

# C# tutorial

## 1-Version History

C# is a simple & powerful object-oriented programming language developed by Microsoft.

C# has evolved much since its first release in 2002. C# was introduced with .NET Framework 1.0 and the current version of C# is 6.0.

The following table lists important features introduced in each version of C#:

Version	.NET Framework	Visual Studio	Important Features
C# 1.0	.NET Framework 1.0/1.1	Visual Studio .NET 2002	<ul style="list-style-type: none"><li>• Basic features</li><li>•</li></ul>
C# 2.0	.NET Framework 2.0	Visual Studio 2005	<ul style="list-style-type: none"><li>• Generics</li><li>•</li><li>• Partial types</li><li>•</li><li>• Anonymous methods</li><li>•</li><li>• Iterators</li><li>•</li><li>• Nullable types</li><li>•</li><li>• Private setters (properties)</li><li>•</li><li>• Method group conversions (delegates)</li><li>•</li><li>• Covariance and Contra-variance</li><li>•</li><li>• Static classes</li><li>•</li></ul>
C# 3.0	.NET Framework 3.0\3.5	Visual Studio 2008	<ul style="list-style-type: none"><li>• Implicitly typed local variables</li><li>•</li><li>•</li></ul>

			<ul style="list-style-type: none"> <li>Object and collection initializers</li> <li>•</li> <li>•</li> <li>Auto-Implemented properties</li> <li>•</li> <li>•</li> <li>Anonymous types</li> <li>•</li> <li>•</li> <li>Extension methods</li> <li>•</li> <li>•</li> <li>Query expressions</li> <li>•</li> <li>•</li> <li>Lambda expressions</li> <li>•</li> <li>•</li> <li>Expression trees</li> <li>•</li> <li>•</li> <li>Partial Methods</li> <li>•</li> </ul>
<b>C# 4.0</b>	.NET Framework 4.0	Visual Studio 2010	<ul style="list-style-type: none"> <li>•</li> <li>Dynamic binding (late binding)</li> <li>•</li> <li>•</li> <li>Named and optional arguments</li> <li>•</li> <li>•</li> <li>Generic co- and contravariance</li> <li>•</li> <li>•</li> <li>Embedded interop types</li> <li>•</li> </ul>
<b>C# 5.0</b>	.NET Framework 4.5	Visual Studio 2012/2013	<ul style="list-style-type: none"> <li>•</li> <li>Async features</li> <li>•</li> <li>•</li> <li>Caller information</li> <li>•</li> </ul>
<b>C# 6.0</b>	.NET Framework 4.6	Visual Studio 2013/2015	<ul style="list-style-type: none"> <li>•</li> </ul>

			Expression Bodied Methods
			•
			•
			Auto-property initializer
			•
			•
			nameof Expression
			•
			•
			Primary constructor
			•
			•
			Await in catch block
			•
			•
			Exception Filter
			•
			•
			String Interpolation
			•

Page Break

## 2-Setup Development Environment for C#

C# is used for server side execution for different kind of application like (web applications), (window forms applications) or (console applications) etc. In order to use C# with your .Net application, you need two things:

- 1- .NET Framework
- 2- IDE (Integrated Development Environment).

### A-.NET Framework

The .NET Framework is a platform where you can write different types of web and desktop based applications.

You can use :

- 1-C#
- 2- Visual Basic
- 3- F#

( With Jscript to write these applications.)

NOTE : If you have the Windows operating system, the .NET framework might already be installed in your PC. Check MSDN to learn about [.NET Framework dependencies](#).

### B-Integrated development Environment(IDE)

An IDE is a tool that helps you write your programs. Visual Studio is an IDE provided by Microsoft to write the code in languages such as C#, F#, VisualBasic , etc. Use Visual Studio 2010/2012/2013 based on the C# version you want to work with.

[Visual Studio](#) is a licensed product, so you must buy a license for commercial use. However, Visual Studio Express is free for learning purpose. Download and install Visual Studio Express from [www.visualstudio.com](http://www.visualstudio.com). We will use Visual Studio Express 2012 for all the C# tutorials.

Page Break

We Will work with console applications to learn c#

---

## 1-First Program(Visual Studio IDE)

1-Create a new project

create a C# console project by clicking VS2012 -> File -> New Project...

From the popup, select Visual C# under Templates (in the right column) and select Console Application from the middle column.

In the name section, give any appropriate project name, location where you want to create all the project files and solution name.

Click OK to create the console project. Program.cs will be created as default .cs file in Visual Studio where you can write your C# code in Program class as shown below.

Now, let's write simple C# code to understand important building blocks. Every console application starts from the Main() method of Program class. The following example code displays "Hello World!!" on the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace CSharpTutorials
{
    Class Program
    {
        static void Main(string[] args)
        {
            string message
= "HelloWorld!!";
le.WriteLine(message);
        }
    }
```

```
}  
}  
}
```

The following image illustrates the important parts of the above example.

Explanation of above points:

1.  
Every .NET application takes the reference of the necessary .NET framework namespaces that it is planning to use with the "using" keyword e.g. *using System.Text*

2.  
1.  
Declare the namespace for the current class using the "namespace" keyword e.g. *namespace CSharpTutorials.FirstProgram*

2.  
1.  
We then declared a class using the "class" keyword: *class Program*

2.  
1.  
The Main() is a method of Program class which is the entry point of the console application.

2.  
1.  
String is a data type.

2.  
1.  
'message' is a variable, that holds a value of a specified data type.

2.  
1.  
"Hello World!!" is the value of the message variable.

2.  
1.  
Console is a .NET framework class. WriteLine() is a method which you can use to display messages to the console

2.  
  
Page Break **2-Class(C#)**

A class is like a blueprint of specific object. In the real world, every object has some attributes like:( color, shape and functionalities).

For example, the luxury car Ferrari.

Ferrari is an object of the luxury car type.

The luxury car is a class that specify certain characteristic like

1-speed

2- color

3- shape interior

.....etc

So any company that makes a car that meet those requirements is an object of the luxury car type.

For example, every single car of BMW, lamborghini, cadillac are an object of the class called 'Luxury Car'.

'Luxury Car' is a class and every single physical car is an object of the luxury car class.

in object oriented programming, a class defines certain properties, fields, events, method etc. A class defines the kinds of data and the functionality their objects will have.

#### Access Modifiers:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members. Public, private, protected and internal are access modifiers in C#. We will learn about it in the keyword section.

#### Field:

Field is a class level variable that can holds a value. Generally field members should have a private access modifier and used with a property.

#### Constructor:

A class can have parameterized or parameter less constructors. The constructor will be called when you create an instance of a class. Constructors can be defined by using an access modifier and class name: `<access modifiers> <class name>(){ }`

```
class MyClass
{
    public MyClass()
    {
    }
}
```

#### Method:

A method can be defined using the following template:

```
{access modifier} {return type} MethodName({parameterType parameterName})
public void MyMethod(int parameter1, string parameter2)
{
    // write your method code here..
}
```

#### Property:

A property can be defined using getters and setters, as below:

```
private int _myPropertyVar;
public int MyProperty
{
    get { return _myPropertyVar; }
    set { _myPropertyVar = value; }
}
```

Property encapsulates a private field. It provides getters (get{ }) to retrieve the value of the underlying field and setters (set{ }) to set the value of the underlying field. In the above example, `_myPropertyVar` is a private field which cannot be accessed directly. It will only be accessed via `MyProperty`. Thus, `MyProperty` encapsulates `_myPropertyVar`.

You can also apply some addition logic in get and set, as in the below example.

```
private int _myPropertyVar;
public int MyProperty
{
    Get { return _myPropertyVar / 2; }
    set { if (value > 100)
        _myPropertyVar = 100;
        else _myPropertyVar = value;
    }
}
```

Auto-implemented Property:

From C# 3.0 onwards, property declaration has been made easy if you don't want to apply some logic in get or set.

The following is an example of an auto-implemented property:

```
public int MyAutoImplementedProperty { get; set; }
```

Notice that there is no private backing field in the above property example. The backing field will be created automatically by the compiler. You can work with an automated property as you would with a normal property of the class. Automated-implemented property is just for easy declaration of the property when no additional logic is required in the property accessors.

Namespace:

Namespace is a container for a set of related classes and namespaces. Namespace is also used to give unique names to classes within the namespace name. Namespace and classes are represented using a dot (.).

```
namespace CSharpTutorials
{
    class MyClass { }
}
```

```
string message;
// value can be assigned after it declared
message = "Hello World!!";
```

The variable in C# is nothing but a name given to a data value. In the above example, message is the name of the variable that stores the string data value "Hello World!!". As the name suggests, the contents of a variable can vary, i.e., you can change the value of a variable at any time.

In C#, a variable is always defined with a **data type**. The following is the syntax variable declaration and initialization.

Syntax:

```
<data type> <variable name>;
<datatype> <variable name> = <value>;
```

A variable can be declared and initialized later or it can be declared and initialized at the same time.

Page Break **4-Data Types**

```
string message = "Hello World!!";
```

string is a data type, message is a variable, and "Hello World!!" is a string value assigned to a variable - message.

The data type tells a C# compiler what kind of value a variable can hold. C# includes many in-built data types for different kinds of data, e.g., String, number, float, decimal,

```
string stringVar = "Hello World!!";
int intVar = 100;
float floatVar = 10.2f;
char charVar = 'A';
bool boolVar = true;
```

Alias	.NET Type	Type	Size (bits)	Range (values)
<b>byte</b>	Byte	Unsigned integer	8	0 to 255
<b>sbyte</b>	SByte	Signed integer	8	-128 to 127
<b>int</b>	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
<b>uint</b>	UInt32	Unsigned integer	32	0 to 4294967295
<b>short</b>	Int16	Signed integer	16	-32,768 to 32,767
<b>ushort</b>	UInt16	Unsigned integer	16	0 to 65,535
<b>long</b>	Int64	Signed integer	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>ulong</b>	UInt64	Unsigned integer	64	0 to 18,446,744,073,709,551,615
<b>float</b>	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
<b>double</b>	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
<b>char</b>	Char	A single Unicode character	16	Unicode symbols used in text
<b>bool</b>	Boolean	Logical Boolean type	8	True or False
<b>object</b>	Object	Base type of all		



		other types		
<b>string</b>	String	A sequence of characters		
<b>decimal</b>	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	(+ or -)1.0 x 10e-28 to 7.9 x 10e28
<b>DateTime</b>	DateTime	Represents date and time		0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

Page Break

## 5-Key words

Access Modifier Keywords:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members.

Access Modifiers	Usage
<b>public</b>	The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members.
<b>private</b>	The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it.
<b>internal</b>	The Internal modifier allows other program code in the same assembly to access the type or its members.
<b>protected</b>	The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members.

Statement Keywords:

Statement keywords are related to program flow.

Statement Keywords
<b>if</b>
<b>else</b>
<b>switch</b>
<b>case</b>
<b>do</b>
<b>for</b>

<b>foreach</b>
<b>in</b>
<b>while</b>
<b>break</b>
<b>continue</b>
<b>default</b>
<b>goto</b>
<b>return</b>
<b>yield</b>
<b>throw</b>
<b>try</b>
<b>catch</b>
<b>finally</b>
<b>checked</b>
<b>unchecked</b>
<b>fixed</b>
<b>lock</b>

Method parameter keywords:

These keywords are applied on the parameters of a method.

Method Parameter Keywords
<b>params</b>
<b>ref</b>
<b>out</b>

Access keywords:

Access keywords are used to access the containing class or the base class of an object or class.

Access keywords
<b>base</b>
<b>this</b>

Type keywords:

Type keywords are used for data types.

Type keywords
<b>bool</b>
<b>byte</b>
<b>char</b>
<b>class</b>
<b>decimal</b>
<b>double</b>
<b>enum</b>

float
int
long
sbyte
short
string
struct
uint
ulong
ushort

Query keywords:

Query keywords are contextual keywords used in LINQ queries.

Query Keywords
from
where
select
group
into
orderby
join
let
in
on
equals
by
ascending
descending

Page Break

## 6-Interface

An interface in C# contains only the declaration of the methods, properties, and events, but not the implementation. It is left to the class that implements the interface by providing implementation for all the members of the interface. Interface makes it easy to maintain a program.

