

## Unit-2: C# Language Basics

*Writing console and GUI applications; Identifiers and Keywords; Writing Comments; Data Types; Expressions and Operators; Strings and Characters; Arrays; Variables and Parameters; Statements (Declaration, Expression, Selection, Iteration and Jump Statements); Namespaces*

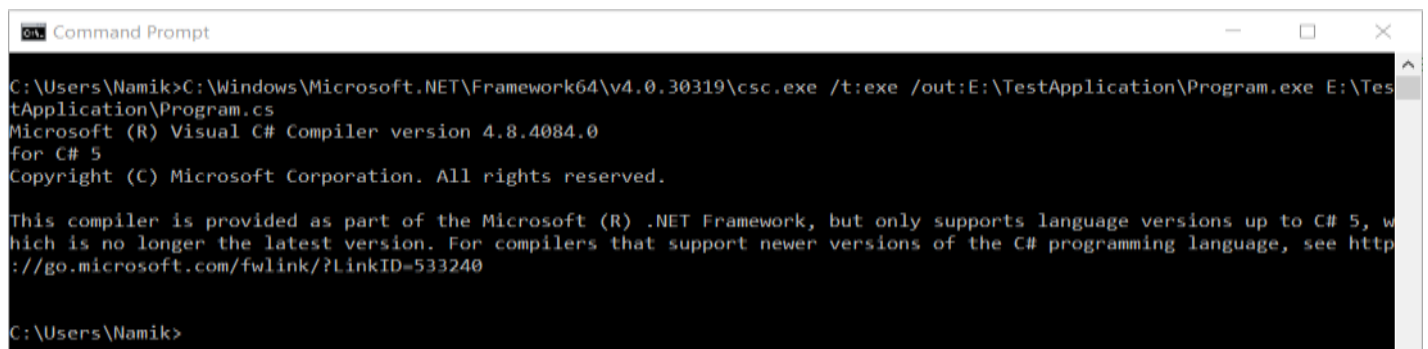
### Console Application and Compilation:

<pre>using System; namespace ConsoleApp1 {     internal class Program     {         static void Main()         {             int x,y,a;             Console.WriteLine("Enter length of rectange:");             x=Convert.ToInt32(Console.ReadLine());             Console.WriteLine("Enter width of rectange:");             y=Convert.ToInt32(Console.ReadLine());             a=x*y;             Console.WriteLine("Aera="+a.ToString());         }     } }</pre>	<pre>//Importing Namespace //Namespace  //Class declaration  //Method/Function</pre>
--	--

- The name of C# compiler is **csc.exe**
- .CS source code can be compile using **IDE (like: visual studio)** or **manually using by calling csc.exe**.

### Steps to Compile:

1. Save a program file such with extension .cs (Syntax: <program\_name>.cs)
2. Go to command window.
3. Invoke csc.exe (location like: *C:\Windows\Microsoft.NET\Framework64\v4.0.30319\*)
4. Run csc.exe to compile source code : *C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe /out:E:\TestApplication\Program.exe E:\TestApplication\Program.cs*



```
Command Prompt
C:\Users\Namik>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe /out:E:\TestApplication\Program.exe E:\TestApplication\Program.cs
Microsoft (R) Visual C# Compiler version 4.8.4084.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

C:\Users\Namik>
```

Or

To produce .dll

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /target:library E:\TestApplication\Program.cs
```

DLL: Dynamic link libraries are files that contain data, code, or resources needed for the running of applications. These are files that are created by the windows ecosystem and can be shared between two or more applications.

When a program or software runs on Windows, much of how the application works depends on the DLL files of the program. For instance, if a particular application had several modules, then how each module interacts with each other is determined by the Windows DLL files.

By using a DLL, a program can be modularized into separate components. For example, an accounting program may be sold by module. Each module can be loaded into the main program at run time if that module is installed. Because the modules are separate, the load time of the program is faster, and a module is only loaded when that functionality is requested.

DLL files was created so that *multiple applications could use their information at the same time*.

```
E:\TestApplication\C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /target:library E:\TestApplication\Program.cs
Microsoft (R) Visual C# Compiler version 4.8.0084.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# p
rogramming language, see http://go.microsoft.com/fwlink/?linkID=533248

E:\TestApplication>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe /out:E:\TestApplication\Program.exe E:\TestApplication\Program.cs
Microsoft (R) Visual C# Compiler version 4.8.0084.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# p
rogramming language, see http://go.microsoft.com/fwlink/?linkID=533248

E:\TestApplication>dir
Volume in drive E has no label.
Volume Serial Number is 2C48-78F5

Directory of E:\TestApplication

08/28/2022  01:09 PM    <DIR>          .
08/28/2022  01:09 PM    <DIR>          ..
08/28/2022  02:00 PM                296 P2.ca
08/28/2022  02:59 PM             1,342 Program.ca
08/28/2022  03:08 PM             3,584 Program.dll
08/28/2022  03:09 PM             4,096 Program.exe
                4 File(s)              9,416 bytes
                2 Dir(s)      86,278,533,882 bytes free

E:\TestApplication>Program.exe
Enter length of rectangle:
5
Enter width of rectangle:
4
Area=20

E:\TestApplication>
```

Or

To select main function from different class at the time of compilation:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /main:ConsoleApp1.Program /t:exe
/out:E:\TestApplication\Program.exe E:\TestApplication\Program.cs
```

## Identifiers and Keywords:

- **Identifiers** are the name given to entities such as variables, methods, classes, etc. They are tokens in a program which **uniquely identify** an element. For example: **int value;**
- Reserved keywords cannot be used as identifiers unless @ is added as prefix.
- Rules for Naming an Identifier:
  1. An identifier **cannot be a C# keyword**.
  2. An identifier must **begin with a letter, an underscore or @ symbol**. The remaining part of identifier can contain letters, digits and underscore symbol.
  3. **Whitespaces are not allowed**. Neither it can have symbols other than letter, digits and underscore.
  4. **Identifiers are case-sensitive**. So, getName, GetName and getname represents 3 different identifiers.

Identifiers	Remarks
number	Valid
calculateMarks	Valid
hello\$	Invalid (Contains \$)
name1	Valid
@if	Valid (Keyword with prefix @)
if	Invalid (C# Keyword)
My name	Invalid (Contains whitespace)
_hello_hi	Valid

- **Keywords** are predefined, reserved identifiers that have special meanings to the compiler.
- They cannot be used as identifiers in your program **unless they include @ as a prefix**.
- For example, **@if** is a **valid** identifier, **but if is not** because **if is a keyword**.
- C# has a total of **77 keywords**. All these keywords are in **lowercase**. Here is a complete list of all C# keywords.

abstract	continue	finally	is	protected	struct	using
as	decimal	fixed	lock	public	switch	virtual
base	default	float	long	readonly	this	void
bool	delegate	for	namespace	ref	throw	volatile
break	do	foreach	new	return	true	while
byte	double	goto	null	sbyte	try	
case	else	if	object	sealed	typeof	
catch	enum	implicit	operator	short	uint	
char	event	in	out	sizeof	ulong	
checked	explicit	int	override	stackalloc	unchecked	
class	extern	interface	params	static	unsafe	
const	false	internal	private	string	ushort	

Ref: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>

Although keywords are reserved words, **they can be used as identifiers if @ is added as prefix.**  
For example: `int @void`

**Contextual Keywords:** Besides regular keywords, C# has **many (44)** contextual keywords. Contextual keywords **have specific meaning in a limited program context** and can be used as identifiers outside that context. **They are not reserved words in C#.**

add	get	notnull	set
and	global	nuint	unmanaged (function pointer calling convention)
alias	group	on	unmanaged (generic type constraint)
ascending	init	or	value
args	into	orderby	var
async	join	partial (type)	when (filter condition)
await	let	partial (method)	where (generic type constraint)
by	managed (function pointer calling convention)	record	where (query clause)
descending	nameof	remove	
dynamic	nint	required	
equals	not	select	
from	with	yield	

Ref: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>

## Comments:

Comments are used in a program to help us understand a piece of code. They are human readable words intended to make the code readable. Comments are **completely ignored by the compiler.**

In C#, there are 3 types of comments:

1. Single Line Comments ( // )
2. Multi Line Comments (/\* \*/)
3. XML Comments ( /// )

**Single Line Comments:** Single line comments start with a double slash //. The compiler ignores everything after // to the end of the line.

```
// Hello World Program
using System;
namespace HelloWorld
{
    class Program
    {
        public static void Main(string[] args) // Execution Starts from Main method
        {
            Console.WriteLine("Hello World!"); // Prints Hello World
        }
    }
}
```

```
    }  
    }  
}
```

**Multi Line Comments:** Multi line comments start with `/*` and ends with `*/`. Multi line comments can span over multiple lines. `/* ... */` can be used **instead of single line comments**.

```
/*  
    This is a Hello World Program in C#.  
    This program prints Hello World.  
*/  
using System;  
namespace HelloWorld  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            /* Prints Hello World    */  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

**XML Documentation Comments:** XML documentation comment is a special feature in C#. It starts with a triple slash `///` and is used to categorically describe a piece of code.. This is done using XML tags within a comment. These comments are then, used to create a separate XML documentation file.

