1. **Boxing and Unboxing**

**Boxing**

The process of converting from a value type to reference type is called boxing

Boxing is an Implicit Conversion

Ex :

int num=123;

object obj=num;

console.writeline(obj);

**Unboxing**

The process of converting from a Reference type to value type is called unboxing

unboxing is an explicit Conversion

Ex :

object obj=123;

int num=(int)obj;

console.writeline(num);

1. **C# Keywords**

Keywords are predefined sets of reserved words that have special meaning in a program. The meaning of keywords cannot be changed, neither can they be directly used as identifiers in a program.

For example,

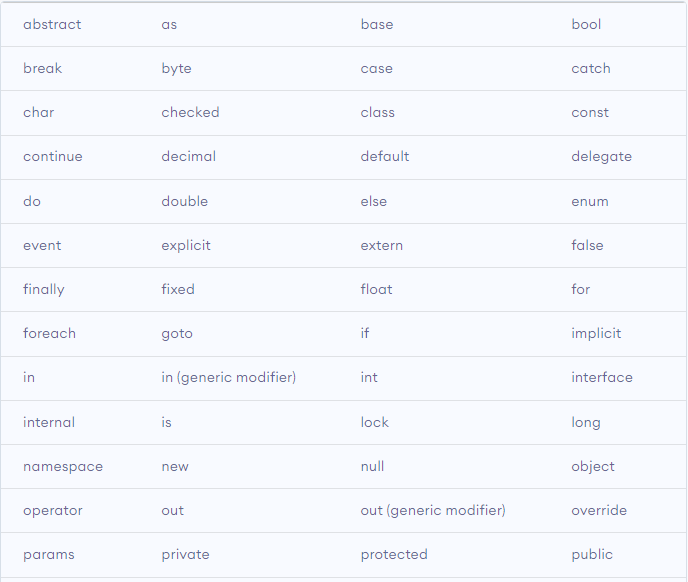
**long mobileNum;**

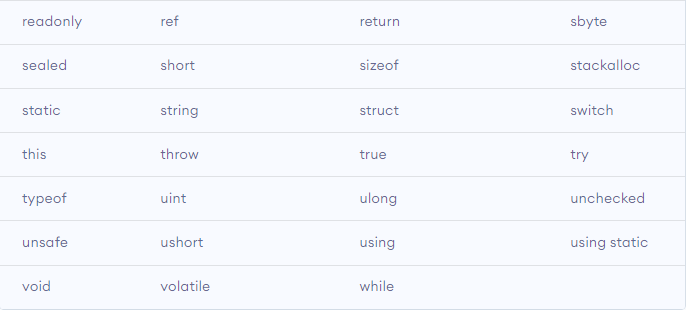
Here, long is a keyword and mobileNum is a variable (identifier). long has a special meaning in C#   
  
i.e. it is used to declare variables of type long and this function cannot be changed.

Also, keywords like long, int, char, etc cannot be used as identifiers. So, we cannot have something like:

**long long;**

C# has a total of 79 keywords.

All these keywords are in lowercase. Here is a complete list of all C# keywords  
  




1. **Control/Conditional Statement**C# provides many decision-making statements that help the flow of the C# program based on certain logical conditions.

Here, you will learn about if, else if, else, and nested if else statements to control the flow based on the conditions.

types are:

1. if

2. multiple if

3. nested if

4. if else

5. else if

**If Statement**

The if statement contains a Boolean condition followed by a single or multi-line code block to be executed. At runtime, if a Boolean condition evaluates to true, then the code block will be executed, otherwise not.

Syntax:

if(condition)

{

// code block to be executed when if condition evaluates to true

}

Example: if Statement

int i = 10, j = 20;

if (i < j)

{

Console.WriteLine("i is less than j");

}

if (i > j)

{

Console.WriteLine("i is greater than j");

}

**else Statement**

The else statement can come only after if or else if statement and can be used only once in the if-else statements. The else statement cannot contain any condition and will be executed when all the previous if and else if conditions evaluate to false.

