The **Common Language Runtime** (**CLR**), the [**virtual machine**](https://en.wikipedia.org/wiki/Virtual_machine)**component of**[**Microsoft**](https://en.wikipedia.org/wiki/Microsoft)[**. NET Framework**](https://en.wikipedia.org/wiki/.NET_Framework), manages the execution of .NET programs.

[Just-in-time compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation) converts the [managed code](https://en.wikipedia.org/wiki/Managed_code) (compiled [intermediate language](https://en.wikipedia.org/wiki/Intermediate_language) code) into [machine instructions](https://en.wikipedia.org/wiki/Machine_instructions) which are then executed on the [CPU](https://en.wikipedia.org/wiki/CPU) of the computer.

**The CLR provides additional services including**

[memory management](https://en.wikipedia.org/wiki/Memory_management) | [type safety](https://en.wikipedia.org/wiki/Type_safety) | [exception handling](https://en.wikipedia.org/wiki/Exception_handling) | [garbage collection](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) | security | [thread management](https://en.wikipedia.org/wiki/Thread_management).

All programs written for the .NET Framework, regardless of [programming language](https://en.wikipedia.org/wiki/Programming_language), are executed by the CLR. All versions of the .NET Framework include CLR.

**CLR implements the**[**Virtual Execution System**](https://en.wikipedia.org/wiki/Virtual_Execution_System)**(VES)** as defined in the [Common Language Infrastructure](https://en.wikipedia.org/wiki/Common_Language_Infrastructure) (CLI) standard, initially developed by Microsoft itself. A public standard defines the Common Language Infrastructure specification.

With Microsoft's move to [.NET Core](https://en.wikipedia.org/wiki/.NET_Core), the CLI VES implementation is known as [CoreCLR](https://en.wikipedia.org/wiki/CoreCLR) instead of CLR.

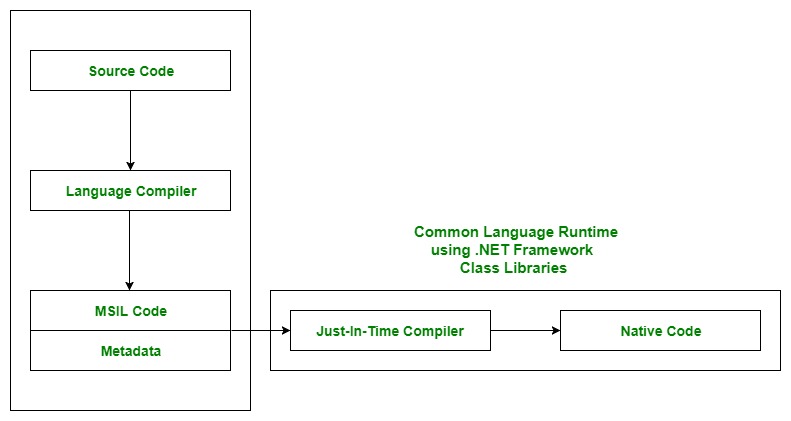
# Role of CLR in the execution of a C# program

Suppose you have written a C# program and save it in a file which is known as the **Source Code.**

**Language specific compiler** compiles the source code into the **MSIL(Microsoft Intermediate Language)**|  **CIL(Common Intermediate Language)** | **IL(Intermediate Language)** along with its **metadata.**

Metadata includes all the types, actual implementation of each function of the program. MSIL is machine-independent code.

Now CLR comes into existence. CLR provides the services and runtime environment to the MSIL code. Internally CLR includes the JIT(Just-In-Time) compiler which converts the MSIL code to machine code which further executed by CPU. CLR also uses the .NET Framework class libraries. Metadata provides information about the programming language, environment, version, and class libraries to the CLR by which CLR handles the MSIL code. As CLR is common so it allows an instance of a class that written in a different language to call a method of the class which written in another language.



CLR

.NET is an open-source developer platform (Languages + Libraries) for building different types of apps.