```
/// <summary>  
/// This is a hello world program.  
/// </summary>  
using System;  
namespace HelloWorld  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

## C# Variables and Data Types:

- A variable is a symbolic name given to a memory location. Variables are used to store data in a computer program. Example: **int age;**
- In this example, a variable age of type int (integer) is declared and it can only store integer values.
- The variable can also be initialized to some value during declaration (called **initialization**). Example: **int age = 24;**
- Variables in C# must be declared before they can be used. This means, **the name and type of variable must be known before they can be assigned a value**. This is why C# is called a **statically-typed language**.
- The datatype of a variable **cannot be changed within a scope**.

```
int age;  
age = 24;  
... ..  
float age; //compilation error
```

- **Implicitly typed variables:** Alternatively in C#, we can declare a variable without knowing its type using **var** keyword. Such variables are called implicitly typed local variables. **Variables declared using var keyword must be initialized at the time of declaration.** Example: `var value = 5;`
- Compiler determines the type of variable from the value that is assigned to the variable in the case of implicitly typed variables.

## **Rules for Naming Variables in C#**

The rules for naming a variable in C# are:

- The variable name can contain letters (uppercase and lowercase), underscore( \_ ) and digits only.
- The variable name must start with either letter, underscore or @ symbol.
- C# is case sensitive. It means age and Age refers to 2 different variables.
- A variable name must not be a C# keyword. For example, if, for, using cannot be a variable name.

Variable Names	Remarks
name	Valid
subject101	Valid
_age	Valid (Best practice for naming private member variables)
@break	Valid (Used if name is a reserved keyword)
101subject	Invalid (Starts with digit)
your_name	Valid
your name	Invalid (Contains whitespace)

## C# Primitive (Predefined/Built-In) Data Types:

### 1) Boolean (bool)

- a. Boolean data type has two possible values: true or false
- b. Default value: false
- c. Boolean variables are generally used to check conditions such as in if statements, loops, etc.

```
using System;
namespace DataType
{
    class BooleanExample
    {
        public static void Main(string[] args)
        {
            bool isValid = true;
            Console.WriteLine(isValid);
        }
    }
}
```

**2) Signed Integral:** These data types hold integer values (both positive and negative). Out of the total available bits, one bit is used for sign.

#### a) sbyte

- Size: 8 bits
- Range: -128 to 127.
- Default value: 0

```
using System;
namespace DataType
{
    class SByteExample
    {
        public static void Main(string[] args)
        {
            sbyte level = 23;
            Console.WriteLine(level);
        }
    }
}
```

#### b) short

- Size: 16 bits
- Range: -32,768 to 32,767
- Default value: 0

```
using System;
namespace DataType
{
    class ShortExample
    {
        public static void Main(string[] args)
        {
            short value = -1109;
            Console.WriteLine(value);
        }
    }
}
```

**c) int**

- Size: 32 bits
- Range:  $-2^{31}$  to  $2^{31}-1$
- Default value: 0

```
using System;
namespace DataType
{
    class IntExample
    {
        public static void Main(string[] args)
        {
            int score = 51092;
            Console.WriteLine(score);
        }
    }
}
```

**d) long**

- Size: 64 bits
- Range:  $-2^{63}$  to  $2^{63}-1$
- Default value: 0L [L at the end represent the value is of long type]

```
using System;
namespace DataType
{
    class LongExample
    {
        public static void Main(string[] args)
        {
            long range = -7091821871L;
        }
    }
}
```



```
        Console.WriteLine(range);
    }
}
}
```

**3) Unsigned Integral:** These data types only hold values equal to or greater than 0. We generally use these data types to store values when we are sure, we won't have negative values.

**a) byte**

- Size: 8 bits
- Range: 0 to 255.
- Default value: 0

```
using System;
namespace DataType
{
    class ByteExample
    {
        public static void Main(string[] args)
        {
            byte age = 62;
            Console.WriteLine(level);
        }
    }
}
```

**b) ushort**

- Size: 16 bits
- Range: 0 to 65,535
- Default value: 0

```
using System;
namespace DataType
{
    class UShortExample
    {
        public static void Main(string[] args)
        {
            ushort value = 42019;
            Console.WriteLine(value);
        }
    }
}
```

**c) uint**

- Size: 32 bits
- Range: 0 to  $2^{32}-1$
- Default value: 0

```
using System;
namespace DataType
{
    class UIntExample
    {
        public static void Main(string[] args)
        {
            uint totalScore = 1151092;
            Console.WriteLine(totalScore);
        }
    }
}
```

**d) ulong**

- Size: 64 bits
- Range: 0 to  $2^{64}-1$
- Default value: 0

```
using System;
namespace DataType
{
    class ULongExample
    {
        public static void Main(string[] args)
        {
            ulong range = 17091821871L;
            Console.WriteLine(range);
        }
    }
}
```

**4) Floating Point:** These data types hold floating point values i.e. numbers containing decimal values. For example, 12.36, -92.17, etc.

**a) float**

- Single-precision floating point type
- Size: 32 bits
- Range:  $1.5 \times 10^{-45}$  to  $3.4 \times 10^{38}$
- Default value: 0.0F [F at the end represent the value is of float type]

```

using System;
namespace DataType
{
    class FloatExample
    {
        public static void Main(string[] args)
        {
            float number = 43.27F;
            Console.WriteLine(number);
        }
    }
}

```

### b) double

- Double-precision floating point type. What is the difference between single and double precision floating point?
- Size: 64 bits
- Range:  $5.0 \times 10^{-324}$  to  $1.7 \times 10^{308}$
- Default value: 0.0D [D at the end represent the value is of double type]

```

using System;
namespace DataType
{
    class DoubleExample
    {
        public static void Main(string[] args)
        {
            double value = -11092.53D;
            Console.WriteLine(value);
        }
    }
}

```

### c) decimal

- Decimal type has more precision and a smaller range as compared to floating point types (double and float). So it is appropriate for monetary calculations.
- Size: 128 bits
- Default value: 0.0M [M at the end represent the value is of decimal type]
- Range:  $(-7.9 \times 10^{28}$  to  $7.9 \times 10^{28}) / (100$  to  $28)$

```

using System;
namespace DataType
{

```

```

class DecimalExample
{
    public static void Main(string[] args)
    {
        decimal bankBalance = 53005.25M;
        Console.WriteLine(bankBalance);
    }
}

```

**5) Character (char):** It represents a 16 bit unicode character.

- Size: 16 bits
- Default value: '\0'
- Range: U+0000 ('\u0000') to U+FFFF ('\uffff')

```

using System;
namespace DataType
{
    class CharExample
    {
        public static void Main(string[] args)
        {
            char ch1 = "\u0042";
            char ch2 = 'x';
            Console.WriteLine(ch1);
            Console.WriteLine(ch2);
        }
    }
}

```

**C# Literals:** Literals are fixed values that appear in the program. They do not require any computation. For example, 5, false, 'w' are literals that appear in a program directly without any computation.