```
interface ILog{    void Log(string msgToLog);}
```

Implement interface using- : <interface name > syntax.

```
class ConsoleLog: ILog
{
    public void Log(string msgToPrint)
    {
        Console.WriteLine(msgToPrint);
    }
}
class FileLog :ILog
```

```

{
    public void Log(string msgToPrint)
    {
        File.AppendText(@"C:\Log.txt").Write(msgToPrint);
    }
}

```

Page Break

## 7-Operators

Operator in C# is a special symbol that specifies which operations to perform on operands. For example, in mathematics the plus symbol (+) signifies the sum of the left and right numbers. In the same way, C# has many operators that have different meanings based on the data types of the operands. C# operators usually have one or two operands. Operators that have one operand are called Unary operators.

Operator category	Operators
Primary	x.y
Unary	+x
Multiplicative	x * y
Additive	x + y
Shift	x << y
Relational and type testing	x < y
Equality	x == y
Logical AND	x & y
Logical XOR	x ^ y
Logical OR	x   y
Conditional AND	x && y
Conditional OR	x    y
Null-coalescing	x ?? y
Conditional	?:
Assignment and lambda expression	

```

static void Main(string[] args)
{
    string message1 = "Hello";
    string message2 = message1
+ "World!!";    Console.WriteLine(message2);
    int i = 10, j = 20;
    int sum = i + j;
    Console.WriteLine("{0} + {1} = {2}", i, j, sum);
}

```

Page Break

## 8-Condition

## if statement:

C# provides many decision making statements that help the flow of the C# program based on certain logical conditions. C# includes the following decision making statements.

1.

if statement

2.

1.

if-else statement

2.

1.

switch statement

2.

1.

Ternary operator :?

2.

Here, you will learn about the if statements.

Syntax:

```
if(boolean expression){    // execute this code block if expression evalutes to true}
```

Code:

```
if(true)
{
    Console.WriteLine("This will be displayed.");
}
if(false)
{
    Console.WriteLine("This will not be displayed.");
}
```

## If-else statement:

C# also provides for a second part to the `if` statement, that is `else`. The else statement must follow `if` or `else if` statement. Also, `else` statement can appear only one time in a if-else statement chain.

Syntax:

```
if(boolean expression){    // execute this code block if expression evalutes to true}else{    // always execute this code block when above if expression is false}
```

Code:

```
int i = 10, j = 20;
if (i > j)
{
    Console.WriteLine("i is greater than j");
}
Else
{
    Console.WriteLine("i is either equal to or less than j");
}
```

## else if statement:

The 'if' statement can also follow an 'else' statement, if you want to check for another condition in the else part.

```
static void Main(string[] args)
{
    int i = 10, j = 20;
    if (i > j)
    {
        Console.WriteLine("i is greater than j");
    }
    else if (i < j)
    {
        Console.WriteLine("i is less than j");
    }
    Else
    {
        Console.WriteLine("i is equal to j");
    }
}
```

Page Break

## 9-Ternary Operator

var result = Boolean conditional expression ? first statement : second statement

Code

```
int x = 20, y = 10;
var result = x > y ? "x is greater than y" : "x is less than or equal to y";
Console.WriteLine(result);
```

If x>y then result="x is greater than y" else result="x less than or equal to y".

Nested Ternary operator:

Nested ternary operators are possible by including conditional expression as a second (after ?) or third part (after :) of the ternary operator..Consider the following example.

```
int x = 2, y = 10;
string result = x > y ? "x is greater than y" : x < y?"x is less than y" :
x == y ? "x is equal to y" : "No result";
```

Page Break

## 10-Switch

C# includes another decision making statement called switch. The switch statement executes the code block depending upon the resulted value of an expression.

```
switch(expression){
```

```

case <value1>      // code block   break;
case <value2>      // code block   break;
case <valueN>      // code block   break;
default           // code block
break;}

```

```

int x = 10;
switch (x)
{
    case 5:
        Console.WriteLine("Value of x is 5");
        break;
    case 10:
        Console.WriteLine("Value of x is 10");
        break;
    case 15:
        Console.WriteLine("Value of x is 15");
        break;
    default:
        Console.WriteLine("Unknown value");
        break;
}

```

Goto with Switch:

```

string statementType = "switch";
switch (statementType){
    case "DecisionMaking":
        Console.Write(" is a decision making statement.");
        break;
    case "if.else":
        Console.Write("if-else");
        break;
    case "ternary":
        Console.Write("Ternary operator");
        break;
    case "switch":
        Console.Write("switch statement");
        goto case "DecisionMaking";
}

```

Nested switch:

```

int j = 5;
switch (j){
    case 5:
        Console.WriteLine(5);
        switch (j - 1)
        {
            case 4:
                Console.WriteLine(4);
                switch (j - 2)
                {
                    case 3:
                        Console.WriteLine(3);

```

```

        break;
    }
    break;
}
break;
case 10:
    Console.WriteLine(10);
    break;
case 15:
    Console.WriteLine(15);
    break;
default:
    Console.WriteLine(100);
    break;
}

```

Page Break

## 11-C# for loop

The **for** keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false.

```

for (variable initialization; condition; steps)
{
    //execute this code block as long as condition is satisfied
}

```

As per the syntax above, the for loop contains three parts: initialization, conditional expression and steps, which are separated by a semicolon.

1. variable initialization: Declare & initialize a variable here which will be used in conditional expression and steps part.

- 2.

- 1.

condition: The condition is a boolean expression which will return either true or false.

- 2.

- 1.

steps: The steps defines the incremental or decremental part

- 2.

Consider the following example of a simple for loop.

```

for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Value of i: {0}", i);
}

```

Page Break

## 12-C# While loop

C# includes the while loop to execute a block of code repeatedly.

```

While(boolean expression){    //execute code as long as condition returns
                             true    }

int i = 0;
while (i < 10){    Console.WriteLine("Value
of i: {0}", i);    i++;}

```



### 13-C# do-while:

The do-while loop is the same as a 'while' loop except that the block of code will be executed at least once, because it first executes the block of code and then it checks the condition.

```
do{
    //execute code block
} while(boolean expression);
```

Nested Do While Loop:

```
int i = 0;
do{
    Console.WriteLine("Value of i: {0}", i);
    int j = i;
    i++;
    do {
        Console.WriteLine("Value of j: {0}", j);
        j++;
    } while (j < 2);
} while (i < 2);
```

### 14-Structure

We have learned [class](#) in the previous section. Class is a [reference type](#). C# includes a value type entity, which is called Structure. A structure is a value type that can contain constructors, constants, fields, methods, properties, indexers, operators, events and nested types. A structure can be defined using the *struct* keyword.

Declaration:

```
public struct Discounts{
    public int Cloths { get; set; }
    public int HomeDecor { get; set; }
    public int Grocery { get; set; }
}
```

Initialization:

```
Discounts saleDiscounts = new Discounts();
saleDiscounts.Cloths = 10;
saleDiscounts.HomeDecor = 5;
saleDiscounts.Grocery = 2;
```

A struct is a value type so it is faster than a class object. Use struct whenever you want to just store the data. Generally structs are good for game programming. However, it is easier to transfer a class object than a struct. So do not use struct when you are passing data across the wire or to other classes.

Characteristics of Structure:

- Structure can include constructors, constants, fields, methods, properties, indexers, operators, events & nested types.
- 
-

Structure cannot include default constructor or destructor.

- 
- 

Structure can implement interfaces.

- 
- 

A structure cannot inherit another structure or class.

- 
- 

Structure members cannot be specified as abstract, virtual, or protected.

- 
- 

Structures must be initialized with new keyword in order to use its properties, methods or events.

- 

Difference between struct and class:

- 

Class is reference type whereas struct is value type

- 
- 

Struct cannot declare a default constructor or destructor. However, it can have parametrized constructors.

- 
- 

Struct can be instantiated without the new keyword. However, you won't be able to use any of its methods, events or properties if you do so.

- 
- 

Struct cannot be used as a base or cannot derive another struct or class.

- 

Page Break

```
struct Point{
    private int _x, _y;
    public int x, y;
    public static int X, Y;
    public int XPoint {
        Get
        {
            return _x;
        }
        Set
        {
            _x = value;
            PointChanged(_x);
        }
    }
    public int YPoint
    {
        Get
        {
            return _y;
        }
    }
}
```

```

        Set
        {
            _y = value;
            PointChanged(_y);
        }
    }
    public event Action<int> PointChanged;
    public void PrintPoints()
    {
        Console.WriteLine(" x: {0}, y: {1}", _x, _y);
    }
    public static void StaticMethod()
    {
        Console.WriteLine("Inside Static method");
    }
}

```

Page Break

## 15-Enum

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.

```

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);

```

Enum methods:

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

Enum method	Description
<b>Format</b>	Converts the specified value of enum type to the specified string format.
<b>GetName</b>	Returns the name of the constant of the specified value of specified enum.
<b>GetNames</b>	Returns an array of string name of all the constant of specified enum.
<b>GetValues</b>	Returns an array of the values of all the constants of specified enum.
<b>object Parse(type, string)</b>	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.
<b>bool TryParse(string, out TEnum)</b>	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.

```
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);
```

Page Break

## 16-StringBuilder

A String is immutable, meaning String cannot be changed once created. For example, new string "Hello World!!" will occupy a memory space on the heap. Now, by changing the initial string "Hello World!!" to "Hello World!! from Tutorials Teacher" will create a new string object on the memory heap instead of modifying the initial string at the same memory address. This behaviour will hinder the performance if the same string changes multiple times by replacing, appending, removing or inserting new strings in the initial string.

To solve this problem, C# introduced StringBuilder. StringBuilder is a dynamic object that allows you to expand the number of characters in the string. It doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string.

### Initialization

StringBuilder can be initialized the same way as class.

```
StringBuilder sb = new StringBuilder(); //orStringBuilder
sb = new StringBuilder("Hello World!!");
```

### initial capacity of characters:

You can give an initial capacity of characters by passing an int value in the constructor. For example, the following will allocate memory of 50 characters sequentially on the memory heap. The memory allocation automatically expands once it reaches the capacity.

