

C# BASIC TUTORIAL (2)

GVC Ths. Nguyễn Minh Đạo

BM CNPM - Khoa CNTT - ĐHSPKT TP.HCM



OBJECTIVES

- C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg.
- This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.



C# - ENCAPSULATION

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.

OVERVIEW

- **Abstraction** and **encapsulation** are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.
- **Encapsulation** is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member.

PUBLIC ACCESS SPECIFIER

- C# supports the following access specifiers:
 - Public
 - Private
 - Protected
 - Internal
 - Protected internal
- Public access specifier allows a class to expose its member variables and member functions to other functions and objects.
- Any public member can be accessed from outside the class.

PUBLIC ACCESS SPECIFIER

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        public double length;
        public double width;

        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
}
//end class Rectangle
```

```
class ExecuteRectangle
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();
        r.length = 4.5;
        r.width = 3.5;
        r.Display();
        Console.ReadLine();
    }
}
```

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

```
Length: 4.5
Width: 3.5
Area: 15.75
```

PUBLIC ACCESS SPECIFIER

- In the preceding example, the member variables `length` and `width` are declared **public**, so they can be accessed from the function **Main()** using an instance of the `Rectangle` class, named `r`.
- The member function **Display()** and **GetArea()** can also access these variables directly without using any instance of the class.
- The member functions **Display()** is also declared **public**, so it can also be accessed from **Main()** using an instance of the `Rectangle` class, named `r`.

PRIVATE ACCESS SPECIFIER

- Private access specifier allows a class to hide its member variables and member functions from other functions and objects.
- Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.


```
using System;
```

```
namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        private double length;
        private double width;

        public void Acceptdetails()
        {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
}
//end class Rectangle
```

PRIVATE ACCESS SPECIFIER

```
class ExecuteRectangle
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
        r.Display();
        Console.ReadLine();
    }
}
```

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

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

PRIVATE ACCESS SPECIFIER

- In the preceding example, the member variables **length** and **width** are declared **private**, so they cannot be accessed from the function **Main()**.
- The member functions ***AcceptDetails()*** and ***Display()*** can access these variables. Since the member functions ***AcceptDetails()*** and ***Display()*** are declared **public**, they can be accessed from ***Main()*** using an instance of the Rectangle class, named **r**.

PROTECTED ACCESS SPECIFIER

- Protected access specifier allows a child class to access the member variables and member functions of its base class.
- This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

INTERNAL ACCESS SPECIFIER

- Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly.
- In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

INTERNAL ACCESS SPECIFIER

```
using System;
```

```
namespace RectangleApplication  
{
```

```
    class Rectangle
```

```
    {
```

```
        //member variables
```

```
        internal double length;
```

```
        internal double width;
```

```
        double GetArea()  
        {
```

```
            return length * width;  
        }
```

```
        public void Display()  
        {
```

```
            Console.WriteLine("Length: {0}", length);
```

```
            Console.WriteLine("Width: {0}", width);
```

```
            Console.WriteLine("Area: {0}", GetArea());  
        }
```

```
    }  
} //end class Rectangle
```

```
class ExecuteRectangle
```

```
{
```

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

```
    {
```

```
        Rectangle r = new Rectangle();
```

```
        r.length = 4.5;
```

```
        r.width = 3.5;
```

```
        r.Display();
```

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

```
}
```

```
}
```

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

```
Length: 4.5
```

```
Width: 3.5
```

```
Area: 15.75
```

INTERNAL ACCESS SPECIFIER

- In the preceding example, notice that the member function ***GetArea()*** is not declared with any access specifier.
- Then what would be the default access specifier of a class member if we don't mention any? It is **private**.

PROTECTED INTERNAL ACCESS SPECIFIER

- The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.



C# - METHODS

A method is a group of statements that together perform a task.

C# - METHODS

- A method is a group of statements that together perform a task. Every C# program has at least one class with a method named **Main**.
- To use a method, you need to:
 - Define the method
 - Call the method

DEFINING METHODS IN C#

- When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)  
{  
    Method Body  
}
```

DEFINING METHODS IN C#

- Following are the various elements of a method:
 - **Access Specifier:** This determines the visibility of a variable or a method from another class.
 - **Return type:** A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
 - **Method name:** Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
 - **Parameter list:** Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
 - **Method body:** This contains the set of instructions needed to complete the required activity.

