School of Computer science &

Information Technology

Department of Master of Computer Applications C# & .NET Technologies – 18MCA304 Unit II – Methods, Arrays and OOP’s

Kathiresan.V Assistant Professor

School of Computer Science & IT JGI Knowledge Campus Jayanagar 9th block

Bengaluru - 69

#### Mail Id:

[**v.Kathiresan@jainuniversity.ac.in**](mailto:v.Kathiresan@jainuniversity.ac.in)

**Class**

Classes are the user defined data types that represent the **state** and **behaviour** of an object. State represents the properties and **behavior** is the action that objects can perform.

Classes can be declared using the following access specifiers that limit the accessibility of classes to other classes, however some classes does not require any access modifiers.

1. Public
2. Private
3. Protected
4. Internal
5. Protected internal For example:

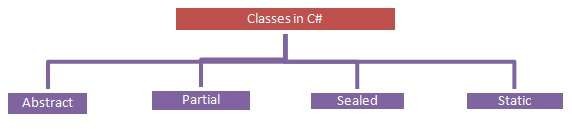
public class Accounts

{ }

#### Some Key points about classes

* + Classes are reference types that hold the object created dynamically in a heap.
  + All classes have a base type of **System.Object**.
  + The default access modifier of a class is **Internal**.
  + The default access modifier of methods and variables is **Private**.
  + Directly inside the namespaces declarations of private classes are not allowed.

The following are types of classes in C#:



#### Abstract class

An Abstract class is a class that provides a common definition to the subclasses and this is the type of class whose object is not created.

Some key points of Abstract classes are:

* + Abstract classes are declared using the abstract keyword.
  + We cannot create an object of an abstract class.
  + If you want to use it then it must be inherited in a subclass.
  + An Abstract class contains both abstract and non-abstract methods.
  + The methods inside the abstract class can either have an implementation or no implementation.
  + We can inherit two abstract classes; in this case the base class method implementation is optional.
  + An Abstract class has only one subclass.
  + Methods inside the abstract class cannot be private.
  + If there is at least one method abstract in a class then the class must be abstract.

For example:

abstract class Accounts

{ }

#### Partial Classes

It is a type of class that allows dividing their properties, methods and events into multiple source files and at compile time these files are combined into a single class.

The following are some key points:

* + All the parts of the partial class must be prefixed with the partial keyword.
  + If you seal a specific part of a partial class then the entire class is sealed, the same as for an abstract class.
  + Inheritance cannot be applied on partial classes.
  + The classes that are written in two class files are combined together at run time.

For example:

partial class Accounts

{ }

#### Sealed Class

A Sealed class is a class that cannot be inherited and used to restrict the properties. The following are some key points:

* + A Sealed class is created using the sealed keyword.
  + Access modifiers are not applied to a sealed class.
  + To access the sealed members we must create an object of the class.

For example:

sealed class Accounts

{ }

#### Static Class

It is the type of class that cannot be instantiated, in other words we cannot create an object of that class using the new keyword, such that class members can be called directly using their class name.

The following are some key points:

* + Created using the static keyword.
  + Inside a static class only static members are allowed, in other words everything inside the static class must be static.
  + We cannot create an object of the static class.
  + A Static class cannot be inherited.
  + It allows only a static constructor to be declared.
  + The methods of the static class can be called using the class name without creating the instance.

For example:

static class Accounts

{ }

# Objects

* Classes are special kinds of templates from which you can create objects.
* Each object contains data and methods to manipulate and access that data.
* The class defines the data and the functionality that each object of that class can contain. Example:

class customer

{

static void Main(string[] args)

{

customer obj = new customer(); obj.displayData();

Console.WriteLine(obj.CustID); Console.WriteLine(obj.Name);

Console.WriteLine(obj.Address);

} }

## Formatting Strings

#### Method 1:

The basic mechanism for formatting is the default implementation of the Object.ToString method, which is discussed in the Default Formatting Using the ToString Method section later in this topic. However, .NET provides several ways to modify and extend its default formatting support.

Defining format specifiers that enable the string representation of an object’s value to take multiple forms. For example, the "X" format specifier in the following statement converts an integer to the string representation of a hexadecimal value.

int integerValue = 60312; Console.WriteLine(integerValue.ToString("X")); // Displays EB98. **Method 2:**

Following on from the format string is a comma separated list of arguments. There must be an argument for each of the place holders.

Example:

string newString;

newString = String.Format("There are {0} cats in my {1} and no {2}", 2, "house", "dogs"); System.Console.WriteLine (newString);

When run, the above code will result in the following output:

*There are 2 cats in my house and no dogs*

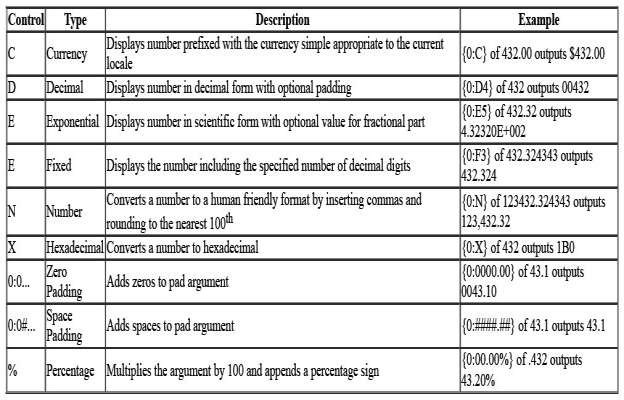
As Simple Format Control Example

The following example uses the X format control to display a number in Hexadecimal format:

newString = String.Format("The number {0} in Hexadecimal is {0:X}", 432); System.Console.WriteLine (newString);

The above example displays argument 0 (the number 432) in two formats. The first is the default decimal format and the second uses the X format control to display the argument as a hexadecimal number.

The String.Format() method provides a wide range of format controls which are outlined in the following section.



## Constants

* Constants are immutable values which are known at compile time and do not change for the life of the program.
* Constants are declared with the const modifier.
* Only the C# built-in types (excluding System.Object) may be declared as const .
* User-defined types, including classes, structs, and arrays, cannot be const .
* Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.
* C# does not support const methods, properties, or events. Example 1:

class Calendar1

{

public const int months = 12;

}

Example 2:

class Calendar2

{

const int months = 12, weeks = 52, days = 365;

}

Example 3:

class Calendar3

{

const int months = 12; const int weeks = 52; const int days = 365;

const double daysPerWeek = (double) days / (double) weeks; const double daysPerMonth = (double) days / (double) months;

}

Constants can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the constant.

#### C# Access Modifiers , C# Access Specifiers

Access Modifiers ( **Access Specifiers** ) describes as the scope of accessibility of an Object and its members. All C# types and type members have an **accessibility level** . We can control the scope of the member object of a class using access specifiers. We are using access modifiers for providing security of our applications. When we specify the accessibility of a type or member we have to declare it by using any of the **access modifiers** provided by C# language.

C# provide six access modifiers , they are as follows :

* private
* public
* protected
* internal
* protected internal
* Private Protected (C# version 7.2 and later)

#### Private Access Modifier

The scope of the accessibility is limited only inside the **classes or struct** in which they are declared. The private members cannot be accessed outside the class and it is the least **permissive access level** .

using System;

namespace ConsoleApplication1

{

class PrivateAccess

{

private string msg = "This variable is private "; private void disp(string msg)

{

Console.WriteLine("This function is private : " + msg);

}

}

class Program

{

static void Main(string[] args)

{

PrivateAccess privateTest = new PrivateAccess(); Console.WriteLine(privateTest.msg);// Cannot Access private variable here privateTest.disp("Hello !!"); // Cannot Access private function here

}

}

}

Output:

Error 1 'ConsoleApplication1.PrivateAccess.msg' is inaccessible due to its protection level Error 2 'ConsoleApplication1.PrivateAccess.disp(string)' is inaccessible due to its protection level

Here we can see the private variable and function in the **PrivateAccess class** cannot access in the main() function because main() function is in a separated class**"Program"** . The scope of the accessibility of private access members limited only inside the classes or struct in which they are

declared. So, the private members **cannot be accessed** outside the class and it is the least permissive access level.

using System;

namespace ConsoleApplication1

{

class Program

{

private string msg = "This variable is private "; private void disp(string msg)

{

Console.WriteLine("This function is private : " + msg);

}

static void Main(string[] args)

{

Program pr = new Program();

Console.WriteLine(pr.msg);// / Accessing private variable inside the class pr.disp("Hello !!"); // Accessing private function inside the class

}

}

}

Output:

This variable is private

This function is private : Hello !!

