# UNIT-II Introduction to C#

## *Introduction-

**C#** is a Pure Object Oriented Programming Language under .NET Framework developed by Microsoft.
**C#** combines ***concept of C, Power of C++, Elegance of Java and Productivity of VB.***
**C#** is designed and developed by ***Anders Hejlsberg*** & his team members at Microsoft.

## *Language Elements-

## C# Variables-

**Def-** ***A Variable is an identifier that denotes a storage location used to store data value.*** ***OR***
***Variables are containers used to store data values.***
A variable may take different values at different time during the execution of program.

### Variable naming conventions-

1. Variable name may consist of alphabet, digit and underscore
2. They must not begin with digit.
3. Uppercase and Lowercase are different/distinct i.e. the variable SUM is not same as Sum or sum.
4. It should not be Keyword.
5. White spaces is not allowed.

### Variable Declaration Syntax-

**<data type> <variable list>;**

Here, *data type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable list may consist of one or more identifier names separated by commas.*

**Some *valid* variable definitions are shown here −**

```
int  i, j, k;
char c, ch;
float f, salary;
double d;
```

## Initializing Variables

***Variables are initialized (assigned a value) with an equal sign followed by a constant expression***.

**The general form of initialization is −**

**<variable_name> = value;**

***Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as −***

**<data_type> <variable_name> = value;**

**Some examples are −**

```
int d = 3, f = 5;    /* initializing d and f. */
byte z = 22;         /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';        /* the variable x has the value 'x'. */
```

## Accepting Values from User

***The Console class in the System namespace provides a function ReadLine() for accepting input from the user and store it into a variable.***

For example,

```
String name=Console.ReadLine();
int num;
num = Convert.ToInt32(Console.ReadLine());
                    OR
num=Int32.Parse (Console.ReadLine());
```

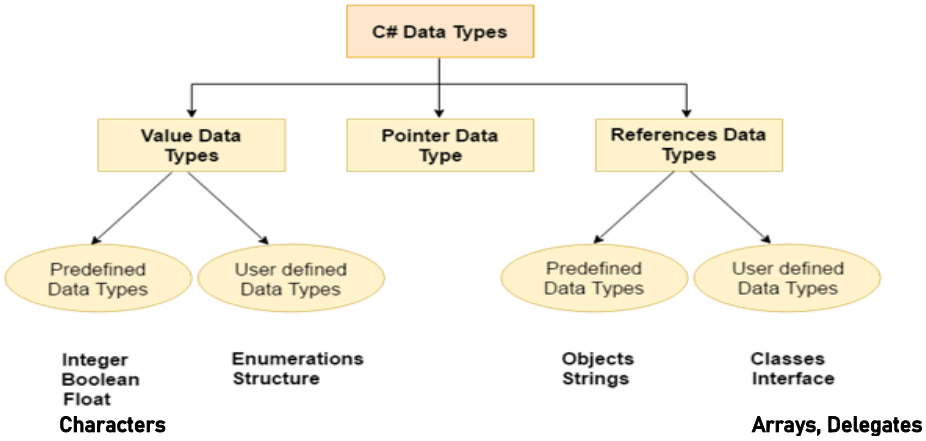
**Note-**The function ***Convert.ToInt32()/ Int32.Parse*** converts the data entered by the user to ***int*** data type, ***because Console.ReadLine() accepts the data in string format.***

## C# Data Types-

***Data type specify the size and type of values that can be stored.***

Every variable in C# is associated with a data type.

C# having rich set of data types.




## Value Types:

In C#, *Value Types are stored on the* ***Stack*** *and having* ***Fixed Length***.
***When a value of variable is assigned to another variable, the value is actually copied. It means that two identical copies of the value are available in memory***.