DEFINING METHODS IN C#

- Following code snippet shows a function ***FindMax*** that takes two integer values and returns the larger of the two.
- It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator
{
    public int FindMax(int num1, int num2)
    {
        /* local variable declaration */
        int result;

        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
    ...
}
```

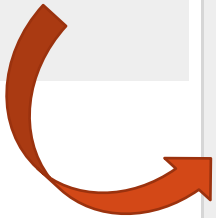
CALLING METHODS IN C#

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }
}
```



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

    //calling the FindMax method
    ret = n.FindMax(a, b);
    Console.WriteLine("Max value is : {0}", ret );
    Console.ReadLine();
}
}
```

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

```
Max value is : 200
```

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }
}
```

CALLING METHODS IN C#

```
class Test
{
    static void Main(string[] args)
    {
        /* local variable definition */
        int a = 100;
        int b = 200;
        int ret;
        NumberManipulator n = new NumberManipulator();
        //calling the FindMax method
        ret = n.FindMax(a, b);
        Console.WriteLine("Max value is : {0}", ret );
        Console.ReadLine();
    }
}
```

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

```
Max value is : 200
```

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int factorial(int num)
        {
            /* local variable declaration */
            int result;

            if (num == 1)
            {
                return 1;
            }
            else
            {
                result = factorial(num - 1) * num;
                return result;
            }
        }
    }
}
```

RECURSIVE METHOD CALL

```
static void Main(string[] args)
{
    NumberManipulator n = new NumberManipulator();
    //calling the factorial method
    Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));
    Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));
    Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));
    Console.ReadLine();
}
}
```

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

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

PASSING PARAMETERS TO A METHOD

- When method with parameters is called, you need to pass the parameters to the method. In C#, there are three ways that parameters can be passed to a method:

Mechanism	Description
Value parameters	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Reference parameters	This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
Output parameters	This method helps in returning more than one value.

C# - PASSING PARAMETERS BY VALUE

- This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter.
- The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument.

```
using System;
```

```
namespace CalculatorApplication
```

```
{
```

```
    class NumberManipulator
```

```
    {
```

```
        public 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)
```

```
        {
```

```
            NumberManipulator n = new NumberManipulator();
```

```
            /* local variable definition */
```

```
            int a = 100;
```

```
            int b = 200;
```



C# - PASSING PARAMETERS BY VALUE

```
Console.WriteLine("Before swap, value of a : {0}", a);  
Console.WriteLine("Before swap, value of b : {0}", b);
```

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

```
Console.WriteLine("After swap, value of a : {0}", a);  
Console.WriteLine("After swap, value of b : {0}", b);
```

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

```
}
```

```
}
```

```
}
```

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

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

C# - PASSING PARAMETERS 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.
- In C#, you declare the reference parameters using the **ref** keyword.

C# - PASSING PARAMETERS BY REFERENCE

```
using System;
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public 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)
        {
            NumberManipulator n = new NumberManipulator();
            /* local variable definition */
            int a = 100;
            int b = 200;

            Console.WriteLine("Before swap, value of a : {0}", a);
            Console.WriteLine("Before swap, value of b : {0}", b);
```

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

        Console.WriteLine("After swap, value of a : {0}", a);
        Console.WriteLine("After swap, value of b : {0}", b);

        Console.ReadLine();
    }
}
```

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

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

C# - PASSING PARAMETERS BY **OUTPUT**


- A return statement can be used for returning only one value from a function. However, using **output parameters**, you can return two values from a function.
- Output parameters are like reference parameters, except that they transfer data out of the method rather than into it.

C# - PASSING PARAMETERS BY OUTPUT

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void getValue(out int x )
        {
            int temp = 5;
            x = temp;
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* local variable definition */
            int a = 100;
```



```
        Console.WriteLine("Before method call, value of a : {0}", a);

        /* calling a function to get the value */
        n.getValue(out a);

        Console.WriteLine("After method call, value of a : {0}", a);
        Console.ReadLine();
    }
}
```

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

```
Before method call, value of a : 100
After method call, value of a : 5
```

C# - PASSING PARAMETERS BY OUTPUT

- The variable supplied for the output parameter need not be assigned a value the method call.
- Output parameters are particularly useful when you need to return values from a method through the parameters without assigning an initial value to the parameter.

C# - PASSING PARAMETERS BY OUTPUT


```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void getValues(out int x, out int y )
        {
            Console.WriteLine("Enter the first value: ");
            x = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Enter the second value: ");
            y = Convert.ToInt32(Console.ReadLine());
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* local variable definition */
            int a , b;
        }
    }
}
```

```
/* calling a function to get the values */
n.getValues(out a, out b);
```

```
Console.WriteLine("After method call, value of a : {0}", a);
Console.WriteLine("After method call, value of b : {0}", b);
Console.ReadLine();
```



When the above code is compiled and executed, it produces the following result (depending upon the user input):