Here we made a change to the above program that we move the **private variable** and function in the same class. So, we can access the private variable and private function from main() because **private access modifiers** scope is limited only inside the classes or struct in which they are declared.

#### Public Access Modifier

public is the most common **access specifier** in C# . It can be access from anywhere, that means there is no restriction on accessibility. The scope of the **accessibility** is inside class as well as outside. The type or member can be accessed by any other code in the same assembly or another assembly that references it.

using System;

namespace ConsoleApplication1

{

class PublicAccess

{

public string msg = "This variable is public"; public void disp(string msg)

{

Console.WriteLine("This function is public : " + msg);

}

}

class Program

{

static void Main(string[] args)

{

PublicAccess pAccess = new PublicAccess(); Console.WriteLine(pAccess.msg); // Accessing public variable pAccess.disp("Hello !!"); // Accessing public function

}

}

}

Output:

This variable is public

This function is public : Hello !!

Here we can access the variable and function in the **main()** , because the variable and function declared as public.

#### Protected Access Modifier

The scope of **accessibility** is limited within the class or struct and the class derived (Inherited

)from this class.

using System;

namespace ConsoleApplication1

{

class ProtectedAccess

{

protected string msg = "This variable is protected "; protected void disp(string msg)

{

Console.WriteLine("This function is protected : " + msg);

}

}

class Program

{

static void Main(string[] args)

{

ProtectedAccess prAccess = new ProtectedAccess(); Console.WriteLine(prAccess.msg); // Cannot access protected variable here prAccess.disp("Hello !!"); // Cannot access protected function here

}

}

}

Output:

Error 1 'ConsoleApplication1.ProtectedAccess.msg' is inaccessible due to its protection level Error 2 'ConsoleApplication1.ProtectedAccess.disp(string)' is inaccessible due to its protection level

Here we can see the **protected variable** and function in the ProtectedAccess class cannot access in the main() function because main() function is in a separated class **"Program"** . The following program will make you clear that how a protected variable and function can access from inherited calss.

using System;

namespace ConsoleApplication1

{

class ProtectedAccess

{

protected string msg = "This variable is protected "; protected void disp(string msg)

{

Console.WriteLine("This function is protected : " + msg);

}

}

class Program : ProtectedAccess

{

static void Main(string[] args)

{

Program pr = new Program();

Console.WriteLine(pr.msg); // Accessing protected variable pr.disp("Hello !!"); // Accessing protected function

}

}

}

Here we made a change to the above program that the class program is in from class**"ProtectedAccess"** .

class Program : ProtectedAccess

So, we can access the **protected variable** and protected function from main() because Protected access modifiers scope of accessibility is limited within the **class or struct**and the class derived (Inherited )from this class.

#### Internal Access Modifier

The **internal access modifiers** can access within the program that contain its declarations and also access only within files in the same assembly level but not from another assembly.

using System;

namespace ConsoleApplication1

{

class InternalAccess

{

internal string msg = "This variable is internal"; internal void disp(string msg)

{

Console.WriteLine("This function is internal : " + msg);

}

}

class Program

{

static void Main(string[] args)

{

InternalAccess iAccess = new InternalAccess(); Console.WriteLine(iAccess.msg); // Accessing internal variable iAccess.disp("Hello !!"); // Accessing internal function Console.ReadKey();

}

}

}

Output:

This variable is internal

This function is internal : Hello !! Protected Internal Access Modifier

Protected internal is the same access levels of both **protected and internal** . It can access anywhere in the same assembly also be accessed within a derived class in another assembly.

using System;

namespace ConsoleApplication1

{

class InternalAccess

{

protected internal string msg = "This variable is protected internal"; protected internal void disp(string msg)

{

Console.WriteLine("This function is protected internal : " + msg);

}

}

class Program

{

static void Main(string[] args)

{

InternalAccess iAccess = new InternalAccess(); Console.WriteLine(iAccess.msg); // Accessing internal variable iAccess.disp("Hello !!"); // Accessing internal function Console.ReadKey();

}

}

}

Output:

This variable is protected internal

This function is protected internal : Hello !!

## Arrays

#### Introduction

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. The position of the last item on an array will total number of items - 1. So if an array has 10 items, the last 10th item is at 9th position.

In C#, arrays can be declared as fixed length or dynamic.

A *fixed length* array can store a predefined number of items.

A *dynamic array* does not have a predefined size. The size of a *dynamic array* increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined.

*Syntax: int[] intArray;*

*As you can see from the above code snippet, the declaration of an array starts with a type of array followed by a square bracket ([]) and name of the array.*

The following code snippet declares an array that can store 5 items only starting from index 0 to 4.

int[] intArray; intArray = new int[5];

The following code snippet declares an array that can store 100 items starting from index 0 to 99.

int[] intArray;

intArray = new int[100];

#### Defining arrays of different types

In the previous code snippet, we saw how to define a simple array of integer type. Similarly, we can define arrays of any type such as double, character, and string.

In C#, arrays are objects. That means that declaring an array doesn't create an array. After declaring an array, you need to instantiate an array by using the "new" operator.

The following code snippet defines arrays of double, char, bool, and string data types. double[] doubleArray = new double[5];

char[] charArray = new char[5];

bool[] boolArray = new bool[2]; string[] stringArray = new string[10];

#### Initializing Arrays

Once an array is declared, the next step is to initialize an array. The initialization process of an array includes adding actual data to the array.

The following code snippet creates an array of 3 items and values of these items are added when the array is initialized.

// Initialize a fixed array

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

Alternative, we can also add array items one at a time as listed in the following code snippet.

// Initialize a fixed array one item at a time int[] staticIntArray = new int[3]; staticIntArray[0] = 1;

staticIntArray[1] = 3;

staticIntArray[2] = 5;

The following code snippet declares a dynamic array with string values.

// Initialize a dynamic array items during declaration

string[] strArray = new string[] { "Mahesh Chand", "Mike Gold", "Raj Beniwal", "Praveen Kumar", "Dinesh Beniwal" };

#### Accessing Arrays

We can access an array item by passing the item index in the array. The following code snippet creates an array of three items and displays those items on the console.

// Initialize a fixed array one item at a time int[] staticIntArray = new int[3]; staticIntArray[0] = 1;

staticIntArray[1] = 3;

staticIntArray[2] = 5;

// Read array items one by one Console.WriteLine(staticIntArray[0]); Console.WriteLine(staticIntArray[1]);

This method is useful when you know what item you want to access from an array. If you try to pass an item index greater than the items in array, you will get an error.

#### Accessing an array using a foreach Loop

The foreach control statement (loop) is used to iterate through the items of an array. For example, the following code uses foreach loop to read all items of an array of strings.

// Initialize a dynamic array items during declaration

string[] strArray = new string[] { "Mahesh Chand", "Mike Gold", "Raj Beniwal", "Praveen Kumar", "Dinesh Beniwal" };

// Read array items using foreach loop foreach(string str in strArray)

{

Console.WriteLine(str);

}

This approach is used when you do not know the exact index of an item in an array and needs to loop through all the items.

#### Array Types

Arrays can be divided into the following four categories.

* + Single-dimensional arrays
  + Multidimensional arrays or rectangular arrays
  + Jagged arrays
  + Mixed arrays.

#### Single Dimension Arrays

Single-dimensional arrays are the simplest form of arrays. These types of arrays are used to store number of items of a predefined type. All items in a single dimension array are stored contiguously starting from 0 to the size of the array -1.

The following code declares an integer array that can store 3 items. As you can see from the code, first I declare the array using [] bracket and after that I instantiate the array by calling the new operator.

int[] intArray; intArray = new int[3];

Array declarations in C# are pretty simple. You put array items in curly braces ({}). If an array is not initialized, its items are automatically initialized to the default initial value for the array type if the array is not initialized at the time it is declared.