## Reference Types:

In C#, *Reference Types are stored on the* ***Heap*** *and having* ***variable (Not Fixed) Length.***
***When a value of reference variable is assigned to another reference variable, the reference (address) is actually copied. It means that both variables are refers to same memory location.***

## C# Operators-

***Operators in C# are symbols that are used to perform operations on operands.***
*For example, consider the expression* ***2 + 3 = 5****, here 2 and 3 are operands and + and = are called operators. So, the Operators in C# are used to manipulate the variables and values in a program.*

**Types of Operators**

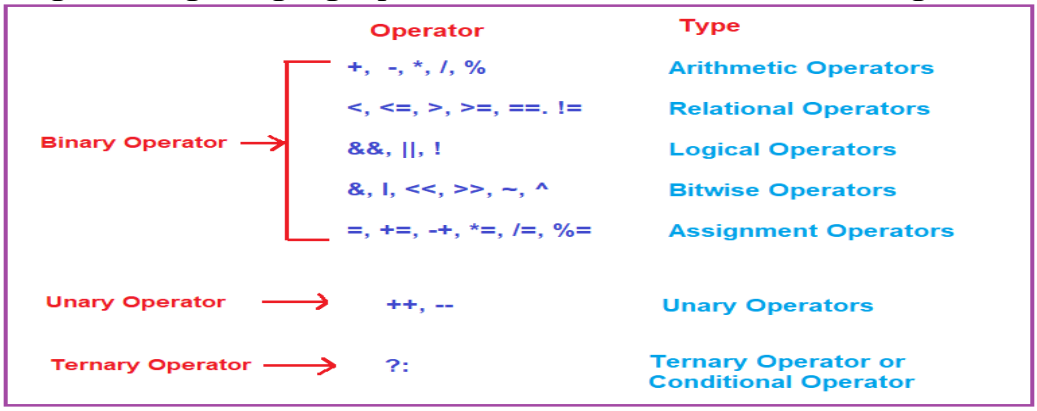
1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**
4. **Assignment Operators**
5. **Conditional Operator**
6. **Special Operator**

***In C#, the Operators can also be categorized based on the Number of Operands:***

1. **Unary Operator**: The Operator that requires one operand (variable or value) to perform the operation is called Unary Operator.(Ex. i++,-9)

2. **Binary Operator**: Then Operator that requires two operands (variables or values) to perform the operation is called Binary Operator. (Ex. a+b)
3. **Ternary Operator**: The Operator that requires three operands (variables or values) to perform the operation is called Ternary Operator. Ternary Operator is also called Conditional Operator.( Ex. a>b?a:b)

For a better understanding of the different types of operators supported in C# Programming Language, please have a look at the below image.




## 1. Arithmetic Operators

***Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.***

For example,

```
2. int x = 5;
3. int y = 10;
4. int z = x + y;// z = 15
```

| 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 |

## 2. 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 |

## 3.Logical Operators

***Logical operators are used to determine the logic between variables or values:***

| Operator | Name | Description | Example |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 && x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |

## 4. Assignment Operators

There are following assignment operators supported by C# −

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B assigns value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | **C += A is** equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | **C -= A is** equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | **C *= A** is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | **C /= A** is equivalent to C = C / A |

## 5. Conditional OR Ternary Operator

***The Conditional or ternary operator (? :) operates on three operands. It is a shorthand for if-then-else statement.***

Syntax- **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(number + “is” +result);
                }
        }
}
```

When we run the program, the output will be:

```
10 is Even Number
```

## 6.Special Operator-

***Special operators are operators which are used to perform specific task.***

C# supports the following special operators.

| | |
|---|---|
| **is** | (relational operator) |
| **as** | (relational operator) |
| **typeof** | (type operator) |
| **sizeof** | (size operator) |
| **new** | (object creator) |
| **.(dot)** | (member-access operator) |
| **checked** | (overflow checking) |
| **unchecked** | (prevention of overflow checking) |

These operators will be discussed as and when they are encountered.

## C# Decision Making and Branching-

### 1) if Statement

***The C# if statement tests the condition. It is executed if condition is true.***

**Syntax:**

```
If (condition)
{
//code to be executed
}
```

```
using System;
public class IfExample
  {
    public static void Main(string[] args)
    {
       int num = 10;
       if (num % 2 == 0)
       {
          Console.WriteLine("It is even number");
       }