```
Enter the first value:
7
Enter the second value:
8
After method call, value of a : 7
After method call, value of b : 8
```




C# - NULLABLES

C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.

C# - NULLABLES

- C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.
- For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a **Nullable< Int32 >** variable.
- Similarly, you can assign true, false or null in a **Nullable< bool >** variable.
- Syntax for declaring a nullable type is as follows:
 - **< data_type> ? <variable_name> = null;**

The following example demonstrates use of nullable data types:

C# - NULLABLES

```
using System;
namespace CalculatorApplication
{
    class NullablesAtShow
    {
        static void Main(string[] args)
        {
            int? num1 = null;
            int? num2 = 45;
            double? num3 = new double?();
            double? num4 = 3.14157;

            bool? boolval = new bool?();
```

```
// display the values
```

```
Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}",
                  num1, num2, num3, num4);
Console.WriteLine("A Nullable boolean value: {0}", boolval);
Console.ReadLine();
```

```
    }
}
```

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

```
Nullables at Show: , 45, , 3.14157
A Nullable boolean value:
```

THE NULL COALESCING OPERATOR (??)

- The null coalescing operator is used with the **nullable** value types and reference types. It is used for converting an operand to the type of another **nullable** (or not) value type operand, where an implicit conversion is possible.
- If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand.

```
using System;
namespace CalculatorApplication
{
    class NullablesAtShow
    {
        static void Main(string[] args)
        {
            double? num1 = null;
            double? num2 = 3.14157;
            double num3;
            num3 = num1 ?? 5.34;
            Console.WriteLine(" Value of num3: {0}", num3);
            num3 = num2 ?? 5.34;
            Console.WriteLine(" Value of num3: {0}", num3);
            Console.ReadLine();
        }
    }
}
```

THE NULL COALESCING OPERATOR (??)

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

```
Value of num3: 5.34
Value of num3: 3.14157
```



C# - ARRAYS

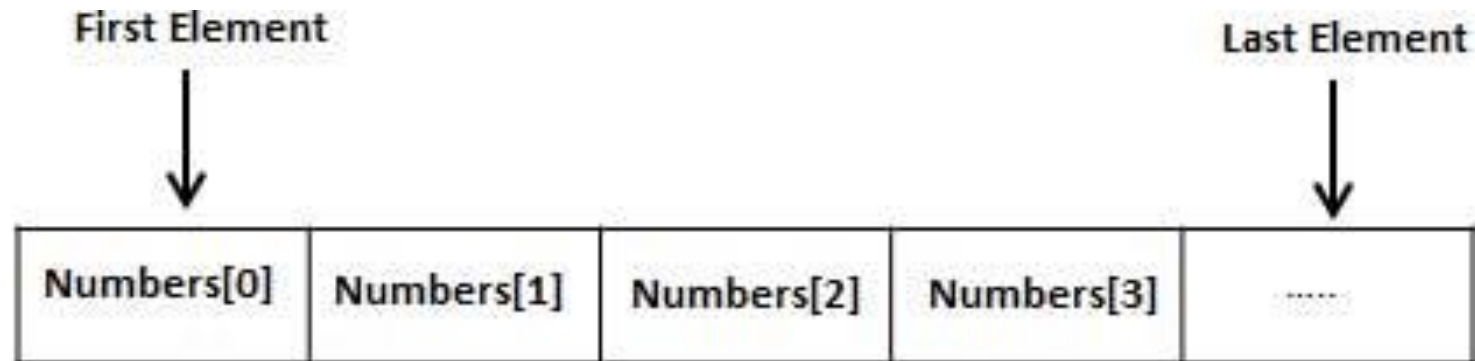
An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

C# - ARRAYS

- 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.

C# - ARRAYS

- 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 to be stored 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, like:
 - **double[] balance = { 2340.0, 4523.69, 3421.0};**

ASSIGNING VALUES TO AN ARRAY

- You can also create and initialize an array, like:
 - **int [] marks = new int[5] { 99, 98, 92, 97, 95};**
- In the preceding case, you may also omit the size of the array, like:
 - **int [] marks = new int[] { 99, 98, 92, 97, 95};**

ASSIGNING VALUES TO AN ARRAY

- You can also copy an array variable into another target array variable. In that case, both the target and source would 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 would be 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];**

ACCESSING ARRAY ELEMENTS

```
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 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;
            }
        }
    }
}
```

USING THE *FOREACH* LOOP



```
/* output each array element's value */
foreach (int j in n )
{
    int i = j-100;
    Console.WriteLine("Element[{0}] = {1}", i, j);
    i++;
}
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
```


C# ARRAYS IN DETAIL

- Arrays are important to C# and should need lots of more detail.
- There are following few important concepts related to array which should be clear to a C# programmer:

Concept	Description
Multi-dimensional arrays	C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Jagged arrays	C# supports multidimensional arrays, which are arrays of arrays.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Param arrays	This is used for passing unknown number of parameters to a function.
The Array Class	Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

C# - MULTIDIMENSIONAL ARRAYS

- C# allows multidimensional arrays. Multidimensional 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:
 - **int [, ,] m;**

TWO-DIMENSIONAL ARRAYS

- The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is, in essence, a list of one-dimensional arrays.
- A 2-dimensional array can be thought of as a table, which will have x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

INITIALIZING TWO-DIMENSIONAL ARRAYS

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