```
StringBuilder sb = new StringBuilder(50); //orStringBuilder sb = new
StringBuilder("Hello World!!", 50);
```

## Important Methods of StringBuilder:

Method Name	Description
<b>StringBuilder.Append(valueToAppend)</b>	Appends the passed values to the end of the current StringBuilder object.
<b>StringBuilder.AppendFormat()</b>	Replaces a format specifier passed in a string with formatted text.
<b>StringBuilder.Insert(index, valueToAppend)</b>	Inserts a string at the specified index of the current StringBuilder object.
<b>StringBuilder.Remove(int startIndex, int length)</b>	Removes the specified number of characters from the given starting position of the current StringBuilder object.
<b>StringBuilder.Replace(oldValue, newValue)</b>	Replaces characters with new characters

```
StringBuilder sb = new StringBuilder("Hello ",50);
sb.Append("World!!");
sb.AppendLine("Hello C#!");
sb.AppendLine("This is new line.");
Console.WriteLine(sb);
```

Page Break

## 17-C# Array

An array is a special type of data type which can store fixed number of values sequentially using special syntax.

The following image shows how an array stores values sequentially.

Array Representation

**Array Declaration:**

An array can be declare using a type name followed by square brackets [].

Example: Array declaration in C#

```
int[] intArray; // can store
int valuesbool[] boolArray; // can store
boolean valuesstring[] stringArray; // can store
string valuesdouble[] doubleArray; // can store
double valuesbyte[] byteArray; // can store
byte valuesStudent[] customClassArray; // can store instances of
Student class
```

## Initialization:

```
// defining array with size 5. add values later on
int[] intArray1 = new int[5];
// defining array with size 5 and adding values at the same time
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};
// defining array with 5 elements which indicates the size of an array
int[] intArray3 = {1, 2, 3, 4, 5};
```

```
string[] strArray1, strArray2;
strArray1 = new string[5]{ "1st Element",
                           "2nd Element",
                           "3rd Element",
                           "4th Element",
                           "5th Element" };
strArray2 = new string[] { "1st Element",
                           "2nd Element",
                           "3rd Element",
                           "4th Element",
                           "5th Element" };
};
```

Page Break

## 18-Multi-dimensional Array:

We have learned about single dimensional arrays in the previous section. C# also supports multi-dimensional arrays. A multi-dimensional array is a two dimensional series like rows and columns.

Example: Multi-dimensional Array:

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

int[,] intArray = { {1, 1}, {1, 2}, {1, 3} };
```

Page Break

## 19-Jagged Array:

A jagged array is an array of an array. Jagged arrays store arrays instead of any other data type value directly.

A jagged array is initialized with two square brackets []. The first bracket specifies the size of an array and the second bracket specifies the dimension of the array which is going to be stored as values. (Remember, jagged array always store an array.)

```
int[][] intJaggedArray = new int[2][];  
intJaggedArray[0] = new int[3]{1,2,3};  
intJaggedArray[1] = new int[2]{4,5};  
Console.WriteLine(intJaggedArray[0][0]); //  
1Console.WriteLine(intJaggedArray[0][2]); //  
3    Console.WriteLine(intJaggedArray[1][1]); // 5
```

The following jagged array stores a multi-dimensional array as a value. Second bracket [,] indicates multi-dimension.

```
int[,] intJaggedArray = new int[3][,];  
intJaggedArray[0] = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };  
intJaggedArray[1] = new int[2, 2] { { 3, 4 }, { 5, 6 } };  
intJaggedArray[2] = new int[2,  
2];Console.WriteLine(intJaggedArray[0][1,1]); //  
4Console.WriteLine(intJaggedArray[1][1,0]); //  
5Console.WriteLine(intJaggedArray[1][1,1]); // 6
```

Page Break

## 20-C# Collection:

We have learned about an array in the previous section. C# also includes specialized classes that hold many values or objects in a specific series, that are called 'collection'.

There are two types of collections available in C#: non-generic collections and [generic collections](#). We will learn about non-generic collections in this section.

Every collection class implements the IEnumerable interface so values from the collection can be accessed using a **foreach** loop.

Non-generic Collections	Usage
<a href="#">ArrayList</a>	ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically.
<a href="#">SortedList</a>	SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection.

<b>Stack</b>	Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack.
<b>Queue</b>	Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue.
<b>HashTable</b>	Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys.
<b>BitArray</b>	BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).

Page Break

## 21-ArrayList

ArrayList is a non-generic type of collection in C#. It can contain elements of any data types. It is similar to an array, except that it grows automatically as you add items in it. Unlike an array, you don't need to specify the size of ArrayList.

Initializing:

```
ArrayList myArrayList = new ArrayList();
```

Important Properties and methods of ArrayList:

Property	Description
<b>Capacity</b>	Gets or sets the number of elements that the ArrayList can contain.
<b>Count</b>	Gets the number of elements actually contained in the ArrayList.
<b>IsFixedSize</b>	Gets a value indicating whether the ArrayList has a fixed size.
<b>IsReadOnly</b>	Gets a value indicating whether the ArrayList is read-only.
<b>Item</b>	Gets or sets the element at the specified index.



Method	Description
<a href="#"><u>Add()/AddRange()</u></a>	Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList.
<a href="#"><u>Insert()/InsertRange()</u></a>	Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList.
<a href="#"><u>Remove()/RemoveRange()</u></a>	Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList.
<a href="#"><u>RemoveAt()</u></a>	Removes the element at the specified index from the ArrayList.
<a href="#"><u>Sort()</u></a>	Sorts entire elements of the ArrayList.
<a href="#"><u>Reverse()</u></a>	Reverses the order of the elements in the entire ArrayList.
<a href="#"><u>Contains</u></a>	Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false.
<b>Clear</b>	Removes all the elements in ArrayList.
<b>CopyTo</b>	Copies all the elements or range of elements to compatible Array.
<b>GetRange</b>	Returns specified number of elements from specified index from ArrayList.
<b>IndexOf</b>	Search specified element and returns zero based index if found. Returns -1 if element not found.
<b>ToArray</b>	Returns compatible array from an ArrayList.

Add elements into ArrayList:

```
ArrayList arrayList1 = new ArrayList();
arrayList1.Add(1);
arrayList1.Add("Two");
arrayList1.Add(3);
arrayList1.Add(4.5);
ArrayList arrayList2 = new ArrayList();
arrayList2.Add(100);
arrayList2.Add(200); //adding entire arrayList2 into
arrayList1arrayList1.AddRange(arrayList2);
```

Access ArrayList Elements:

ArrayList elements can be accessed using indexer, in the same way as an array. However, you need to cast it to the appropriate type or use the implicit type `var` keyword while accessing it.

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(1);
myArrayList.Add("Two");
myArrayList.Add(3);
myArrayList.Add(4.5f);
//Access individual item using indexer
int firstElement = (int) myArrayList[0]; //returns 1
string secondElement = (string) myArrayList[1]; //returns "Two"
int thirdElement = (int) myArrayList[2]; //returns 3
float fourthElement = (float) myArrayList[3]; //returns
4.5//use var keywordvar firstElement = myArrayList[0]; //returns 1
```

Use a `foreach` or a `for` loop to iterate an ArrayList.

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(1);
myArrayList.Add("Two");
myArrayList.Add(3);
myArrayList.Add(4.5);
foreach (var val in myArrayList)
    Console.WriteLine(val);
//Orfor(int i =
0 ; i< myArrayList.Count; i++) Console.WriteLine(myArrayList[i]);
```

Insert elements into ArrayList:

Use the `Insert()` method to insert a single item at the specified index.

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(1);
myArrayList.Add("Two");
myArrayList.Add(3);
myArrayList.Add(4.5);
myArrayList.Insert(1, "Second Item");
myArrayList.Insert(2, 100);
foreach (var val in myArrayList)
    Console.WriteLine(val);
```

Use the `InsertRange()` method to insert all the values from another collection into ArrayList at the specified index.

```
ArrayList arrayList1 = new ArrayList();
arrayList1.Add(100);
arrayList1.Add(200);
ArrayList arrayList2 = new ArrayList();
```

```

arryList2.Add(10);
arryList2.Add(20);
arryList2.Add(30);
arryList2.InsertRange(2, arryList1);
foreach(var item in arryList2)
    Console.WriteLine(item);

```

Remove elements from an ArrayList:

```

ArrayList arryList1 = new ArrayList();
arryList1.Add(100);
arryList1.Add(200);
arryList1.Add(300);
arryList1.Remove(100); //Removes 1 from ArrayList
foreach (var item in arryList1)
    Console.WriteLine(item);

```

Use the RemoveAt() method to remove an element from the specified index location.

```

ArrayList arryList1 = new ArrayList();
arryList1.Add(100);
arryList1.Add(200);
arryList1.Add(300);
arryList1.RemoveAt(1); //Removes the first element from an ArrayList
foreach (var item in arryList1)
    Console.WriteLine(item);

```

Page Break

## 22-Sorted List

The SortedList collection stores key-value pairs in the ascending order of key by default. SortedList class implements IDictionary & ICollection interfaces, so elements can be accessed both by key and index.

Important Properties and Methods of SortedList:

Property	Description
Capacity	Gets or sets the number of elements that the SortedList instance can store.
Count	Gets the number of elements actually contained in the SortedList.
IsFixedSize	Gets a value indicating whether the SortedList has a fixed size.

<b>IsReadOnly</b>	Gets a value indicating whether the SortedList is read-only.
<b>Item</b>	Gets or sets the element at the specified key in the SortedList.
<b>Keys</b>	Get list of keys of SortedList.
<b>Values</b>	Get list of values in SortedList.

Method	Description
<b>void Add(object key, object value)</b>	Add key-value pairs into SortedList.
<b>void Remove(object key)</b>	Removes element with the specified key.
<b>void RemoveAt(int index)</b>	Removes element at the specified index.
<b>bool Contains(object key)</b>	Checks whether specified key exists in SortedList.
<b>void Clear()</b>	Removes all the elements from SortedList.
<b>object GetByIndex(int index)</b>	Returns the value by index stored in internal array
<b>object GetKey(int index)</b>	Returns the key stored at specified index in internal array
<b>int IndexOfKey(object key)</b>	Returns an index of specified key stored in internal array
<b>int IndexOfValue(object value)</b>	Returns an index of specified value stored in internal array

Add elements in SortedList:

Use the Add() method to add key-value pairs into a SortedList.

Key cannot be null but value can be null. Also, datatype of all keys must be same, so that it can compare otherwise it will throw runtime exception.

```
SortedList sortedList1 = new SortedList();
sortedList1.Add(3, "Three");
sortedList1.Add(4, "Four");
sortedList1.Add(1, "One");
sortedList1.Add(5, "Five");
sortedList1.Add(2, "Two");
SortedList sortedList2 = new SortedList();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
SortedList sortedList3 = new SortedList();
sortedList3.Add(1.5, 100);
sortedList3.Add(3.5, 200);
sortedList3.Add(2.4, 300);
sortedList3.Add(2.3, null);
sortedList3.Add(1.1, null);
```

SortedList key can be of any data type, but you cannot add keys of different data types in the same SortedList. The following example will throw runtime exception because we are trying to add the second item with a string key:

```
SortedList sortedList = new SortedList();
sortedList.Add(3, "Three");
sortedList.Add("Four", "Four"); // Throw
exception: InvalidOperationException
sortedList.Add(1, "One");
sortedList.Add(8, "Five");
sortedList.Add(2, "Two");
```

Access SortedList:

SortedList can be accessed by index or key. Unlike other collection, SortedList requires key instead of index to access a value for that key.