    }
  }
```

**Output:**

```
It is even number
```

### 2) if-else Statement

***The C# if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.***

**Syntax:**

```
if(condition) {
//code if condition is true
}
else {
//code if condition is false
}
```

```
using System;
public class IfExample
  {       public static void Main(string[] args)
    {  int num = 11;
       if (num % 2 == 0)
       {
            Console.WriteLine("It is even number");
       }
       else
       {
          Console.WriteLine("It is odd number");
       }

    }
```

**Output-**

It is odd number

## 3) Nested if...else Statement

***An if...else statement can exist within another if...else statement. Such statements are called nested if...else statement.***

**Syntax:**

```
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
                }
            }
```

Ex.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5, y = 20;
            if (x > y)
            {
                if (x >= 10)
                {
            Console.WriteLine("x value greater than or equal to 10");
                }
                else
                {
                    Console.WriteLine("x value less than 10");
                }
            }
            else
            {
                if (y <= 20)
                {
                  Console.WriteLine("y value less than or equal to 20");
                }
                else
                {
                    Console.WriteLine("y value greater than 20");
                }
            }
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

## 4) If – else – if ladder Statement

***The if-else-if ladder statement executes one condition from multiple statements***.

The execution starts from top and checked for each if condition.

The statement of ***if*** block will be executed which evaluates to be true. If none of the ***if*** condition evaluates to be true then the last ***else*** block is evaluated.

Syntax-

```
if(condition 1)
    {
      // code to be executed if condition1 is true
    }
else if(condition 2)
    {
      // code to be executed if condition2 is true
    }
else if(condition n)
    {
      // code to be executed if condition3 is true
    }
    ...
else
    {
      // code to be executed if all the conditions are false
    }
```

```
class Program {
    public static void Main(String[] args)
   {     int i = 20;
        if (i == 10)
           Console.WriteLine("i is 10");
       else if (i == 15)
           Console.WriteLine("i is 15");
       else if (i == 20)
           Console.WriteLine("i is 20");
       else
           Console.WriteLine("i is not present");
   }
}            OutPut-  i is 20
```

## 5) Switch Case-

***Switch statement is an alternative to long if-else-if ladders.***

***The expression is checked for different cases and the one match is executed.***

**break** statement is used to move out of the switch. If the break is not used, the control will flow to all cases below it until break is found or switch comes to an end.

There is **default case (optional)** at the end of switch, if none of the case matches then default case is executed.

**Syntax-**

```
switch (expression)
 {
case value1: // statement sequence
        break;
case value2: // statement sequence
        break;
.
.
.
case value N : // statement sequence
        break;
default: // default statement sequence
}
```

```
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;
        }
     }
  }
}
```

Output- Enter an alphabet

X
Not a vowel

# C# Decision Making and Looping-

***Loops are used to execute one or more statements multiple times until a specified condition is fulfilled.***

There are many loops in C# such as for loop, while loop, do while loop etc. Details about these are given as follows:

## 1.while loop

***The while loop continuously executes loop body until loop condition is TRUE.***

***While loop is Entry Controlled Loop***

If the loop condition is true, the body of the loop is executed. Otherwise, the control flow jumps to the next statement after the while loop.

The while loop may never run if the condition is false the first time it is tested. The control skips the loop and goes directly to the next statement.

The syntax of the while loop is given as follows:

```
while (condition)
{
  // These statements will be executed if the condition evaluates to true
}
```

A program that demonstrates the while loop is given as follows:

*Source Code: Program that demonstrates while loop in C#*

```
using System;
namespace LoopDemo
{  class Example {
    static void Main(string[] args) {
      int i = 1;
      Console.WriteLine("First 10 Natural Numbers are: ");
      while (i <= 10)
      {
         Console.WriteLine( i );
         i++;
      }
    }
  }
}
```

**The output of the above program is as follows:**

```
First 10 Natural Numbers are:
1
2
3
4
5
6
7
8
9
10
```

## 2. do-while loop

***Exit Controlled Loop***

***The do while loop executes one or more statements multiple times as long as the loop condition is satisfied.***

It is similar to the while loop but the while loop has the test condition at the start of the loop and ***the do while loop has the test condition at the end of the loop. So it executes at least once always.***

The syntax of the do while loop is given as follows:

```
do
{
  // These statements will be executed if the condition evaluates to true
}while (condition);
```

A program that demonstrates the do while loop is given as follows:

*Source Code: Program that demonstrates do while loop in C#*

```
using System;
namespace LoopDemo
{
 class Example {
    static void Main(string[] args) {
       int i = 1;
 Console.WriteLine("First 10 Natural Numbers are: ");
       do
       {
```

```
            Console.WriteLine( i );
            i++;
        } while (i <= 10);
    }
  }
}
```

The output of the above program is as follows:

```
First 10 Natural Numbers are:
1
2
3
4
5
6
7
8
9
10
```

**3. for loop -*The for loop executes one or more statements multiple times as long as the loop condition is satisfied*.**

If the loop condition is true, the body of the for loop is executed. Otherwise, the control flow jumps to the next statement after the for loop.

In For loop, first the initialization is done. This declares and initializes the loop variables, if there are any. Then the condition is evaluated. If it is true, loop body is executed. If it is false, the control passes to the next statement after the for loop body. After the loop body is executed, the loop variables are updated. Then the condition is checked again and the cycle continues.

The syntax of the for loop is given below:

```
for ( initialization, condition, increment )
{
   // These statements will be executed if the condition evaluates to true
}
```

A program that demonstrates the for loop is as follows:

*Source Code: Program that demonstrates for loop in C#*

```
using System;
```

```
namespace LoopDemo
{
  class Example {
     static void Main(string[] args) {
        Console.WriteLine("First 10 Natural Numbers are: ");
        for (int i = 1; i <= 10; i++)
          Console.WriteLine(i);
     }
  }
}
```

**The output of the above program is as follows:**

```
First 10 Natural Numbers are:
1
2
3
4
5
6
7
8
9
10
```

## 4. Nested loops

***Nested loops are loop where one loop is written inside another.***

**Nested loops can be created from for loops, while loops and do while loops**.

The syntax of nested loops is given as follows:

```
Outer loop
{
     // Statements in the outer loop
              Inner Loop
              {
                 // Statements in the inner loop
              }
      // Statements in the outer loop
}
```

A program that demonstrates nested for loop is given as follows:

```
namespace LoopDemo
{
  class Example {
     static void Main(string[] args) {
        Console.WriteLine("Nested for loop");
        for( int i = 0; i < 5; i++)
        {
                    for(int j=0; j<=i; j++)
                     {
                      Console.Write("  *");
                     }
           Console.WriteLine();
        }
     }
  }
}
```

The output of the above program is as follows:

```
Nested for loop
*
* *
* * *
* * * *
* * * * *
```

## 5. foreach loop

***The foreach loop executes one or more statements for all the elements in an Array/Collection such as List, Hash Table etc***.

Syntax

```
foreach(datatype loopvariable in array,collectionname)
  {
       //body
   }
```

*Source Code: Program that demonstrates foreach loop in C#*

```
using System;
namespace LoopDemo
{
  class Example {
     static void Main(string[] args) {
       Console.WriteLine("First 10 Natural Numbers are: ");
```

```
      int[] naturalNos = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      foreach (int i in naturalNos)
      {
         Console.WriteLine(i);
      }
    }
  }
}
```

**The output of the above program is as follows:**

```
First 10 Natural Numbers are:
1
2
3
4
5
6
7
8
9
10
```

## Arrays in C#

***Array is a collection of data elements of the same type***.

These data elements are stored in contiguous memory locations. Some of the arrays in C# are 1-D arrays, 2-D arrays etc. Details about these are given as follows:

### 1-D Arrays

***One dimensional arrays consist of a single row with as many elements as required.***

These arrays can be declared using the following syntax:

**data_type[ ] name_of_array;**

In the above syntax, data_type is the data type of array elements and name_of_array is the name given to the array.

The syntax to create the array is given as follows:

**data_type[ ] name_of_array = new data_type[array_size];**

In the above syntax, data_type is the data type of array elements, name_of_array is the name given to the array, new is a keyword that creates an instance of the array and array_size is the size of the array.

There are many methods to assign values to the array. Some of these are given as follows:

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

A program that demonstrates one dimensional arrays is given as follows:

*Source Code: Program that demonstrates one dimensional arrays in C#*

