**Unit-2 (The C# Language Basic)**

**Writing Console Application:**

# **Comments in C#**

Comments are used for explaining the code and are used in a similar manner as in Java, C or C++. Compilers ignore the comment entries and do not execute them. Generally, programming languages contain two types of comments but in C#, there are **3 Types** of comments :

**Single Line Comments :** It is used to comment a single line. These comment can be written in a separate line or along with the codes in the same line. But for better understanding always use the comment in a separate line.

// Single Line Comments

**Multiline Comments :** It is used to comment more than one line. Generally this is used to comment out an entire block of code statements.

/\* Multiline

Comment \*/

**XML Documentation Comments :** It is a special type of comment in C# and used to create the documentation of C# code by adding XML elements in the source code. XML elements are added in XML Documentation Comments of C#.

/// <summary>

/// This class does something of program Summary.

/// </summary>

**C# Data Types**

1. **Value Types:**

**Integral Types:**

* + - * **byte: 8-bit unsigned integer.**
      * **sbyte: 8-bit signed integer.**
      * **short: 16-bit signed integer.**
      * **ushort: 16-bit unsigned integer.**
      * **int: 32-bit signed integer.**
      * **uint: 32-bit unsigned integer.**
      * **long: 64-bit signed integer.**
      * **ulong: 64-bit unsigned integer.**
      * **Floating-Point Types:**
      * **float: 32-bit floating-point number.**
      * **double: 64-bit double-precision floating-point number.**
      * **decimal: 128-bit decimal type for financial and monetary calculations.**

**Other Value Types:**

* **char: 16-bit Unicode character.**
* **bool: Represents a Boolean value (true or false).**

**Enumerations:**

* **enum: A user-defined type that consists of named constant values.**

1. **Reference Types:**

**Class Types:**

**class: Reference type that can contain data members, methods, properties, etc.**

**Interface Types:**

**interface: Defines a contract for classes to implement.**

**Array Types:**

**array: A collection of elements of the same type.**

**Delegate Types:**

**delegate: Defines a reference type that can encapsulate a method with a specific signature.**

**String Type:**

**string: Represents a sequence of characters.**

1. **Nullable Value Types:**

**Nullable<T>: Allows value types to have a value of null.**

**int? nullableInt = null;**

1. **Dynamic Type:**

**dynamic: Represents an object whose operations are resolved at runtime.**

**var i=1;**

**Example:**

namespace ConsoleApp9

{

internal class Program

{

static void Main(string[] args)

{

// Value Types

int integerNumber = 42;

double floatingPointNumber = 3.14;

char character = 'A';

bool isTrue = true;

// Reference Types

string text = "Hello, C#!";

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

MyClass myObject = new MyClass();

// Nullable Value Type

int? nullableInt = null;

// Dynamic Type

dynamic dynamicVariable = 10;

dynamicVariable = "Now I'm a string.";

Console.WriteLine("Integer:”+integerNumber);

Console.WriteLine("Float:”+ floatingPointNumber);

+ “Float: +floatingPointNumber+”Character:” +character);

Console.WriteLine($"Integer: {integerNumber}, Float: {floatingPointNumber}, Character: {character}");

Console.WriteLine($"Boolean: {isTrue}");

Console.WriteLine($"String: {text}");

Console.WriteLine($"Array: [{string.Join(", ", numbersArray)}]");

Console.WriteLine($"Nullable Int: {nullableInt.GetValueOrDefault()}");

Console.WriteLine($"Dynamic Variable: {dynamicVariable}");

Console.ReadLine();

}

}

class MyClass

{

// Class Definition

}

**Expressions and Operators in C#**

**Expressions**   
Much of the program’s work is the calculation of expressions. **Expressions are sequences of operators, literals and variables** that are calculated to a value of some type (number, string, object or other type).

**Operator:**  
An operator is a symbol that represents an operation to be performed on one or more operands. Operands are the values or variables on which the operation is applied.