The following code declares and initializes an array of three items of integer type.

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

The following code declares and initializes an array of 5 string items.

string[] strArray = new string[5] { "Mahesh", "Mike", "Raj", "Praveen", "Dinesh" }; You can even directly assign these values without using the new operator.

string[] strArray = { "Mahesh", "Mike", "Raj", "Praveen", "Dinesh" }; You can initialize a dynamic length array as follows:

string[] strArray = new string[] { "Mahesh", "Mike", "Raj", "Praveen", "Dinesh" };

#### Multi-Dimensional Arrays

A multi-dimensional array, also known as a rectangular array is an array with more than one dimension. The form of a multi-dimensional array is a matrix.

#### Declaring a multi-dimensional array

A multi dimension array is declared as following: string[,] mutliDimStringArray;

A multi-dimensional array can be fixed-sized or dynamic sized.

#### Initializing multi-dimensional arrays

The following code snippet is an example of fixed-sized multi-dimensional arrays that defines two multi dimension arrays with a matrix of 3x2 and 2x2. The first array can store 6 items and second array can store 4 items. Both of these arrays are initialized during the declaration.

int[,] numbers = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };

string[,] names = new string[2, 2] { { "Rosy", "Amy" }, { "Peter", "Albert" } };

Now let's see examples of multi-dimensional dynamic arrays where you are not sure of the number of items of the array. The following code snippet creates two multi-dimensional arrays with no limit.

int[,] numbers = new int[, ] { { 1, 2 }, { 3, 4 }, { 5, 6 } };

string[,] names = new string[, ] { { "Rosy", "Amy" }, { "Peter", "Albert" } };

You can also omit the new operator as we did in single dimension arrays. You can assign these values directly without using the new operator. For example:

int[, ] numbers = { {1,2},{3,4},{5,6}};

string[, ] names = { {"Rosy","Amy"},{"Peter","Albert"} };

We can also initialize the array items one item at a time. The following code snippet is an example of initializing array items one at a time.

int[, ] numbers = new int[3, 2]; numbers[0, 0] = 1;

numbers[1, 0] = 2;

numbers[2, 0] = 3;

numbers[0, 1] = 4;

numbers[1, 1] = 5;

#### Accessing multi-dimensional arrays

A multi-dimensional array items are represented in a matrix format and to access it's items, we need to specify the matrix dimension. For example, item(1,2) represents an array item in the matrix at second row and third column.

The following code snippet shows how to access numbers array defined in the above code. Console.WriteLine(numbers[0, 0]);

Console.WriteLine(numbers[0, 1]);

Console.WriteLine(numbers[1, 0]);

Console.WriteLine(numbers[1, 1]);

Console.WriteLine(numbers[2, 0]);

Console.WriteLine(numbers[2, 2]);

**Jagged Arrays**

Jagged arrays are arrays of arrays. The elements of a jagged array are other arrays.

#### Declaring Jagged Arrays

Declaration of a jagged array involves two brackets. For example, the following code snippet declares a jagged array that has three items of an array.

*int[][] intJaggedArray = new int[3][];*

The following code snippet declares a jagged array that has two items of an array.

*string[][] stringJaggedArray = new string[2][];*

#### Initializing Jagged Arrays

Before a jagged array can be used, its items must be initialized. The following code snippet initializes a jagged array; the first item with an array of integers that has two integers, second item with an array of integers that has 4 integers, and a third item with an array of integers that has 6 integers.

// Initializing jagged arrays intJaggedArray[0] = new int[2]; intJaggedArray[1] = new int[4]; intJaggedArray[2] = new int[6];

We can also initialize a jagged array's items by providing the values of the array's items. The following code snippet initializes item an array's items directly during the declaration.

// Initializing jagged arrays

Int JaggedArray[0] = new int[2] {2,12};

Int JaggedArray[1] = new int[4] {4,14,24,34};

Int JaggedArray[2] = new int[6] {6,16,26,36,46,56};

#### Accessing Jagged Arrays

We can access a jagged array's items individually in the following way:

Console.Write(intJaggedArray3[0][0]); Console.WriteLine(intJaggedArray3[2][5]);

We can also loop through all of the items of a jagged array. The Length property of an array helps a lot; it gives us the number of items in an array. The following code snippet loops through all of the items of a jagged array and displays them on the screen.

// Loop through all itesm of a jagged array

for (int i = 0; i < intJaggedArray3.Length; i++) { System.Console.Write("Element({0}): ", i);

for (int j = 0; j < intJaggedArray3[i].Length; j++) {

System.Console.Write("{0}{1}", intJaggedArray3[i][j], j == (intJaggedArray3[i].Length - 1) ? ""

: " ");

}

System.Console.WriteLine();

}

### Mixed Arrays

Mixed arrays are a combination of multi-dimension arrays and jagged arrays. The mixed arrays type is removed from .NET 4.0. I have not really seen any use of mixed arrays. You can do anything you want with the help of multi-dimensional and jagged arrays.

#### A Simple Example

Here is a complete example listed in Listing 1 that demonstrates how to declare all kinds of arrays then initialize them and access them.

To test this code, create a console application using Visual Studio 2010 or Visual C# Express and copy and paste this code.

Console.WriteLine("Single Dimension Array Sample");

// Single dim array

string[] strArray = new string[] { "Mahesh Chand", "Mike Gold", "Raj Beniwal", "Praveen Kumar", "Dinesh Beniwal" };

// Read array items using foreach loop foreach(string str in strArray)

{

Console.WriteLine(str);

}

Console.WriteLine(" ");

Console.WriteLine("Multi-Dimension Array Sample");

string[, ] string2DArray = new string[2, 2] { { "Rosy", "Amy" }, { "Peter", "Albert" }}; foreach(string str in string2DArray)

{

Console.WriteLine(str);

}

Console.WriteLine(" ");

Console.WriteLine("Jagged Array Sample"); int[][] intJaggedArray3 = {

new int[] { 2, 12 },

new int[] { 14, 14, 24, 34 },

new int[] { 6,

16,

26,

36,

46,

56

}

};

// Loop through all itesm of a jagged array

for (int i = 0; i < intJaggedArray3.Length; i++) { Console.Write("Element({0}): ", i);

for (int j = 0; j < intJaggedArray3[i].Length; j++) {

Console.Write("{0}{1}", intJaggedArray3[i][j], j == (intJaggedArray3[i].Length - 1) ? "" : " ");

}

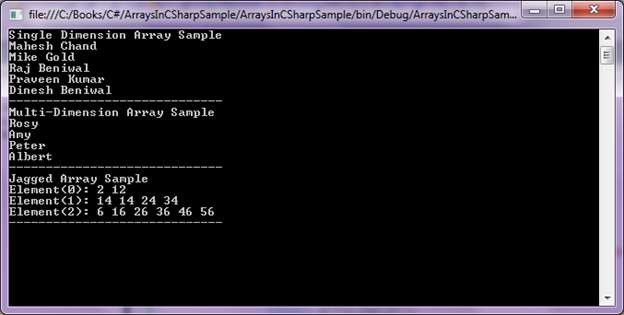
Console.WriteLine();

}

Console.WriteLine(" ");

*Listing 1*

The output of Listing 1 looks like Figure 1.



### Array Class

Array class is the mother of all arrays and provides functionality for creating, manipulating, searching, and sorting arrays in .NET Framework.

**Array** class, defined in the System namespace, is the base class for arrays in C#. Array class is an abstract base class that means we cannot create an instance of the Array class.

#### Creating an Array

Array class provides the CreateInstance method to construct an array. The CreateInstance method takes first parameter as the type of items and second and third parameters are the

dimension and their range. Once an array is created, we use SetValue method to add items to an array.

The following code snippet creates an array and adds three items to the array. As you can see the type of the array items is string and range is 3. You will get an error message if you try to add 4th item to the array.