```
using System;
namespace ArrayDemo
{
 class Example
 {
   static void Main(string[ ] args)
   {
     int [ ] arr = new int[5] {1, 2, 3, 4, 5};
     int i;
     Console.WriteLine("Elements in the array are:");
     for ( i = 0; i < 5; i++ )
     {
       Console.WriteLine(arr[i]);
     }
   }
 }
}
```

**The output of the above program is as follows:**

```
Elements in the array are:
1
2
3
4
5
```

## 2-D Arrays

***Two dimensional arrays contain multiple rows and columns***.

Each element of the 2-D array is identified as ***arr[i,j] where i and j are the subscripts for row and column index respectively and the name of the array is arr.***

The 2-D arrays can be declared using the following syntax:

**data_type[ , ] name_of_array = new data_type[row_size, column_size];**

In the above syntax, data_type is the data type of array elements, name_of_array is the name given to the array, new is a keyword that creates an instance of the array,row_size is the number of rows and column_size is the number of columns.

One of the methods to assign values in a 2-D array is given as follows:

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

A program that demonstrates two dimensional arrays is given as follows:

```
using System;
namespace ArrayDemo
{
 class Example
 {
   static void Main(string[] args)
   {
     int[,] arr = new int[2, 3] { {4,1,7}, {3, 8, 2} };
     int i, j;
     Console.WriteLine("The elements in the 2-D array are:");
     for (i = 0; i < 2; i++)
     {
       for (j = 0; j < 3; j++)
       {
         Console.Write( “   “+arr[i,j]);
        }
       Console.WriteLine();
     }
   }
 }
}
```

**The output of the above program is as follows:**

```
The elements in the 2-D array are:
4 1 7
3 8 2
```

# Array Class

***The Array class is defined in the System namespace***.

Using Array class, you can easily work with arrays. ***It is the base class for all the arrays in C#.'***

## Most Common Properties in Array Class

| *Property* | *Description* |
|---|---|
| ***IsFixedSize*** | *This property gets a value indicating whether the Array has a fixed size.* |
| ***IsReadOnly*** | *This property gets a value indicating whether the Array is read-only.* |
| ***Length*** | *This property gets the total number of elements in all the dimensions of the Array.* |
| ***Rank*** | *This property gets the rank (number of dimensions) of the Array. For example, a one-dimensional array returns 1, a two-dimensional array returns 2, and so on.* |

## Most Common Methods in Array Class

| Methods | Description |
|---|---|
| ***Clear(Array, Int32, Int32)*** | ***This method sets a range of elements in an array to the default value of each element type.*** |
| *Clone()* | *This method creates a shallow copy of the Array.* |
| ***Copy(Array, Array, Int32)*** | ***This method copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.*** |
| *GetLength(Int32)* | *This method gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.* |
| *GetUpperBound(Int32)* | *This method gets the index of the last element of the specified dimension in the array.* |

| Methods | Description |
|---|---|
| *GetValue(Int32)* | *This method gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.* |
| ***IndexOf(Array, Object)*** | ***This method searches for the specified object and returns the index of its first occurrence in a one-dimensional array.*** |
| ***Reverse(Array)*** | ***This method reverses the sequence of the elements in the entire one-dimensional Array.*** |
| *SetValue(Object, Int32)* | *This method sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.* |
| ***Sort(Array)*** | ***This method sorts the elements in an entire one-dimensional Array using the IComparableimplementation of each element of the Array.*** |
| ***ToString()*** | ***This method returns a string that represents the current object.*** |

## Examples On Array Class Methods Sort(),Reverse()-

1. Sort one dimensional array in descending order using Array Class method