Example: else Statement

int i = 20, j = 20;

if (i > j)

{

Console.WriteLine("i is greater than j");

}

else if (i < j)

{

Console.WriteLine("i is less than j");

}

else

{

Console.WriteLine("i is equal to j");

}

**Switch Statement**

Switch statement can be used to replace the if else and else if statement in C#.

The advantage of using switch over if else and else if statement is the codes will look much cleaner and readable with switch.

**The syntax of switch statement is:**

*switch (variable/expression)*

*{*

*case value1:*

*// Statements executed if expression(or variable) = value1*

*break;*

*case value2:*

*// Statements executed if expression(or variable) = value1*

*break;*

*... ... ...*

*... ... ...*

*default:*

*// Statements executed if no case matches*

*}*The switch statement evaluates the expression (or variable) and compare its value with the values (or expression) of each case (value1, value2, …). When it finds the matching value, the statements inside that case are executed.

But, if none of the above cases matches the expression, the statements inside default block is executed. The default statement at the end of switch is similar to the else block in if else statement.

However, a problem with the switch statement is, when the matching value is found, it executes all statements after it until the end of switch block.

To avoid this, we use break statement at the end of each case. The break statement stops the program from executing non-matching statements by terminating the execution of switch statement.

**Example :**

using System;

namespace \_1st\_program

{

class Program

{

static void Main(string[] args)

{

char ch;

Console.WriteLine("Enter an alphabet");

ch = Convert.ToChar(Console.ReadLine());

switch (Char.ToLower(ch))

{

case 'a':

Console.WriteLine("Vowel");

break;

case 'e':

Console.WriteLine("Vowel");

break;

case 'i':

Console.WriteLine("Vowel");

break;

case 'o':

Console.WriteLine("Vowel");

break;

case 'u':

Console.WriteLine("Vowel");

break;

default:

Console.WriteLine("Not a vowel");

break;

}

}

}

}

**Difference between Switch and If else Example:**

using System;

namespace \_1st\_program

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("enter a valid number");

string number = Console.ReadLine();

long a = Convert.ToInt64(number);

if (a == 1)

{

Console.WriteLine(a);

}

else if (a == 2)

{

Console.WriteLine(a);

}

else if (a == 3)

{

Console.WriteLine(a);

}

else if (a == 4)

{

Console.WriteLine(a);

}

else

{

Console.WriteLine("Not a valid number");

}

switch (a)

{

case 1:

{

Console.WriteLine(a);

break;

}

case 2:

{

Console.WriteLine(a);

break;

}

case 3:

{

Console.WriteLine(a);

break;

}

case 4:

{

Console.WriteLine(a);

break;

}

default:

Console.WriteLine("Not a valid number");

break;

}

Console.ReadLine();

}

}

}

**Practice Example**

Simple calculator program using C# switch Statement

1. **Loops**  
     
     
   **for Loop**  
     
   The for keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false

Syntax:

for (initializer; condition; iterator)

{

//code block

}

The for loop contains the following three optional sections, separated by a semicolon. Those are

Initializer:

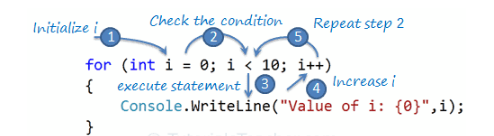
The initializer section is used to initialize a variable that will be local to a for loop and cannot be accessed outside loop. It can also be zero or more assignment statements, method call, increment, or decrement expression e.g., ++i or i++, and await expression.

Condition:

The condition is a boolean expression that will return either true or false. If an expression evaluates to true, then it will execute the loop again; otherwise, the loop is exited.

Iterator:

The iterator defines the incremental or decremental of the loop variable



Example for Loop:

// forward numbers

for (int i = 0; i < 10; i++)

{

Console.WriteLine("forward Value of i: {0}", i);

}

// reverse numbers

for (int i = 10; i > 0; i--)

{

Console.WriteLine("reverse Value of i: {0}", i);

}

// Nested Loop

for (int i = 0; i < 2; i++)

{

for (int j = i; j < 4; j++)

Console.WriteLine("Value of i: {0}, J: {1} ", i, j);

}

**While Loop**

the while loop to repeatedly execute a block of code as long as the specified condition returns true.