```
SortedList sortedList = new SortedList();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", "Four");
int i = (int) sortedList["one"];
int j = (int) sortedList["two"];
string str = (string)sortedList["four"];
Console.WriteLine(i);
Console.WriteLine(j);
Console.WriteLine(str);
```

Access SortedList using foreach loop:

```
SortedList sortedList1 = new SortedList();
sortedList1.Add("one", 1);
sortedList1.Add("two", 2);
sortedList1.Add("three", 3);
sortedList1.Add("four", 4);
foreach(DictionaryEntry kvp in sortedList1 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );
```

Remove elements from SortedList:

Use the Remove() or RemoveAt() method to remove elements from a SortedList.

Remove() signature: `void Remove(object key)`

RemoveAt() signature: `void RemoveAt(int index)`

```
SortedList sortedList1 = new SortedList();
sortedList1.Add("one", 1);
sortedList1.Add("two", 2);
sortedList1.Add("three", 3);
sortedList1.Add("four", 4);
sortedList1.Remove("one");//removes element whose key is 'one'
sortedList1.RemoveAt(0);//removes element at zero index i.e first
element: four
foreach(DictionaryEntry kvp in sortedList1 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );
```

Check for existing key in SortedList:

The Contains() & ContainsKey() methods determine whether the specified key exists in the SortedList collection or not.

Contains() signature: `bool Contains(object key)`

ContainsKey() signature: `bool ContainsKey(object key)`

The ContainsValue() method determines whether the specified value exists in the SortedList or not.

ContainValue() signature: `bool ContainValue(object value)`

```
SortedList sortedList = new SortedList();
sortedList.Add(3, "Three");
sortedList.Add(2, "Two");
sortedList.Add(4, "Four");
sortedList.Add(1, "One");
sortedList.Add(8, "Five");
sortedList.Contains(2); // returns true
sortedList.Contains(4); // returns
true
sortedList.Contains(6); //returns false
sortedList.ContainsKey(2); // returns
true
sortedList.ContainsKey(6); // returns
false
sortedList.ContainsValue("One"); // returns
true
sortedList.ContainsValue("Ten"); // returns false
```

Page Break

## 23-Stack

C# includes a special type of collection which stores elements in LIFO style (Last In First Out). C# includes a generic and non-generic Stack. Here, you are going to learn about the non-generic stack.

Stack allows null value and also duplicate values. It provides a Push() method to add a value and Pop() or Peek() methods to retrieve values.

## Important Properties and Methods of Stack:

Property	Usage
<b>Count</b>	Returns the total count of elements in the Stack.
Method	Usage
<u>Push</u>	Inserts an item at the top of the stack.
<u>Peek</u>	Returns the top item from the stack.
<u>Pop</u>	Removes and returns items from the top of the stack.
<u>Contains</u>	Checks whether an item exists in the stack or not.
<u>Clear</u>	Removes all items from the stack.

## Add values into Stack:

The Push() method adds values into the Stack. It allows value of any datatype.

```
Stack myStack = new Stack();
myStack.Push("Hello!!");
myStack.Push(null);
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
```

## Access Stack Elements:

You can retrieve stack elements by various ways. Use a foreach statement to iterate the Stack collection and get all the elements in LIFO style.

```
Stack myStack = new Stack();
myStack.Push("Hello!!");
myStack.Push(null);
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
foreach (var itm in myStack)
    Console.Write(itm);
Peek():
```

The Peek() method returns the last (top-most) value from the stack. Calling Peek() method on empty stack will throw InvalidOperationException. So always check for elements in the stack before retrieving elements using the Peek() method.

```
Stack myStack = new Stack();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
Console.WriteLine(myStack.Peek());
Console.WriteLine(myStack.Peek());
Console.WriteLine(myStack.Peek());
```

Pop():

You can also retrieve the value using the Pop() method. The Pop() method removes and returns the value that was added last to the Stack. The Pop() method call on an empty stack will raise an InvalidOperationException. So always check for number of elements in stack must be greater than 0 before calling Pop() method.

```
Stack myStack = new Stack();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
Console.WriteLine("Number of elements in Stack: {0}", myStack.Count);
while (myStack.Count > 0)
    Console.WriteLine(myStack.Pop());
Console.WriteLine("Number of elements in Stack: {0}", myStack.Count);
```

Contains

The Contains() method checks whether the specified item exists in a Stack collection or not. It returns true if it exists; otherwise it returns false.

Contains() method signature: `bool Contains(object obj);`

```
Stack myStack = new Stack();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
myStack.Contains(2); // returns true
myStack.Contains(10); // returns false
```



Clear:

The Clear() method removes all the values from the stack.

Clear() signature: `void Clear();`

```
Stack myStack = new Stack();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);
myStack.Push(5);
myStack.Clear(); // removes all elements
Console.WriteLine("Number of elements in Stack: {0}", myStack.Count);
```

## 24-Queue

C# includes a Queue collection class in the *System.Collection* namespace. Queue stores the elements in FIFO style (First In First Out), exactly opposite of the [Stack](#) collection. It contains the elements in the order they were added.

Queue collection allows multiple null and duplicate values. Use the Enqueue() method to add values and the Dequeue() method to retrieve the values from the Queue.

Important Properties and Methods of Queue:

Property	Usage
<b>Count</b>	Returns the total count of elements in the Queue.

Method	Usage
<a href="#">Enqueue</a>	Adds an item into the queue.
<a href="#">Dequeue</a>	Removes and returns an item from the beginning of the queue.
<a href="#">Peek</a>	Returns an first item from the queue
<a href="#">Contains</a>	Checks whether an item is in the queue or not
<a href="#">Clear</a>	Removes all the items from the queue.
<b>TrimToSize</b>	Sets the capacity of the queue to the actual number of items in the queue.

Add elements in Queue:

Queue is a non-generic collection. So you can add elements of any datatype into a queue using the Enqueue() method.

```
Queue queue = new Queue();
queue.Enqueue(3);
queue.Enqueue(2);
queue.Enqueue(1);
queue.Enqueue("Fuor");
```

Access Queue:

Dequeue() method is used to retrieve the top most element in a queue collection. Dequeue() removes and returns a first element from a queue because the queue stores elements in FIFO order. Calling Dequeue() method on empty queue will throw InvalidOperationException exception. So always check that the total count of a queue is greater than zero before calling the Dequeue() method on a queue.

```
Queue queue = new Queue();
queue.Enqueue(3);
queue.Enqueue(2);
queue.Enqueue(1);
queue.Enqueue("Four");
```

```
Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);
```

```
while (queue.Count > 0)
    Console.WriteLine(queue.Dequeue());
```

```
Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);
```

## Peek()

The Peek() method always returns the first item from a queue collection without removing it from the queue. Calling Peek() and Dequeue() methods on an empty queue collection will throw a run time exception "InvalidOperationException".

```
Queue queue = new Queue();
queue.Enqueue(3);
queue.Enqueue(2);
queue.Enqueue(1);
queue.Enqueue("Four");
```

```
Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);
```

```
Console.WriteLine(queue.Peek());
Console.WriteLine(queue.Peek());
Console.WriteLine(queue.Peek());
```

```
Console.WriteLine("Number of elements in the Queue:  
{0}", queue.Count);
```

### Contains():

The Contains() method checks whether an item exists in a queue. It returns true if the specified item exists; otherwise it returns false.

```
Queue queue = new Queue();  
queue.Enqueue(3);  
queue.Enqueue(2);  
queue.Enqueue(1);  
queue.Enqueue("Four");
```

```
queue.Contains(2); // true  
queue.Contains(100); //false
```

### CLEAR():

The Clear() method removes all the items from a queue.

```
Queue queue = new Queue();  
queue.Enqueue(3);  
queue.Enqueue(2);  
queue.Enqueue(1);  
queue.Enqueue("Four");
```

```
Console.WriteLine("Number of elements in the Queue:  
{0}", queue.Count);
```

```
queue.Clear();
```

```
Console.WriteLine("Number of elements in the Queue:  
{0}", queue.Count);
```

## 25-HashTable

C# includes Hashtable collection in *System.Collections* namespace, which is similar to generic [Dictionary](#) collection. The Hashtable collection stores key-value pairs. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

### Important Properties and Methods of Hashtable:

Property	Description
Count	Gets the total count of key/value pairs in the Hashtable.
IsReadOnly	Gets boolean value indicating whether the Hashtable is read-only.
Item	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection of keys in the Hashtable.
Values	Gets an ICollection of values in the Hashtable.

Methods	Usage
<a href="#">Add</a>	Adds an item with a key and value into the hashtable.
<a href="#">Remove</a>	Removes the item with the specified key from the hashtable.
<a href="#">Clear</a>	Removes all the items from the hashtable.
<a href="#">Contains</a>	Checks whether the hashtable contains a specific key.
<a href="#">ContainsKey</a>	Checks whether the hashtable contains a specific key.
<a href="#">ContainsValue</a>	Checks whether the hashtable contains a specific value.
<a href="#">GetHash</a>	Returns the hash code for the specified key.

## Add key-value into Hashtable:

The Add() method adds an item with a key and value into the Hashtable. Key and value can be of any data type. Key cannot be null whereas value can be null.

```
Hashtable ht = new Hashtable();

ht.Add(1, "One");
ht.Add(2, "Two");
ht.Add(3, "Three");
ht.Add(4, "Four");
ht.Add(5, null);
ht.Add("Fv", "Five");
ht.Add(8.5F, 8.5);
```

You can also assign key and value at the time of initialization using object initializer syntax:

```
Hashtable ht = new Hashtable()
{
    { 1, "One" },
    { 2, "Two" },
    { 3, "Three" },
    { 4, "Four" },
    { 5, null },
    { "Fv", "Five" },
    { 8.5F, 8.5 }
};
```

Hashtable can include all the elements of Dictionary as shown below.

```
Dictionary<int, string> dict = new Dictionary<int, string>();

dict.Add(1, "one");
dict.Add(2, "two");
dict.Add(3, "three");
Hashtable ht = new Hashtable(dict);
```

## Access Hashtable:

You can retrieve the value of an existing key from the Hashtable using indexer. Please note that the hashtable indexer requires a key.

```

Hashtable ht = new Hashtable();

ht.Add(1, "One");
ht.Add(2, "Two");
ht.Add(3, "Three");
ht.Add(4, "Four");
ht.Add("Fv", "Five");
ht.Add(8.5F, 8.5F);
string strValue1 = (string)ht[2];string strValue2 = (string)ht["Fv"];
float fValue = (float) ht[8.5F];
Console.WriteLine(strValue1);
Console.WriteLine(strValue2);
Console.WriteLine(fValue);

```

Hashtable elements are key-value pairs stored in DictionaryEntry. So you cast each element in Hashtable to DictionaryEntry. Use the foreach statement to iterate the Hashtable, as shown below:

```

Hashtable ht = new Hashtable();

ht.Add(1, "One");
ht.Add(2, "Two");
ht.Add(3, "Three");
ht.Add(4, "Four");
ht.Add("Fv", "Five");
ht.Add(8.5F, 8.5);
foreach (DictionaryEntry item in ht)
    Console.WriteLine("key:{0}, value:{1}",item.Key, item.Value);

```

Hashtable has a Keys and a Values property that contain all the keys and values respectively. You can use these properties to get the keys and values.

```

Hashtable ht = new Hashtable();
ht.Add(1, "One");
ht.Add(2, "Two");
ht.Add(3, "Three");
ht.Add(4, "Four");
ht.Add("Fv", "Five");
ht.Add(8.5F, 8.5);
foreach (var key in ht.Keys ){
    Console.WriteLine("Key:{0}, Value:{1}",key , ht[key]);}
Console.WriteLine("***All Values***");
foreach (var value in ht.Values){
    Console.WriteLine("Value:{0}", value);}

```

*Remove elements in Hashtable:*

The Remove() method removes the item with the specified key from the hashtable.