Literals	Values	Example
Boolean Literals	true, false	bool isValid = true; bool isPresent = false;
Integer Literals		long value1 = 4200910L; long value2 = -10928190L; int decimalValue = 25;
Floating Point Literals		double number = 24.67; float value = -12.29F; double scientificNotation = 6.21e2

Character and String Literals		char ch1 = 'R';// character char ch2 = "\x0072";// hexadecimal char ch3 = "\u0059";// unicode char ch4 = (char)107;// casted from integer string firstName = "Richard"; string lastName = " Feynman";
-------------------------------	--	--

### Escape sequence characters:

Character	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\r	Carriage return
\t	Horizontal Tab
\a	Alert
\b	Backspace

### Custom data types:

- Data type that is created using primitive data types is called custom data types.
- We can use the **struct, class, interface, enum, and record constructs** to create your own custom types.
- The .NET class library itself is **a collection of custom types** that we can use in our applications.

```

using System;
namespace ConsoleApp1
{
    public class Converter
    {
        int exchange; //variable or data
        public Converter (int rate) //Constructor
        {
            exchange= rate;
        }
        public int Convert (int unit) //method
        {
            return unit*exchange;
        }
    }
    class Test
    {
        static void Main()
        {

```

```

        Converter indianToNepaliConverter = new Converter(2);
        Converter poundToNepaliConverter = new Converter(135);
        Console.WriteLine(indianToNepaliConverter.Convert(100));
        Console.WriteLine(poundToNepaliConverter.Convert(100));
        Console.WriteLine(indianToNepaliConverter.Convert(poundToNepaliConverter.Convert(1)));
    }
}

```

## C# Type Conversion:

The process of converting the value of one type (int, float, double, etc.) to another type is known as type conversion. In C#, there are two basic types of type conversion:

- 1) Implicit Type Conversions
- 2) Explicit Type Conversions

### **Implicit Type Conversion in C#:**

- In implicit type conversion, the C# compiler automatically converts one type to another.
- Generally, smaller types like int (having less memory size) are automatically converted to larger types like double (having larger memory size).
- In implicit type conversion, **smaller types are converted to larger types**. Hence, there is **no loss of data during the conversion**.

```

using System;
namespace MyApplication {
    class Program {
        static void Main(string[] args) {
            int numInt = 500;

            // get type of numInt
            Type n = numInt.GetType();

            // Implicit Conversion
            double numDouble = numInt;

            // get type of numDouble
            Type n1 = numDouble.GetType();

            // Value before conversion
            Console.WriteLine("numInt value: " + numInt);
            Console.WriteLine("numInt Type: " + n);

            // Value after conversion
            Console.WriteLine("numDouble value: "+numDouble);

```

```

        Console.WriteLine("numDouble Type: " + n1);
        Console.ReadLine();
    }
}
}

```

## C# Explicit Type Conversion:

- In explicit type conversion, we explicitly convert one type to another.
- Generally, larger types like double (having large memory size) are converted to smaller types like int (having small memory size).
- The explicit type conversion is also called type casting.

```

using System;
namespace MyApplication {
    class Program {
        static void Main(string[] args) {

            double numDouble = 1.23;

            // Explicit casting
            int numInt = (int) numDouble;

            // Value before conversion
            Console.WriteLine("Original double Value: "+numDouble);

            // Value before conversion
            Console.WriteLine("Converted int Value: "+numInt);
            Console.ReadLine();
        }
    }
}

```

Method	Description
ToBoolean()	converts a type to a Boolean value
ToChar()	converts a type to a char type
ToDouble()	converts a type to a double type
ToInt16()	converts a type to a 16-bit int type
ToString()	converts a type to a string

## C# Type Conversion using Parse():

- In C#, we can also use the Parse() method to perform type conversion.
- Generally, while performing type conversion between non-compatible types like int and string, we use Parse().
- Note: We cannot use Parse() to convert a textual string like "test" to an int. For example:

```
String str = "test";  
int a = int.Parse(str); // Error Code
```

```
using System;  
namespace Conversion {  
    class Program {  
        static void Main(string[] args) {  
            string n = "100";  
            // converting string to int type  
            int a = int.Parse(n);  
            Console.WriteLine("Original string value: "+n);  
            Console.WriteLine("Converted int value: "+a);  
            Console.ReadLine();  
        }  
    }  
}
```

**Implicit Conversion** are allowed when both of the following are true;

- The compiler can guarantee they will always succeed.
- No information is lost in conversions.

**Explicit Conversion** are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

Implicit conversion happen **automatically** and explicit conversion require a **cast**.

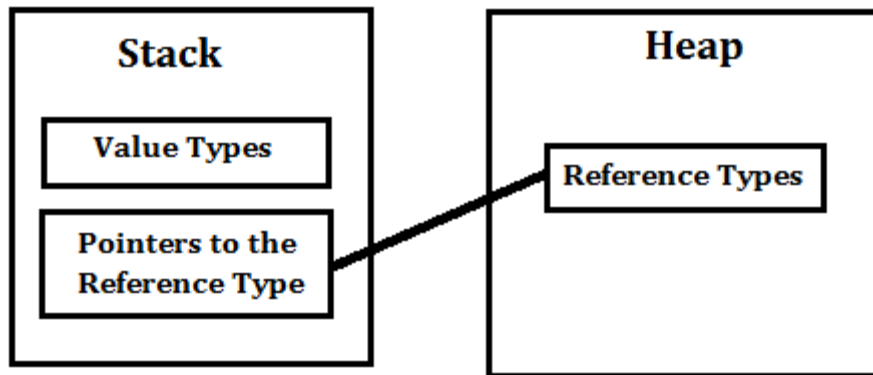
## Value Types Vs Reference Types:

C# types fall into the following categories:

1. Value Types
2. Reference Types
3. Generic Type Parameters
4. Pointer Types

A Value Type holds the data within its own memory allocation and a Reference Type contains a pointer to another memory location that holds the real data. Reference Type variables are stored in the heap while Value Type variables are stored in the stack.





## Value Type:

A Value Type stores its contents in memory allocated on the stack. When you created a Value Type, a single space in memory is allocated to store the value and that variable directly holds a value. If you assign it to another variable, the value is copied directly and both variables work independently. Predefined datatypes, structures, enums are also value types, and work in the same way. **Value types can be created at compile time and Stored in stack memory, because of this, Garbage collector can't access the stack.**

e.g. `int x = 10;`

Here the value 10 is stored in an area of memory called the stack.

```

public struct point
{
    public int x;
    public int y;
}
  
```

Point struct

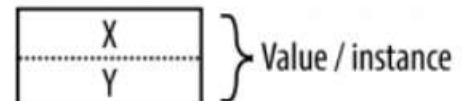


Figure 2-1. A value-type instance in memory

```

static void main()
{
    point p1 = new point();
    p1.x=7;
    point p2 =p1;
    console.WriteLine(p1.x); //7
    console.WriteLine(p2.x); //7

    p1.x=9;
    console.WriteLine(p1.x); //9
    console.WriteLine(p2.x); //7
}
  
```

Point struct



Figure 2-2. Assignment copies a value-type instance

## Reference Type:

Reference Types are used by a reference which holds a reference (address) to the object but not the object itself. Because reference types represent the address of the variable rather than the data itself, assigning a reference variable to another doesn't copy the data. Instead it creates a second copy of the reference, which refers to the same location of the heap as the original value. Reference Type variables are stored in a different area of memory called the heap. This means that when a reference type variable is no longer used, it can be marked for garbage collection. Examples of reference types are Classes, Objects, Arrays, Indexers, Interfaces etc.

e.g. `int[] iArray = new int[20];`

Explore From: <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/>

## Operators in C #:

Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants. For example, in  $2+3$ ,  $+$  is an operator that is used to carry out addition operation, while 2 and 3 are operands.

### Types:

- a) **Arithmetic Operators:** Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.

C# Arithmetic Operators		
Operator	Operator Name	Example
+	Addition Operator	$6 + 3$ evaluates to 9
-	Subtraction Operator	$10 - 6$ evaluates to 4
*	Multiplication Operator	$4 * 2$ evaluates to 8
/	Division Operator	$10 / 5$ evaluates to 2
%	Modulo Operator (Remainder)	$16 \% 3$ evaluates to 1

```
using System;
namespace Operator
{
    class ArithmeticOperator
    {
        public static void Main(string[] args)
        {
            double firstNumber = 14.40, secondNumber = 4.60, result;
            int num1 = 26, num2 = 4, rem;
            // Addition operator
            result = firstNumber + secondNumber;
```

```

Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);
// Subtraction operator
result = firstNumber - secondNumber;
Console.WriteLine("{0} - {1} = {2}", firstNumber, secondNumber, result);

// Multiplication operator
result = firstNumber * secondNumber;
Console.WriteLine("{0} * {1} = {2}", firstNumber, secondNumber, result);

// Division operator
result = firstNumber / secondNumber;
Console.WriteLine("{0} / {1} = {2}", firstNumber, secondNumber, result);

// Modulo operator
rem = num1 % num2;
Console.WriteLine("{0} % {1} = {2}", num1, num2, rem);
}
}

```

**b) Relational Operators:** Relational operators are used to check the relationship between two operands. If the relationship is true the result will be true, otherwise it will result in false. Relational operators are used in decision making and loops.