Array stringArray = Array.CreateInstance(typeof(String), 3); stringArray.SetValue("Mahesh Chand", 0);

stringArray.SetValue("Raj Kumar", 1);

stringArray.SetValue("Neel Beniwal", 2);

*Note: Calling SetValue on an existing item of an array overrides the previous item value with the new value.*

The code snippet in Listing 2 creates a multi-dimensional array. Array intArray3D = Array.CreateInstance(typeof(Int32), 2, 3, 4);

for (int i = intArray3D.GetLowerBound(0); i <= intArray3D.GetUpperBound(0); i++)

for (int j = intArray3D.GetLowerBound(1); j <= intArray3D.GetUpperBound(1); j++) for (int k = intArray3D.GetLowerBound(2); k <= intArray3D.GetUpperBound(2); k++) { intArray3D.SetValue((i \* 100) + (j \* 10) + k, i, j, k);

}

foreach(int ival in intArray3D) { Console.WriteLine(ival);

}

*Listing 2*

#### Array Properties

Table 1 describes Array class properties.

|  |  |
| --- | --- |
| **IsFixedSize** | **Return a value indicating if an array has a fixed size or not.** |
| IsReadOnly | Returns a value indicating if an array is read-only or not. |
| LongLength | Returns a 64-bit integer that represents total number of items in all the dimensions of an array. |

|  |  |
| --- | --- |
| Length | Returns a 32-bit integer that represents the total number of items in all the dimensions of an array. |
| Rank | Returns the number of dimensions of an array. |

*Table 1*

The code snippet in Listing 3 creates an array and uses Array properties to display property values.

int[] intArray = new int[3] { 0,

1,

2

};

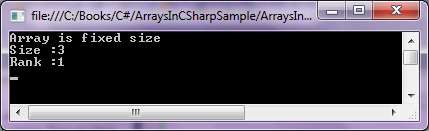
if (intArray.IsFixedSize) { Console.WriteLine("Array is fixed size");

Console.WriteLine("Size :" + intArray.Length.ToString()); Console.WriteLine("Rank :" + intArray.Rank.ToString());

}

*Listing 3*

The output of Listing looks like Figure 2.



*Figure 2*

#### Searching for an Item in an Array

The BinarySearch static method of Array class can be used to search for an item in an array. This method uses the binary search algorithm to search for an item. The method takes at least two parameters. First parameter is the array in which you would like to search and the second parameter is an object that is the item you are looking for. If an item is found in the array, the

method returns the index of that item (based on first item as 0th item). Otherwise method returns a negative value.

#### Note

*You must sort an array before searching. See comments in this article.*

Listing 4 uses BinarySearch method to search an array for a string.

// Create an array and add 5 items to it

Array stringArray = Array.CreateInstance(typeof(String), 5); stringArray.SetValue("Mahesh", 0);

stringArray.SetValue("Raj", 1);

stringArray.SetValue("Neel", 2);

stringArray.SetValue("Beniwal", 3);

stringArray.SetValue("Chand", 4);

// Find an item

object name = "Neel";

int nameIndex = Array.BinarySearch(stringArray, name);

if (nameIndex >= 0) Console.WriteLine("Item was at " + nameIndex.ToString() + "th position"); else Console.WriteLine("Item not found");

*Listing 4*

#### Sorting Items in an Array

The Sort static method of the Array class can be used to sort array items. This method has many overloaded forms. The simplest form takes as a parameter the array you want to sort. Listing 5 uses the Sort method to sort array items. Using the Sort method, you can also sort a partial list of items.

// Create an array and add 5 items to it

Array stringArray = Array.CreateInstance(typeof(String), 5); stringArray.SetValue("Mahesh", 0);

stringArray.SetValue("Raj", 1);

stringArray.SetValue("Neel", 2);

stringArray.SetValue("Beniwal", 3);

stringArray.SetValue("Chand", 4);

// Find an item

object name = "Neel";

int nameIndex = Array.BinarySearch(stringArray, name);

if (nameIndex >= 0) Console.WriteLine("Item was at " + nameIndex.ToString() + "th position");

else Console.WriteLine("Item not found"); Console.WriteLine(); Console.WriteLine("Original Array"); Console.WriteLine(" ");

foreach(string str in stringArray) { Console.WriteLine(str);

}

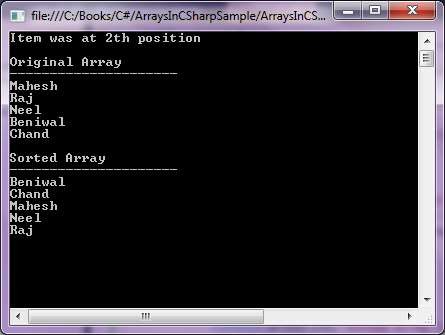
Console.WriteLine(); Console.WriteLine("Sorted Array"); Console.WriteLine(" ");

Array.Sort(stringArray); foreach(string str in stringArray) { Console.WriteLine(str);

}

*Listing 5*

The output of Listing 5 looks like Figure 3.

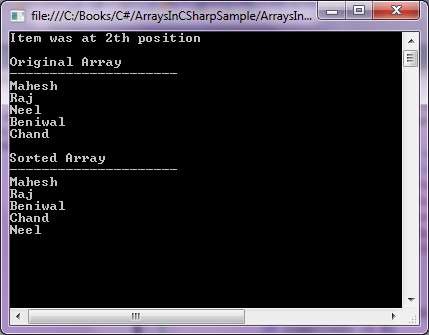


*Figure 3*

Alternatively the Sort method takes starting index and number of items after that index. The following code snippet sorts 3 items starting at 2nd position.

1. Array.Sort(stringArray, 2, 3);

The new output looks like Figure 4.



*Figure 4*

#### Getting and Setting Values

The GetValue and SetValue methods of the Array class can be used to get and set values of an array's items. The code listed in Listing 4 creates a 2-dimensional array instance using the CreateInstance method. After that I use the SetValue method to add values to the array.

In the end, I find number of items in both dimensions and use GetValue method to read values and display on the console.

Array names = Array.CreateInstance(typeof(String), 2, 4);

names.SetValue("Rosy", 0, 0);

names.SetValue("Amy", 0, 1);

names.SetValue("Peter", 0, 2);

names.SetValue("Albert", 0, 3);

names.SetValue("Mel", 1, 0);

names.SetValue("Mongee", 1, 1);

names.SetValue("Luma", 1, 2);

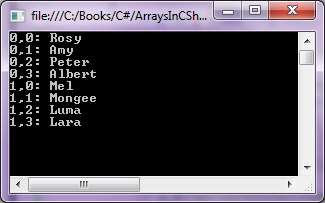
names.SetValue("Lara", 1, 3);

int items1 = names.GetLength(0); int items2 = names.GetLength(1); for (int i = 0; i < items1; i++)

for (int j = 0; j < items2; j++) Console.WriteLine(i.ToString() + "," + j.ToString() + ": " + names.GetValue(i, j));

*Listing 6*

The output of Listing 6 generates Figure 5.



*Figure 5*

#### Reverse an array items

The Reverse static method of the Array class reverses the order of items in an array. Similar to the Sort method, you can just pass an array as a parameter of the Reverse method.

Array stringArray = Array.CreateInstance(typeof(String), 5); stringArray.SetValue("Mahesh", 0);

stringArray.SetValue("Raj", 1);

stringArray.SetValue("Neel", 2);

stringArray.SetValue("Beniwal", 3);

stringArray.SetValue("Chand", 4); Console.WriteLine("Original Array"); Console.WriteLine(" ");

foreach(string str in stringArray) { Console.WriteLine(str);

}

Console.WriteLine(); Console.WriteLine("Reversed Array"); Console.WriteLine(" ");

Array.Reverse(stringArray);

// Array.Sort(stringArray, 2, 3); foreach(string str in stringArray) { Console.WriteLine(str);

}

Console.WriteLine(); Console.WriteLine("Double Reversed Array"); Console.WriteLine(" ");

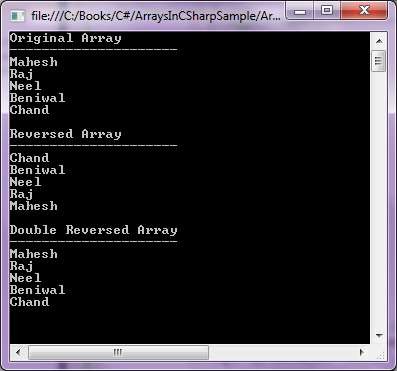
Array.Reverse(stringArray);

// Array.Sort(stringArray, 2, 3); foreach(string str in stringArray) { Console.WriteLine(str);

}

*Listing 7*

The output of Listing 7 generates Figure 6.