**Syntax:**

Initialization;

While(condition)

{

//code block

Inc/dec

}

The while loop starts with the while keyword, and it must include a boolean conditional expression inside brackets that returns either true or false. It executes the code block until the specified conditional expression returns false.

The for loop contains the initialization and increment/decrement parts. When using the while loop, initialization should be done before the loop starts, and increment or decrement steps should be inside the loop.

**Example:**

int i = 0; // initialization

while (i < 10) // condition

{

Console.WriteLine("i = {0}", i);

i++; // increment

}

**do while Loop**

do while loop is the same as while loop except that it executes the code block at least once.

**Syntax:**

Initialization;

do

{

//code block

//inc/dec

}

while(condition);

The do-while loop starts with the do keyword followed by a code block and a boolean expression with the while keyword. The do while loop stops execution exits when a boolean condition evaluates to false. Because the while(condition) specified at the end of the block, it certainly executes the code block at least once.

**Example**

int i = 0;

do

{

Console.WriteLine("i = {0}", i);

i++;

} while (i < 5);

**Break**

In c#, Break statement is useful to break or terminate the execution of loops (for, while, do-while, etc.) or switch statements.

Syntax :

break;

**Continue**

continue statement is used to terminate the current iteration and start the next iteration

Syntax :

Continue;

**Example for above 2** int i;

for (i = 0; i <= 10; i++)

{

if (i == 5)

continue;

if (i == 8)

break;

Console.WriteLine("value is" +i);

}

Console.ReadLine();

**Goto**

The C# goto statement is also known jump statement.

It is used to transfer control to the other part of the program.

It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

**Note**: Currently, it is avoided to use goto statement in C# because it makes the program complex.

**Example**:

repeat:

Console.WriteLine("You are not eligible to vote!");

Console.WriteLine("Enter your age:");

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

if (age < 18)

{

goto repeat;

}

else

{

Console.WriteLine("You are eligible to vote!");

}

**Logical Programs:**

1.Exchange Two integer variable value without using third variable

2.Write a Program Reverse Number - 12345 to 54321 ?

3.Print 1- 100 Prime Numbers ?

4.Fibonacci series?

5. Factorial?

6.Sum of Digits?

7.Armstrong number?

8.Traingle

9.Char Print

10.Palindrome

**Arrays**

an array is a structure representing a fixed length ordered collection of values or objects with the same type.

Arrays make it easier to organize and operate on large amounts of data.

For example, rather than creating 100 integer variables, you can just create one array that stores all those integers!

1.Single Dimensional Array

2.Multidimensional Array

3.Jagged Array

**Syntax:**

An array can be declared using by specifying the type of its elements with square brackets.

datatype[] arrayName;

**Example Array Declaration:**

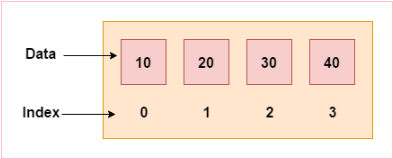
int[] evenNums; // integer array

string [] cities; // string array

**Example Array Declaration & Initialization:**

int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

string[] cities = new string[3]{ "Mumbai", "London", "New York" };



**Advantages:**

Code Optimization (less code)

Random Access

Easy to sort data etc.

**Disadvantages:**

Fixed size and must specify the size also

**Access Array Elements:**

if you want to get the particular value in an array we need to go with Index. based on Index Value we can get any location value from an array.

so, we can get values in an Array

we can change value in an array

from all using Index

index always Start always with \*0\*

**Examples:**

using System;

namespace AccessArray {

class Program {

static void Main(string[] args) {

// create an array

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

//access first element

Console.WriteLine("Element in first index : " + numbers[0]);

//access second element

Console.WriteLine("Element in second index : " + numbers[1]);

//access third element

Console.WriteLine("Element in third index : " + numbers[2]);