1. **Arithmetic Operators:**

**Addition (**+**):** Adds two operands.

**Subtraction (**-**):** Subtracts the right operand from the left operand.

**Multiplication (**\***):** Multiplies two operands.

**Division (**/**):** Divides the left operand by the right operand.

**Modulus (**%**):** Returns the remainder of the division of the left operand by the right operand.

1. **Relational Operators:**

Equal to (==): **Checks if two operands are equal.**Not equal to (!=): **Checks if two operands are not equal.**Greater than (>): **Checks if the left operand is greater than the right operand.**Less than (<): **Checks if the left operand is less than the right operand.**Greater than or equal to (>=): **Checks if the left operand is greater than or equal to the right operand.**Less than or equal to (<=): **Checks if the left operand is less than or equal to the right operand.**

**Example:**

static void Main(string[] args)

{

int x = 10, y = 5;

Console.WriteLine("x > y : " + (x > y)); // True

Console.WriteLine("x < y : " + (x < y)); // False

Console.WriteLine("x >= y : " + (x >= y)); // True

Console.WriteLine("x <= y : " + (x <= y)); // False

Console.WriteLine("x == y : " + (x == y)); // False

Console.WriteLine("x != y : " + (x != y)); // True

Console.ReadLine();

}

1. **Logical Operators:**

Logical AND (&&): **Returns true if both operands are true.**

Logical OR (||): **Returns true if at least one operand is true.**

Logical NOT (!): **Returns true if the operand is false and false if the operand is true.**

**Example:**

static void Main(string[] args)

{

bool a = true;

bool b = false;

Console.WriteLine(a && b); // False

Console.WriteLine(a || b); // True

Console.WriteLine(!b); // True

Console.WriteLine(b || true); // True

Console.WriteLine((5 > 7) ^ (a == b)); // False

Console.ReadLine();

}

1. **Assignment Operators:**

Assignment (=): **Assigns the value on the right to the variable on the left.**Addition assignment (+=), Subtraction assignment (-=), etc.: **Performs the specified operation and assigns the result to the variable.**

1. **Conditional (Ternary) Operator:**

Conditional (? :): **Evaluates a boolean expression and returns one of two values based on whether the expression is true or false.**

static void Main(string[] args)

{

int a = 6;

int b = 4;

Console.WriteLine(a > b ? "a>b" : "b<=a"); // a>b

int num = a == b ? 1 : -1; // num will have value -1

Console.ReadLine();

}

1. **Other Operators:**

Increment (++): **Increments a variable by 1.**Decrement (--): **Decrements a variable by 1.**Unary Minus (-): **Negates the value of an operand.**

**Keywords and Identifier:**

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

For example,

long mobileNum;

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

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

long long;

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

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

int @void;

The above statement will create a variable @void of type int.

**Contextual Keywords(2021 Exam)**

Besides regular keywords, C# has 25 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 | alias | ascending |
| async | await | descending |
| dynamic | from | get |
| global | group | into |
| join | let | orderby |
| partial (type) | partial (method) | remove |
| select | set | value |
| var | when (filter condition) | where (generic type constraint) |
| yield |  |  |

**Rules for Naming an Identifier**

* An identifier can not be a C# keyword.
* An identifier must begin with a letter, an underscore or @ symbol. The remaining part of identifier can contain letters, digits and underscore symbol.
* Whitespaces are not allowed. Neither it can have symbols other than letter, digits and underscore.
* Identifiers are case-sensitive. So, **getName**, **GetName** and **getname** represents 3 different identifiers.

Here are some of the valid and invalid 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 |

**Example: Find list of keywords and identifiers in a program**

using System;

namespace HelloWorld

{

class Hello

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

}

|  |  |
| --- | --- |
| Keywords | Identifiers |
| using | System |
| namespace | HelloWorld (namespace) |
| class | Hello (class) |
| static | Main (method) |
| void | args |
| string | Console |
|  | WriteLine |

# **String and Character**

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

char c = 'A'; // Simple character

***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.

For example:

char newLine = '\n';

char backSlash = '\\';

*Table 2-2. Escape sequence characters*

|  |  |
| --- | --- |
| **Char** | **Meaning** |
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \0 | Null |
| \a | Alert |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
|  |  |

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

|  |  |  |  |
| --- | --- | --- | --- |
| char copyrightSymbol | | = '\u00A9'; | |
| har | omegaSymbol | = | '\u03A9'; |
| char | newLine | = | '\u000A'; |

**Char Conversions**

An implicit conversion from a **char** to a numeric type works for the numeric types that can accommodate an unsigned **short**. For other numeric types, an explicit con‐ version is required.