C# Relational Operators		
Operator	Operator Name	Example
==	Equal to	6 == 4 evaluates to false
>	Greater than	3 > -1 evaluates to true
<	Less than	5 < 3 evaluates to false
>=	Greater than or equal to	4 >= 4 evaluates to true
<=	Less than or equal to	5 <= 3 evaluates to false
!=	Not equal to	10 != 2 evaluates to true

```

using System;
namespace Operator
{
    class RelationalOperator
    {
        public static void Main(string[] args)
        {
            bool result;
            int firstNumber = 10, secondNumber = 20;

```

```

result = (firstNumber==secondNumber);
Console.WriteLine("{0} == {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber > secondNumber);
Console.WriteLine("{0} > {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber < secondNumber);
Console.WriteLine("{0} < {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber >= secondNumber);
Console.WriteLine("{0} >= {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber <= secondNumber);
Console.WriteLine("{0} <= {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber != secondNumber);
Console.WriteLine("{0} != {1} returns {2}",firstNumber, secondNumber, result);
}
}
}

```

c) **Logical Operators:** Logical operators are used to perform logical operation such as and, or. Logical operators operates on boolean expressions (true and false) and returns boolean values. Logical operators are used in decision making and loops.

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

```

using System;

namespace Operator
{
    class LogicalOperator
    {
        public static void Main(string[] args)
        {

```

```

        bool result;
        int firstNumber = 10, secondNumber = 20;

        // OR operator
        result = (firstNumber == secondNumber) || (firstNumber > 5);
        Console.WriteLine(result);

        // AND operator
        result = (firstNumber == secondNumber) && (firstNumber > 5);
        Console.WriteLine(result);
    }
}

```

**d) Unary Operators:** Unary operators operates on a single operand.

C# unary operators		
Operator	Operator Name	Description
+	Unary Plus	Leaves the sign of operand as it is
-	Unary Minus	Inverts the sign of operand
++	Increment	Increment value by 1
--	Decrement	Decrement value by 1
!	Logical Negation (Not)	Inverts the value of a boolean

```

using System;

namespace Operator
{
    class UnaryOperator
    {
        public static void Main(string[] args)
        {
            int number = 10, result;
            bool flag = true;

            result = +number;
            Console.WriteLine("+number = " + result);

            result = -number;
            Console.WriteLine("-number = " + result);

            result = ++number;
            Console.WriteLine("++number = " + result);
        }
    }
}

```

```

        result = --number;
        Console.WriteLine("--number = " + result);

        Console.WriteLine("!flag = " + (!flag));
    }
}

```

e) **Ternary Operator:** The ternary operator `?` : operates on three operands. It is a shorthand for if-then-else statement. Ternary operator can be used as follows:

variable = Condition? Expression1 : Expression2;

The ternary operator works as follows:

- If the expression stated by Condition is true, the result of Expression1 is assigned to variable.
- If it is false, the result of Expression2 is assigned to variable.

```

using System;
namespace Operator
{
    class TernaryOperator
    {
        public static void Main(string[] args)
        {
            int number = 10;
            string result;

            result = (number % 2 == 0)? "Even Number" : "Odd Number";
            Console.WriteLine("{0} is {1}", number, result);
        }
    }
}

```

f) **Bitwise and Bit Shift Operators:** Bitwise and bit shift operators are used to perform bit manipulation operations.

C# Bitwise and Bit Shift operators	
Operator	Operator Name
~	Bitwise Complement
&	Bitwise AND
	Bitwise OR

^	Bitwise Exclusive OR
<<	Bitwise Left Shift
>>	Bitwise Right Shift

```

using System;
namespace Operator
{
    class BitOperator
    {
        public static void Main(string[] args)
        {
            int firstNumber = 10;
            int secondNumber = 20;
            int result;

            result = ~firstNumber;
            Console.WriteLine("~{0} = {1}", firstNumber, result);

            result = firstNumber & secondNumber;
            Console.WriteLine("{0} & {1} = {2}", firstNumber, secondNumber, result);

            result = firstNumber | secondNumber;
            Console.WriteLine("{0} | {1} = {2}", firstNumber, secondNumber, result);

            result = firstNumber ^ secondNumber;
            Console.WriteLine("{0} ^ {1} = {2}", firstNumber, secondNumber, result);

            result = firstNumber << 2;
            Console.WriteLine("{0} << 2 = {1}", firstNumber, result);

            result = firstNumber >> 2;
            Console.WriteLine("{0} >> 2 = {1}", firstNumber, result);
        }
    }
}

```

#### g) Assignment Operators:

Operator	Operator Name	Example	Equivalent To
+=	Addition Assignment	x += 5	x = x + 5
-=	Subtraction Assignment	x -= 5	x = x - 5
*=	Multiplication Assignment	x *= 5	x = x * 5
/=	Division Assignment	x /= 5	x = x / 5

%=	Modulo Assignment	x %= 5	x = x % 5
&=	Bitwise AND Assignment	x &= 5	x = x & 5
=	Bitwise OR Assignment	x  = 5	x = x   5
^=	Bitwise XOR Assignment	x ^= 5	x = x ^ 5
<<=	Left Shift Assignment	x <<= 5	x = x << 5
>>=	Right Shift Assignment	x >>= 5	x = x >> 5
=>	Lambda Operator	x => x*x	Returns x*x

```

using System;
namespace Operator
{
    class BitOperator
    {
        public static void Main(string[] args)
        {
            int number = 10;

            number += 5;
            Console.WriteLine(number);

            number -= 3;
            Console.WriteLine(number);

            number *= 2;
            Console.WriteLine(number);

            number /= 3;
            Console.WriteLine(number);

            number %= 3;
            Console.WriteLine(number);

            number &= 10;
            Console.WriteLine(number);

            number |= 14;
            Console.WriteLine(number);

            number ^= 12;
            Console.WriteLine(number);

            number <<= 2;
            Console.WriteLine(number);
        }
    }
}

```



```

        number >>= 3;
        Console.WriteLine(number);
    }
}

```

### h) Miscellaneous Operators:

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
is	Determines whether an object is of a certain type.	If( Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello");

Reference From: <https://www.programiz.com/csharp-programming/operators>

### C# Operator Precedence:

Operator precedence is a set of rules which defines how an expression is evaluated. In C#, each C# operator has an assigned priority and based on these priorities, the expression is evaluated.

For example, the precedence of multiplication (\*) operator is higher than the precedence of addition (+) operator. Therefore, operation involving multiplication is carried out before addition.

int x = 4 + 3 \* 5; Output will be 19

C# Associativity of operators		
Category	Operators	Associativity
Postfix Increment and Decrement	++, --	Left to Right
Prefix Increment, Decrement and Unary	++, --, +, -, !, ~	Right to Left
Multiplicative	*, /, %	Left to Right
Additive	+, -	Left to Right
Shift	<<, >>	Left to Right
Relational	<, <=, >, >=	Left to Right
Equality	==, !=	Left to Right
Bitwise AND	&	Left to Right
Bitwise XOR	^	Left to Right
Bitwise OR		Left to Right
Logical AND	&&	Left to Right

Logical OR		Left to Right
Ternary	? :	Right to Left
Assignment	=, +=, -=, *=, /=, % =, &=,  =, ^=, <<=, >>=	Right to Left

```
using System;
namespace Operator
{
    class OperatorPrecedence
    {
        public static void Main()
        {
            int a = 5, b = 6, c = 3;
            int result = a * b / c;
            Console.WriteLine(result);
        }
    }
}
```

Output: 10

### **Null-Coalescing Operator in C#:**

**?? Operator** is known as Null-coalescing operator. It will return the value of its left-hand operand if it is not null. If it is null, then it will evaluate the right-hand operand and returns its result. Or if the left-hand operand evaluates to non-null, then it does not evaluate its right-hand operand.

p ?? q

Here, p is the left and q is the right operand of ?? operator. The value of p can be nullable type, but the value of q must be non-nullable type. If the value of p is null, then it returns the value of q. Otherwise, it will return the value of p.

```
using System;
namespace example {
class Program {
    static void Main(string[] args)
    {
        string item_1 = null;
        string item_2 = "GeeksforGeeks";
        string item_3 = "GFG";

        string item_4 = item_1 ?? item_2;
        item_3 = item_4 ?? item_2;
    }
}
```

```

Console.WriteLine("Value of item_4 is: {0} \n"+"Value of item_3 is: {1}", item_4, item_3);

// Value types
int ? item_5 = null;
int ? item_6 = item_5 ?? obj.Add(10, 30);
Console.WriteLine("Value of item_6 is: {0}", item_6);
}

// Method
public static int Add(int a, int b)
{
    int result = a + b;
    return result;
}
}
}

```

### Output:

```

Value of item_4 is: GeeksforGeeks
Value of item_3 is: GeeksforGeeks
Value of item_6 is: 40

```

### String and Characters:

C# char type (aliasing the System.Char type) represents a Unicode character and occupies 2 byte. A char literal is specified inside single quotes:

```
Char c= 'A';
```

**Escape sequences** express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning.

```
Char newline= '\n';
```

Escape sequence	Character name	Unicode encoding
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A

\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

### String Type:

In C#, a string is a sequence of characters. We use the **string** keyword to create a string.

```
// create a string
string str = "C# Programming";
```

A string variable in C# is not of primitive types like int, char, etc. Instead, it is an object of the String class (**System.String**).

