // Output: 3 (Decimal part is lost)
```

Conversion Using Methods

The Convert class provides methods to convert between different types.

```
Ex. string str = "123";

int number = Convert.ToInt32(str); // String to int

Console.WriteLine(number); // Output: 123
```

Parse Method:

Converts a string to a numeric type (like int, float, etc.).

```
Ex. string str = "456";

int num = int.Parse(str); // Parse string to int

Console.WriteLine(num); // Output: 456
```

TryParse Method

A safer way to parse strings, as it doesn't throw an exception if the conversion fails.

Ex.

```
string str = "789";
if (int.TryParse(str, out int result))
{
    Console.WriteLine(result); // Output: 789
}
else
{
    Console.WriteLine("Invalid input");
}
```

Operators & Expressions:

Types of Operators in C#

(a) Arithmetic Operators

+ (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;

(b) Relational Operators

Used to compare two values.

== (Equal): a == b

!= (Not Equal): a != b

> (Greater Than): a > b

< (Less Than): a < b

(c) Logical Operators

Used to perform logical operations (AND, OR, NOT).

− && (AND): a > b && c > d

|| (OR): a > b || c > d

! (NOT): !isTrue

(d) Assignment Operators

Used to assign values to variables.

++a (Pre-Increment) –

a++ (Post-Increment) –

--a (Pre-Decrement) –

a-- (Post-Decrement)

(f) Bitwise Operators

Used for bit-level operations.

& (AND): a & b –

| (OR): a | b –

^ (XOR): a ^ b

Statements:

A statement in C# is a single instruction or command that the compiler can execute. It typically ends with a semicolon (;) and can perform an action like declaring a variable, making a decision, or looping through data.

Types of Statements in C#

1. **Declaration Statements**
    - Used to declare variables or constants.
    - Syntax: `datatype variableName = value;`

```
int number = 10; // Declaration statement
const double PI = 3.14; // Constant declaration
```

2 .Expression Statements

- Any valid expression followed by a semicolon.
- Includes method calls, assignments, and operations.

```
number = number + 5;  // Assignment statement
Console.WriteLine(number); // Method call statement
```

3. Control Flow Statements

- **Control the execution flow of the program based on conditions or loops.**

If-Else Statement:

```
if (number > 5)
{
    Console.WriteLine("Number is greater than 5");
}
else
{
    Console.WriteLine("Number is 5 or less");
}
```

Switch Statement:

```
switch (number)

{

    case 1:

        Console.WriteLine("One");

        break;

    case 2:

        Console.WriteLine("Two");

        break;

    default:

        Console.WriteLine("Other number");

        break;

}
```

Looping Statements

Used to repeat a block of code.

Loop : forloop , while, do-while , foreach

4 . Jump Statements

- Used to transfer control from one part of the program to another.

  Break , continue, return , goto

Understanding Arrays in C#

An **array** in C# is a collection of elements of the same type stored in contiguous memory locations.

syntax:

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

Types of Arrays in C#:

1).Single-Dimensional Array

- A simple list of elements

```
int[] numbers = { 10, 20, 30, 40, 50 };
```

2). Multi-Dimensional Array

- Arrays with more than one dimension, like a table or matrix.

```
int[,] matrix = { { 1, 2, 3 }, { 4, 5, 6 } };
```

3)**Jagged Array**

- An array of arrays, where each sub-array can have a different size.

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[] { 1, 2, 3 };
jaggedArray[1] = new int[] { 4, 5 };
jaggedArray[2] = new int[] { 6, 7, 8, 9 };
Console.WriteLine(jaggedArray[1][1]);
```

Array method:

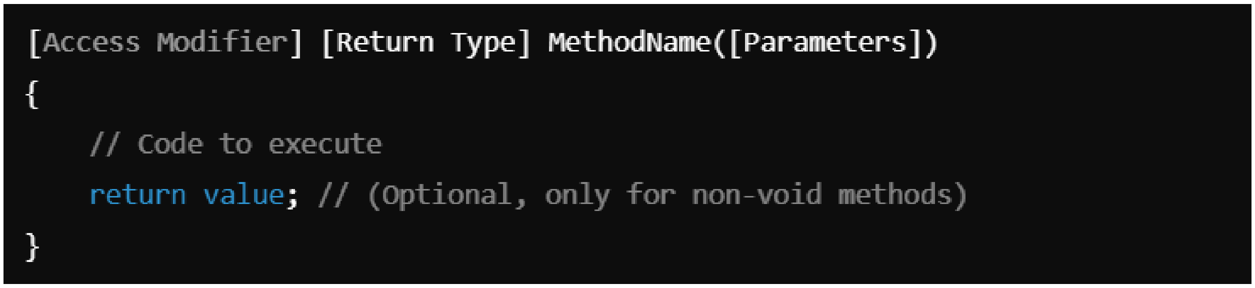
sort(),reverse(),copy(),clear(),indexof(),etc..

Array method are shown in code.

Methods in C#:

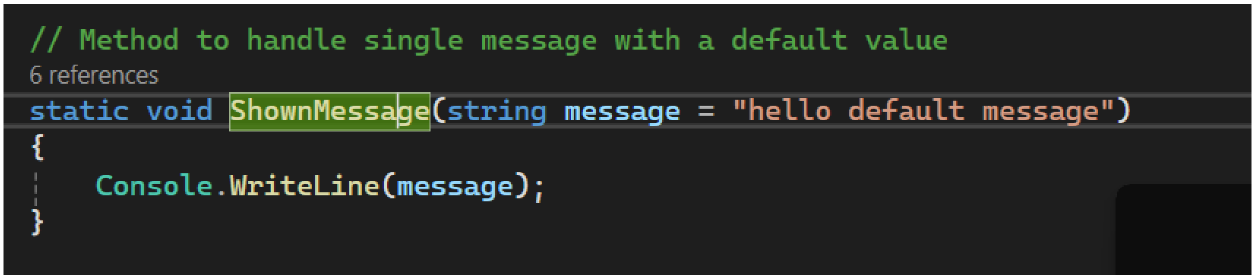
A **method** in C# is a block of code that performs a specific task. Methods help to organize code, avoid repetition, and improve readability and reusability.

Syntax:



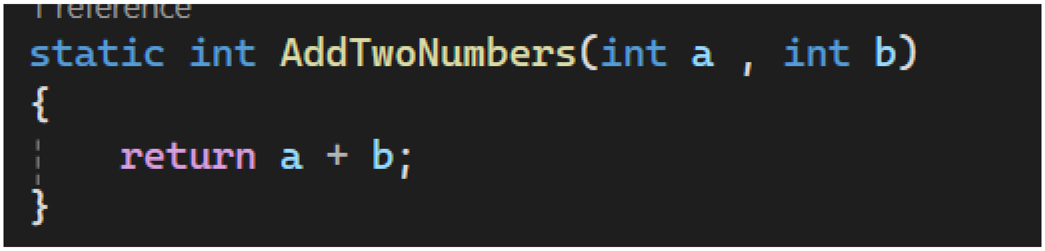
```
[Access Modifier] [Return Type] MethodName([Parameters])
{
    // Code to execute
    return value; // (Optional, only for non-void methods)
}
```