Example:

static void Main(string[] args)

{

Console.WriteLine('\'');

Console.WriteLine("New Line Example" + "\n");

Console.WriteLine("This is a backslash: \\");

Console.WriteLine("She said, \"Hello!\"");

Console.WriteLine("This is a line.\rThis text replaces the first line.");

Console.WriteLine("Hello, \bWorld!");

Console.WriteLine("This is\tindented.");

Console.ReadLine();

}

Output:

New Line Example

This is a backslash: \

She said, "Hello!"

This text replaces the first line.

Hello,World!

This is indented.

**String Type**

C#’s string type (aliasing the **System.String** type, A string literal is specified inside double quotes:  
string a = "Heat";  
string is a reference type, rather than a value type. Its equality operators, however, follow value-type semantics:  
string a = "test";  
string b = "test";  
Console.Write (a == b); // True

The escape sequences that are valid for char literals also work inside strings:  
string a = "Here's a tab:**\t**";

The cost of this is that whenever you need a literal backslash, you must write it twice:  
string a1 = "\\\\server\\fileshare\\helloworld.cs";

To avoid this problem, C# allows *verbatim* string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:  
string a2 = **@**"\\server\fileshare\helloworld.cs";

A verbatim string literal can also span multiple lines:  
string escaped = "First Line\r\nSecond Line";   
string verbatim = @"First Line Second Line";  
  
True if your IDE uses CR-LF line separators:   
Console.WriteLine(escaped == verbatim);

You can include the double-quote character in a verbatim literal by writing it twice:  
string xml = @"<customer id=""123""></customer>";

**String concatenation**The + operator concatenates two strings:

string s = "a" + "b";

One of the operands may be a nonstring value, in which case ToString is called on that value. For example:

string s = "a" + 5; // a5

Using the + operator repeatedly to build up a string is inefficient: a better solution is to use the System.Text.StringBuilder type .

**String interpolation(Exam 2021)**

A string preceded with the $ character is called an *interpolated string*. Interpolated strings can include expressions inside braces:

int x = 4;  
Console.Write (**$**"A square has **{x}** sides"); // Prints: A square has 4 sides  
Any valid C# expression of any type can appear within the braces, and C# will convert the expression to a string by calling its ToString method or equivalent. You can change the formatting by appending the expression with a colon and a *format string*

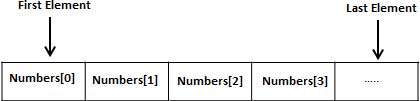
string s = $"255 in hex is {byte.MaxValue**:X2**}"; // X2 = 2-digit Hexadecimal // Evaluates to "255 in hex is FF"

# **Array**

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



# **Declaring Arrays**

To declare an array in C#, you can use the following syntax −

datatype[] arrayName;

where,

* *datatype* is used to specify the type of elements in the array.
* *[ ]* specifies the rank of the array. The rank specifies the size of the array.
* *arrayName* specifies the name of the array.

For example,

double[] balance;

# **Initializing an Array**

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.

Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

double[] balance = new double[10];

# **Assigning** **Values to an Array**

You can assign values to individual array elements, by using the index number, like −

double[] balance = new double[10];

balance[0] = 4500.0;

You can assign values to the array at the time of declaration, as shown –

double[] balance = { 2340.0, 4523.69, 3421.0};

You can also create and initialize an array, as shown –

int [] marks = new int[5] { 99, 98, 92, 97, 95};

You may also omit the size of the array, as shown –

int [] marks = new int[] { 99, 98, 92, 97, 95};

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location –

int [] marks = new int[] { 99, 98, 92, 97, 95};

int[] score = marks;

When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example, for an int array all elements are initialized to 0.

# **Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

double salary = balance[9];

The following example, demonstrates the above-mentioned concepts declaration, assignment, and accessing arrays –

using System;

namespace ArrayApplication {

class MyArray {

static void Main(string[] args) {

int [] n = new int[10]; /\* n is an array of 10 integers \*/

int i,j;

/\* initialize elements of array n \*/

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

n[ i ] = i + 100;

}

/\* output each array element's value \*/

for (j = 0; j < 10; j++ ) {

Console.WriteLine("Element[{0}] = {1}", j, n[j]);

}

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result –

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

# **Using the *foreach* Loop**

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

using System;

namespace ArrayApplication {

class MyArray {

static void Main(string[] args) {

int [] n = new int[10]; /\* n is an array of 10 integers \*/

/\* initialize elements of array n \*/

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

n[i] = i + 100;