```
using System;
namespace Demo
{
  public class MyApplication
  {
     public static void Main(string[] args)
     {
        int [] arr = {34, 76, 1, 99, 68};
        Console.Write("The original unsorted array is: ");
        foreach (int i in arr)
        {
          Console.Write(i + " ");
        }
```

```
        Array.Sort(arr);
        Array.Reverse(arr);
        Console.WriteLine();
        Console.Write("The sorted array is: ");
        for(int i=0; i<arr.Length; i++)
        {
           Console.Write(arr[i] + " ");
        }
     }
  }
}
```

**The output of the above program is as follows:**

```
The original unsorted array is: 34   76   1   99   68
The sorted array is: 99   76   68   34   1
```

**2.A program that demonstrates the copy(, ,) method of array class is given as follows:**

```
using System;
class Example
{
  static void Main()
  {
     int[] arr1 = new int[5] { 45, 12, 9, 77, 34};
     int[] arr2 = new int[5];
     Array.Copy(arr1, arr2, 5);
     Console.WriteLine("arr1: ");
     foreach (int i in arr1)
     {
        Console.Write(i + " ");
     }
     Console.WriteLine();
     Console.WriteLine("arr2: ");
     foreach (int i in arr2)
     {
        Console.Write(i + " ");
     }
  }
}
```

**The output of the above program is as follows:**

```
arr1:
45 12 9 77 34
arr2:
45 12 9 77 34
```

## C# String

***String is a sequence of characters.***

For example, `"hello"` is a string containing a sequence of characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`.

We use the `string` keyword to create a string. For example,

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

Here, we have created a `string` named `str` and assigned the text `"C# Programming"`. We use double quotes to represent strings in C#.

**Example: Create string in C#**

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

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


     // print string
     Console.WriteLine(str1);
     Console.WriteLine(str2);

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

**Output**

```
C# Programming
Programiz
```

In the above example, we have created two strings named `str1` and `str2` and printed them.

> *Note:* ***A string variable in C# is not of primitive types like*** `int`, `char`, ***etc. Instead, it is an object of the*** `String` ***class.***

String Class Methods

| Properties | Description |
|---|---|
| Clone() | Make clone of string. |
| CompareTo() | Compares two specified String objects and returns an integer that indicates their relative position in the sort order. |
| Contains() | Returns a value indicating whether a specified substring occurs within this string. |
| EndsWith() | Determines whether the end of this string instance matches the specified string. |
| Equals() | Determines whether this instance and another specified String object have the same value |
| IndexOf(String) | Reports the zero-based index of the first occurrence of the specified string in this instance. |
| ToLower() | Returns a copy of this string converted to lowercase. |
| ToUpper() | Returns a copy of this string converted to uppercase. |
| Insert() | Returns a new string in which a specified string is inserted at a specified index position in this instance. |
| LastIndexOf(String) | Reports the zero-based index position of the last occurrence of a specified string within this instance. |
| Remove() | This method deletes all the characters from beginning to specified index position. |
| Replace() | This method helps to replace the character. |
| Split() | This method splits the string based on specified value. |
| StartsWith(String) | Determines whether the beginning of this string instance matches the specified string. |
| Substring() | Retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string. |
| Trim() | It removes extra whitespaces from beginning and ending of string. |

## String Operations

C# string provides various methods to perform different operations on strings. We will look into some of the commonly used string operations.

### 1. Get the Length of a string

To find the length of a string, we use the `Length` property. For example,

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

      // create string
```

```
    string str = "C# Programming";
    Console.WriteLine("string: " + str);

    // get length of str
    int length = str.Length;

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

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

**Output**

```
string: C# Programming
Length: 14
```

In the above example, the `Length` property calculates the total number of characters in the string and returns it.

**2. Join two strings in C#**

We can join two strings in C# using the `Concat()` method. For example,

```
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();
    }
  }
}
```

**Output**

```
string str1: C#
string str2: Programming
Joined string: C# Programming
```

In the above example, we have created two strings named `str1` and `str2`. Notice the statement,