Console.ReadLine();

}

}

}

Console.ReadLine();

**Change Array Elements:**

using System;

namespace ChangeArray {

class Program {

static void Main(string[] args) {

// create an array

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

Console.WriteLine("Old Value at index 0: " + numbers[0]);

// change the value at index 0

numbers[0] = 11;

//print new value

Console.WriteLine("New Value at index 0: " + numbers[0]);

Console.ReadLine();

}

}

}

**Iterating C# Array using Loops**

using System;

namespace AccessArrayFor {

class Program {

static void Main(string[] args) {

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

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

Console.WriteLine("Element in index " + i + ": " + numbers[i]);

}

Console.ReadLine();

}

}

}

if you observe in above array i used .Length

Here, the Length property of the array gives the size of the array.

so you can loop dynamically based on length of the Array.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Multidimensional Array**

Arrays can have more than one dimension.

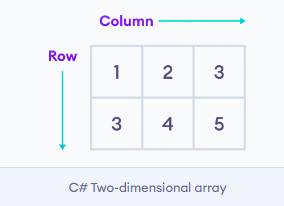
In a multidimensional array, each element of the array is also an array. For example,

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

Here, x is a multidimensional array which has two elements: {1, 2, 3} and {3, 4, 5}. And, each element of the array is also an array with 3 elements.

**Two-dimensional array in C#**

A two-dimensional array consists of single-dimensional arrays as its elements. It can be represented as a table with a specific number of rows and columns.



**Two-Dimensional Array Declaration**

int[ , ] x = new int [2, 3];

**Two-Dimensional Array initialization**

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

**Access Elements from 2D Array**

We use the index number to access elements of a 2D array. For example,

// a 2D array

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

// access first element from first row

x[0, 0]; // returns 1

// access third element from second row

x[1, 2]; // returns 5

// access third element from first row

x[0, 2]; // returns 3

**Change Array Elements**

We can also change the elements of a two-dimensional array. To change the element, we simply assign a new value to that particular index. For example,

using System;

namespace MultiDArray {

class Program {

static void Main(string[] args) {

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

// old element

Console.WriteLine("Old element at index [0, 0] : "+numbers[0, 0]);

// assigning new value

numbers[0, 0] = 222;

// new element

Console.WriteLine("New element at index [0, 0] : "+numbers[0, 0]);

}

}

}

out put :

Old element at index [0, 0] : 2

New element at index [0, 0] : 222

**Iterating C# Array using Loop**

using System;

namespace MultiDArray {

class Program {

static void Main(string[] args) {

int[ , ] numbers = { {2, 3, 9}, {4, 5, 9} };

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

Console.Write("Row "+ i+": ");

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

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

}

Console.WriteLine();

}

}

}

}

Output

Row 0: 2 3 9

Row 1: 4 5 9

In the above example, we have used a nested for loop to iterate through the elements of a 2D array. Here,

numbers.GetLength(0) - gives the number of rows in a 2D array

numbers.GetLength(1) - gives the number of elements in the row

**Jagged Array**

a jagged array consists of multiple arrays as its element. However, unlike multidimensional arrays, each array inside a jagged array can be of different sizes.

**C# Jagged Array Declaration**

Here's a syntax to declare a jagged array in C#.

dataType[ ][ ] nameOfArray = new dataType[rows][ ];

Let's see an example,

// declare jagged array

int[ ][ ] jaggedArray = new int[2][ ];

Here,

int - data type of the array

[][] - represents jagged array

jaggedArray - name of the jagged array

[2][] - represents the number of elements (arrays) inside the jagged array

Since we know each element of a jagged array is also an array, we can set the size of the individual array. For example,

// set size of the first array as 3

jaggedArray[0] = new int[3];

// set size of second array as 2

jaggedArray[1] = new int[2];

Here,

index at the first square bracket represents the index of the jagged array element

index at the second square bracket represents the index of the element inside each element of the jagged array

Initialize without setting size of array elements

**declaring string jagged array**

int[ ][ ] jaggedArray = new int[2] [ ];