*Figure 6*

#### Clear an array items

The Clear static method of the Array class removes all items of an array and sets its length to zero. This method takes three parameters - first an array object, second starting index of the array and third is number of elements. The following code clears two elements from the array starting at index 1 (means second element of the array).

Array.Clear(stringArray, 1, 2);

#### Note

*Keep in mind, the Clear method does not delete items. Just clear the values of the items.*

The code listed in Listing 8 clears two items from the index 1.

Array stringArray = Array.CreateInstance(typeof(String), 5); stringArray.SetValue("Mahesh", 0);

stringArray.SetValue("Raj", 1);

stringArray.SetValue("Neel", 2);

stringArray.SetValue("Beniwal", 3);

stringArray.SetValue("Chand", 4); Console.WriteLine("Original Array"); Console.WriteLine(" ");

foreach(string str in stringArray) { Console.WriteLine(str);

}

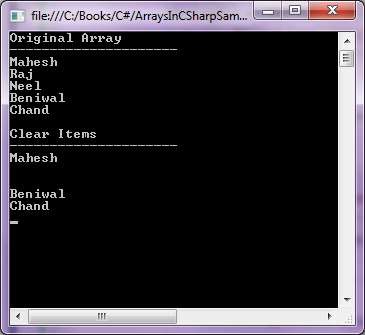
Console.WriteLine(); Console.WriteLine("Clear Items"); Console.WriteLine(" ");

Array.Clear(stringArray, 1, 2); foreach(string str in stringArray) { Console.WriteLine(str);

}

*Listing 8*

The output of Listing 8 generates Figure 7. As you can see from Figure 7, the values of two items from the output are missing but actual items are there.



*Figure 7*

#### Get the size of an array

The GetLength method returns the number of items in an array. The GetLowerBound and GetUppperBound methods return the lower and upper bounds of an array respectively. All these three methods take at least a parameter, which is the index of the dimension of an array. The following code snippet uses all three methods.

Console.WriteLine(stringArray.GetLength(0).ToString()); Console.WriteLine(stringArray.GetLowerBound(0).ToString()); Console.WriteLine(stringArray.GetUpperBound(0).ToString());

#### Copy an array

The Copy static method of the Array class copies a section of an array to another array. The CopyTo method copies all the elements of an array to another one-dimension array. The code listed in Listing 9 copies contents of an integer array to an array of object types.

// Creates and initializes a new Array of type Int32.

Array oddArray = Array.CreateInstance(Type.GetType("System.Int32"), 5); oddArray.SetValue(1, 0);

oddArray.SetValue(3, 1);

oddArray.SetValue(5, 2);

oddArray.SetValue(7, 3);

oddArray.SetValue(9, 4);

// Creates and initializes a new Array of type Object.

Array objArray = Array.CreateInstance(Type.GetType("System.Object"), 5); Array.Copy(oddArray, oddArray.GetLowerBound(0), objArray, objArray.GetLowerBound(0), 4);

int items1 = objArray.GetUpperBound(0);

for (int i = 0; i < items1; i++) Console.WriteLine(objArray.GetValue(i).ToString());

*Listing 9*

You can even copy a part of an array to another array by passing the number of items and starting item in the Copy method. The following format copies a range of items from an Array starting at the specified source index and pastes them to another Array starting at the specified destination index.

public static void Copy(Array, int, Array, int, int);

#### Clone an Array

Clone method creates a shallow copy of an array. A shallow copy of an Array copies only the elements of the Array, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new Array point to the same objects that the references in the original Array point to.

The following code snippet creates a cloned copy of an array of strings. string[] clonedArray = (string[])stringArray.Clone();

## Strings

Strings are one of the most important data types in any modern language including C#. In this article, you will learn how to work with strings in C#. The article discusses the String class, its methods and properties and how to use them.

#### Introduction

In any programming language, to represent a value, we need a data type. The Char data type represents a character in .NET. In .NET, text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0').

The System.String data type is used to represent a string in .NET. A string in C# is an object of type System.String.

The String class in C# represents a string.

The following code creates three strings with a name, number and double values.

// String of characters

System.String authorName = "Mahesh Chand";

// String made of an Integer System.String age = "33";

// String made of a double System.String numberString = "33.23";

Here is the complete example that shows how to use stings in C# and .NET. using System;

namespace CSharpStrings

{

class Program

{

static void Main(string[] args)

{

// Define .NET Strings

// String of characters

System.String authorName = "Mahesh Chand";

// String made of an Integer System.String age = "33";

// String made of a double System.String numberString = "33.23";

// Write to Console.

Console.WriteLine("Name: {0}", authorName);

Console.WriteLine("Age: {0}", age);

Console.WriteLine("Number: {0}", numberString); Console.ReadKey();

} } }

#### String Class

The string class defined in the .NET base class library represents text as a series of Unicode characters. The String class provides methods and properties to work with strings.

The String class has methods to clone a string, compare strings, concatenate strings, and copy strings. This class also provides methods to find a substring in a string, find the index of a character or substring, replace characters, spilt a string, trim a string, and add padding to a string. The string class also provides methods to convert a string characters to uppercase or lowercase.

Check out these links to learn about a specific operation or functionality of strings.

#### What is different between String and System.String?

.NET defines all data types as a class. The System.String class represents a collection of Unicode characters also known as a text. The System.String class also defines the properties and methods to work with string data types.

The String class is equivalent to the System.String in C# language. The string class also inherits all the properties and methods of the System.String class.

#### Create a string

There are several ways to construct strings in C# and .NET.

* + Create a string using a constructor
  + Create a string from a literal
  + Create a string using concatenation
  + Create a string using a property or a method
  + Create a string using formatting

*Create a string using its constructor*

The String class has several overloaded constructors that take an array of characters or bytes. The following code snippet creates a string from an array of characters.

char[] chars = { 'M', 'a', 'h', 'e', 's', 'h' }; string name = new string(chars); Console.WriteLine(name);

*Create a string from a literal*

This is the most common ways to instantiate a string.

You simply define a string type variable and assign a text value to the variable by placing the text value without double quotes. You can put almost any type of characters within double quotes accept some special character limitations.

The following code snippet defines a string variable named firstName and then assigns text value Mahesh to it.

#### string firstName; firstName = "Mahesh";

Alternatively, we can assign the text value direct to the variable. string firstName = “Mahesh”;

Here is a complete sample example of how to create strings using literals. using System;

namespace CSharpStrings

{

class Program

{

static void Main(string[] args)

{

string firstName = "Mahesh"; string lastName = "Chand"; string age = "33";

string numberString = "33.23"; Console.WriteLine("First Name: {0}", firstName); Console.WriteLine("Last Name: {0}", lastName); Console.WriteLine("Age: {0}", age);

Console.WriteLine("Number: {0}", numberString); Console.ReadKey();

} } }

#### Create a string using concatenation

Sting concatenation operator (+) can be used to combine more than one string to create a single string. The following code snippet creates two strings. The first string adds a text Date and current date value from the DateTime object. The second string adds three strings and some hard coded text to create a larger string.

string nowDateTime = "Date: " + DateTime.Now.ToString("D"); string firstName = "Mahesh";

string lastName = "Chand"; string age = "33";

string authorDetails = firstName + " " + lastName + " is " + age + " years old."; Console.WriteLine(nowDateTime);

Console.WriteLine(authorDetails);

#### Create a string using a property or a method

Some properties and methods of the String class returns a string object such as SubString method. The following code snippet takes one sentence string and finds the age within that string. The code returns 33.

string authorInfo = "Mahesh Chand is 33 years old."; int startPosition = sentence.IndexOf("is ") + 1;

string age = authorInfo.Substring(startPosition +2, 2 ); Console.WriteLine("Age: " + age);

#### Create a string with Format

The String.Format method returns a string. The following code snippet creates a new string using the Format method.

string name = "Mahesh Chand"; int age = 33;

string authorInfo = string.Format("{0} is {1} years old.", name, age.ToString()); Console.WriteLine(authorInfo);

#### Create a string using ToString Method

The ToString method returns a string. We can apply ToString on pretty much any data type that can be converted to a string. The following code snippet converts an int data type to a string. string name = "Mahesh Chand";

int age = 33;

string authorInfo = string.Format("{0} is {1} years old.", name, age.ToString()); Console.WriteLine(authorInfo);

#### Get all characters of a string using C#

A string is a collection of characters.