```
string joinedString = string.Concat(str1, str2);
```

Here, the `Concat()` method joins `str1` and `str2` and assigns it to the `joinedString` variable.

We can also join two strings using the `+` operator in C#. To learn more, visit *C# string Concat*.

### 3. C# compare two strings

In C#, we can make comparisons between two strings using the `Equals()` method. The `Equals()` method checks if two strings are equal or not. For example,

```
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();
   }
  }
}
```

**Output**

```
string str1 and str2 are equal: True
string str1 and str3 are equal: False
```

In the above example, we have created 3 strings named `str1`, `str2`, and `str3`. Here, we are using the `Equals()` method to check if one string is equal to another.

***Immutable string***

***A String cannot be changed once it is created***

A new memory is created every time on modification.

String str = "tim";

***Mutable string***

***A String can be changed again and again.***

***StringBuilder is a mutable string in C#. With StringBuilder, you can expand the number of characters in the string..***

StringBuilder str = new StringBuilder("stringliteral");

**Constant Variables**- *in c#, constant field values are set at compile-time, and those values will never change.*

To define constant fields in c#, we need to use ***const*** keyword during the declaration of fields in our application, and we can use constants with numbers, boolean values, strings, or null references.

It's mandatory to initialize constant fields with required values during the declaration itself; otherwise, we will get compile-time errors in our c# application.

In c#, the static modifier is not allowed to use during the declaration of constant variables.

**Syntax const data_type Variable_name = "value";**

```
// Ex. Constant variables
const string name = "Suresh Dasari";
const string location = "Hyderabad";
const int age = 32;
```

## Enumeration –

***Enumeration (or enum)*** **is a value data type in C#.**

***It is mainly used to assign the names or string values to Numbers , that make a program easy to read and maintain.***

The main objective of *enum* is to define our own data types (Enumerated Data Types).

Enumeration is declared using *enum* keyword directly inside a namespace, class, or structure.

**Syntax:**

```
enum     Enum_name
{
    string_1...;
    string_2...;
    .
    .
}
```

Example **: Consider the below code for the enum. Here enum with name month is created and its data members are the name of months like jan, feb, mar, apr, may. Now let's try to print the default integer values of these enums. An explicit cast is required to convert from enum type to an integral type.**

```
using System;
namespace ConsoleApplication1 {

// making an enumerator 'month'
enum month
{

   // following are the data members
   jan,
   feb,
   mar,
   apr,
   may

}

class Program {

   // Main Method
   static void Main(string[] args)
   {

      // getting the integer values of data members..
      Console.WriteLine("The value of jan in month " +
                  "enum is " + (int)month.jan);
      Console.WriteLine("The value of feb in month " +
                  "enum is " + (int)month.feb);
      Console.WriteLine("The value of mar in month " +
                  "enum is " + (int)month.mar);
      Console.WriteLine("The value of apr in month " +
                  "enum is " + (int)month.apr);
      Console.WriteLine("The value of may in month " +
                  "enum is " + (int)month.may);
   }
}
}
```

Output-
**The value of jan in month enum is 0**
**The value of feb in month enum is 1**
**The value of mar in month enum is 2**
**The value of apr in month enum is 3**
**The value of may in month enum is 4**

## Parameter passing in C#-

**There are four different ways of passing parameters to a method in C# which are as:**

1. **Pass By Value**
2. **Pass By Ref (reference)**
3. **Out (reference)**
4. **Params (parameter arrays)**

**1.Pass By Value**

*By default, parameters are passed by value.*

***The values of the actual parameters are copied into formal parameters.***

***Hence, the changes made to the formal parameter inside the method have no effect on the actual parameters.***

*i.e. In this method a duplicate copy is made and sent to the called function. There are two copies of the variables. So if you change the value in the called method it won't be changed in the calling method.*

*Ex.*

```
using System;

namespace CalculatorApplication {
  class NumberManipulator {

     static  void swap(int x, int y)
    {
       int temp;

       temp = x; /* save the value of x */
```

```
      x = y;    /* put y into x */
      y = temp; /* put temp into y */
    }

    static void Main(string[] args)
    {
      /* local variable definition */
      int a = 100;
      int b = 200;