1: A Method With Parameters



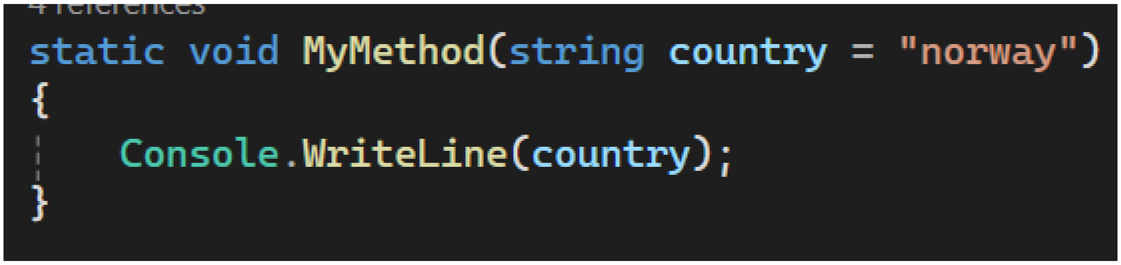
```
// Method to handle single message with a default value
6 references
static void ShownMessage(string message = "hello default message")
{
    Console.WriteLine(message);
}
```

2:A Method With a Return Value



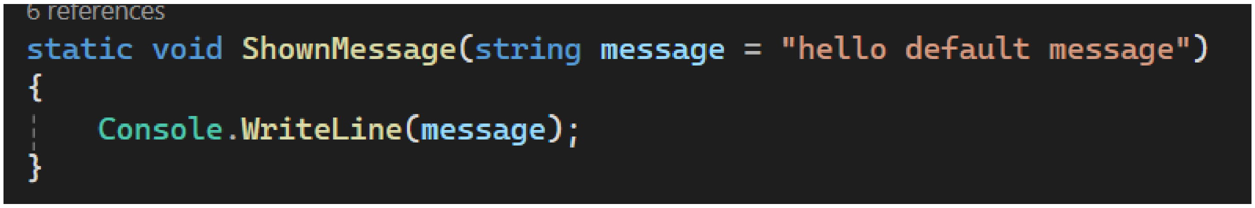
```
1 reference
static int AddTwoNumbers(int a , int b)
{
    return a + b;
}
```

3: A Method Without a Return Value



```
4 references
static void MyMethod(string country = "norway")
{
    Console.WriteLine(country);
}
```

4. Static Methods



```
static void ShownMessage(string message = "hello default message")
{
    Console.WriteLine(message);
}
```

Object-Oriented Programming (OOP) in C#

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data (fields) and methods (functions). C# is a fully object-oriented language

Class and Object

Class

A class is a blueprint for creating objects. It defines the properties (fields) and behaviors (methods) that an object can have.

Object

An object is an instance of a class. It is created based on the class definition.

Encapsulation:

Encapsulation means bundling data (fields) and methods into a single unit (class) and restricting access to them

public: Accessible everywhere.

private: Accessible only within the class.

protected: Accessible within the class and its derived classes.

internal: Accessible within the same assembly.

Ex.

```
public class BankAccount
{
    private double balance; // Private field
    public void Deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
    }

    public double GetBalance()
    {
        return balance;
    }
}
// Using the class
BankAccount account = new BankAccount();
account.Deposit(1000);
Console.WriteLine(account.GetBalance()); // Output: 1000
```

Inheritance:

inheritance allows one class (child class) to inherit properties and methods from another class (parent class)

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

// Child class
public class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("The dog barks.");
    }
}

// Using inheritance
Dog myDog = new Dog();
myDog.Eat(); // Inherited from Animal
myDog.Bark(); // Defined in Dog
```

Polymorphism

Polymorphism means "many forms." It allows methods to perform different tasks based on the context. In C#, polymorphism is achieved through:

1. **Method Overloading**: Same method name, different parameters.
2. **Method Overriding**: A child class modifies a method in the parent class

Method Overloading:

Ex.