```
Hashtable ht = new Hashtable();  
ht.Add(1, "One");  
ht.Add(2, "Two");  
ht.Add(3, "Three");  
ht.Add(4, "Four");  
ht.Add("Fv", "Five");  
ht.Add(8.5F, 8.5);  
  
ht.Remove("Fv"); // removes {"Fv", "Five"}
```

## Check for existing elements:

Contains() and ContainsKey() check whether the specified key exists in the Hashtable collection. ContainsValue() checks whether the specified value exists in the Hashtable.

Contains(), ContainsKey() and ContainsValue() Signatures:

```
bool Contains(object key)
```

```
bool ContainsKey(object key)
```

```
bool ContainsValue(object value)
```

```
Hashtable ht = new Hashtable();  
ht.Add(1, "One");  
ht.Add(2, "Two");  
ht.Add(3, "Three");  
ht.Add(4, "Four");  
  
ht.Contains(2); // returns true  
ht.ContainsKey(2); // returns true  
ht.Contains(5); // returns false  
ht.ContainsValue("One"); // returns true
```

## Clear():

Clear() method removes all the key-value pairs in the Hashtable.

```
Hashtable ht = new Hashtable();  
ht.Add(1, "One");  
ht.Add(2, "Two");  
ht.Add(3, "Three");
```

```
ht.Add(4, "Four");  
ht.Add("Fv", "Five");  
ht.Add(8.5F, 8.5);  
  
ht.Clear(); // removes all elements  
Console.WriteLine("Total Elements: {0}", ht.Count);
```



## 26-Indexer

An Indexer is a special type of property that allows a class or structure to be accessed the same way as array for its internal collection. It is same as property except that it defined with **this** keyword with square bracket and paramters.

```
class StringDataStore
{
    private string[] strArr = new string[10]; // internal data storage

    public StringDataStore()
    {
    }

    public string this[int index]
    {
        get
        {
            if (index < 0 && index >= strArr.Length)
                throw new IndexOutOfRangeException("Cannot store more than 10 objects");

            return strArr[index];
        }
        set
        {
            if (index < 0 && index >= strArr.Length)
                throw new IndexOutOfRangeException("Cannot store more than 10 objects");

            strArr[index] = value;
        }
    }

    public string this[string name]
    {
        get
        {
            foreach (string str in strArr){
                if(str.ToLower() == name.ToLower())
                    return str;
            }

            return null;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        StringDataStore strStore = new StringDataStore();

        strStore[0] = "One";
        strStore[1] = "Two";
        strStore[2] = "Three";
        strStore[3] = "Four";

        Console.WriteLine(strStore["one"]);
        Console.WriteLine(strStore["two"]);
        Console.WriteLine(strStore["Three"]);
        Console.WriteLine(strStore["FOUR"]);
    }
}
```

## Override Indexer:

You can override an indexer by having different index types. The following example shows how an indexer can be of int type as well as string type.

```
class StringDataStore
{
    private string[] strArr = new string[10]; // internal data storage

    public StringDataStore()
    {
    }

    public string this[int index]
    {
        get
        {
            if (index < 0 && index >= strArr.Length)
                throw new IndexOutOfRangeException("Cannot store more than 10 objects");

            return strArr[index];
        }
        set
        {
            if (index < 0 && index >= strArr.Length)
                throw new IndexOutOfRangeException("Cannot store more than 10 objects");

            strArr[index] = value;
        }
    }

    public string this[string name]
    {
        get
        {
            foreach (string str in strArr){
                if(str.ToLower() == name.ToLower())
                    return str;
            }

            return null;
        }
    }
}

class Program
```

```
{
    static void Main(string[] args)
    {
        StringDataStore strStore = new StringDataStore();

        strStore[0] = "One";
        strStore[1] = "Two";
        strStore[2] = "Three";
        strStore[3] = "Four";

        Console.WriteLine(strStore["one"]);
        Console.WriteLine(strStore["two"]);
        Console.WriteLine(strStore["Three"]);
        Console.WriteLine(strStore["FOUR"]);
    }
}
```

## 27-Stream

C# includes following standard IO (Input/Output) classes to read/write from different sources like a file, memory, network, isolated storage, etc.

**Stream:** *System.IO.Stream* is an abstract class that provides standard methods to transfer bytes (read, write, etc.) to the source. It is like a wrapper class to transfer bytes. Classes that need to read/write bytes from a particular source must implement the Stream class.

The following classes inherits Stream class to provide functionality to Read/Write bytes from a particular source:

**FileStream** reads or writes bytes from/to a physical file whether it is a .txt, .exe, .jpg or any other file. FileStream is derived from the Stream class.

**MemoryStream:** MemoryStream reads or writes bytes that are stored in memory.

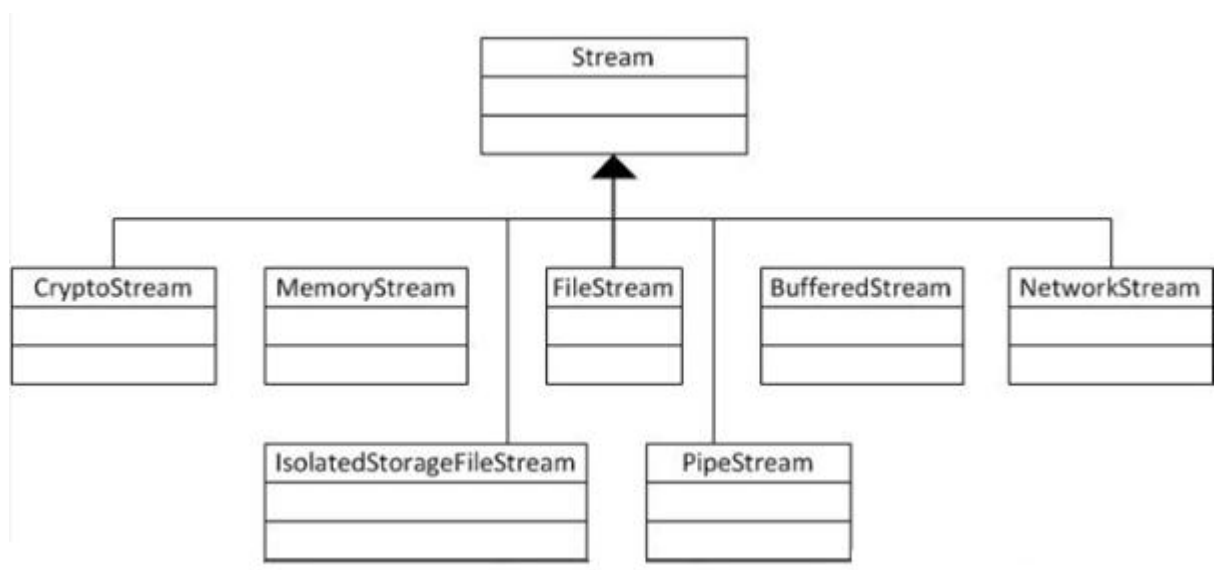
**BufferedStream:** BufferedStream reads or writes bytes from other Streams to improve the performance of certain I/O operations.

**NetworkStream:** NetworkStream reads or writes bytes from a network socket.

**PipeStream:** PipeStream reads or writes bytes from different processes.

**CryptoStream:** CryptoStream is for linking data streams to cryptographic transformations.

The following diagram shows the hierarchy of stream classes:



# Readers and Writers:

**Stream Reader:** Stream Reader is a helper class for reading characters from a Stream by converting bytes into characters using an encoded value. It can be used to read strings (characters) from different Streams like File Stream, Memory Stream, etc.

**Stream Writer:** Stream Writer is a helper class for writing a string to a Stream by converting characters into bytes. It can be used to write strings to different Streams such as File Stream, Memory Stream, etc.

**Binary Reader:** Binary Reader is a helper class for reading primitive data type from bytes.

**Binary Writer:** Binary Writer writes primitive types in binary.



# 28 -Working with Files & Directories:

C# provides the following classes to work with the File system. They can be used to access directories, access files, open files for reading or writing, create a new file or move existing files from one location to another, etc.

Class Name	Usage
File	File is a static class that provides different functionalities like copy, create, move, delete, open for reading or /writing, encrypt or decrypt, check if a file exists, append lines or text to a file's content, get last access time, etc.

<i>FileInfo</i>	<i>The File Info class provides the same functionality as a static File class. You have more control on how you do read/write operations on a file by writing code manually for reading or writing bytes from a file.</i>
<i>Directory</i>	<i>Directory is a static class that provides functionality for creating, moving, deleting and accessing subdirectories.</i>
<i>DirectoryInfo</i>	<i>DirectoryInfo provides instance methods for creating, moving, deleting and accessing subdirectories.</i>
<i>Path</i>	<i>Path is a static class that provides functionality such as retrieving the extension of a file, changing the extension of a file, retrieving the absolute physical path, and other path related functionalities.</i>

## File

C# includes static **File** class to perform I/O operation on physical file system. The static File class includes various utility method to interact with physical file of any type e.g. binary, text etc.

Use this static File class to perform some quick operation on physical file. It is not recommended to use File class for multiple operations on multiple files at the same time due to performance reasons. Use FileInfo class in that scenario.

## Important Methods of Static File Class:

<i>Method</i>	<i>Usage</i>
<i>AppendAllLines</i>	<i>Appends lines to a file, and then closes the file. If the specified file does not exist, this method creates a file, writes the specified lines to the file, and then closes the file.</i>
<i>AppendAllText</i>	<i>Opens a file, appends the specified string to the file, and then closes the file. If the file does not exist, this method creates a file, writes the specified string to the file, then closes the file.</i>
<i>AppendText</i>	<i>Creates a StreamWriter that appends UTF-8 encoded text to an existing file, or to a new file if the specified file does not exist.</i>

<i>Copy</i>	<i>Copies an existing file to a new file. Overwriting a file of the same name is not allowed.</i>
<i>Create</i>	<i>Creates or overwrites a file in the specified path.</i>
<i>CreateText</i>	<i>Creates or opens a file for writing UTF-8 encoded text.</i>
<i>Decrypt</i>	<i>Decrypts a file that was encrypted by the current account using the Encrypt method.</i>
<i>Delete</i>	<i>Deletes the specified file.</i>
<i>Encrypt</i>	<i>Encrypts a file so that only the account used to encrypt the file can decrypt it.</i>
<i>Exists</i>	<i>Determines whether the specified file exists.</i>
<i>GetAccessControl</i>	<i>Gets a FileSecurity object that encapsulates the access control list (ACL) entries for a specified file.</i>
<i>Move</i>	<i>Moves a specified file to a new location, providing the option to specify a new file name.</i>
<i>Open</i>	<i>Opens a FileStream on the specified path with read/write access.</i>
<i>ReadAllBytes</i>	<i>Opens a binary file, reads the contents of the file into a byte array, and then closes the file.</i>
<i>ReadAllLines</i>	<i>Opens a text file, reads all lines of the file, and then closes the file.</i>
<i>ReadAllText</i>	<i>Opens a text file, reads all lines of the file, and then closes the file.</i>
<i>Replace</i>	<i>Replaces the contents of a specified file with the contents of another file, deleting the original file, and creating a backup of the replaced file.</i>
<i>WriteAllBytes</i>	<i>Creates a new file, writes the specified byte array to the file, and then closes the file. If the target file already exists, it is overwritten.</i>
<i>WriteAllLines</i>	<i>Creates a new file, writes a collection of strings to the file, and then closes the file.</i>
<i>WriteAllText</i>	<i>Creates a new file, writes the specified string to the file, and then closes the file. If the target file already exists, it is overwritten.</i>

## Append text lines:

Use `AppendAllLines()` method to append multiple text lines to the specified file as shown below.