### String Operations:

#### 1. *Get the Length of a string*

```
using System;
namespace CsharpString {
    class Test {
        public static void Main(string [] args) {

            // create string
            string str = "Namaraj Adhikari";
            Console.WriteLine("string: " + str);

            // get length of str
            int length = str.Length;
            Console.WriteLine("Length: "+ length);

            Console.ReadLine();
        }
    }
}
```

#### 2. *Join two strings in C#:* The + Operator is used to concatenate C# strings

*Example: String s= "a" + "b";*

```
using System;
namespace CsharpString {
    class Test {
        public static void Main(string [] args) {

            // create string
```

```

string str1 = "C# ";
Console.WriteLine("string str1: " + str1);

// create string
string str2 = "Programming";
Console.WriteLine("string str2: " + str2);

// join two strings
string joinedString = string.Concat(str1, str2);
Console.WriteLine("Joined string: " + joinedString);

Console.ReadLine();
}
}
}

```

**3. C# compare two strings:** In C#, we can make comparisons between two strings using the Equals() method.

```

using System;
namespace CsharpString {
    class Test {
        public static void Main(string [] args) {

            // create string
            string str1 = "C# Programming";
            string str2 = "C# Programming";
            string str3 = "Programiz";

            // compare str1 and str2
            Boolean result1 = str1.Equals(str2);
            Console.WriteLine("string str1 and str2 are equal: " + result1);

            //compare str1 and str3
            Boolean result2 = str1.Equals(str3);
            Console.WriteLine("string str1 and str3 are equal: " + result2);

            Console.ReadLine();
        }
    }
}

```

**4. String interpolation:** In C#, we can use string interpolation to insert variables inside a string. For string interpolation, the string literal must begin with the \$ character.

using System;

```
namespace CsharpString {  
    class Test {  
        public static void Main(string [] args) {  
  
            // create string  
            string name = "Programiz";  
  
            // string interpolation  
            string message = $"Welcome to {name}";  
            Console.WriteLine(message);  
  
            Console.ReadLine();  
        }  
    }  
}
```

### Methods of C# string

Methods	Description
Format()	returns a formatted string
Split()	splits the string into substring
Substring()	returns substring of a string
Compare()	compares string objects
Replace()	replaces the specified old character with the specified new character
Contains()	checks whether the string contains a substring
Join()	joins the given strings using the specified separator
Trim()	removes any leading and trailing whitespaces
EndsWith()	checks if the string ends with the given string
IndexOf()	returns the position of the specified character in the string
Remove()	returns characters from a string
ToUpper()	converts the string to uppercase
ToLower()	converts the string to lowercase
PadLeft()	returns string padded with spaces or with a specified Unicode character on the left
PadRight()	returns string padded with spaces or with a specified Unicode character on the right
StartsWith()	checks if the string begins with the given string
ToCharArray()	converts the string to a char array
LastIndexOf()	returns index of the last occurrence of a specified string

## Variables and Parameters:

A variable represents a storage location that has a modifiable value. A variable can be a local variable, parameter (*value, ref, or out*), field (*instance or static*), or array element.

## **Definite Assignment:**

C# enforces a definite assignment policy. In practice, this means that outside of an unsafe context, it's impossible to access uninitialized memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called (unless marked as optional).
- All other variables (such as fields and array elements) are automatically initialized by the runtime

```
static void Main()
{
    int x;
    Console.WriteLine(x); //compile time error
}
```

**Field and array** elements are automatically initialized with the default values for their type.

Type	Default value
Any reference type	null
Any built-in integral numeric type	0 (zero)
Any built-in floating-point numeric type	0 (zero)
bool	false
char	'\0' (U+0000)
enum	The value produced by the expression (E)0, where E is the enum identifier.
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null.
Any nullable value type	An instance for which the HasValue property is false and the Value property is undefined. That default value is also known as the null value of a nullable value type.

Example:

```
static void Main()
{
    int[] ints =new int[2];
    Console.WriteLine(ints[0]);
}
```

**Or**

## Class Test

```
{  
    static int x;  
    static void Main()  
    {  
        Console.WriteLine(x);  
    }  
}
```

### Default value expressions:

You can obtain the default value for any type with the default keyword.

e.g. decimal d = default(decimal);

### Parameters:

- A method has a sequence of parameters. Parameters **define the set of arguments** that must be provided for that method.
- Methods in C# are generally the block of codes or statements in a program which gives the user the ability to reuse the same code.
- Methods provides better readability of the code.
- Method is a collection of statements that perform some specific tasks and may/may not return the result to the caller.
- Method requires some valuable inputs to execute and complete its tasks. These input values are known as Parameters.

### C# contains the following types of Method Parameters:

#### 1. Value Parameters

```
// C# program to illustrate value parameters  
using System;  
  
public class GFG {  
  
    // Main Method  
    static public void Main()  
    {  
  
        // The value of the parameter  
        // is already assigned  
        string str1 = "Geeks";  
        string str2 = "geeks";  
        string res = addstr(str1, str2);  
        Console.WriteLine(res);  
    }  
  
    public static string addstr(string s1, string s2)
```



```
{  
    return s1 + s2;  
}  
}
```

2. **Ref Parameters:** The ref is a keyword in C# which is used for passing the value types by reference.

```
// C# program to illustrate the  
// concept of ref parameter  
using System;  
class GFG {  
  
    // Main Method  
    public static void Main()  
    {  
  
        // Assigning value  
        string val = "Dog";  
  
        // Pass as a reference parameter  
        CompareValue(ref val);  
  
        // Display the given value  
        Console.WriteLine(val);  
    }  
  
    static void CompareValue(ref string val1)  
    {  
        // Compare the value  
        if (val1 == "Dog")  
        {  
            Console.WriteLine("Matched!");  
        }  
  
        // Assigning new value  
        val1 = "Cat";  
    }  
}
```

**Output:**  
Matched!  
Cat

3. **Out Parameters:** The out is a keyword in C# which is used for the passing the arguments to methods as a reference type. It is generally used when a method returns multiple values. The out parameter does not pass the property. It is not necessary to initialize parameters before it passes to out.

```
// C# program to illustrate the
// concept of out parameter
using System;

class GFG {

    // Main method
    static public void Main()
    {

        // Creating variable
        // without assigning value
        int num;

        // Pass variable num to the method
        // using out keyword
        AddNum(out num);

        // Display the value of num
        Console.WriteLine("The sum of"
            + " the value is: {0}",num);

    }

    // Method in which out parameter is passed
    // and this method returns the value of
    // the passed parameter
    public static void AddNum(out int num)
    {
        num = 40;
        num += num;
    }
}
```

**Output:**

The sum of the value is: 80

4. **Default or Optional Parameters:** It is **not necessary** to pass all the parameters in the method if method used default or optional parameters. This concept is introduced in C# 4.0. Here, each optional parameter contains a **default value** which is the part of its definition. If we

do not pass any arguments to the optional parameters, then it takes its default value. The optional parameters **are always defined at the end of the parameter list**.

```
// C# program to illustrate the
// concept of optional parameters
using System;
class GFG
{
// This method contains two regular
// parameters, i.e. ename and eid
// And two optional parameters, i.e.
// bgrp and dept
static public void detail(string ename,int eid,string bgrp = "A+",string dept = "Review-Team")
{
    Console.WriteLine("Employee name: {0}", ename);
    Console.WriteLine("Employee ID: {0}", eid);
    Console.WriteLine("Blood Group: {0}", bgrp);
    Console.WriteLine("Department: {0}", dept);
}

// Main Method
static public void Main()
{
    // Calling the detail method
    detail("XYZ", 123);
    detail("ABC", 456, "B-");
    detail("DEF", 789, "B+",Software Developer");
}
}
```

**Output:**

Employee name: XYZ  
Employee ID: 123  
Blood Group: A+  
Department: Review-Team  
Employee name: ABC  
Employee ID: 456  
Blood Group: B-  
Department: Review-Team  
Employee name: DEF  
Employee ID: 789  
Blood Group: B+  
Department: Software Developer

5. **Named Parameters:** Using named parameters, you can specify the value of the parameter according to their **names not their order in the method**. Or in other words, it provides us a facility to not remember parameters according to their order. This concept is introduced in C# 4.0. It makes your program easier to understand when you are working with a larger number of parameters in your method. But always remember named parameters are always appear after fixed arguments, if you try to provide fixed argument after named parameter, then the compiler will throw an error.

```
// C# program to illustrate the
// concept of the named parameters
using System;

public class GFG {

    // addstr contain three parameters
    public static void addstr(string s1, string s2, string s3)
    {
        string result = s1 + s2 + s3;
        Console.WriteLine("Final string is: " + result);
    }

    // Main Method
    static public void Main()
    {
        // calling the static method with named
        // parameters without any order
        addstr(s1: "Geeks", s2: "for", s3: "Geeks");
    }
}
```