```
public class MathOperations
{
    public int Add(int a, int b)
    { return a + b; }
    public double Add(double a, double b)
    { return a + b }
}
// Using the class
MathOperations math = new MathOperations();
```

```
Console.WriteLine(math.Add(2, 3));

Console.WriteLine(math.Add(2.5, 3.5));
```

Method Overriding:

```
// Parent class

public class Animal

{

    public virtual void Sound()

    { Console.WriteLine("Animal makes a
sound.");}

}

// Child class

public class Dog : Animal

{

    public override void Sound()

    {  Console.WriteLine("Dog barks."); }

}

// Using overriding

Animal myAnimal = new Dog();
```

```
myAnimal.Sound(); // Output: Dog barks
```

Abstraction:

Abstraction hides implementation details and only exposes essential features. In C#, abstraction is achieved using:

1. **Abstract Classes**
2. **Interfaces**

Abstract Class:

An abstract class cannot be instantiated. It can have both abstract (without implementation) and non-abstract methods.

```
public abstract class Shape

{

    public abstract void Draw(); // Abstract
method

    public void DisplayInfo()

    {Console.WriteLine("This is a shape.");}

}

public class Circle : Shape

{
    public override void Draw()

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

}

// Using abstraction

Shape shape = new Circle();

shape.Draw();

shape.DisplayInfo();
```

Interface

An interface defines a contract. A class implementing an interface must provide implementations for all its methods.

```
public interface IShape

{void Draw();}


public class Square : IShape

{

    public void Draw()

    { Console.WriteLine("Drawing a square.");}
}

// Using an interface

IShape shape = new Square();

shape.Draw();
```

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 refers to the region of the program where a variable or method is accessible

a. Local Scope

ariables declared inside a method or block ({}) have local scope.

They are only accessible within the method or block where they are defined.

Ex.

```
public void PrintMessage()
{
    int number = 10; // Local variable
    Console.WriteLine(number); // Accessible here
}
// Console.WriteLine(number); // Error: 'number' does not access here
```

b. Class Scope

Variables declared inside a class but outside any method are accessible to all methods of the class.

These are often called fields or class members.

ex.

```
public class Example

{

    private int number; // Class-level variable

    public void SetNumber(int value)

    {number = value; // Accessible here}

    public void PrintNumber()

{Console.WriteLine(number); // Accessible here}

}
```

c. Namespace Scope:

Classes, methods, or variables declared at the namespace level are accessible to all other classes within the same namespace.

```
namespace MyNamespace

{

    public class ClassA

   {public int Value = 10;}

    public class ClassB

    {
        public void DisplayValue()
        {
            ClassA obj = new ClassA();
            Console.WriteLine(obj.Value); // Accessible within the same namespace
        }
    }
}
```

d. Global Scope

Members declared as static in a class can be accessed globally without creating an object.

```
public class GlobalExample
{public static int GlobalValue = 100;}
// Accessible globally
Console.WriteLine(GlobalExample.GlobalValue);
```

Accessibility Modifiers:

public: Accessible from anywhere in the program.

ex.public class Example

```
{public int Value = 10; // Accessible from outside}

Example obj = new Example();

Console.WriteLine(obj.Value); // Output: 10
```

private: Accessible only within the class where it is defined.

Ex.

```
public class Example
{
    private int Value = 10; // Accessible only within the class

    public void DisplayValue()
    {
        Console.WriteLine(Value); // Accessible here
    }

}


// obj.Value; // Error: 'Value' is inaccessible due to its protection level
```

protected: Accessible within the class and derived classes.

Ex.

```
public class Parent

{

    protected int Value = 20; // Accessible in derived class

}

public class Child : Parent

{

    public void DisplayValue()