      Console.WriteLine("Before swap, value of a : "+ a);
      Console.WriteLine("Before swap, value of b : "+ b);

      /* calling a function to swap the values */
      swap(a, b);

      Console.WriteLine("After swap, value of a :"+ a);
      Console.WriteLine("After swap, value of b : "+ b);

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

Output-

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

## 2.Pass By Reference-

***A reference parameter is a reference to a memory location of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters.***

***The reference parameters represent the same memory location as the actual parameters that are supplied to the method.***

***Hence, the changes made to the formal parameter inside the method are effects on the actual parameters.***

***You can declare the reference parameters using the ref keyword in front of both formal and actual parameters.***

```
using System;

namespace CalculatorApplication {
  class NumberManipulator {

     public static void swap(ref int x, ref int y)
   {
       int temp;
       temp = x; /* save the value of x */
       x = y;    /* put y into x */
       y = temp; /* put temp into y */
    }
    static void Main(string[] args) {

       /* local variable definition */
       int a = 100;
       int b = 200;

       Console.WriteLine("Before swap, value of a : "+a);
       Console.WriteLine("Before swap, value of b : "+b);

       /* calling a function to swap the values */
       swap(ref a, ref b);

       Console.WriteLine("After swap, value of a : "+ a);
       Console.WriteLine("After swap, value of b : "+b);

       Console.ReadLine();
    }
  }
}
OutPut-
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

### 3)Out (reference) Parameter-

***The out parameter in c# is useful to return more than one value from the methods in the c# programming language.***

***The out paramters are always passed by reference for both, the value type and the reference type data types.***

```
class Program
 {
   static void Main(string[] args)
   {
      int x, y;
      Multiplication(out x, out y);
      Console.WriteLine("x Value: {0}", x);
      Console.WriteLine("y Value: {0}", y);
      Console.WriteLine("Press Enter Key to Exit..");
      Console.ReadLine();
   }
   public static void Multiplication(out int a, out int b)
   {
      a = 10;
      b = 5;
      a *= a;
      b *= b;
   }
 }
```

### 4.Params (parameter arrays)

***In C#,*** *params* ***is a keyword which is used to specify a parameter that takes variable number of arguments****.*
***It is useful when we don't know the number of arguments prior****.*
**Note- Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.**

**Example**

```
class Program{
  public static int Add(params int[] ListNumbers)
   {
      int total = 0;
      foreach(int i in ListNumbers)
      {
         total =total+ i;
      }
      return total;
   }

static void Main(string[] args)
{

   // Calling function by passing 5 arguments as follows
   int y = Add(12,13,10,15,56);

   // Displaying result
   Console.WriteLine(y);

// Calling function by passing 3 arguments as follows
   int r = Add(1,2,3);

   // Displaying result
   Console.WriteLine(r);
}
}
```

Output-

106

6

## Boxing and UnBoxing-

***Boxing is the process of converting a value type on stack in to the object type on heap.***

***Boxing is implicit***.

***A boxing conversion makes a copy of the value. So, changing the value of one variable will not impact others.***

**Example: Boxing**

```
int i = 10;
object o = i; //performs boxing
```

*In the above example, the integer variable that is "i" is assigned to the object "o". Thus the object data type is a reference type and base class of all the other classes in C# ultimately, an integer can be assigned to an object type. And this process of converting from the integer to the object data type is called boxing.*

***Unboxing is the reverse of boxing. It is the process of converting a reference type to value type.***

**Unboxing extract the value from the reference type and assign it to a value type.**

***Unboxing is explicit. It means we have to cast explicitly.***

**Example: Unboxing**

```
int i = 10;
object o = i; //performs boxing
object o = 10;
int i = (int)o; //performs unboxing
```

| Boxing | Unboxing |
|---|---|
| It convert value type into an object type. | It convert an object type into value type. |
| Boxing is an implicit conversion process. | Unboxing is the explicit conversion process. |
| Here, the value stored on the stack copied to the object stored on the heap memory. | Here, the object stored on the heap memory copied to the value stored on the stack . |