After Installing .NET SDK

**Create your .NET app**

In your command prompt, run the following command to create your app:

dotnet new console -o MyApp || dotnet new console –-name MyApp || dotnet new console -n MyApp

### **What do these commands mean?**

* The dotnet new console command creates a new console app for you.
* The -o, -n, --name parameter creates a directory named MyApp where your app is stored and populates it with the required files. (cd MyApp)

**To run the dotnet code:** dotnet run

# What is C#?

It is an **object-oriented programming language** created by **Microsoft** that runs on the .NET Framework.

C# is used for: Mobile applications | Desktop applications | Web applications | Web services | Web sites | Games | VR | Database applications | API application | distributed application

# First Program (Hello World)

using System; // importing other libraries

namespace HelloWorld // namespaces are used to group similar classes

{

class Program

{

static void Main(string[] args) // entry point of code

{

Console.WriteLine("Hello World!");

}

}

}

Result: Hello World!

**Line 1:** using System means that we can use classes from the System namespace.

**Line 2:** A blank line. C# ignores white space. However, multiple lines make the code more readable.

**Line 3:** namespace is used to organize your code, and it is a container for classes and other namespaces.

**Line 4:** The curly braces {} marks the beginning and the end of a block of code.

**Line 5:** class is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

**Line 7:** Another thing that always appear in a C# program, is the Main method. Any code inside its curly brackets {} will be executed.

**Line 9:** Console is a class of the System namespace, which has a WriteLine() method that is used to output/print text. In our example it will output "Hello World!".

If you omit the using System line, you would have to write System.Console.WriteLine() to print/output text.

**Note:** Every C# statement ends with a semicolon ;.

**Note:** C# is case-sensitive: "MyClass" and "myclass" has different meaning.

**Note:** Unlike [Java](https://www.w3schools.com/java/default.asp), the name of the C# file does not have to match the class name, but they often do (for better organization). When saving the file, save it using a proper name and add ".cs" to the end of the filename. (represent c sharp(.cs))

Hello World!

# User Input and Output to the Screen/ Console

The WriteLine() prints the output on a new line each time

while Write() prints on the same line

**Note:** That you should remember to add spaces when needed, for better readability while using Write() method.

The Console.ReadLine() to get user input, it returns a string.

string s = Console.ReadLine(); // string input

int i = Convert.ToInt32(Console.ReadLine()); // int input

# Comments

Single-line Comments = //…

Multiline Comments = /\*…\*/

# Variable

It is a way to represent memory location through symbol so that it can be easily identified.

|  |  |
| --- | --- |
| Variable Type | Example |
| Decimal types | decimal |
| Boolean types | True or false value, as assigned |
| Integral types | int, char, byte, short, long |
| Floating point types | float and double |
| Nullable types | Nullable data types |

You can declare a variable without assigning the value, and assign the value later:

int myNum;

myNum = 15;

Console.WriteLine(myNum);

If you assign a new value to an existing variable, it will overwrite the previous value:

int myNum = 15;

myNum = 20; // myNum is now 20

Console.WriteLine(myNum);

To declare more than one variable of the **same type**, use a comma-separated list:

int x = 5, y = 6, z = 50;

Console.WriteLine(x + y + z);

# Constants

However, you can add the const keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

**const** int myNum = 15;

myNum = 20; // error

The const keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant, is PI (3.14159...).