The following code snippet reads all characters of a string and displays on the console.

string nameString = "Mahesh Chand";

for (int counter = 0; counter <= nameString.Length - 1; counter++)

Console.WriteLine(nameString[counter]);

#### Size of string

The Length property of the string class returns the number of characters in a string including white spaces.

The following code snippet returns the size of a string and displays on the console.

string nameString = "Mahesh Chand"; Console.WriteLine(nameString);

Console.WriteLine("Size of string {0}", nameString.Length);

#### Number of characters in a string

We can use the string.Length property to get the number of characters of a string but it will also count an empty character. So, to find out exact number of characters in a string, we need to remove the empty character occurrences from a string.

The following code snippet uses the Replace method to remove empty characters and then displays the non-empty characters of a string.

string name = "Mahesh Chand"; string name = "Mahesh Chand";

// Get size of string

Console.WriteLine("Size of string: {0}", name.Length );

// Remove all empty characters

string nameWithoutEmptyChar = name.Replace(" ", "");

// Size after empty characters are removed

Console.WriteLine("Size of non empty char string: {0}", nameWithoutEmptyChar.Length);

// Read and print all characters

for (int counter = 0; counter <= nameWithoutEmptyChar.Length - 1; counter++)

Console.WriteLine(nameWithoutEmptyChar[counter]);

#### Convert String to Char Array

ToCharArray method converts a string to an array of Unicode characters. The following code snippet converts a string to char array and displays them.

string sentence = "Mahesh Chand is an author and founder of C# Corner"; char[] charArr = sentence.ToCharArray();

foreach (char ch in charArr)

{

Console.WriteLine(ch);

}

#### Empty String

An empty string is a valid instance of a System.String object that contains zero characters. There are two ways to create an empty string. We can either use the string.Empty property or we can simply assign a text value with no text in it.

The following code snippet creates two empty strings. string empStr = string.Empty;

string empStr2 = "";

Both of the statements above generates the same output.

An empty string is sometimes used to compare the value of other strings. The following code snippet uses an empty string to compare with the name string.

string name = "Mahesh Chand"; if (name != empStr)

{

Console.WriteLine(name);

}

In real world coding, we will probably never create an empty string unless you plan to use it somewhere else as a non-empty string. We can simply use the string.Empty direct to compare a string with an empty string.

if (name != string.Empty)

{

Console.WriteLine(name);

}

Here is a complete example of using an empty string. string empStr = string.Empty;

string empStr2 = "";

string name = "Mahesh Chand"; if (name != empStr)

{ Console.WriteLine(name); } if (name != string.Empty)

{ Console.WriteLine(name); }

#### Null String

A null string is a string variable that has not been initialized yet and has a null value. If you try to call any methods or properties of a null string, you will get an exception. A null string valuable is exactly same as any other variable defined in your code.

A null string is typically used in string concatenation and comparison operations with other strings.

The following code example shows how to use a null string.

string nullStr = null;

string empStr = string.Empty; string name = "Mahesh Chand";

if ((name != nullStr) || (name != empStr))

{

Console.WriteLine(name + " is neither null nor empty");

}

**Enumerations**

In C#, enum is a value type data type. The enum is used to declare a list of named integer constants. It can be defined using the enum keyword directly inside a namespace, class, or structure. The enum is used to give a name to each constant so that the constant integer can be referred using its name.

Example: enum enum WeekDays

{

Monday = 0,

Tuesday =1,

Wednesday = 2,

Thursday = 3,

Friday = 4,

Saturday =5,

Sunday = 6

}

Console.WriteLine(WeekDays.Friday); Console.WriteLine((int)WeekDays.Friday);

Output:

Friday 4

By default, the first member of an enum has the value 0 and the value of each successive enum member is increased by 1. For example, in the following enumeration, Monday is 0, Tuesday is 1, Wednesday is 2 and so forth.

Example: enum enum WeekDays

{

Monday, Tuesday, Wednesday, Thursday,

Friday, Saturday, Sunday

}

Console.WriteLine((int)WeekDays.Monday); Console.WriteLine((int)WeekDays.Friday);

Output:

0

4

An explicit cast is necessary to convert from enum type to an integral type. For example, to get the int value from an enum:

Example: enum

int dayNum = (int)WeekDays.Friday; Console.WriteLine(dayNum);

Output:

4

A change in the value of the first enum member will automatically assign incremental values to the other members sequentially. For example, changing the value of Monday to 10, will assign 11 to Tuesday, 12 to Wednesday, and so on:

Example: enum enum WeekDays

{

Monday = 10, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

}

Console.WriteLine((int)WeekDays.Monday); Console.WriteLine((int)WeekDays.Friday);

Output:

10

14

The enum can includes named constants of numeric data type e.g. byte, sbyte, short, ushort, int, uint, long, or ulong.

enum cannot be used with string type.

Enum is mainly used to make code more readable by giving related constants a meaningful name. It also improves maintainability.

#### Enum methods:

Enum is an abstract class that includes static helper methods to work with enums. Enum method Description

Format Converts the specified value of enum type to the specified string format. GetName Returns the name of the constant of the specified value of specified enum. GetNames Returns an array of string name of all the constant of specified enum.

GetValues Returns an array of the values of all the constants of specified enum.

object Parse(type, string) Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.

bool TryParse(string, out TEnum) Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object. The return value indicates whether the conversion succeeded.

Example: enum mehtods enum WeekDays

{

Monday, Tuesday, Wednesday, Thursday, Friday,

Saturday, Sunday

}

Console.WriteLine(Enum.GetName(typeof(WeekDays), 4)); Console.WriteLine("WeekDays constant names:");

foreach (string str in Enum.GetNames(typeof(WeekDays))) Console.WriteLine(str); Console.WriteLine("Enum.TryParse():");

WeekDays wdEnum; Enum.TryParse<WeekDays>("1", out wdEnum); Console.WriteLine(wdEnum);

Output:

Friday

WeekDays constant names:

Monday Tuesday Wednesday Thursday Friday Saturday Sunday

Enum.TryParse():

Tuesday Examples:

using System;

namespace example\_enum

{

class Program

{

public enum DayofWeek

{

Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday

}

static void Main(string[] args)

{

Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Sunday, DayofWeek.Sunday); Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Monday, DayofWeek.Monday); Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Tuesday, DayofWeek.Tuesday);

Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Wednesday, DayofWeek.Wednesday);

Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Thursday, DayofWeek.Thursday); Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Friday, DayofWeek.Friday); Console.WriteLine("Day of week {0} {1}", (int)DayofWeek.Saturday, DayofWeek.Saturday); Console.ReadLine();

}

}

}

Program showing enum type having same values

using System;

namespace enum\_example4

{

class Program

{

public enum DayofWeek

{

Sunday = 1, Monday, Tuesday = 1, Wednesday, Thursday = 2, Friday, Saturday

}

static void Main(string[] args)

{

string[] values = Enum.GetNames(typeof(DayofWeek)); foreach (string s in values)

{

Console.WriteLine(s);

}

Console.WriteLine();

int[] n = (int[])Enum.GetValues(typeof(DayofWeek)); foreach (int x in n)

{

Console.WriteLine(x);

}

Console.ReadLine();

}

}

}

Program to find out number of values in enum

using System;

namespace enum\_exampl3

{

class Program

{

public enum DayofWeek

{

Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday

}

static void Main(string[] args)

{

string[] values = Enum.GetNames(typeof(DayofWeek)); int total = 0;

foreach (string s in values)