{Console.WriteLine(Value); // Accessible here}

}
```

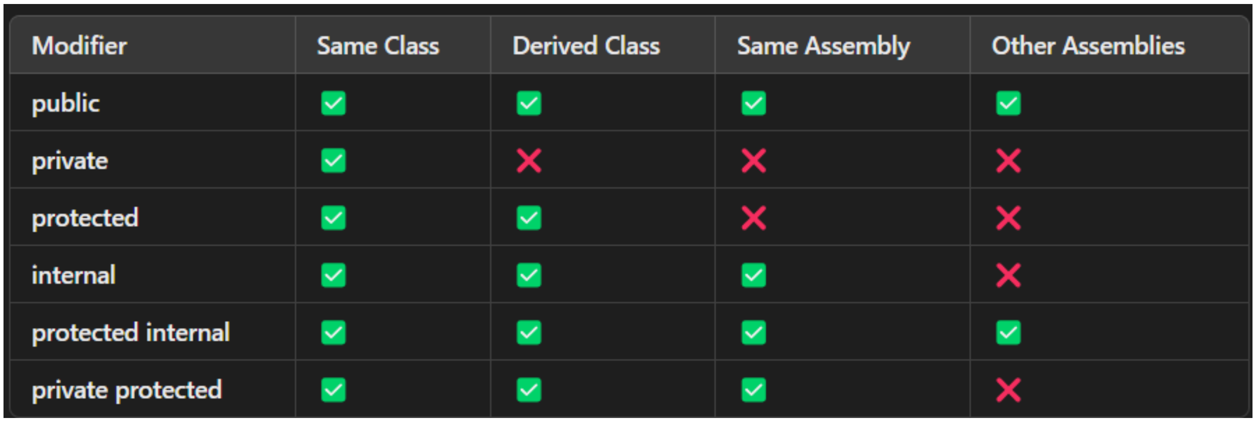
internal: Accessible only within the same assembly.

Ex.

```
internal class InternalExample
{
    public int Value = 30; // Accessible within the same assembly
}
// Accessible in the same assembly but not in another project
```

protected internal: Accessible within the same assembly and by derived classes.

private protected: Accessible only within the containing class and derived classes in the same assembly.



Modifier	Same Class	Derived Class	Same Assembly	Other Assemblies
public	✅	✅	✅	✅
private	✅	❌	❌	❌
protected	✅	✅	❌	❌
internal	✅	✅	✅	❌
protected internal	✅	✅	✅	✅
private protected	✅	✅	✅	❌

Namespaces and .NET Libraries

namespaces and .NET libraries are essential concepts that organize and provide access to reusable code

Namespaces

A namespace is a logical grouping of related classes, interfaces, enums, and other types in C#. It helps avoid naming conflicts and makes code more organized.

You can create a namespace using the namespace keyword.

Ex.

```
namespace MyNamespace

{

    public class MyClass
    {
        public void DisplayMessage()
  {Console.WriteLine("Hello from MyNamespace!");
        }
    }
}
using MyNamespace;
MyClass obj = new MyClass();
obj.DisplayMessage(); // Output: Hello from MyNamespace!
```

You can nest namespaces to create a hierarchy.

System Namespace:

The System namespace is one of the most important namespaces in C#. It contains fundamental classes and base types like Console, String, Int32, etc.

Ex.

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, World!"); // Console is part of the System namespace
    }
}
```

.NET Libraries

The .NET Libraries are a collection of reusable classes, interfaces, and methods provided by the .NET Framework, .NET Core, or .NET 5/6+. These libraries simplify common programming tasks like file handling, database operations, and network communication.

Namespaces: Logical grouping of related types (e.g., `System`, `System.IO`).

Assemblies: Physical files (DLLs) that contain compiled code for these namespaces.

System Namespace:

Contains basic types and fundamental classes.

Common classes: `Console`, `String`, `Math`, `DateTime`

System.Collections Namespace

Provides classes for data structures like lists, queues, dictionaries, and hash tables.

Common classes: `ArrayList`, `Hashtable`, `Queue`

System.Collections.Generic Namespace

Provides type-safe collections (generic collections).

Common classes: `List<T>`, `Dictionary<TKey, TValue>`

System.IO Namespace

Handles input/output operations like reading/writing files and directories.

Common classes: `File`, `Directory`, `StreamReader`, `StreamWriter`

System.Linq Namespace

Provides Language Integrated Query (LINQ) capabilities for querying collections and databases.

Common methods: `Where`, `Select`, `OrderBy`

System.Net Namespace

Supports network operations like sending HTTP requests.