**Output:**

Final string is: GeeksforGeeks

6. **Dynamic Parameters:** Parameters pass dynamically means the **compiler does not check the type of the dynamic type** variable at compile-time, instead of this, the compiler gets the type at the run time. The dynamic type variable is created using a **dynamic** keyword.

```
// C# program to illustrate the concept
// of the dynamic parameters
using System;
class GFG {

    // Method which contains dynamic parameter
    public static void mulval(dynamic val)
```

```

    {
        val *= val;
        Console.WriteLine(val);
    }

    // Main method
    static public void Main()
    {
        // Calling mulval method
        mulval(30);
    }
}

```

**Output:**

900

7. Params: It is useful when the programmer doesn't have any prior knowledge about the number of parameters to be used. By using params you are allowed to pass any variable number of arguments. Only one params keyword is allowed and no additional Params will be allowed in function declaration after a params keyword. The length of params will be zero if no arguments will be passed.

```

// C# program to illustrate params
using System;
namespace Examples {
class Geeks {
    // function containing params parameters
    public static int mulval(params int[] num)
    {
        int res = 1;

        // foreach loop
        foreach(int j in num)
        {
            res *= j;
        }
        return res;
    }

    static void Main(string[] args)
    {

```

```

        // Calling mulval method
        int x = mulval(20, 49, 56, 69, 78);

        // show result
        Console.WriteLine(x);
    }
}
}

```

### **Output:**

295364160

### **C# Arrays:**

- An array is a collection of similar data elements stored at contiguous memory locations.
- It is also known as **homogeneous** data structure.

### **C# Array Declaration:**

Syntax: datatype[] arrayName;

Here,

dataType - data type like int, string, char, etc

arrayName - it is an identifier

e.g. int[] age;

To define the number of elements that an array can hold, we must allocate memory for the array in C#.

For example:

// declare an array

int[] age;

// allocate memory for array

age = new int[5];

*or*

int[] age = new int[5];

### **Array initialization in C#:**

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

If we have not provided the size of the array, C# automatically specifies the size by counting the number of elements in the array.

age[0]	age[1]	age[2]	age[3]	age[4]
12	4	5	2	5

```
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();

        }
    }
}
```

### **Output**

Element in first index : 1  
Element in second index : 2  
Element in third index : 3

### **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();
        }
    }
}
```

**C# Array Operations using System.Linq:** In C#, we have the **System.Linq** namespace that provides different methods to perform various operations in an array. For example,

```
using System;

// provides us various methods to use in an array
using System.Linq;

namespace ArrayMinMax {
    class Program {
        static void Main(string[] args) {

            int[] numbers = {51, 1, 3, 4, 98};

            // get the minimum element
            Console.WriteLine("Smallest Element: " + numbers.Min());

            // Max() returns the largest number in array
            Console.WriteLine("Largest Element: " + numbers.Max());

            Console.ReadLine();
        }
    }
}
```

### **Output**

Smallest Element: 1  
Largest Element: 98

**Note:** It is compulsory to use the **System.Linq** namespace while using **Min()**, **Max()**, **Sum()**, **Count()**, and **Average()** methods.

**C# Multidimensional Array:** In a multidimensional array, each element of the array is also an array.

```
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#:**





Two-Dimensional Array Declaration:

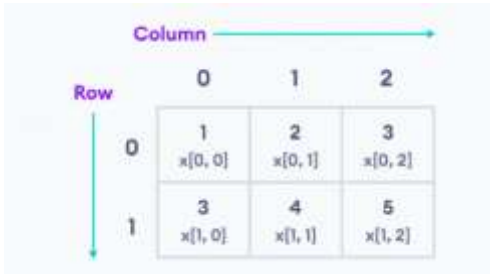
Two-Dimensional Array initialization

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

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

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

Access Elements from 2D Array



### **C# Jagged Array:**

In C#, 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:**

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

```
// declare jagged array
```

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

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

```
// 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];
```

### **Initializing Jagged Array:**

a) Using the index number:

```
// 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] = 4;
```

b) 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};
```

c) 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:

```
jaggedArray[1][0];
```

Differentiate between multidimensional and jagged array with program.

### Statements (C# Programming):

- The actions that a program takes are expressed in statements.
- Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition.
- A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block.
- A statement block is enclosed in { } brackets and can contain nested blocks.

```
static void Main()  
{  
    // Declaration statement.  
    int counter;  
  
    // Assignment statement.  
    counter = 1;  
  
    // Error! This is an expression, not an expression statement.  
    // counter + 1;  
  
    // Declaration statements with initializers are functionally  
    // equivalent to declaration statement followed by assignment statement:  
    int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
```

```

const double pi = 3.14159; // Declare and initialize constant.

// foreach statement block that contains multiple statements.
foreach (int radius in radii)
{
    // Declaration statement with initializer.
    double circumference = pi * (2 * radius);

    // Expression statement (method invocation). A single-line
    // statement can span multiple text lines because line breaks
    // are treated as white space, which is ignored by the compiler.
    System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
        counter, radius, circumference);

    // Expression statement (postfix increment).
    counter++;
} // End of foreach statement block
} // End of Main method body.
} // End of SimpleStatements class.
/*
Output:
Radius of circle #1 = 15. Circumference = 94.25
Radius of circle #2 = 32. Circumference = 201.06
Radius of circle #3 = 108. Circumference = 678.58
Radius of circle #4 = 74. Circumference = 464.96
Radius of circle #5 = 9. Circumference = 56.55
*/

```

## **Control Statements:**

Control statements enable us to specify the flow of program control; i.e., the order in which the instructions in a program must be executed. It make possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

C# program control statements can be put into the following categories: *selection, iteration, and jump*.

- **Selection** statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration** statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- **Jump** statements allow your program to execute in a nonlinear fashion.

## C#'s Selection Statements:

- a) **C# if Statement:** C# if-then statement will execute a block of code if the given condition is true. Syntax:

```
if (boolean-expression)
{
    // statements executed if boolean-expression is true
}
```

- b) **C# if-else Statement:** The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false).

```
if (boolean-expression)
{
    // statements executed if boolean-expression is true
}
else
{
    // statements executed if boolean-expression is false
}
```

### Expression is true

// codes before if-else

```
if (number < 5)
{
    number += 5;
}
else
{
    number -= 5;
}
// codes after if-else
```

### Expression is false

// codes before if-else

```
if (number < 5)
{
    number += 5;
}
else
{
    number -= 5;
}
// codes after if-else
```

- c) **C# if-else-if Statement (if-else ladder):** When we have only one condition to test, if-then and if-then-else statement works fine. But what if we have a multiple condition to test and execute one of the many block of code. The if...else if statement is executed from the top to bottom. As soon as a test expression is true, the code inside of that if ( or else if ) block is executed. Then the control jumps out of the if...else if block. If none of the expression is true, the code inside the else block is executed.

```
if (boolean-expression-1)
{
    // statements executed if boolean-expression-1 is true
}
else if (boolean-expression-2)
{
    // statements executed if boolean-expression-2 is true
}
else if (boolean-expression-3)
{
    // statements executed if boolean-expression-3 is true
}
.
.
.
else
{
    // statements executed if all above expressions are false
}
```

d) **Nested if-else Statement:** An if...else statement can exist within another if...else statement. Such statements are called nested if...else statement.

```
if (boolean-expression)
{
    if (nested-expression-1)
    {
        // code to be executed
    }
    else
    {
        // code to be executed
    }
}
else
{
    if (nested-expression-2)
    {
        // code to be executed
    }
    else
    {
        // code to be executed
    }
}
```

Example:

```
using System;
namespace Conditional
{
    class Nested
    {
        public static void Main()
        {
            int first = 7, second = -23, third = 13;
            if (first > second)
            {
                if (firstNumber > third)
                {
                    Console.WriteLine("{0} is the largest", first);
                }
                else
                {
                    Console.WriteLine("{0} is the largest", third);
                }
            }
            else
            {
                if (second > third)
                {
                    Console.WriteLine("{0} is the largest", second);
                }
                else
                {
                    Console.WriteLine("{0} is the largest", third);
                }
            }
        }
    }
}
```

**Output:**

13 is the largest

**Switch Statement:** It is multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
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.

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.