}

/\* output each array element's value \*/

foreach (int j in n ) {

int i = j-100;

Console.WriteLine("Element[{0}] = {1}", i, j);

}

Console.ReadKey();

}

}

}

# **C# Array Types**

There are 3 types of arrays in C# programming:

1. Single Dimensional Array
2. Multidimensional Array
3. Jagged Array

# **Multidimensional Array**

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array. You can declare a 2-dimensional array of strings as –

string [,] names;

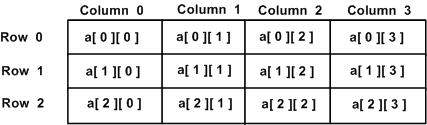
or, a 3-dimensional array of int variables as −

int [ , , ] m;

# **Two-Dimensional Arrays**

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns −



Thus, every element in the array a is identified by an element name of the form a[ i , j ], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in array a.

# **Initializing Two-Dimensional Arrays**

Multidimensional arrays may be initialized by specifying bracketed values for each row. The Following array is with 3 rows and each row has 4 columns.

int [,] a = new int [3,4] {

{0, 1, 2, 3} , /\* initializers for row indexed by 0 \*/

{4, 5, 6, 7} , /\* initializers for row indexed by 1 \*/

{8, 9, 10, 11} /\* initializers for row indexed by 2 \*/

};

# **Accessing Two-Dimensional Array Elements**

An element in 2-dimensional array is accessed by using the subscripts. That is, row index and column index of the array. For example,

int val = a[2,3];

The above statement takes 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check the program to handle a two dimensional array –

Example:

class Program

{

static void Main(string[] args)

{

/\* an array with 5 rows and 2 columns\*/

int[,] a = new int[5, 2] { { 0, 0 }, { 1, 2 }, { 2, 4 }, { 3, 6 }, { 4, 8 } };

int i, j;

/\* output each array element's value \*/

for (i = 0; i < 5; i++)

{

for (j = 0; j < 2; j++)

{

Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i, j]);

}

}

Console.ReadKey();

}

}

Output:

a[0,0]: 0

a[0,1]: 0

a[1,0]: 1

a[1,1]: 2

a[2,0]: 2

a[2,1]: 4

a[3,0]: 3

a[3,1]: 6

a[4,0]: 4

a[4,1]: 8

Example:

static void Main(string[] args)

{

int sum = 0;

int[,] a = new int[3, 4] { { 0, 1, 2, 3 }, { 4, 5, 6, 7 }, { 8, 9, 10, 11 } };

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 4; j++)

{

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

sum = sum + a[i, j];

}

Console.Write(sum);

Console.WriteLine();

sum = 0;

}

Console.ReadLine();

}

**Jagged array**

A Jagged array is an array of arrays. In jagged array no of rows is fixed and no of column may varies. A jagged array (also known as a ragged array) is an array of arrays in which the member arrays in turn can be of different dimensions and sizes. You may implement multi-dimensional arrays are jagged arrays to improve performance. You can declare a jagged array named *scores* of type **int** as −

int [][] scores;

Declaring an array, does not create the array in memory. To create the above array −

int[][] scores = new int[5][];

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

scores[i] = new int[4];

}

You can initialize a jagged array as −

int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};

Where, scores is an array of two arrays of integers - scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

# **Example**

The following example illustrates using a jagged array –

Q.Write a C# program to initialize and display jagged array elements with sum of each row.(2021)

Ans:

namespace ConsoleApp14

{

internal class Program

{

static void Main(string[] args)

{

int sum = 0;

int[][] scores = new int[2][] { new int[] { 92, 93, 94 }, new int[] { 85, 66, 87, 88 } };

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

{

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

{

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

sum += Convert.ToInt32(scores[i][j]);

}

Console.Write("="+sum);

sum = 0;

Console.WriteLine();

}

Console.ReadLine();

}

}

}

# **Advantages of C# Array**

* Code Optimization (less code)
* Random Access
* Easy to traverse data
* Easy to manipulate data
* Easy to sort data etc.