Common classes: `HttpClient`, `WebClient`

System.Text Namespace

Provides classes for text manipulation and encoding.

Common classes: `StringBuilder`, `Encoding`

Key Points

Namespaces organize code logically and prevent naming conflicts.

.NET Libraries provide a vast collection of pre-built classes for almost every task

Enumerations:

An enum is a distinct value type that defines a collection of constants under a single type.

It makes your code more readable and less error-prone by replacing magic numbers or strings with meaningful names.

Defining an Enumeration:

enum keyword to define an enumeration

By default the underlying type of an enum is int, and the values start at 0 and increment by 1.

Syntax:

```
[accessspecifier] enum Enumname: underlying

{

    Val1

    val2

}
```

Specifying Custom Values:

You can assign specific integer values to the enum members

```
enum Severity
{
    Low = 1,
    Medium = 5,
    High = 10,
    Critical = 20
}
```

Ex.

```
enum Days
{
    Sunday,    // 0
    Monday,    // 1
    Tuesday,   // 2
    Wednesday, // 3
    Thursday,  // 4
    Friday,    // 5
    Saturday   // 6
}
```

Converting Enum to Integer:

```
Days day = Days.Friday;
int dayValue = (int)day;
Console.WriteLine(dayValue); // Output: 5
```

Converting Integer to Enum:

```
int dayValue = 3;
Days day = (Days)dayValue;
Console.WriteLine(day); // Output: Wednesday
```

Parsing Enum from String:

```
string input = "Thursday";
Days day = (Days)Enum.Parse(typeof(Days), input);
Console.WriteLine(day); // Output: Thursday
```

Getting All Enum Values:

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

{ Console.WriteLine(day);}
```

Enum with Flags:

The `[Flags]` attribute allows an enum to represent a combination of values, often used for bitwise operations.

```
[Flags]

enum FileAccess

{

    Read = 1,    // 001 in binary

    Write = 2,   // 010 in binary

    Execute = 4  // 100 in binary

}

FileAccess permissions = FileAccess.Read | FileAccess.Write;
```

Data table:

NET framework that represents a single table of in-memory data.

It is part of the **System.Data** namespace and is often used to store, manipulate, and manage data in applications, especially when working with databases or structured data.

Creating a DataTable:

Syntax:

```
Datatable tablevar = new datatable(“tablename”)
```

Adding Columns:

```
DataTable table = new DataTable();
table.Columns.Add("ColumnName",typeof(DataType);
```

Datatype: int,string,decimal,datetime,bool

Adding Rows:

add rows using the Rows.Add method.

```
DataRow newRow = table.NewRow();

newRow["ID"] = 3;

newRow["Name"] = "Charlie";
```

```
newRow["Age"] = 28;

table.Rows.Add(newRow);
```

Accessing Data:

access data in a DataTable using `Rows` and `Columns`.

```
int id = (int)table.Rows[0]["ID"];

string name = table.Rows[0]["Name"].ToString();
```

Manipulating DataTable:

1. **Filtering Data**

filter rows using the `Select` method.

```
DataRow[] filteredRows = table.Select("Age > 25");

foreach (DataRow row in filteredRows)

{

    Console.WriteLine(row["Name"]);

}
```

2.**Sorting Data**

sort rows using the `DefaultView` property.

```
table.DefaultView.Sort = "Name ASC";
foreach (DataRowView row in table.DefaultView)
{
    Console.WriteLine(row["Name"]);
}
```

3. **Deleting Rows:**

delete rows using the Delete method.

```
table.Rows[0].Delete();
```

Primary Key:

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

Cloning and Copying:

Clone: Copies the structure of the DataTable.

```
DataTable clonedTable = table.Clone();
```

Copy: Copies both the structure and data.