```
using System;

namespace Conditional
{
    class SwitchCase
    {
        public 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;
    }
}
}
}

```

```

using System;
namespace Conditional
{
    class SwitchCase
    {
        public static void Main(string[] args)
        {
            char ch;
            Console.WriteLine("Enter an alphabet");
            ch = Convert.ToChar(Console.ReadLine());

            switch(Char.ToLower(ch))
            {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    Console.WriteLine("Vowel");
                    break;
                default:
                    Console.WriteLine("Not a vowel");
                    break;
            }
        }
    }
}

```



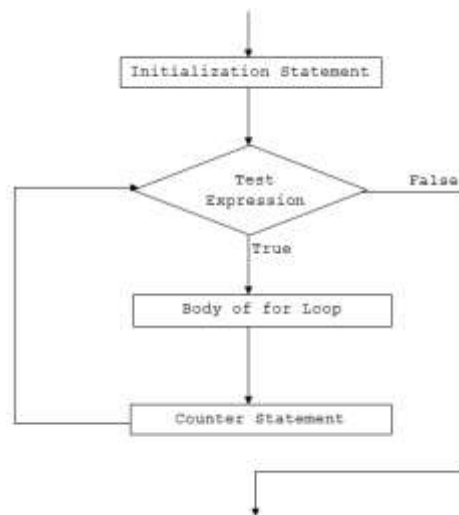


```
}  
}  
}
```

**Iteration Loop:** Iteration statements are used to execute a particular set of instructions repeatedly until a particular condition is met or for a fixed number of iterations (a loop repeatedly executes the same set of instructions until a termination condition is met).

a) **for loop:** The for keyword is used to create for loop in C#.

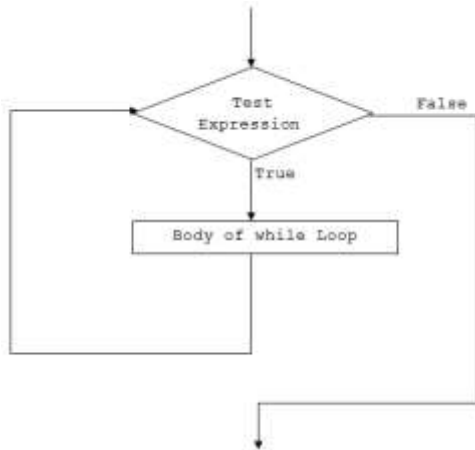
```
for (initialization; condition; iterator)  
{  
    // body of for loop  
}
```



```
using System;  
namespace Loop  
{  
    class ForLoop  
    {  
        public static void Main(string[] args)  
        {  
            for (int i=1; i<=5; i++)  
            {  
                Console.WriteLine("C# For Loop: Iteration {0}", i);  
            }  
        }  
    }  
}
```

b) **while loop:** The while keyword is used to create while loop in C#. Also known as **entry control loop**.

```
while (expression)
{
    // body of while
    //increment/decrement
}
```



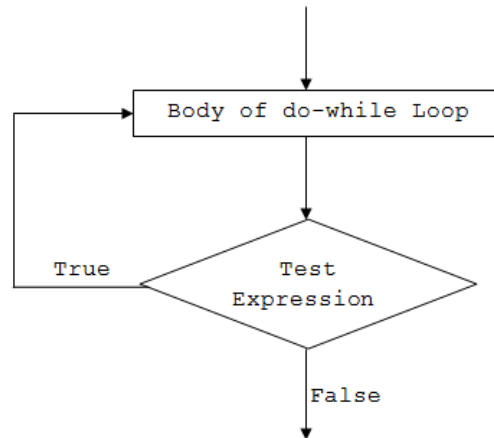
```
using System;
namespace Loop
{
    class WhileLoop
    {
        public static void Main(string[] args)
        {
            int i=1, sum=0;

            while (i<=5)
            {
                sum += i;
                i++;
            }
            Console.WriteLine("Sum = {0}", sum);
        }
    }
}
```

c) **do while loop:** **do** and **while** keyword is used to create a do...while loop. In while loop, the condition is checked **before the body is executed**. In do...while loop, condition is checked

after the body is executed. This is why, the body of do...while loop will execute at least once irrespective to the test-expression. Do-while loop is also known as **exit control** loop.

```
do
{
    // body of do while loop
    //increment/decrement
} while (expression);
```



```
using System;
namespace Loop
{
    class DoWhileLoop
    {
        public static void Main(string[] args)
        {
            int i = 1, n = 5, product;

            do
            {
                product = n * i;
                Console.WriteLine("{0} * {1} = {2}", n, i, product);
                i++;
            } while (i <= 10);
        }
    }
}
```

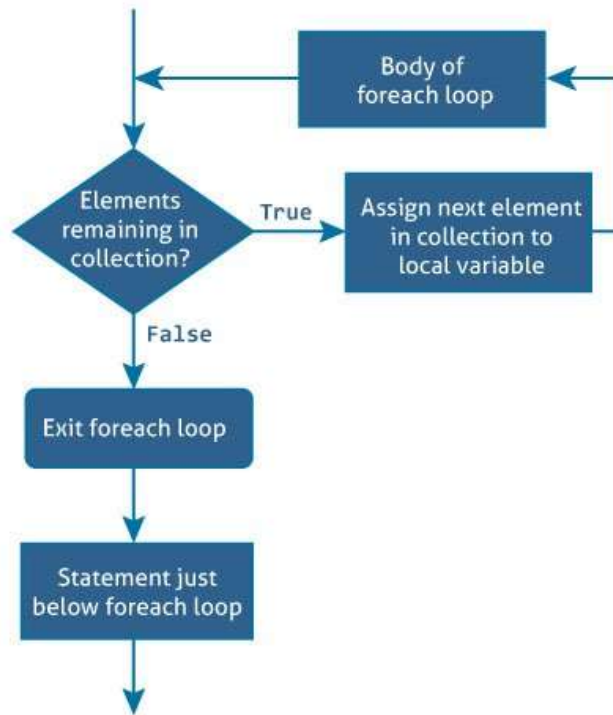
d) **foreach loop:** C# provides an easy to use and more readable alternative to for loop. The foreach loop work with arrays and collections to iterate through the items of arrays/collections. The foreach loop iterates through each item, hence called foreach loop.

```
foreach (element in iterable-item)
```

```
{
```

```
    // body of foreach loop
```

```
}
```



```
using System;
```

```
namespace Loop
```

```
{
```

```
    class ForEachLoop
```

```
    {
```

```
        public static void Main(string[] args)
```

```
        {
```

```
            char[] gender = {'m','f','m','m','m','f','f','m','m','f'};
```

```
            int male = 0, female = 0;
```

```
            foreach (char g in gender)
```

```
            {
```

```
                if (g == 'm')
```

```
                    male++;
```

```
                else if (g == 'f')
```

```
                    female++;
```

```
            }
```

```
            Console.WriteLine("Number of male = {0}", male);
```

```
            Console.WriteLine("Number of female = {0}", female);
```

```
        }
```

```
    }
```

```
}
```

**Nested Loops in C#:** A loop within another loop is called nested loop.

```
Outer-Loop
{
    // body of outer-loop
    Inner-Loop
    {
        // body of inner-loop
    }
    ... ..
}
```

The inner loop is a part of the outer loop and must start and finish within the body of outer loop. On each iteration of outer loop, the inner loop is executed completely.

```
using System;
namespace Loop
{
    class NestedForLoop
    {
        public static void Main(string[] args)
        {
            for (int i=1; i<=5; i++)
            {
                for (int j=1; j<=i; j++)
                {
                    Console.Write(j + " ");
                }
                Console.WriteLine();
            }
        }
    }
}
```

**Output:**

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

## Generating Flag of Nepal Using Stars:

using System;

namespace Loop

```
{
    class NestedForLoop
    {
        public static void Main(string[] args)
        {
            triangleShape(3);
            triangleShape(3);
            flagPole(row);
        }

        static void triangleShape(int n)
        {
            int i,j;
            for(i=1;i<=n;i++)
            {
                for(j=1;j<=i;j++)
                {
                    Console.Write ("* ");
                }
                Console.Write ("\n");
            }
        }
        static void flagPole(int n)
        {
            int i;
            for(i=1;i<=n;i++)
            {
                Console.Write f("*\n");
            }
        }
    }
}
```

### Output:

```
*
* *
* * *
*
* *
* * *
*
*
*
```

## Jumping Statement in C#:

- a) **Break Statement:** In C#, we use the break statement **to terminate the** loop. Sometimes we may need to terminate the loop immediately without checking the test expression. In such cases, the break statement is used.