**Note:** You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

# Datatypes

|  |  |
| --- | --- |
| Types | Data Types |
| Value Data Type | short, int, char, float, double etc. |
| Reference Data Type | String, Class, Object and Interface |
| Pointer Data Type | Pointers |

int myNum = 5; // Integer (whole number)

double myDoubleNum = 5.99D; // Floating point number

char myLetter = 'D'; // Character

bool myBool = true; // Boolean

string myText = "Hello"; // String

Note that you should end the value Long with an "L", Float with a “F”, Double with a “D”

***Integer***

|  |  |  |
| --- | --- | --- |
| int | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| short | 2 bytes | -32,768 to 32,767 |
| long | 8 bytes | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

***Floating Point***

A floating point number can also be a scientific number with an "e" to indicate the power of 10

|  |  |  |
| --- | --- | --- |
| float | 4 bytes | 3.4e−038 to 3.4e+038, 6 to 7 decimal digits |
| double | 8 bytes | 1.7e−308 to 1.7e+308, 15 decimal digits |

***Booleans***

|  |  |  |
| --- | --- | --- |
| bool | 1 bit | Stores true or false values |

***Characters***

|  |  |  |
| --- | --- | --- |
| char | 2 bytes | Stores a single character/letter, surrounded by **single quotes** |

***Strings***

|  |  |  |
| --- | --- | --- |
| string | 2 bytes per character | Stores a sequence of characters, surrounded by **double quotes** |

# Integer (Int)

int max = int.MaxValue;

int min = int.MinValue;

Console.WriteLine($"Max Value is {max} & Min Value is {min}");

int maxA = max + 3;

Console.WriteLine(maxA);// Example of integer overflow

Max Value is 2147483647 & Min Value is -2147483648

-2147483646

If a calculation produces a value that exceeds those limits, you have an**underflow** or **overflow condition.** The answer appears to wrap from one limit to the other.

Notice that the answer is very close to the minimum (negative) integer. It's the same as min + 2. The addition operation overflowed the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

# Double

The decimal type has a smaller range but greater precision than double.

float f1 = 1.0F;

float f2 = 3.0F;

double d1 = 1.0D;

double d2 = 3.0D;

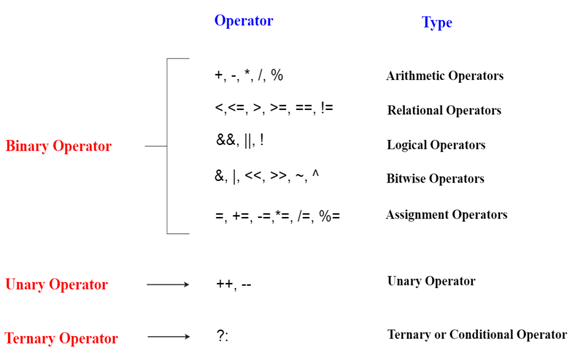
decimal m1 = 1M;

decimal m2 = 3M;

Console.WriteLine($"Float = {f1/f2}  double = {d1/d2}  decimal = {m1/m2}");

Float = 0.33333334 double = 0.3333333333333333 decimal=0.3333333333333333333333333333

# Operators



A unary operation is an operation with only one operand, i.e. a single input.

A binary operation or dyadic operation is a rule for combining two elements to produce another element.

# Type Casting

* **Implicit Casting** (automatically) - converting a smaller type to a larger type size  
  char -> int -> long -> float -> double

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble); // Outputs 9

***Note*** *- cannot implicitly convert type 'string' to 'int'*.

* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

Console.WriteLine(myDouble); // Outputs 9.78

Console.WriteLine(myInt); // Outputs 9

# Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long)

int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt));    // Convert int to string // 10

Console.WriteLine(Convert.ToDouble(myInt));    // Convert int to double // 10

Console.WriteLine(Convert.ToInt32(myDouble));  // Convert double to int // 5

Console.WriteLine(Convert.ToString(myBool));   // Convert bool to string // True

# String Interpolation

Another option of string concatenation is **string interpolation**, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation:

string firstName = "John";

string lastName = "Doe";

string name = $"My full name is: {firstName} {lastName}";

Console.WriteLine(name);

Result: name is: John Doe

# Ternary Operator

variable *= (*condition*) ?* expressionTrue *:*  expressionFalse*;*

# Looping through a Array

Method 1 for loop

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.Length; i++) // Length is a property