```
DataTable copiedTable = table.Copy();
```

```
}
```

Exception Handling:

An exception is an error or unexpected event that occurs during the execution of a program.

Ex.

Division by zero

Accessing a null object

File not found

Invalid type conversion

1. try-catch Block:

```
try
{
    int result = 10 / 0; // This will throw a DivideByZeroException
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: Division by zero is not allowed.");
}
```

2. Multiple catch Blocks:

```
try
{
    int[] numbers = { 1, 2, 3 };
    Console.WriteLine(numbers[5]); // Throws IndexOutOfRangeException
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Error: Index is out of range.");
}
catch (Exception ex)
{
    Console.WriteLine($"General Error: {ex.Message}");
}
```

3. finally Block

The finally block is optional and is used to execute code regardless of whether an exception occurs or not

```
try
{
    Console.WriteLine("Opening a file...");
    // Simulate file operation
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
finally
{
    Console.WriteLine("Closing the file...");
}
```

```
}
```

4. throw Statement

```
try
{
    throw new InvalidOperationException("Custom error occurred.");
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.Message);
}
```

Throw:

The throw statement rethrows the original exception without altering its **stack trace**.

Ex.

```
try
{
    int result = 10 / 0; // Throws DivideByZeroException
}
```

```
catch (Exception ex)
{
    Console.WriteLine("Caught exception. Rethrowing...");
    throw; // Rethrows the original exception
}
```

Throw ex:

The `throw ex` statement resets the **stack trace** of the exception.

```
try
{
    int result = 10 / 0; // Throws DivideByZeroException
}
catch (Exception ex)
{
    Console.WriteLine("Caught exception. Throwing with 'throw ex'...");
    throw ex; // Resets the stack trace
```

}

String Class:

String class is a part of the **System** namespace and is used to work with text.

Strings in C# are immutable, meaning once created, their value cannot be changed.

String Literal: Text enclosed in double quotes

```
string str1 = "Hello, World!";
```

String Object: Using the String class

```
String str2 = new String("Hello, C#!");
```

String method:

concat(),substring(),replace(),trim(),toUpper(),

toLower(),contains(),split(),join(),startwith(),

endswith(),indexof()

Interpolated Strings:

```
int age = 25;

string name = "John";
string message = $"My name is {name} and I am {age} years old.";
```

```
Console.WriteLine(message);
```

DateTime Class:

The **DateTime** class is used to work with dates and times.

It provides various methods and properties to handle tasks like getting the current date, formatting dates, calculating differences, and more.

Creating DateTime Objects:

Using DateTime.Now:

Syntax:

```
Datetime var = datetime.method;
```

```
DateTime now = DateTime.Now;
```

Using DateTime.Today

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

Specifying a Date and Time

You can create a DateTime object by specifying the year, month, day, hour, minute, second, and millisecond.

```
DateTime specificDate = new DateTime(2024, 12, 15, 14, 30, 45);

Console.WriteLine(specificDate); // Output: 12/15/2024 2:30:45 PM
```

Using DateTime.Parse

Converts a string representation of a date and time into a DateTime object.

```
DateTime parsedDate = DateTime.Parse("15 December 2024");

Console.WriteLine(parsedDate); // Output: 12/15/2024 12:00:00 AM
```

Using DateTime.TryParse

Safely parses a string into a DateTime object without throwing exceptions.

```
string dateString = "15/12/2024";

if (DateTime.TryParse(dateString, out DateTime result))
```

```
{
    Console.WriteLine(result); // Output: 12/15/2024 12:00:00 AM
}
else
{
    Console.WriteLine("Invalid date format");
}
```

Common properties:

now(),Today(),Utcnow(),Day(),Hour(),Minute(),

Month(),Year(),Second()

ex.

```
DateTime now = DateTime.Now;
Console.WriteLine($"Year: {now.Year}, Month: {now.Month}, Day: {now.Day}");
Console.WriteLine($"Hour: {now.Hour}, Minute: {now.Minute}, Second: {now.Second}");
```

Common Methods:

1. Add Methods:

AddDays(),AddMonths(),AddYears(),AddHours(),

AddMinutes()

Ex.