```
using System;
namespace CSharpBreak
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 4; ++i)
            {
                if (i == 3)
                {
                    break;
                }
                Console.WriteLine(i);
            }
        }
    }
}
```

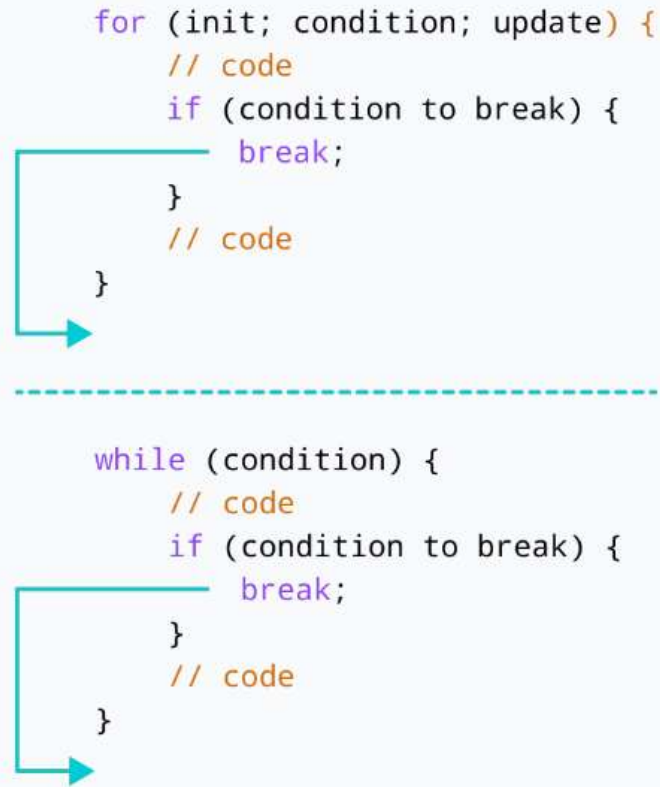
Output:

1  
2

```
static void Main(string[] args)
{
    int i = 1;
    while (i <= 5)
    {
        Console.WriteLine(i);
        i++;
        if (i == 4)
        {
            break;
        }
    }
    Console.ReadLine();
}
```

Output ?





```
static void Main(string[] args)  
{  
    int sum = 0;  
    for(int i = 1; i <= 3; i++)  
    { //outer loop  
        // inner loop  
        for(int j = 1; j <= 3; j++)  
        {  
            if (i == 2)  
            {  
                break;  
            }  
            Console.WriteLine("i = " + i + " j = " + j);  
        }  
    }  
}
```

Output:

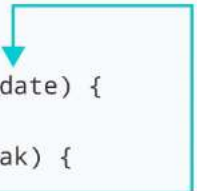
```
i = 1 j = 1  
i = 1 j = 2  
i = 1 j = 3  
i = 3 j = 1  
i = 3 j = 2  
i = 3 j = 3
```

**b) Continue Statement:** In C#, we use the continue statement to **skip a current iteration** of a loop. When our program encounters the continue statement, the program control moves to the end of the current loop.

```
using System;
namespace ContinueLoop {
class Program {
    static void Main(string[] args){
        for (int i = 1; i <= 5; ++i)
        {
            if (i == 3)
            {
                continue;
            }
            Console.WriteLine(i);
        }
    }
}
```

Output:

```
1
2
4
5
```



```
for (init; condition; update) {
    // code
    if (condition to break) {
        continue;
    }
    // code
}
```

---



```
while (condition) {
    // code
    if (condition to break) {
        continue;
    }
    // code
}
```

```

using System;
namespace ContinueNested {
    class Program {
        static void Main(string[] args) {

            int sum = 0;

            // outer loop
            for(int i = 1; i <= 3; i++) {

                // inner loop
                for(int j = 1; j <= 3; j++) {
                    if (j == 2) {
                        continue;
                    }

                    Console.WriteLine("i = " + i + " j = " + j);
                }
            }
        }
    }
}

```

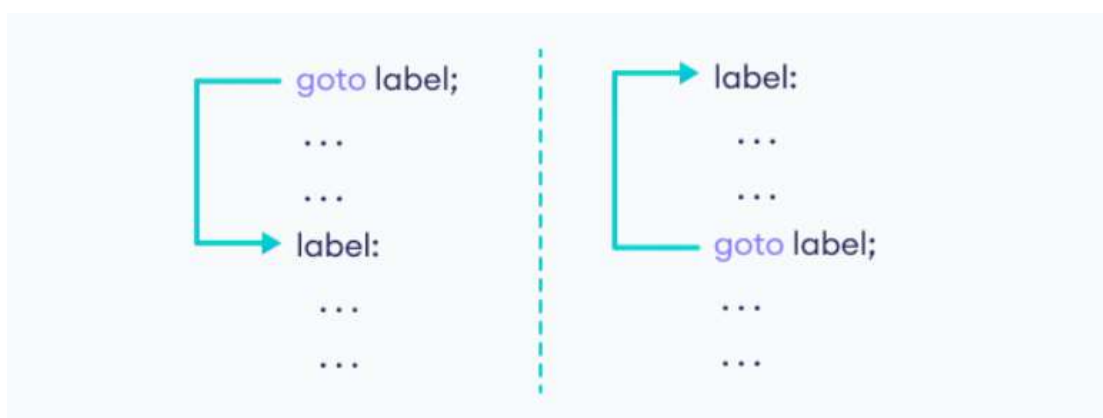
Output:

```

i = 1 j = 1
i = 1 j = 3
i = 2 j = 1
i = 2 j = 3
i = 3 j = 1
i = 3 j = 3

```

c) **Go to Statement:** In C#, the goto statement transfers control to some other part of the program.



```

using System;
namespace CSharpGoto {
    class Program {
        static void Main() {
            for(int i = 0; i <= 10; i++) {
                if(i == 5) {
                    // transfers control to End label
                    goto End;
                }

                Console.WriteLine(i);
            }

            // End label
        End:
            Console.WriteLine("Loop End");
        }
    }
}

```

Output:

```

0
1
2
3
4
Loop End

```

**d) Return:** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```

class A
{
    int a,b,sum;
    public int add()
    {
        a=10;
        b=15;
        sum=a+b;
        return sum;
    }
}

```

```
class B
{
    public static void main()
    {
        A obj= new A();
        int res=obj.add();
        Console.WriteLine(res);
    }
}
```

**Output:**

25

**e) Throw:** The throw statement throws an exception to indicate an error has occurred.

```
If(age<18)
    throw new ArithmeticException("Not Eligible to Vote")
```

**Namespaces in C# Programming:** Namespaces are used in C# to organize and provide a level of separation of codes. They can be considered as a container which consists of other namespaces, classes, etc.

A namespace can have following types as its members:

- a) Namespaces (Nested Namespace)
- b) Classes
- c) Interfaces
- d) Structures
- e) Delegates

**Namespaces are not mandatory in a C# program**, but they do play an important role in writing cleaner codes and managing larger projects.

```
namespace MyNamespace
{
    class MyClass
    {
        public void MyMethod()
        {
            System.Console.WriteLine("Creating my namespace");
        }
    }
}
```

In the above example, a namespace MyNamespace is created. It consists of a class MyClass as its member. MyMethod is a method of class MyClass.

### Accessing Members of Namespace in C#:

*Syntax: Namespace-Name.Member-Name*

```
using System;
namespace MyNamespace
{
    public class SampleClass
    {
        public static void myMethod()
        {
            Console.WriteLine("Creating my namespace");
        }
    }
}

namespace MyProgram
{
    public class MyClass
    {
        public static void Main()
        {
            MyNamespace.SampleClass.myMethod();
        }
    }
}
```

**Using a Namespace in C#:** A namespace can be included in a program using the using keyword. The syntax is,

**using Namespace-Name;**

**Two classes with the same name can be created inside 2 different namespaces in a single program**

### Rules within a Namespace:

#### a) Name scoping:

```
namespace outer
{
    class class1 {}
    namespace inner
```

```
{  
  class class2 : class1 {}  
}  
}
```

- b) Name hiding:** If the same type name appears in both an inner and an outer namespace, the inner name wins.

```
namespace outer  
{  
  class foo {}  
  namespace inner  
  {  
    class foo {}  
    class test  
    {  
      Foo f1; //outer.inner.foo  
      Outer.Foo f2; //outer.foo  
    }  
  }  
}
```

- c) Nested using directive:** You can nest a using directive within a namespace. This allows you to scope the using directive within a namespace declaration. In the following example, Class1 is visible in one scope, but not in another.

```
namespace N1  
{  
  class Class1 {}  
}  
  
namespace N2  
{  
  class Class2: Class1 {}  
}  
  
namespace N3  
{  
  class Class3: Class1 {} //compile- time -error  
}
```