```
int [,] a = 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


```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        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();
        }
    }
}
```

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



```
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
```

C# - JAGGED ARRAYS

- A Jagged array is an array of arrays. You can declare a jagged array scores of int values 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}};
```

```
using System;
```

```
namespace ArrayApplication
```

```
{
```

```
    class MyArray
```

```
    {
```

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

```
        {
```

```
            /* a jagged array of 5 array of integers*/
```

```
            int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},
```

```
            new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 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();
```


```
        }
```

```
    }
```

```
}
```

C# - JAGGED ARRAYS

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



```
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
```


C# - PASSING ARRAYS AS FUNCTION ARGUMENTS

```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        double getAverage(int[] arr, int size)
        {
            int i;
            double avg;
            int sum = 0;

            for (i = 0; i < size; ++i)
            {
                sum += arr[i];
            }

            avg = (double)sum / size;
            return avg;
        }
    }
}
```



```
static void Main(string[] args)
{
    MyArray app = new MyArray();
    /* an int array with 5 elements */
    int [] balance = new int[]{1000, 2, 3, 17, 50};
    double avg;

    /* pass pointer to the array as an argument */
    avg = app.getAverage(balance, 5 );

    /* output the returned value */
    Console.WriteLine( "Average value is: {0} ", avg );
    Console.ReadKey();
}
}
```


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

```
Average value is: 214.4
```


C# - PARAM ARRAYS

```
using System;

namespace ArrayApplication
{
    class ParamArray
    {
        public int AddElements(params int[] arr)
        {
            int sum = 0;
            foreach (int i in arr)
            {
                sum += i;
            }
            return sum;
        }
    }
}
```



```
class TestClass
{
    static void Main(string[] args)
    {
        ParamArray app = new ParamArray();
        int sum = app.AddElements(512, 720, 250, 567, 889);
        Console.WriteLine("The sum is: {0}", sum);
        Console.ReadKey();
    }
}
```

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

```
The sum is: 2938
```



C# - ARRAY CLASS

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

PROPERTIES OF THE ARRAY CLASS

The following table provides some of the most commonly used properties of the Array class:

S.N	Property Name & Description
1	IsFixedSize Gets a value indicating whether the Array has a fixed size.
2	IsReadOnly Gets a value indicating whether the Array is read-only.
3	Length Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	LongLength Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	Rank Gets the rank (number of dimensions) of the Array.

METHODS OF THE ARRAY CLASS

The following table provides some of the most commonly used methods of the Array class:

S.N	Method Name & Description
1	Clear Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	Copy(Array, Array, Int32) 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.
3	CopyTo(Array, Int32) Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	GetLength Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	GetLongLength Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.

METHODS OF THE ARRAY CLASS

The following table provides some of the most commonly used methods of the Array class:

S.N	Method Name & Description
6	GetLowerBound Gets the lower bound of the specified dimension in the Array.
7	GetType Gets the Type of the current instance. (Inherited from Object.)
8	GetUpperBound Gets the upper bound of the specified dimension in the Array.
9	GetValue(Int32) Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
10	IndexOf(Array, Object) Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.

METHODS OF THE ARRAY CLASS

The following table provides some of the most commonly used methods of the Array class:

S.N	Method Name & Description
11	Reverse(Array) Reverses the sequence of the elements in the entire one-dimensional Array.
12	SetValue(Object, Int32) Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
13	Sort(Array) Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.
14	ToStringk Returns a string that represents the current object. (Inherited from Object.)

METHODS OF THE ARRAY CLASS

```
using System;
namespace ArrayApplication
{
    class MyArray
    {

        static void Main(string[] args)
        {
            int[] list = { 34, 72, 13, 44, 25, 30, 10 };
            int[] temp = list;

            Console.Write("Original Array: ");
            foreach (int i in list)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();

            // reverse the array
            Array.Reverse(temp);
            Console.Write("Reversed Array: ");
            foreach (int i in temp)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
        }
    }
}
```

```
//sort the array
Array.Sort(list);
Console.Write("Sorted Array: ");
foreach (int i in list)
{
    Console.Write(i + " ");
}
Console.WriteLine();

Console.ReadKey();
}
```

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

```
Original Array: 34 72 13 44 25 30 10
Reversed Array: 10 30 25 44 13 72 34
Sorted Array: 10 13 25 30 34 44 72
```