```
DateTime now = DateTime.Now;

DateTime futureDate=now.AddDays(10).AddHours(5);

Console.WriteLine(futureDate);
```

2. Subtract:

Used to calculate the difference between two dates.

```
DateTime startDate = new DateTime(2024, 12, 1);

DateTime endDate = new DateTime(2024, 12, 15);

TimeSpan difference = endDate - startDate;

Console.WriteLine($"Days Difference:
{difference.Days}"); // Output: 14
```

3. Compare:

Compares two dates and returns:

-1 if the first date is earlier.

0 if the dates are equal.

1 if the first date is later.

Ex.

```
DateTime date1 = new DateTime(2024, 12, 1);

DateTime date2 = new DateTime(2024, 12, 15);

int comparison = DateTime.Compare(date1, date2);

Console.WriteLine(comparison); // Output: -1
```

4. ToString:

Formats a DateTime object as a string

```
DateTime now = DateTime.Now;

Console.WriteLine(now.ToString("MM/dd/yyyy"));
// Output: 12/15/2024

Console.WriteLine(now.ToString("dddd, dd MMMM
yyyy")); // Output: Sunday, 15 December 2024
```

Formatting Dates:

“d”:Short date

“D”:Long date

“t”:Short time

“T”:Long time

“f”:Full date and short time

“F”:Full date and time

C# provides three types of DateTime values:

DateTime.Local: Represents local time.

DateTime.Utc: Represents UTC (Coordinated Universal Time).

DateTime.Unspecified: No time zone specified.

Basic File Operations:

C# provides robust support for working with files and directories using classes from the **System.IO** namespace.

You can perform basic file operations like creating, reading, writing, appending, copying, moving, and deleting files easily.

System.IO:

file:Provides static methods for file operations.

FileInfo: Provides instance methods for file operations

StreamReader: Reads data from a file.

StreamWriter: Writes data to a file.

1. Creating a File:

Using File.Create

```
using System.IO;

string filePath = "example.txt";

File.Create(filePath).Close(); // Close is required to release the file

Console.WriteLine("File created successfully.");
```

2. Writing to a File:

Using File.WriteAllText

```
string filePath = "example.txt";
```

```
File.WriteAllText(filePath, "This is new content.");

Console.WriteLine("Content written to file.");
```

3. Reading from a File:

Using `File.ReadAllText`

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

Console.WriteLine("File Content:");

Console.WriteLine(content);
```

4. Checking if a File Exists:

```
string filePath = "example.txt";

if (File.Exists(filePath))

{

    Console.WriteLine("File exists.");

}

else

{

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

5. Deleting a File:

```
string filePath = "example.txt";

if (File.Exists(filePath))

{

    File.Delete(filePath);

    Console.WriteLine("File deleted
successfully.");

}

else

{

    Console.WriteLine("File not found.");

}
```

6. Copying a File:

Copies a file to a new location.

```
string sourceFile = "example.txt";

string destinationFile = "example_copy.txt";
```

```
File.Copy(sourceFile, destinationFile, overwrite: true);

Console.WriteLine("File copied successfully.");
```

7. Moving or Renaming a File:

Moves a file to a new location or renames it.

```
string sourceFile = "example.txt";

string destinationFile = "new_example.txt";

File.Move(sourceFile, destinationFile);

Console.WriteLine("File moved/renamed successfully.");
```

8. File Information:

Using FileInfo

```
FileInfo fileInfo = new FileInfo("example.txt");

Console.WriteLine($"File Name: {fileInfo.Name}");

Console.WriteLine($"File Size: {fileInfo.Length} bytes");

Console.WriteLine($"Created On: {fileInfo.CreationTime}");
Console.WriteLine($"Last Accessed:
{fileInfo.LastAccessTime}");
```