```

string dummyLines = "This is first line." + Environment.NewLine +
                    "This is second line." + Environment.NewLine +
                    "This is third line.";
/*Opens DummyFile.txt and append lines. If file is not exists then
create and open.*/
File.AppendAllLines
(
@"C:\DummyFile.txt",
dummyLines
.Split(Environment.NewLine.ToCharArray())
.ToList<string>()
);

```

## Append text string:

Use `File.AppendAllText()` method to append string to a file in single line of code as shown below.

```

//Opens DummyFile.txt and append Text. If file is not exists then cre
ate and open.File.AppendAllText(
@"C:\ DummyFile.txt",
"This is File testing");

```

## Overwrite existing texts:

Use `File.WriteAllText()` method to write texts to the file. Please note that it will not append text but overwrite existing texts.

```

//Opens DummyFile.txt and write texts. If file is not exists then cre
ate and open.File.WriteAllText(
@"C:\DummyFile.txt"
, "This is dummy text");

```

```

//Check whether file is exists or not at particular location
bool isFileExists = File.Exists(@"C:\ DummyFile.txt");
// returns false
//Copy DummyFile.txt as new file DummyFileNew.txt
File.Copy(@"C:\DummyFile.txt", @"D:\NewDummyFile.txt");
//Get when the file was accessed last time
DateTime lastAccessTime = File.GetLastAccessTime
(@"C:\DummyFile.txt");
//get when the file was written last time
DateTime lastWriteTime = File.GetLastWriteTime(@"C:\DummyFile.txt");
// Move file to new location
File.Move(@"C:\DummyFile.txt", @"D:\DummyFile.txt");
//Open file and returns FileStream for reading bytes from the file
FileStream fs = File.Open(@"D:\DummyFile.txt",

```

```
FileMode.OpenOrCreate);  
//Open file and return StreamReader for reading string from the file  
StreamReader sr = File.OpenText(@"D:\DummyFile.txt");  
//Delete file  
File.Delete(@"C:\DummyFile.txt");
```

## 29-C# FileInfo

You have learned how to perform different tasks on physical files using static File class in the previous section. Here, we will use FileInfo class to perform read/write operation on physical files.

The FileInfo class provides the same functionality as the static File class but you have more control on read/write operations on files by writing code manually for reading or writing bytes from a file.

### Important properties and methods of FileInfo:

Property	Usage
Directory	<i>Gets an instance of the parent directory.</i>
DirectoryName	<i>Gets a string representing the directory's full path.</i>
Exists	<i>Gets a value indicating whether a file exists.</i>
Extension	<i>Gets the string representing the extension part of the file.</i>
FullName	<i>Gets the full path of the directory or file.</i>
IsReadOnly	<i>Gets or sets a value that determines if the current file is read only.</i>
LastAccessTime	<i>Gets or sets the time the current file or directory was last accessed</i>
LastWriteTime	<i>Gets or sets the time when the current file or directory was last written to</i>
Length	<i>Gets the size, in bytes, of the current file.</i>
Name	<i>Gets the name of the file.</i>

Method	Usage
AppendText	<i>Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo.</i>
CopyTo	<i>Copies an existing file to a new file, disallowing the overwriting of an existing file.</i>

Create	Creates a file.
CreateText	Creates a StreamWriter that writes a new text file.
Decrypt	Decrypts a file that was encrypted by the current account using the Encrypt method.
Delete	Deletes the specified file.
Encrypt	Encrypts a file so that only the account used to encrypt the file can decrypt it.
GetAccessControl	Gets a FileSecurity object that encapsulates the access control list (ACL) entries for a specified file.
MoveTo	Moves a specified file to a new location, providing the option to specify a new file name.
Open	Opens a in the specified FileMode.
OpenRead	Creates a read-only FileStream.
OpenText	Creates a StreamReader with UTF8 encoding that reads from an existing text file.
OpenWrite	Creates a write-only FileStream.
Replace	Replaces the contents of a specified file with the file described by the current FileInfo object, deleting the original file, and creating a backup of the replaced file.
ToString	Returns a path as string.

The following example shows how to read bytes from a file manually and then convert them to a string using UTF8 encoding:

```
//Create object of FileInfo for specified path
    FileInfo fi = new FileInfo(@"D:\DummyFile.txt");
//Open file for Read\Write
FileStream fs = fi.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
    FileShare.ReadWrite);
//create byte array of same size as FileStream length
byte[] fileBytes = new byte[fs.Length];
/*define counter to check how much bytes to read.Decrease the counter
as you read each byte*/
int numBytesToRead = (int)fileBytes.Length;
//Counter to indicate number of bytes already read
int numBytesRead = 0;
//iterate till all the bytes read from FileStream
while (numBytesToRead > 0)
{
    int n = fs.Read(fileBytes, numBytesRead, numBytesToRead);
    if (n == 0)
        break;
}
```

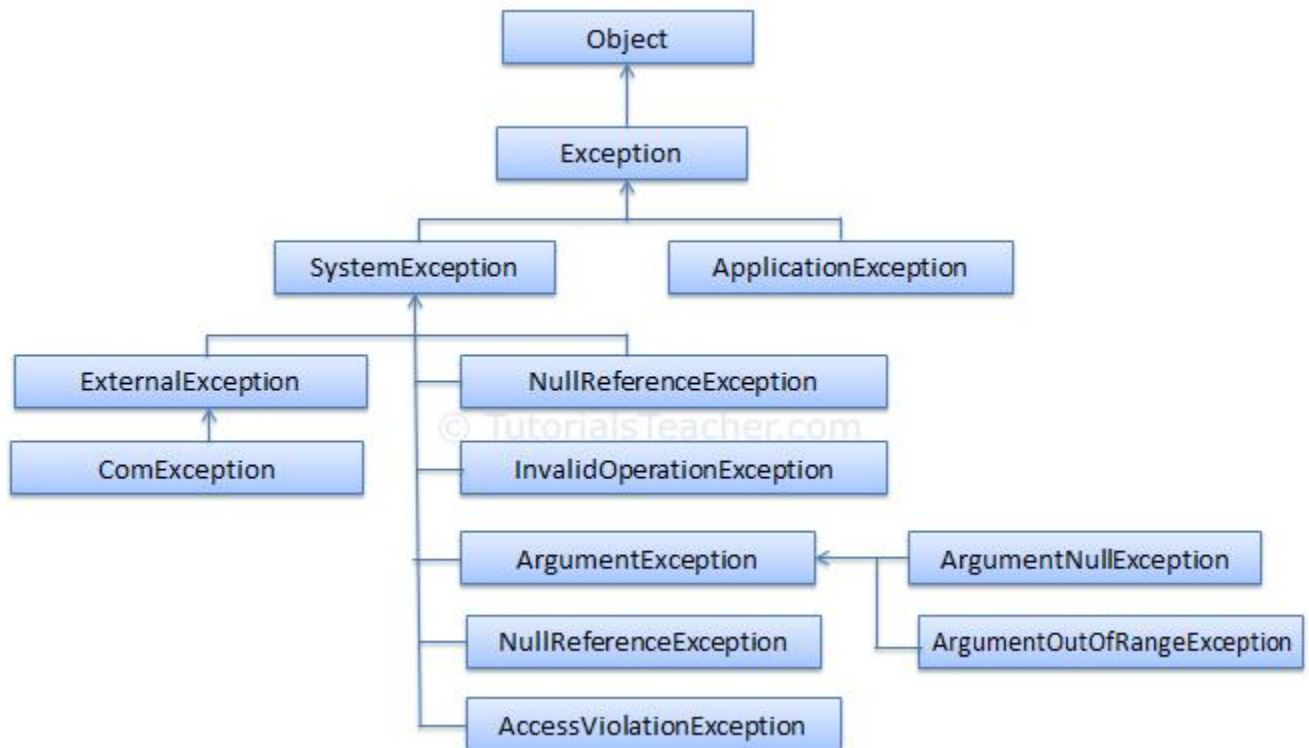
```
    numBytesRead += n;  
    numBytesToRead -= n;  
}  
//Once you read all the bytes from FileStream, you can convert it into string using UTF8 encoding  
string filestring = Encoding.UTF8.GetString(fileBytes);
```

## 30-Exception

An application may encounter an error during the execution. When an error occurs, either CLR or program code throws an exception which contains necessary information about the error. There are two types of exceptions in .Net, exceptions generated by the executing program and exceptions generated by the CLR.

C# includes built-in classes for every possible exception. All the exception classes are directly or indirectly derived from the **Exception** class. There are two main classes for exceptions - **SystemException** and **ApplicationException**. SystemException is a base class for all CLR generated errors whereas ApplicationException serves as a base class for all application related exceptions, which you want to raise on business rule violation.

The following is a hierarchy of some of the exception classes in .Net:



*Important Exceptions:*

Exception	Description
ArgumentException	Raised when a non-null argument that is passed to a method is invalid.
ArgumentNullException	Raised when null argument is passed to a method.
ArgumentOutOfRangeException	Raised when the value of an argument is outside the range of valid values.
DivideByZeroException	Raised when an integer value is divide by zero.
FileNotFoundException	Raised when a physical file does not exist at the specified location.
FormatException	Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse.
IndexOutOfRangeException	Raised when an array index is outside the lower or upper bounds of an array or collection.
InvalidOperationException	Raised when a method call is invalid in an object's current state.
InvalidCastException	Raised when incompatible types are being converted.
KeyNotFoundException	Raised when the specified key for accessing a member in a collection is not exists.
NotSupportedException	Raised when a method or operation is not supported.
NullReferenceException	Raised when program access members of null object.
OverflowException	Raised when an arithmetic, casting, or conversion operation results in an overflow.
OutOfMemoryException	Raised when a program does not get enough memory to execute the code.
StackOverflowException	Raised when a stack in memory overflows.
TimeoutException	The time interval allotted to an operation has expired.

## Exception Class:

Every exception class in .Net is derived from the base Exception class. It includes the following important properties using which you can use to get information about the exception when you handle the exception.

Property	Description
Message	Provides details about the cause of the exception.
StackTrace	Provides information about where the error occurred.
InnerException	Provides information about the series of exceptions that might have occurred.
HelpLink	This property can hold the help URL for a particular exception.
Data	This property can hold arbitrary data in key-value pairs.
TargetSite	Provides the name of the method where this exception was thrown

## Exception Handling :

We have seen in the previous section that an exception is thrown by the CLR or program code if there is an error in the program. These exceptions need to be handle to prevent crashing of program. C# provides built-in support to handle the exception using try, catch & finally block.

```
try
{
    // code that may raise exceptions
}
catch(Exception ex)
{
    // handle exception
}
```



```

}
finally
{
    // final cleanup code
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Console.Write("Enter Student Name: ");

        string studentName = Console.ReadLine();

        try
        {
            IList<String> studentList = FindAllStudentFromDatabase
                                      (studentName);

            Console.WriteLine("Total {0}: {1}",
                             studentName,
                             studentList.Count());
        }
        catch(Exception ex)
        {
            Console.Write("No Students exists for the specified name.
");
        }

        Console.ReadKey();
    }

    private static IList<String> FindAllStudentFromDatabase
                                   (string studentName)
    {
        var studentList =
        // find all students with same name from the database

        return studentList;
    }
}

```

# 31-Delegates

A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter? How does C# handle the callback functions or event handler? The answer is - **delegate**.

A delegate is like a pointer to a function. It is a reference type data type and it holds the reference of a method. All the delegates are implicitly derived from `System.Delegate` class.

*<access modifier> delegate <return type> <delegate\_name>(<parameters>)*

```
public delegate void Print(int value);
```

The Print delegate shown above, can be used to point to any method that has same return type & parameters declared with Print. Consider the following example that declares and uses Print delegate.