{

Console.WriteLine(s); total++;

}

Console.WriteLine("Total values in enum type is : {0}", total); Console.WriteLine();

int[] n = (int[])Enum.GetValues(typeof(DayofWeek)); foreach (int x in n)

{

Console.WriteLine(x);

}

Console.ReadLine();

}

}

}

# Structures

A structure is a value type data type. It helps you to make a single variable hold related data of various data types. The struct keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure − struct Books {

public string title;

public string author; public string subject; public int book\_id;

};

The following program shows the use of the structure −

using System; struct Books {

public string title; public string author; public string subject; public int book\_id;

};

public class testStructure {

public static void Main(string[] args) {

Books Book1; /\* Declare Book1 of type Book \*/ Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/ Book1.title = "C Programming"; Book1.author = "Nuha Ali";

Book1.subject = "C Programming Tutorial"; Book1.book\_id = 6495407;

/\* book 2 specification \*/ Book2.title = "Telecom Billing"; Book2.author = "Zara Ali";

Book2.subject = "Telecom Billing Tutorial"; Book2.book\_id = 6495700;

/\* print Book1 info \*/

Console.WriteLine( "Book 1 title : {0}", Book1.title); Console.WriteLine("Book 1 author : {0}", Book1.author);

Console.WriteLine("Book 1 subject : {0}", Book1.subject);

Console.WriteLine("Book 1 book\_id :{0}", Book1.book\_id);

/\* print Book2 info \*/

Console.WriteLine("Book 2 title : {0}", Book2.title);

Console.WriteLine("Book 2 author : {0}", Book2.author);

Console.WriteLine("Book 2 subject : {0}", Book2.subject);

Console.WriteLine("Book 2 book\_id : {0}", Book2.book\_id);

Console.ReadKey();

}

}

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

Book 1 title : C Programming Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial Book 1 book\_id : 6495407

Book 2 title : Telecom Billing Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial Book 2 book\_id : 6495700

#### Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features −

* Structures can have methods, fields, indexers, properties, operator methods, and events.
* Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
* Unlike classes, structures cannot inherit other structures or classes.
* Structures cannot be used as a base for other structures or classes.
* A structure can implement one or more interfaces.
* Structure members cannot be specified as abstract, virtual, or protected.
* When you create a struct object using the New operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
* If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

#### Class versus Structure

Classes and Structures have the following basic differences −

* classes are reference types and structs are value types
* structures do not support inheritance
* structures cannot have default constructor

In the light of the above discussions, let us rewrite the previous example − using System;

struct Books { private string title;

private string author; private string subject; private int book\_id;

public void getValues(string t, string a, string s, int id) { title = t;

author = a; subject = s; book\_id = id;

}

public void display() { Console.WriteLine("Title : {0}", title); Console.WriteLine("Author : {0}", author); Console.WriteLine("Subject : {0}", subject); Console.WriteLine("Book\_id :{0}", book\_id);

}

};

public class testStructure {

public static void Main(string[] args) {

Books Book1 = new Books(); /\* Declare Book1 of type Book \*/ Books Book2 = new Books(); /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/ Book1.getValues("C Programming",

"Nuha Ali", "C Programming Tutorial",6495407);

/\* book 2 specification \*/ Book2.getValues("Telecom Billing",

"Zara Ali", "Telecom Billing Tutorial", 6495700);

/\* print Book1 info \*/ Book1.display();

/\* print Book2 info \*/

Book2.display(); Console.ReadKey();

}

}

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

Title : C Programming Author : Nuha Ali

Subject : C Programming Tutorial Book\_id : 6495407

Title : Telecom Billing Author : Zara Ali

Subject : Telecom Billing Tutorial Book\_id : 6495700

**Custom Namespaces**

Namespaces are elements that allow us to create a system to organize our code. One more very important use is to avoid name clashes between two sets of code. Using namespaces in our code is a good habit because it is likely to save us from problems later when we want to reuse some of our code.

In this article we will talk about namespaces.

1. delimited by . (dot) operator and
2. by using 'using' directive

Let's look at an example; in a console application to print/write a line on the console we use 'Console.WriteLine("Hello");'. This is only possible, if you use 'using System' directive in code. What is 'System' here? That's the namespace. Let's look at the ways to use namespaces in programs.

1. Dot Operator class Program

{

static void Main(string[] args)

{

System.Console.WriteLine("Using dot operator."); System.Console.ReadKey();

}

}

/\* output:-

Using dot operator.

\*/

1. Using the "using" directive using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Using 'using' directive."); Console.ReadKey();

}

}

/\* output:-

Using 'using' directive.

\*/

Look at the difference in both programs; you will notice the use of dot operator and using directive.

1. Using dot operator

Let's look at the way to use dot operator in C#.

**Program1.cs** using System; namespace demo

{

class Program1

{

public static void myMethod1()

{

Console.WriteLine("Hello demo Namespace using dot operator.");

} } }

#### Program.cs (From below, anyone may be used)

|  |  |
| --- | --- |
| using System; | using System; |
|  | class Program |
| namespace demo | { |
| { | static void Main(string[] |
| class Program | args) |
| { | { |
| static void Main(string[] |  |
| args) | demo.Program1.myMethod1(); |
| { | Console.ReadKey(); |
|  | } |
| demo.Program1.myMethod1(); | } |
| Console.ReadKey(); | /\* output:- |
| } } } | Hello demo Namespace using |
|  | dot operator. |
| /\* output:- | \*/ |
| Hello demo Namespace using |  |
| dot operator. |  |
| \*/ |  |

In the above programs, I have organized the same program in two different C# files. In the first program, I have used a namespace 'demo' and inside this I have a class 'Program1' and inside this a method "myMethod1()". Now to use "myMethod1()" method in Main, I have used its fully qualified path like:

namespace\_name.class\_name.myMethod1();

Actually, this way is not much useful when we need to call myMethod1() many-times. Let's re- write Program.cs code a bit simpler.

1. Using the "using" directive

**Program.cs** using System; using demo;

class Program

{

static void Main(string[] args)

{

Program1.myMethod1(); Console.ReadKey();

} }

/\* output:-

Hello demo Namespace using dot operator.

\*/

Please note, in the above program I am using just "class\_name.myMethod1()" instead of fully qualified name "namespace\_name.class\_name.myMethod1()". And for this I am using a new namespace "using demo;" at the top.

#### Nested Namespace

Namespaces can be nested as well. A good way to organize our namespaces is via a hierarchical system. We put the more general names at the top of the hierarchy and get more specific as we go down. This hierarchical system can be represented by nested namespaces. Look at the program, which has a nested system.

using System; namespace demo

{ namespace test

{ namespace itorian

{ class Program

{

static void Main(string[] args)

{

Console.WriteLine("Welcome to Nested Namespace."); Console.ReadKey();

} } } }}

/\* output:-

Welcome to Nested Namespace.

\*/

**Inheritance**

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in C# by which one class is allowed to inherit the features(fields and methods) of another class.

#### Important terminology:

* + **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
  + **Sub Class:** The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
  + **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want,

we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

#### How to use inheritance

The symbol used for inheritance is **:**.

#### Syntax:

class derived-class : base-class

{

// methods and fields

.

.

}

**Example:** In below example of inheritance, class RR is a base class, class RaghuRamesh is a derived class which extends RR class and class Sudo is a driver class to run program.

// C# program to illustrate the

// concept of inheritance using System;

namespace ConsoleApplication1 {

// Base class class RR {

// data members public string name; public string subject;

// public method of base class

public void readers(string name, string subject)

{

this.name = name; this.subject = subject;

Console.WriteLine("Myself: " + name); Console.WriteLine("My Favorite Subject is: " + subject);

}

}

// inheriting the RR class using : class RaghuRamesh : RR {

// constructor of derived class public RaghuRamesh()

{

Console.WriteLine("RaghuRamesh");

}

}

// Driver class class Sudo {

// Main Method

static void Main(string[] args)

{

// creating object of derived class RaghuRamesh g = new RaghuRamesh();

// calling the method of base class

// using the derived class object g.readers("Harshal", "C#");

}

}

}

#### Output:

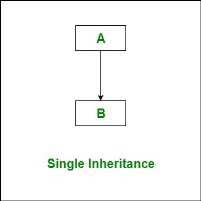
RaghuRamesh Myself: Kirti

My Favorite Subject is: C#

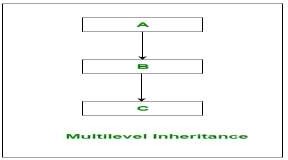
#### Types of Inheritance in C#

Below are the different types of inheritance which is supported by C# in different combinations.