# **Disadvantages of C# Array**

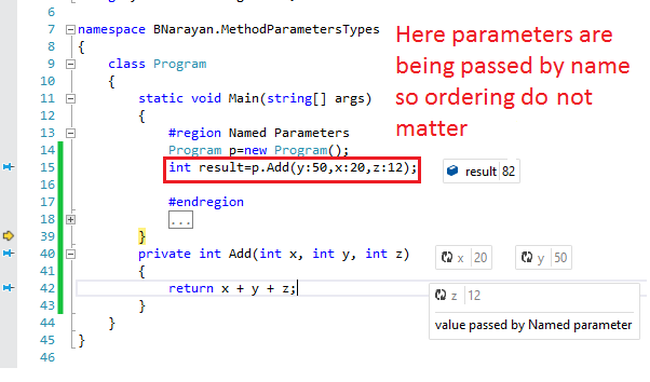
* Fixed size

**Parameters**

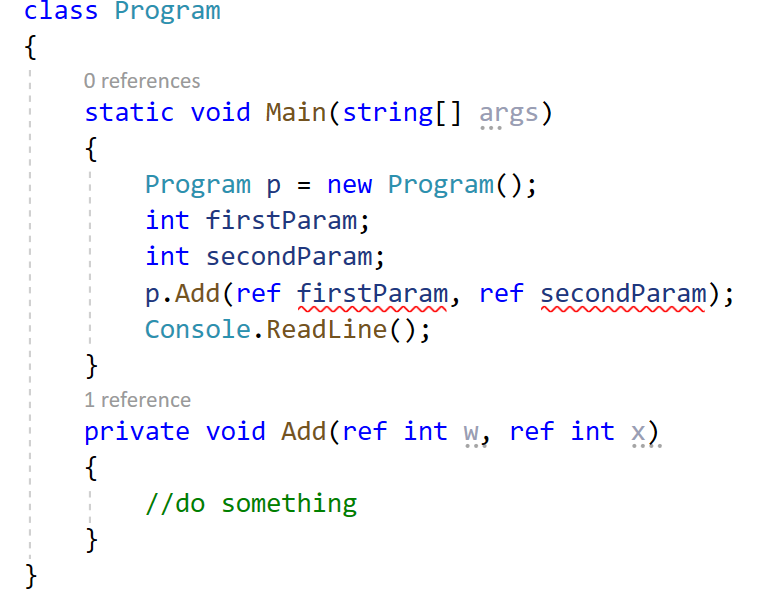
A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. Method parameter is one of the integral parts of programming.

We can categorize method parameters in various parts. Some of them are:

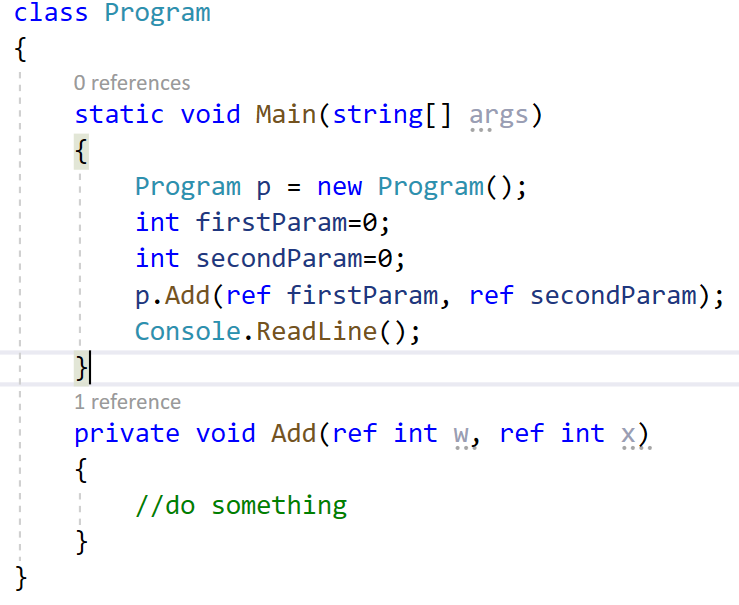
* Named Parameters (C# 4.0 and above)
* Ref Parameter (Passing Value Types by Reference)
* Out Parameters
* Default Parameters or Optional Arguments (C# 4.0 and above)
* Dynamic parameter (dynamic keyword).
* Value parameter or Passing Value Types by Value (normal C# method param are value parameter)
* Params (params)

**Named Parameters**Using this feature we can specify the value of a parameter by parameter name regardless of its ordering in method. Have a look at the following screenshot.  
  