```
class Program
{
    // declare delegate
    public delegate void Print(int value);

    static void Main(string[] args)
    {
        // Print delegate points to PrintNumber
        Print printDel = PrintNumber;

        printDel(100000);
        printDel(200);

        // Print delegate points to PrintMoney
        printDel = PrintMoney;

        printDel(10000);
        printDel(200);
    }

    public static void PrintNumber(int num)
    {
```

```

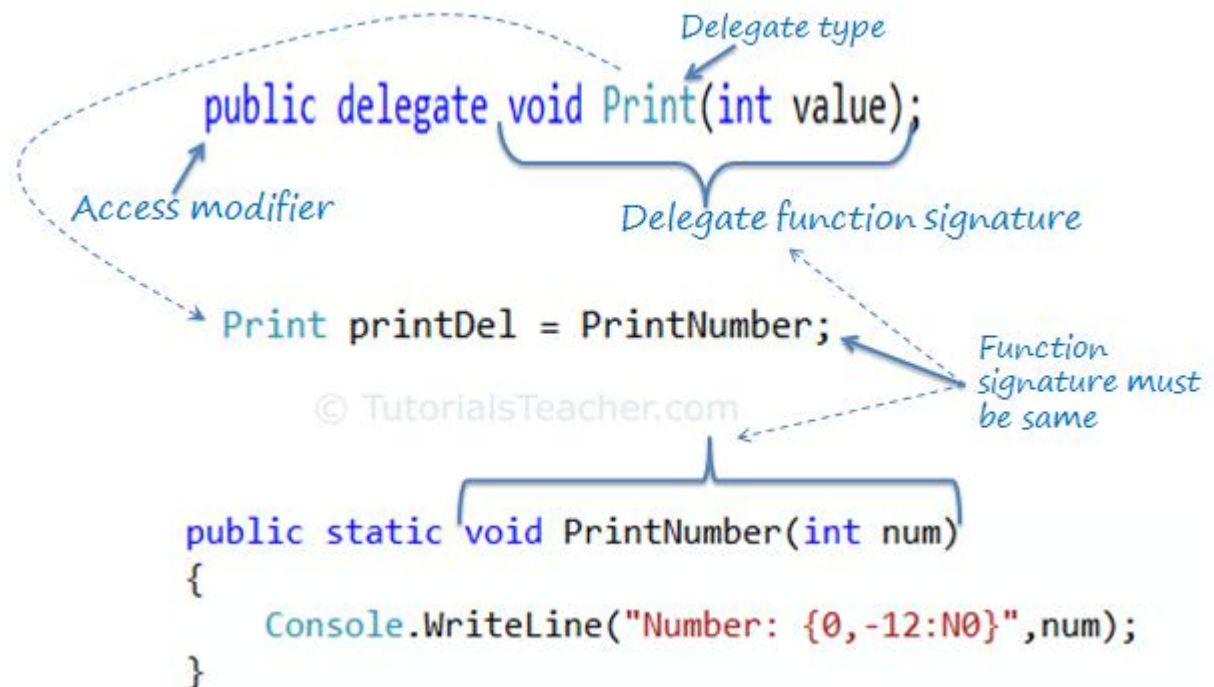
        Console.WriteLine("Number: {0,-12:N0}", num);
    }

    public static void PrintMoney(int money)
    {
        Console.WriteLine("Money: {0:C}", money);
    }
}

```

In the above example, we have declared Print delegate that accepts *int* type parameter and returns void. In the Main() method, a variable of Print type is declared and assigned a PrintNumber method name. Now, invoking Print delegate will in-turn invoke PrintNumber method. In the same way, if the Print delegate variable is assigned to the PrintMoney method, then it will invoke the PrintMoney method.

The following image illustrates the delegate.



## Invoking Delegate:

The delegate can be invoked like a method because it is a reference to a method. Invoking a delegate will in-turn invoke a method which is referred to. The delegate can be invoked by two ways: using () operator or using the Invoke() method of delegate as shown below.

```

Print printDel = PrintNumber;
printDel.Invoke(10000);
//or

```

```
printDel (10000);
```

## *Delegate as a parameter:*

A method can have a parameter of a delegate type and can invoke the delegate parameter.

```
public static void PrintHelper(Print delegateFunc, int numToPrint)
{
    delegateFunc(numToPrint);
}
```

## 32-Event

In general terms, an event is something special that is going to happen. For example, Microsoft launches events for developers, to make them aware about the features of new or existing products. Microsoft notifies the developers about the event by email or other advertisement options. So in this case, Microsoft is a publisher who launches (raises) an **event** and **notifies** the developers about it and developers are the **subscribers** of the event and attend (**handle**) the event.

Events in C# follow a similar concept. An event has a publisher, subscriber, notification and a handler. Generally, UI controls use events extensively. For example, the button control in a Windows form has multiple events such as click, mouseover, etc. A custom class can also have an event to notify other subscriber classes about something that has happened or is going to happen. Let's see how you can define an event and notify other classes that have event handlers.

An event is nothing but an encapsulated delegate. As we have learned in the previous section, a delegate is a reference type data type. You can declare the delegate as shown below:

```
public delegate void someEvent();  
public someEvent someEvent;
```

Now, to declare an event, use the **event** keyword before declaring a variable of delegate type, as below:

```
public delegate void someEvent();  
public event someEvent someEvent;
```

```
public class PrintHelper  
{  
    // declare delegate  
    public delegate void BeforePrint();  
  
    //declare event of type delegate  
    public event BeforePrint beforePrintEvent;
```

```
public PrintHelper()
{
}

public void PrintNumber(int num)
{
    //call delegate method before going to print
    if (beforePrintEvent != null)
        beforePrintEvent();

    Console.WriteLine("Number: {0,-12:N0}", num);
}

public void PrintDecimal(int dec)
{
    if (beforePrintEvent != null)
        beforePrintEvent();

    Console.WriteLine("Decimal: {0:G}", dec);
}

public void PrintMoney(int money)
{
    if (beforePrintEvent != null)
        beforePrintEvent();

    Console.WriteLine("Money: {0:C}", money);
}

public void PrintTemperature(int num)
{
    if (beforePrintEvent != null)
        beforePrintEvent();

    Console.WriteLine("Temperature: {0,4:N1} F", num);
}

public void PrintHexadecimal(int dec)
{
    if (beforePrintEvent != null)
        beforePrintEvent();

    Console.WriteLine("Hexadecimal: {0:X}", dec);
}
}
```

## 33-Generics

Generics introduced in C# 2.0. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

A generic class can be defined using angle brackets <>. For example, the following is a simple generic class with a generic member variable, generic method and property.

```
class MyGenericClass<T>
{
    private T genericMemberVariable;

    public MyGenericClass(T value)
    {
        genericMemberVariable = value;
    }

    public T genericMethod(T genericParameter)
    {
        Console.WriteLine("Parameter type: {0}, value: {1}", typeof
(T).ToString(),genericParameter);
        Console.WriteLine("Return type: {0}, value: {1}", typeof(T).T
oString(), genericMemberVariable);

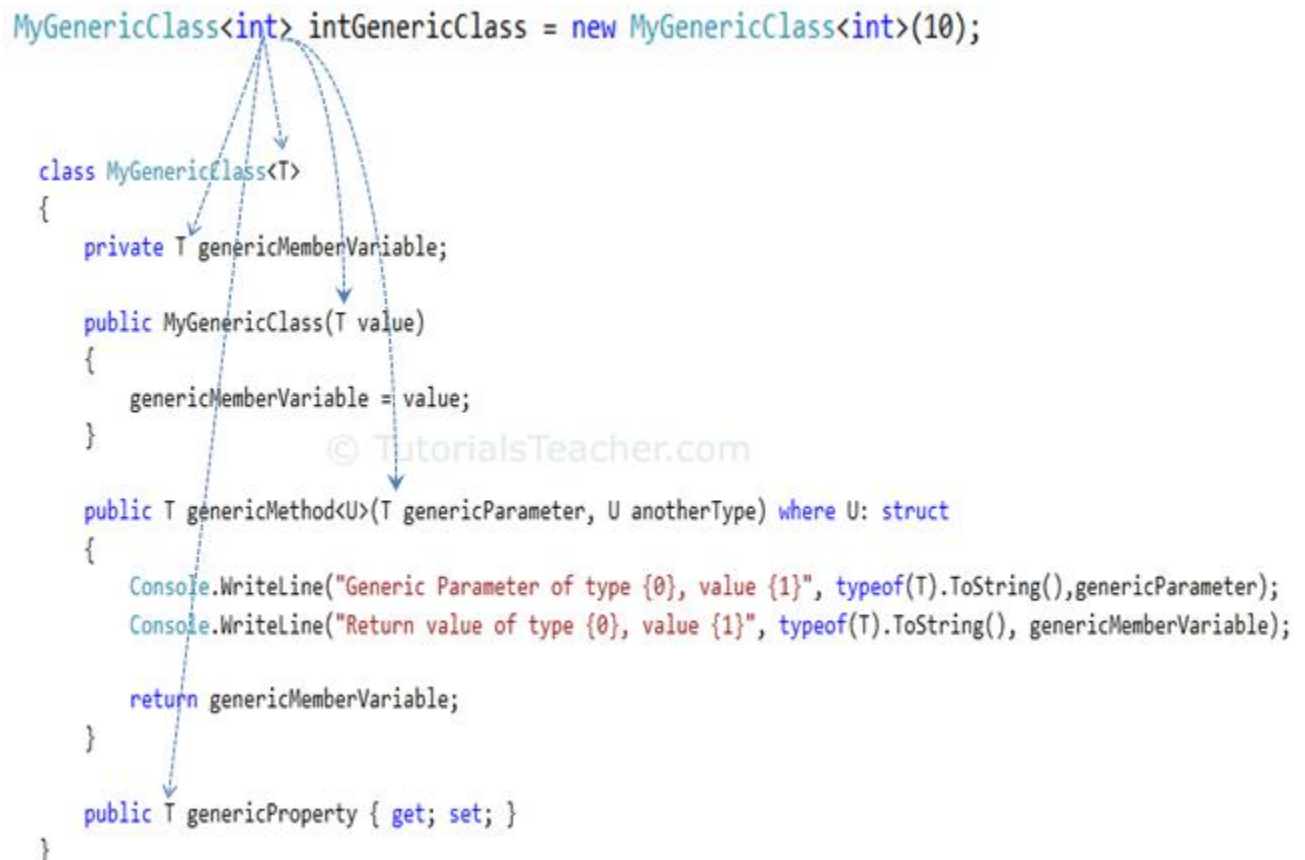
        return genericMemberVariable;
    }

    public T genericProperty { get; set; }
}
```

Now, the compiler assigns the type based on the type passed by the caller when instantiating a class. For example, the following code uses the int data type:

```
MyGenericClass<int> intGenericClass = new MyGenericClass<int>(10);  
int val = intGenericClass.genericMethod(200);
```