**initialize each array**

jaggedArray[0] = new int[] {1, 3, 5};

jaggedArray[1] = new int[] {2, 4};

**Initialize while declaring Jagged Array**

int[ ][ ] jaggedArray = {

new int[ ] {10, 20, 30},

new int[ ] {11, 22},

new int[ ] {88, 99}

};

Accessing elements of a jagged array

We can access the elements of the jagged array using the index number. For example,

// access first element of second array

jaggedArray[1][0];

// access second element of the second array

jaggedArray[1][1];

// access second element of the first array

jaggedArray[0][1];

Example: C# Jagged Array

using System;

namespace JaggedArray {

class Program {

static void Main(string[] args) {

// create a jagged array

int[ ][ ] jaggedArray = {

new int[] {1, 3, 5},

new int[] {2, 4},

};

// print elements of jagged array

Console.WriteLine("jaggedArray[1][0]: " + jaggedArray[1][0]);

Console.WriteLine("jaggedArray[1][1]: " + jaggedArray[1][1]);

Console.WriteLine("jaggedArray[0][2]: " + jaggedArray[0][2]);

Console.ReadLine();

}

}

}

Output

jaggedArray[1][0]: 2

jaggedArray[1][1]: 4

jaggedArray[0][2]: 5

Here, inside a jagged array,

jaggedArray[1][0] - first element of the second array

jaggedArray[1][1] - second element of the second array

jaggedArray[0][2] - third element of the first array

**Iterating through a jagged array**

In C#, we can use loops to iterate through each element of the jagged array. For example,

using System;

namespace JaggedArray {

class Program {

static void Main(string[] args) {

// declare a jagged array

int[][] jaggedArray = new int[2][];

// set size of Jagged Array Elements

jaggedArray[0] = new int[3];

jaggedArray[1] = new int[2];

// initialize the first array

jaggedArray[0][0] = 1;

jaggedArray[0][1] = 3;

jaggedArray[0][2] = 5;

// initialize the second array

jaggedArray[1][0] = 2;

jaggedArray[1][1] = 2;

// outer for loop

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

Console.Write("Element "+ i +": ");

// inner for loop

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

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

}

Console.WriteLine();

}

Console.ReadLine();

}

}

}

Output

Element 0: 1 3 5

Element 1: 2 2

In the above example, we have used a nested for loop to iterate through the jagged array. Here,

1. Outer for loop

to access the elements (arrays) of the jagged array

jaggedArray.Length - gives the size of jagged array

2. Inner for loop

to access the elements of the individual array inside the jagged array.

jaggedArray[i].Length - gives the size of elements of the ith array inside the jagged array

**Compile time and Runtime Errors**

**Compile Time Errors:**

These errors occur when we violate the rules present in a syntax. The compile-time error indicates something that we need to fix before compiling the code. A compiler can easily detect these errors. It is the reason why we call them compile-time errors. Here are the most frequent errors (compile-time):

Terminator- missing semicolon.

Missing parenthesis.

Printing the overall value of a variable with no declaration.

**Runtime Errors:**

These errors occur during the run-time program execution after a successful compilation.

It is very difficult for a compiler to find out a runtime error because it cannot point out the exact line at which this particular error occurs.

**Example:**

Divide by Zero

**Object Class**

C# is an object-oriented program. In object-oriented programming (OOP), we solve complex problems by dividing them into objects.

To work with objects, we need to perform the following activities:

create a class

create objects from the class

**Class**

A class is a Collection of Objects / properties and methods etc.

**Syntax for Class:**

class ClassName {

}

**Note: In C#, fields and methods inside a class are called members of a class.**

**C# Objects**

An object is an instance of a class.

Suppose, we have a class Veda. StudentName and StudentNumber are the objects of class

**Syntax for Object Creation :**

ClassName obj = new ClassName();

in the above used "**new**" keyword to create an object of the class.

And, obj is the name of the object.

**Access Class Members using Object**

We use the name of objects along with the" ." operator to access members of a class.