  
  
It provides an ease for user because sometimes we may have 10-20 parameters in a method. In that case we need to remember ordering of all those parameters. But if we use the default parameter feature of C# then we do not need to remember any ordering. This feature is available in C# 4.0 and above versions (C#, C# 6).

**Ref Parameter (Passing Value Types by Reference)**In C# by default we pass the parameter by value also known as **value parameter**. But a parameter also can be passed by **reference**. To pass an argument with reference we need to use **"ref"** keyword. If any argument passed as **reference**, then any change to the parameter reflected in the calling **method**.  
  
Any argument being passed by reference must be assigned before passing it to the method. If it is not assigned then it will give compile time error.



Now assign value for those parameters and error will gone.



**Example ref Parameter**

class Program

{

static void Main(string[] args)

{

int a = 10;

//Creating an object of A class

A ob = new A();

//Printing the value of local variable,a

//Before calling the Add() method

Console.WriteLine("Before calling the Add() method, the value in a is : " + a);

//The Add() method is called and passed a value type(int) by reference

//by using the ref parameter

ob.Add(ref a);

//Printing the value of local variable,a

//After calling the Add() method

Console.WriteLine("Afrer calling the Add() method, the value in a is : " + a);

Console.ReadLine();

}

}

public class A

{

public void Add(ref int i)

{

a = i + 10;

}

}

**Swap Two Number Using ref**

class Program

{

static void Main(string[] args)

{

int a = 100;

int b = 500;

Console.WriteLine("Value of a and b before sawapping");

Console.WriteLine();

Console.WriteLine("a=" + " " + a);

Console.WriteLine("b=" + " " + b);

SwapNum(ref a, ref b);

Console.WriteLine();

Console.WriteLine("Value of a and b after sawapping");

Console.WriteLine();

Console.WriteLine("a=" + " " + a);

Console.WriteLine("b=" + " " + b);

Console.ReadLine();

}

static void SwapNum(ref int x, ref int y)

{

int tempswap = x;

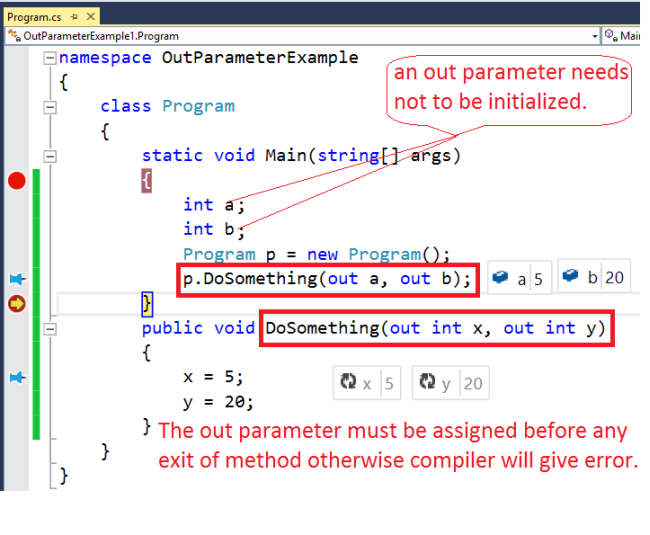
x = y;

y = tempswap;

}

}

**Out Parameters**An out param is similar to **ref param**. But any argument passed as out parameter **need not to be initialized** whereas in case of ref it must be initialized. Another difference is that ref parameter may be assigned inside the method in which it is called (not necessary), but in case of “out” it must be assigned in the calling method before any exit.



**Dynamic parameter (dynamic keyword):**A parameter can also be passed dynamically like the following code snippet.

n **C# 4.0**, a new type of parameters is introduced that is known as a dynamic parameter. Here the 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.

class Program

{

static void Main(string[] args)

{

mulval(30);//output 900

Console.ReadKey();

}

public static void mulval(dynamic val)

{

val \*= val;

Console.WriteLine(val);

}

}

**Value parameter or Passing Value Types by Value (normal C# method param are value parameter):**  
Normal C# method param are value parameter. It is also known as “passing value types by value” or we can say that when a variable is passed as value type then it contains its data directly not the reference. If any changes made in the value type parameter it will not reflect the original data stored as argument. If we need to modify original data we can pass it using 'ref' or 'out' keyword. e.g.

class Program

{

static void Main(string[] args)

{

int a = 10;

int b = 25;

int res = Add(a, b);

Console.ReadKey();

}

public static int Add(int x, int y)

{

return x = y;

}

}

**Params (params)**The ‘param’ keyword can be used to specify a method parameter that takes a variable number of arguments. When we use ‘param’ keyword to pass variable number of arguments we can send any number of arguments or no arguments. But all the arguments passed should be of same type.  
  