{

Console.WriteLine(cars[i]); // Can modify the value of i

}

Method 2 for each loop

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

foreach (string i in cars)

{

Console.WriteLine(i); // Can’t modify the value of i

}

## Ways to Create an Array

// Create an array of four elements, and add values later

string[] cars = new string[4];

// Create an array of four elements and add values right away

string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size

string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

**Note** If you declare an array and initialize it later, you have to use the new keyword:

// Declare an array

string[] cars;

// Add values, using new

cars = new string[] {"Volvo", "BMW", "Ford"};

// Add values without using new (this will cause an error)

cars = {"Volvo", "BMW", "Ford"};

To create multidimensional array, we need to use comma inside the square brackets. For example:

1. **int**[,] arr=**new** **int**[3,3];//declaration of 2D array
2. **int**[,,] arr=**new** **int**[3,3,3];//declaration of 3D array

# C# Multidimensional Array Example: Declaration and initialization at same time

There are 3 ways to initialize multidimensional array in C# while declaration.

**int**[,] arr = **new** **int**[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

We can omit the array size.

**int**[,] arr = **new** **int**[,]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

We can omit the new operator also.

**int**[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

# Minimum and Maximum of array

using System;

namespace DemoApp {

class Program

{

public int[] Print(int[] a)

{

var ans = new int[2];

bool first = true;

int min=-1, max=-1;

for (int \_ = 0; \_ < a.Length; \_++)

{

if (first)

{

first = false;

min = a[\_];

max = a[\_];

}

else

{

if (min > a[\_])

{

min = a[\_];

}

if (max < a[\_])

{

max = a[\_];

}

}

}

ans[0] = min;

ans[1] = max;

return ans;

}

static void Main(string[] args)

{

int[] arr = {-121,1890098, 1, 2, 39, 3, 7,3232,2323,4,42};

Program pObj = new Program();

int[] x = pObj.Print(arr);

Console.WriteLine($"{x[0]} {x[1]}");

}

}

}

# Method / Functions

When a **parameter** is passed to the method, it is called an **argument**.

The reason is simple: a static method can be accessed without creating an object of the class, while public methods can only be accessed by objects.

# Method Overloading

*Using different data type*

        static int add(int a, int b){

            return a + b;

        }

        static double add(double a, double b){

            return a + b;

        }

        static void Main(string[] args){

            int z = add(1, 2);

            double y = add(1.1, 2.3);

            Console.WriteLine($"{z} {y}");// 3 3.3

*Using different parameter*

static int add(int a, int b){

            return a + b ;

        }

        static int add(int a, int b, int c){

            return a + b + c;

        }

        static void Main(string[] args){

            int z = add(1, 2);

            int y = add(1, 2, 3);

            Console.WriteLine($"{z} {y}");// 3 6

# Call by Reference & Call by Value

using System;

namespace DemoApp {

class Program

{

public void change(ref int x)

{

x += 1;

}

static void Main(string[] args)

{

int a = 12;

Program pObj = new Program();

Console.WriteLine(a);

pObj.change(ref a);

Console.WriteLine(a);

}

}

}

For call by reference use ref or out keyword (Reference of an arguments are passed as a parameters | Change actual value of parameters)

Output 12 13

For call by value remove ref keyword (Copy of an arguments are passed as a parameters | Do not change actual value of parameters)

Output 12 12

# Objects and Class

using System;

namespace DemoApp {

public class Student

{

private int Id;

private string Name;

public Student(int i, string n) // Encapsulation

{

Id = i;

Name = n;

}

public void showData()

{

Console.WriteLine($"{Id} {Name}");

}

}

public class Program

{

static void Main(string[] args)

{

Student s1 = new Student(3, "saranj");

Student s2 = new Student(11, "mohan");

s1.showData();

s2.showData();

}

}

}

**Output**

3 saranj

11 mohan

# Constructors

A constructor is a **special method** that is used to **initialize objects**. The advantage of a constructor is that it is called when an object of a class is created. It can be used to set initial values for fields:

// Create a Car class

class Car{

public string model; // Create a field

public Car(){ // Create a **class constructor** for the Car class

model = "Mustang"; // Set the initial value for model

}

static void Main(string[] args){

Car Ford = new Car(); // Create an object of the Car Class (this will **call the constructor**)

Console.WriteLine(Ford.model); // Print the value of model

}

}

// Outputs "Mustang"

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

## Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

class Car{

public string model;

public string color;

public int year;

// Create a class constructor with multiple parameters

public Car(string modelName, string modelColor, int modelYear){

model = modelName;

color = modelColor;

year = modelYear;

}

static void Main(string[] args){

Car Ford = new Car("Mustang", "Red", 1969);

Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);

}

}

// Outputs Red 1969 Mustang

Constructor saves time, provide code reusability.

# **Access Modifier**

**Access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

|  |  |
| --- | --- |
| Modifier | Description |
| public | The code is accessible for all classes |
| private | The code is only accessible within the same class |
| protected | The code is accessible within the same class, or in a class that is inherited from that class. |
| internal | The code is only accessible within its own assembly, but not from another assembly. |

There are also two combinations: protected internal and private protected.

### **Why Access Modifiers?**

To control the visibility of class members (the security level of each individual class and class member).

To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private.

# Encapsulation

The meaning of Encapsulation is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare fields/variables as private
* provide public get and set methods, through **properties**, to access and update the value of a private field

class Person{

public string Name // property

{ get; set; }

}

class Program{

static void Main(string[] args){

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

The output will be: Liam

## Why Encapsulation?

* Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
* Fields can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data

# Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

* **Derived Class** (child) - the class that inherits from another class
* **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

class Vehicle // base class (parent)

{

public string brand = "Ford"; // Vehicle field

public void honk() // Vehicle method

{

Console.WriteLine("Tuut, tuut!");

}

}

class Car : Vehicle // derived class (child)

{

public string modelName = "Mustang"; // Car field

}

class Program

{

static void Main(string[] args)

{

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (From the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class

Console.WriteLine(myCar.brand + " " + myCar.modelName);

}

}

#### **Why And When To Use "Inheritance"?**

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

## The sealed Keyword

If you don't want other classes to inherit from a class, use the sealed keyword

# Polymorphism

class Animal // Base class (parent)

{

public **virtual** void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

The output will be:

The animal makes a sound  
The pig says: wee wee  
The dog says: bow wow

The output will be without a virtual keyword:

The animal makes a sound  
The animal makes a sound  
The animal makes a sound

### **Not The Output I Was Looking For**

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the virtual keyword to the method inside the base class, and by using the override keyword for each derived class methods:

# Abstraction

## Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/cs/cs_interface.asp).

The abstract keyword is used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

// Abstract class

abstract class Animal{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep(){

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound(); // Call the abstract method

myPig.sleep(); // Call the regular method

}

}

The pig says: wee wee  
Zzz

# Interface

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies).

// Interface

interface IAnimal {

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal {

public void animalSound() {// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program {

static void Main(string[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

The pig says: wee wee

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class).
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interfaces can contain properties and methods, but not fields/variables
* Interface members are by default abstract and public
* 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. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

# Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

interface IFirstInterface {

void myMethod(); // interface method

}

interface ISecondInterface {

void myOtherMethod(); // interface method

}

// Implement multiple interfaces

class DemoClass : IFirstInterface, ISecondInterface {

public void myMethod() {

Console.WriteLine("Some text..");

}

public void myOtherMethod() {

Console.WriteLine("Some other text...");

}

}

class Program

{

static void Main(string[] args)

{

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

# Enums

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

Enum is short for "enumerations", which means "specifically listed".

class Program{

enum Level{

Low,

Medium,

High

}

static void Main(string[] args){

Level myVar = Level.Medium;

Console.WriteLine(myVar);