**Example**

using System;

namespace ClassObject {

class Veda

{

String StudentName{get;set;}  
}

static void Main(string[] args) {

// create Veda Object

Veda vda=new Veda();

// access student name of the veda

vda.StudentName = "Raju";

Console.WriteLine(vda.StudentName);

Console.ReadLine();

}

}

}

**Creating Multiple Objects of a Class**

using System;

namespace ClassObject {

class Employee {

string department;

static void Main(string[] args) {

// create Employee object

Employee sheeran = new Employee();

// set department for sheeran

sheeran.department = "Development";

Console.WriteLine("Sheeran: " + sheeran.department);

// create second object of Employee

Employee taylor = new Employee();

// set department for taylor

taylor.department = "Content Writing";

Console.WriteLine("Taylor: " + taylor.department);

Console.ReadLine();

}

}

}

**Creating objects in a different class**

We can Create Class in a Class and we can create multiple class in different files and we can use that as Reference using Name Space

**Properties**

property is member of class using which we can expose values associated with a class to the outside of environment

**Syntax:**

Modifier datatype Name {get; set;}

**Example**

Public int Marks

{

get {return marks;} // represents a value returning method without parameter

set{marks=value;} // represents a non-value returning method without parameter

}

**Functions**

A method is a block of code that performs a specific task.

**Syntax :**

returnType methodName()

{

// method body

}

**returnType** - It specifies what type of value a method returns. For example, if a method has an int return type then it returns an int value.

if the method does not return a value, its return type is void.

**methodName** - It is an identifier that is used to refer to the particular method in a program.

**method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }

1. Declaring a Method

2. Calling a Method

**Example:**

using System;

namespace Method {

class Program {

// method declaration

public void display() {

Console.WriteLine("Hello World");

}

static void Main(string[] args) {

// create class object

Program p1 = new Program();

//call method

p1.display();

Console.ReadLine();

}

}

}

**Method Return Type:**

A C# method may or may not return a value. If the method doesn't return any value, we use the void keyword.

If the method returns any value, we use the return statement to return any value.

**With void**

Void addNumbers() {

...

}

**With Return type**

int addNumbers() {

...

return sum;

}

**Example of Method with Return type:**

using System;

namespace Method {

class Program {

// method declaration

static int addNumbers() {

int sum = 5 + 14;

return sum;

}

static void Main(string[] args) {

// call method

int sum = addNumbers();

Console.WriteLine(sum);

Console.ReadLine();

}

}

}

**C# Methods Parameters:**

we can also create a method that accepts some value. These values are called method parameters.

Example:

int addNumber(int a, int b) {

//code

}

Here, a and b are two parameters passed to the addNumber() function.

If a method is created with parameters, we need to pass the corresponding values(arguments) while calling the method

// call the method

addNumber(100, 100);

**C# Access Modifiers**

In this tutorial, we will learn about the public, private, protected,

and internal access modifiers in C# with the help of examples.

**In C#, access modifiers specify the accessibility of types (classes, interfaces, etc)**

**and type members (fields, methods, etc).**

==> For example :-

class Student s

{

public string name;

private int num;

}

Here,

name - public field that can be accessed from anywhere

num - private field can only be accessed within the Student class

Types of Access Modifiers :-

In C#, there are 6 basic types of access modifiers.

**1)public**

**2)private**

**3)protected**

**4)internal**

**5)protected internal access modifier**

**6)private protected access modifier**

**1)Pubilc access modifier :-**

When we declare a type or type member public,

it can be accessed from anywhere.

For example :-

using System;

namespace MyApplication {

class Student {

public string name = "Sheeran";

public void print() {

Console.WriteLine("Hello from Student class");

}

}

class Program {

static void Main(string[] args) {

// creating object of Student class

Student student1 = new Student();sss

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Console.ReadLine();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

}

}

}

Note:We have used the object student1 of the Student class to access its members.

To learn more, visit the C# class and objects.

==================================================================

**2)Private access modifier :-**

When we declare a type member with the private access modifier,

it can only be accessed within the same class or struct.