If a method is using param keyword it cannot have any other parameter i.e. no additional parameter allowed.

class Program

{

static void Main(string[] args)

{

string my1stLine = "This is my first Line";

string my2ndLine = "This is my second Line";

string my3rdline = "This is my Third Line";

string my4thline = "This is my fourth Line";

PrintPassedList(my1stLine, my2ndLine, my3rdline, my4thline);

Console.ReadKey();

}

public static void PrintPassedList(params string[] listtoPrint)

{

foreach (var item in listtoPrint)

{

Console. WriteLine(item);

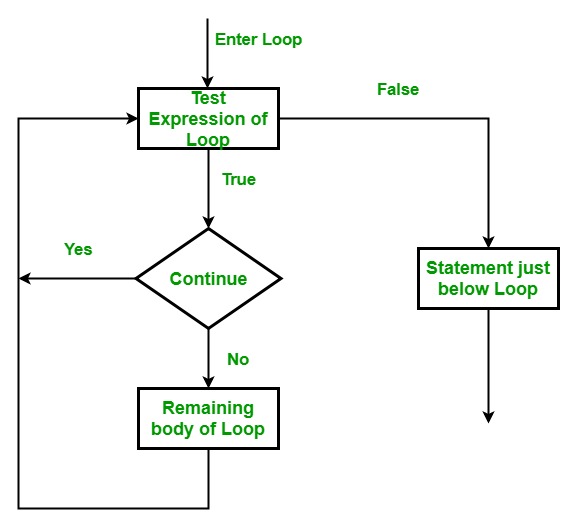
}

}

}

**Continue statement**

This statement is used to skip over the execution part of the loop on a certain condition. After that, it transfers the control to the beginning of the loop. Basically, it skips its following statements and continues with the next iteration of the loop.



static void Main(string[] args)

{

for (int i = 1; i <= 10; i++)

{

// if the value of i becomes 4 then

// it will skip 4 and send the

// transfer to the for loop and

// continue with 5

if (i == 4)

continue;

Console.WriteLine(i);

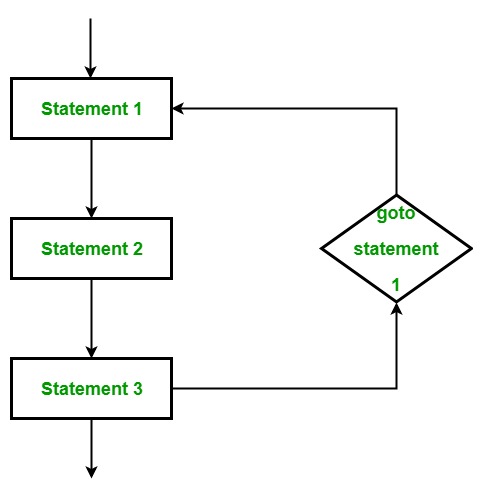
}

Console.ReadLine();

}

**goto statement**

This statement is used to transfer control to the labeled statement in the program.The label is the valid identifier and placed just before the statement from where the control is transferred.



static void Main(string[] args)

{

int number = 20;

switch (number)

{

case 5:

Console.WriteLine("case 5");

break;

case 10:

Console.WriteLine("case 10");

break;

case 20:

Console.WriteLine("case 20");

// goto statement transfer

// the control to case 5

goto case 5;

default:

Console.WriteLine("No match found");

break;

}

Console.ReadLine();

}

**return statement**

This statement terminates the execution of the method and returns the control to the calling method. It returns an optional value.If the type of method is void, then the return statement can be excluded.

class Program

{

static void Main(string[] args)

{

int number = 2;

// calling addition function

int result = Addition(number);

Console.WriteLine("The addition is {0}", result);

Console.ReadLine();

}

static int Addition(int a)

{

// add two value and

// return the result of addition

int add = a + a;

// using return statement

return add;

}

}