}

}

Output : Medium

enum Months{

January, // 0

February, // 1

March=6, // 6

April, // 7

May, // 8

June, // 9

July // 10

}

static void Main(string[] args){

int myNum = (int) Months.April;

Console.WriteLine(myNum); // Output = 7

}

# Working With Files

The File class from the System.IO namespace, allows us to work with files:

|  |  |
| --- | --- |
| Method | Description |
| AppendText() | Appends text at the end of an existing file |
| Copy() | Copies a file |
| Create() | Creates or overwrites a file |
| Delete() | Deletes a file |
| Exists() | Tests whether the file exists |
| ReadAllText() | Reads the contents of a file |
| Replace() | Replaces the contents of a file with the contents of another file |
| WriteAllText() | Creates a new file and writes the contents to it. If the file already exists, it will be overwritten. |

# Try, catch and finally

try

{

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

Console.WriteLine(myNumbers[10]);

}

catch (Exception e)

{

Console.WriteLine("Something went wrong.");

Console.WriteLine(e.Message);

}

finally

{

Console.WriteLine("The 'try catch' is finished.");

}

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

static void checkAge(int age){

if (age < 18){

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else{

Console.WriteLine("Access granted - You are old enough!");

}

}

static void Main(string[] args){

checkAge(15);

}

# Fibonacci Sequence

using System;

namespace FibonacciSeries

{

    class Fibo{

        static void Main(string[] args){

            //int[] arr = {1, 1};

            var arr = new *List*<int> {1, 1};

            int input = Convert.ToInt32(Console.ReadLine());

            while(arr.Count != input){

                int lastElement = arr[arr.Count - 1];

                int secondLastElement = arr[arr.Count - 2];

                arr.Add(lastElement + secondLastElement);

            }

            foreach (var i in arr) //(int i in arr)

            {

            Console.WriteLine(i); // Can’t modify the value of i

            }

        }

    }

}

# Record Type

A record type in C# 9 is a lightweight, immutable data type (or a lightweight class) that has read-only properties only. Because a record type is immutable, it is thread-safe and cannot mutate or change after it has been created and helps improve memory management. You can initialize a record type only inside a constructor.

It also makes your code more readable and easier to maintain. An immutable object is defined as an object that, once created, can’t change.

**Features**

* Used for immutable objects
* With expression support
* Value based equality support

Init-only properties can be used to make the individual properties of an object immutable, and records can be used to make the entire object immutable.

using System;

namespace DemoProject

{

public record Member

{

public int Id { get; init; }

public string FirstName { get; init; }

public string LastName { get; init; }

public string Address { get; init; }

}

class Program

{

public static void Main(string[] args)

{

// Can take parameterized or non-parameterized constructor

// Then value can given as below or Member() or Member {}

var member = new Member

{

Id = 1,

FirstName = "Kirtesh",

LastName = "Shah",

Address = "Vadodara"

};

Console.WriteLine(member.Id);

Console.WriteLine(member.FirstName);

Console.WriteLine(member.LastName);

Console.WriteLine(member.Address);

}

}

}

# GUID

GUID stands for Global Unique Identifier. A GUID is a 128-bit integer (16 bytes) that you can use across all computers and networks wherever a unique identifier is required.

int? is shorthand for Nullable<int>.

int? a = null; // (Nullable integer)

int b = 20;

var x = a ?? b; // if a is null x = b, if a is not null x = a

Console.WriteLine(x); //20

String.Empty == “”

# Async

 An async method uses the [await operator](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await) to do potentially long-running work without blocking the caller's thread,

# IEnumerable and IEnumerator

They are collections (list collection, objects collection etc)

IEnumerable uses IEnumerator internally and it has simple syntax.

IEnumerator track the current cursor position.

# Task

A task in C# is used to implement Task-based Asynchronous Programming and was introduced with the . NET Framework 4. The Task object is typically executed asynchronously on a thread pool thread rather than synchronously on the main thread of the application.