For example :-

using System;

namespace MyApplication {

class Student {

private string name = "Sheeran";

private void print() {

Console.WriteLine("Hello from Student class");

}

}

----------------------------------------------------------------------------------------------------------------------------------------

class Program

{

static void Main(string[] args) {

// creating object of Student class

Student student1 = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

// accessing print method from Student

student1.print();

Console.ReadLine();

}

}

}

==========================================================================

**3)protected access modifier**

When we declare a type member as protected,

it can only be accessed from the same class and its derived classes.

For example:-

using System;

namespace MyApplication {

class Student {

protected string name = "Sheeran";

}

class Program {

static void Main(string[] args) {

// creating object of student class

Student student1 = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + student1.name);

Console.ReadLine();

}

}

}

Now, let's try to access the protected member from a derived class.

using System;

namespace MyApplication {

class Student {

protected string name = "Sheeran";

}

// derived class

class Program : Student {

static void Main(string[] args) {

// creating object of derived class

Program program = new Program();

// accessing name field and printing it

Console.WriteLine("Name: " + program.name);

Console.ReadLine();

}

}

}

==================================================

**4)Internal access modifier :-**

When we declare a type or type member as internal,

it can be accessed only within the same assembly.

An assembly is a collection of types (classes, interfaces, etc)

and resources (data). They are built to work together and form a

logical unit of functionality.

That's why when we run an assembly all classes and interfaces inside

the assembly run together. To learn more, visit the C# Assembly.

using System;

namespace Assembly

{

class Student

{

internal string name = "Sheeran";

}

class Program

{

static void Main(string[] args)

{

// creating object of Student class

Student theStudent = new Student();

// accessing name field and printing it

Console.WriteLine("Name: " + theStudent.name);

Console.ReadLine();

}

}

}

If we use internal within a single assembly,

it works just like the public access modifier.

**Example: internal in different Assembly**

using System;

namespace Assembly1 {

public class StudentName {

internal string name = "Sheeran";

}

class Program {

static void Main(string[] args) {

}

}

}

Here, this code is in Assembly1. We have created an internal field

name inside the class StudentName. Now, this field can only be accessed

from the same assembly Assembly1.

// Code on Assembly2

using System;

// access Assembly1

using Assembly1;

namespace Assembly2 {

class Program {

static void Main(string[] args) {

StudentName student = new StudentName();

// accessing name field from Assembly1

Console.Write(student.name);

Console.ReadLine();

}

}

}

====================================

**5. protected internal access modifier**

The protected internal is a combination of protected and internal access modifiers.

When we declare a member protected internal,

it can be accessed from the same assembly and the derived

class of the containing class from any other assembly.

// Code on Assembly1

using System;

namespace Assembly1 {

public class Greet {

protected internal string msg="Hello";

}

class Program {

static void Main(string[] args) {

Greet greet = new Greet();

Console.WriteLine(greet.msg);

Console.ReadLine();

}

}

}

In the above example, we have created a class named Greet with a

field msg. Since the field is protected internal, we are able to

access it from the Program class as they are in the same assembly.

Let's derive a class from Greet in another assembly and try to access

the protected internal field msg from it.

// Code on Assembly2

using System;

// access Assembly1

using Assembly1;

namespace Assembly2 {

// derived class of Greet

class Program: Greet {

static void Main(string[] args) {

Program greet = new Program();

// accessing name field from Assembly1

Console.Write(greet.msg);

Console.ReadLine();

}

}

}

---------------------------------------------------------------------------------------------------------------------------------

**6. private protected access modifier**

The private protected access modifier is a combination of private

and protected.It is available from the C# version 7.2 and later.

When we declare a member private protected,

it can only be accessed within the same class,

and its derived class within the same assembly.

For example,

// Code in Assembly1

using System;

namespace Assembly1 {

public class StudentName {

private protected string name = "Sheeran";

}

//derived class of StudentName class

class Program1 : StudentName {

static void Main(string[] args) {

Program1 student = new Program1();

// accessing name field from base class

Console.Write(student.name);

Console.ReadLine();

}

}

}