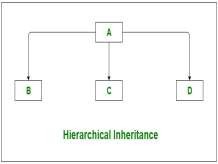
1. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



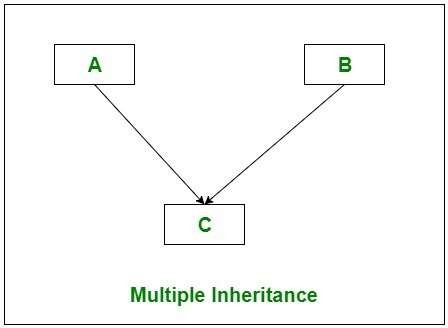
1. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



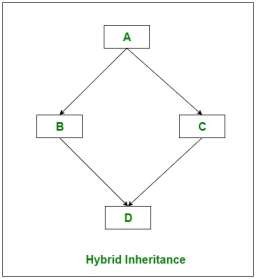
1. **Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.



1. **Multiple Inheritance(Through Interfaces):**In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that **C# does not support multiple inheritance** with classes. In C#, we can achieve multiple inheritance only through Interfaces. In the image below, Class C is derived from interface A and B.



1. **Hybrid Inheritance(Through Interfaces):** It is a mix of two or more of the above types of inheritance. Since C# doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In C#, we can achieve hybrid inheritance only through Interfaces.



#### Important facts about inheritance in C#

* + **Default Superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass(single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
  + **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because C# does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by C#.
  + **Inheriting Constructors:** A subclass inherits all the members (fields, methods) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
  + **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has properties(get and set methods) for accessing its private fields, then a subclass can inherit.

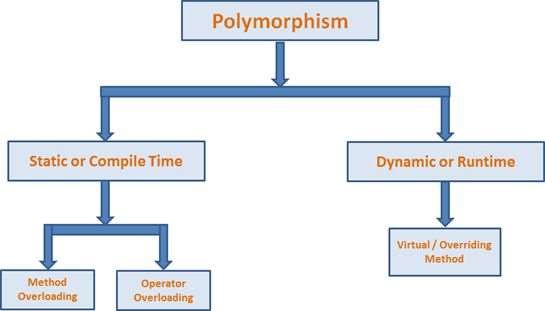
**Polymorphism**

Polymorphism is a Greek word, meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to a class to have multiple implementations with the same name. It is one of the core principles of Object Oriented Programming after encapsulation and inheritance. In this article, you'll learn what polymorphism is, how it works, and how to implement polymorphism in C#.

Types of Polymorphism

There are two types of polymorphism in C#:

* + - Static / Compile Time Polymorphism.
    - Dynamic / Runtime Polymorphism.



Static or Compile Time Polymorphism

It is also known as Early Binding. Method overloading is an example of Static Polymorphism. In overloading, the method / function has a same name but different signatures. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time. Overloading is the concept in which method names are the same with a different set of parameters.

Here C# compiler checks the number of parameters passed and the type of parameter and make the decision of which method to call and it throw an error if no matching method is found.

In the following example, a class has two methods with the same name "Add" but with different input parameters (the first method has three parameters and the second method has two parameters).

public class TestData

{

public int Add(int a, int b, int c)

{

return a + b + c;

}

public int Add(int a, int b)

{

return a + b;

}

}

class Program

{

static void Main(string[] args)

{

TestData dataClass = new TestData(); int add2 = dataClass.Add(45, 34, 67);

int add1 = dataClass.Add(23, 34);

}

}

#### Dynamic / Runtime Polymorphism

Dynamic / runtime polymorphism is also known as late binding. Here, the method name and the method signature (number of parameters and parameter type must be the same and may have a different implementation). Method overriding is an example of dynamic polymorphism.

Method overriding can be done using inheritance. With method overriding it is possible for the base class and derived class to have the same method name and same something. The compiler would not be aware of the method available for overriding the functionality, so the compiler does not throw an error at compile time. The compiler will decide which method to call at runtime and if no method is found then it throws an error.

public class Drawing

{

public virtual double Area()

{

return 0;

}

}

public class Circle : Drawing

{

public double Radius { get; set; } public Circle()

{

Radius = 5;

}

public override double Area()

{

return (3.14) \* Math.Pow(Radius, 2);

}

}

public class Square : Drawing

{

public double Length { get; set; } public Square()

{

Length = 6;

}

public override double Area()

{

return Math.Pow(Length, 2);

}

}

public class Rectangle : Drawing

{

public double Height { get; set; } public double Width { get; set; } public Rectangle()

{

Height = 5.3;

Width = 3.4;

}

public override double Area()

{

return Height \* Width;

}

}

class Program

{

static void Main(string[] args)

{

Drawing circle = new Circle(); Console.WriteLine("Area :" + circle.Area());

Drawing square = new Square(); Console.WriteLine("Area :" + square.Area());

Drawing rectangle = new Rectangle(); Console.WriteLine("Area :" + rectangle.Area());

}

}

The compiler requires an Area() method and it compiles successfully but the right version of the Area() method is not being determined at compile time but determined at runtime. Finally the overriding methods must have the same name and signature (number of parameters and type), as the virtual or abstract method defined in the base class method and that it is overriding in the derived class.

A method or function of the base class is available to the child (derived) class without the use of the "overriding" keyword. The compiler hides the function or method of the base class. This concept is known as shadowing or method hiding. You may find the difference between overriding and shadowing here.

Preventing Derived class from overriding virtual members

Virtual members remain “virtual” indefinitely. In other words, virtual members remain “virtual” regardless of how many classes have been between virtual members and the class that originally declared it. For example, if class X has the virtual method "A" and the class Y is derived from X and the class Z is derived from Y, class Z inherits the virtual method "A" and override it.

public class X

{

public virtual void A()

{

}

}

public class Y : X

{

public override void A()

{

}

}

A derived class is able to stop virtual inheritance by declaring an override member as "sealed".

public class Y : X

{

public sealed override void A()

{

}

}

Accessing Base class virtual member

Using the "base" keyword, the derived class is able to access the method. public class X

{

public virtual void A()

{

}

}

public class Y : X

{

public override void A()

{

base.A();

}

}

Summary

* + - The meaning of Polymorphism is one name having multiple forms.
    - The following are the two types of Polymorphism:
      * Static or compile-time polymorphism (for example, method overloading and operator overloading).
      * Dynamic or runtime polymorphism (for example, overriding).
    - Method Overriding differs from shadowing.
    - Using the "new" keyword, we can hide the base class member.
    - We can prevent a derived class from overriding virtual members.
    - We can access a base class virtual member from the derived class.

**public class** Drawing

{

**public virtual double** Area()

{ **return** 0; }

}

**public class** Circle : Drawing

{

**public double** Radius { **get**; **set**; }

**public** Circle()

{ Radius = 5; }

**public override double** Area()

{

**return** (3.14) \* Math.Pow(Radius, 2);

} }

**public class** Square : Drawing

{

**public double** Length { **get**; **set**; }

**public** Square()

{

Length = 6;

}

**public override double** Area()

{

**return** Math.Pow(Length, 2);

}}

**public class** Rectangle : Drawing

{

**public double** Height { **get**; **set**; }

**public double** Width { **get**; **set**; }

**public** Rectangle()

{

Height = 5.3;

Width = 3.4;

}

**public override double** Area()

{

**return** Height \* Width;

}

}

**class** Program

{

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

{

Drawing circle = **new** Circle(); Console.WriteLine("Area :" + circle.Area());

Drawing square = **new** Square(); Console.WriteLine("Area :" + square.Area());

Drawing rectangle = **new** Rectangle(); Console.WriteLine("Area :" + rectangle.Area());

}

}