```
MyGenericClass<int> intGenericClass = new MyGenericClass<int>(10);  
  
class MyGenericClass<T>  
{  
    private T genericMemberVariable;  
  
    public MyGenericClass(T value)  
    {  
        genericMemberVariable = value;  
    }  
  
    public T genericMethod<U>(T genericParameter, U anotherType) where U: struct  
    {  
        Console.WriteLine("Generic Parameter of type {0}, value {1}", typeof(T).ToString(), genericParameter);  
        Console.WriteLine("Return value of type {0}, value {1}", typeof(T).ToString(), genericMemberVariable);  
  
        return genericMemberVariable;  
    }  
  
    public T genericProperty { get; set; }  
}
```



Generics can be applied to the following:

- Interface
- Abstract class
- Class
- Method
- Static method
- Property
- Event
- Delegates
- Operator



## *Advantages of Generic:*

1. Increases the reusability of the code.
2. Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition.
3. Generic has a performance advantage because it removes the possibilities of boxing and unboxing

## 34-Generic Collections

You have learned about the [collection](#) in the previous section, e.g. [ArrayList](#), [BitArray](#), [SortedList](#), [Queue](#), [Stack](#) and [Hashtable](#). These types of collections can store any type of items. For example, ArrayList can store items of different data types:

```
ArrayList arList = new ArrayList();

arList.Add(1);
arList.Add("Two");
arList.Add(true);
arList.Add(100.45);
arList.Add(DateTime.Now);
```

The limitation of these collections is that while retrieving items, you need to cast into the appropriate data type, otherwise the program will throw a runtime exception. It also affects on performance, because of boxing and unboxing.

To overcome this problem, C# includes generic collection classes in the ***System.Collections.Generic*** namespace.

The following are widely used generic collections:

Generic Collections	Description
<a href="#">List&lt;T&gt;</a>	Generic List<T> contains elements of specified type. It grows automatically as you add elements in it.
<a href="#">Dictionary&lt;TKey,TValue&gt;</a>	Dictionary<TKey,TValue> contains key-value pairs.
<a href="#">SortedList&lt;TKey,TValue&gt;</a>	SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default.
<a href="#">Hashset&lt;T&gt;</a>	Hashset<T> contains non-duplicate elements. It eliminates duplicate elements.
<a href="#">Queue&lt;T&gt;</a>	Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection.

*Stack<T>*

*Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values*

## 35-List<T>:

You have already learned about ArrayList in the previous section. An ArrayList resizes automatically as it grows. The List<T> collection is the same as an ArrayList except that List<T> is a generic collection whereas ArrayList is a non-generic collection.

List<T> can be initialized in the following two ways.

```
List<int> intList = new List<int>();  
//Or  
IList<int> intList = new List<int>();
```

In the above example, the first statement uses List type variable, whereas the second statement uses IList type variable to initialize List. List<T> is a concrete implementation of IList<T> interface. In the object-oriented programming, it is advisable to program to interface rather than concrete class. So use IList<T> type variable to create an object of List<T>.

List<T> includes more helper methods than IList<T> interface. The table shown below lists important properties and methods of List<T>, which are initialized using a List<T>:

Property	Usage
Items	<i>Gets or sets the element at the specified index</i>
Count	<i>Returns the total number of elements exists in the List&lt;T&gt;</i>
Method	Usage
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the specified element exists or not in a List<T>.

Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
InsertRange	Inserts elements of another collection at the specified index.
Remove	Removes the first occurrence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match with the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.
TrueForAll	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

## Add Elements into List:

Use the Add() method to add an element into a List collection. The following example adds int value into a List<T> of *int* type.

```
IList<int> intList = new List<int>();
intList.Add(10);
intList.Add(20);
intList.Add(30);
intList.Add(40);
IList<string> strList = new List<string>();
strList.Add("one");
strList.Add("two");
strList.Add("three");
strList.Add("four");
strList.Add("four");
strList.Add(null);
strList.Add(null);
IList<Student> studentList = new List<Student>();
studentList.Add(new Student());
studentList.Add(new Student());
studentList.Add(new Student());
```

You can also add elements at the time of initialization using object initializer syntax as below:

```
IList<int> intList = new List<int>(){ 10, 20, 30, 40 };
//Or
IList<Student> studentList = new List<Student>() {
    new Student(){ StudentID=1, StudentName="Bill"},
    new Student(){ StudentID=2, StudentName="Steve"},
    new Student(){ StudentID=3, StudentName="Ram"},
    new Student(){ StudentID=1, StudentName="Moin"}
};
```

## AddRange

The AddRange() method adds all the elements from another collection.

```
IList<int> intList1 = new List<int>();
```

```
intList1.Add(10);
intList1.Add(20);
intList1.Add(30);
intList1.Add(40);
List<int> intList2 = new List<int>();

intList2.AddRange(intList1);
```

## *Access List collection:*

Use a foreach or for loop to iterate a List<T> collection.

```
List<int> intList = new List<int>() { 10, 20, 30 };

intList.ForEach(el => Console.WriteLine(el));
```

If you have initialized the List<T> with an IList<T> interface then use separate foreach statement with implicitly typed variable:

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };
foreach (var el in intList)
    Console.WriteLine(el);
```

Access individual items by using an indexer (i.e., passing an index in square brackets):

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };

int elem = intList[1]; // returns 20
```

List<T> implements IList<T>, so List<T> implicitly type cast to IList<T>.

```
static void Print(IList<string> list)
{
    Console.WriteLine("Count: {0}", list.Count);
    foreach (string value in list)
    {
        Console.WriteLine(value);
    }
}
static void Main(string[] args)
```

```

{
    string[] strArray = new string[2];
    strArray[0] = "Hello";
    strArray[1] = "World";
    Print(strArray);

    List<string> strList = new List<string>();
    strList.Add("Hello");
    strList.Add("World");
    Print(strList);
}

```

## Insert into List:

The Insert() method inserts an element into a List<T> collection at the specified index.

```

IList<int> intList = new List<int>(){ 10, 20, 30, 40 };

intList.Insert(1, 11); // inserts 11 at 1st index: after 10.
foreach (var el in intList)
    Console.Write(el);

```

## Remove Elements from List:

The Remove() and RemoveAt() methods remove items from a List<T> collection.

```

IList<int> intList = new List<int>(){ 10, 20, 30, 40 };

intList.Remove(10); // removes the 10 from a list

intList.RemoveAt(2); //removes the 3rd element (index starts from 0)
foreach (var el in intList)
    Console.Write(el);

```



## TrueForAll:

TrueForAll() is a method of the List<T> class. It returns true if the specified condition turns out to be true, otherwise false. Here, the condition can be specified as a predicate type delegate or lambda expression.

```
List<int> intList = new List<int>(){ 10, 20, 30, 40 };  
bool res = intList.TrueForAll(el => el%2 == 0);// returns true
```

The following example uses isPositiveInt() as a Predicate<int> type delegate as a parameter to TrueForAll.

```
static bool isPositiveInt(int i)  
{  
    return i > 0;  
}  
static void Main(string[] args)  
{  
    List<int> intList = new List<int>(){10, 20, 30, 40};  
  
    bool res = intList.TrueForAll(isPositiveInt);  
}
```

## 36-Generic Sortedlist

A generic SortedList (SortedList<TKey,TValue>) represents a collection of key-value pairs that are sorted by key based on associated [IComparer<T>](#). A SortedList collection stores key and value pairs in ascending order of key by default. Generic SortedList implements [IDictionary<TKey,TValue>](#) & [ICollection<KeyValuePair<TKey,TValue>>](#) interfaces so elements can be access by key and index both.

```
SortedList<int,string> mySortedList = new SortedList<int,string>();
```

### Important Properties and Methods of SortedList<TKey, TValue>:

Property	Description
Capacity	Gets or sets the number of elements that the SortedList<TKey,TValue> can store.
Count	Gets the total number of elements exists in the SortedList<TKey,TValue>.
IsReadOnly	Returns a boolean indicating whether the SortedList<TKey,TValue> is read-only.
Item	Gets or sets the element with the specified key in the SortedList<TKey,TValue>.
Keys	Get list of keys of SortedList<TKey,TValue>.
Values	Get list of values in SortedList<TKey,TValue>.

Method	Description
<code>void Add(TKey key, TValue value)</code>	Add key-value pairs into SortedList<TKey, TValue>.
<code>void Remove(TKey key)</code>	Removes element with the specified key.
<code>void RemoveAt(int index)</code>	Removes element at the specified index.
<code>bool ContainsKey(TKey key)</code>	Checks whether the specified key exists in SortedList<TKey, TValue>.
<code>bool ContainsValue(TValue value)</code>	Checks whether the specified key exists in SortedList<TKey, TValue>.

<code>void Clear()</code>	Removes all the elements from <code>SortedList&lt;TKey, TValue&gt;</code> .
<code>int IndexOfKey(TKey key)</code>	Returns an index of specified key stored in internal array of <code>SortedList&lt;TKey, TValue&gt;</code> .
<code>int IndexOfValue(TValue value)</code>	Returns an index of specified value stored in internal array of <code>SortedList&lt;TKey, TValue&gt;</code>
<code>bool TryGetValue(TKey key, out TValue value)</code>	Returns true and assigns the value with specified key, if key does not exists then return false.

## Add Elements into SortedList:

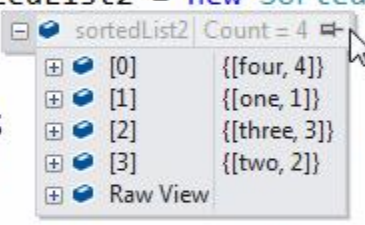
Use the `Add()` method to add key value pairs into a `SortedList`. The key cannot be null, but the value can be null. Also, the datatype of key and value must be same as specified, otherwise it will give compile time error.

```
SortedList<int, string> sortedList1 = new SortedList<int, string>();
sortedList1.Add(3, "Three");
sortedList1.Add(4, "Four");
sortedList1.Add(1, "One");
sortedList1.Add(5, "Five");
sortedList1.Add(2, "Two");
```



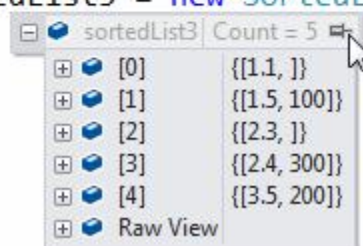
The screenshot shows the `sortedList1` collection in the Solution Explorer. It contains 5 elements, sorted by key. The elements are: {1, One}, {2, Two}, {3, Three}, {4, Four}, and {5, Five}.

```
SortedList<string, int> sortedList2 = new SortedList<string, int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
```



The screenshot shows the `sortedList2` collection in the Solution Explorer. It contains 4 elements, sorted by key. The elements are: {four, 4}, {one, 1}, {three, 3}, and {two, 2}.

```
SortedList<double, int?> sortedList3 = new SortedList<double, int?>();
sortedList3.Add(1.5, 100);
sortedList3.Add(3.5, 200);
sortedList3.Add(2.4, 300);
sortedList3.Add(2.3, null);
sortedList3.Add(1.1, null);
```



The screenshot shows the `sortedList3` collection in the Solution Explorer. It contains 5 elements, sorted by key. The elements are: {1.1, }, {1.5, 100}, {2.3, }, {2.4, 300}, and {3.5, 200}.

## Access SortedList<TKey, TValue>:

The `SortedList` can be accessed by the index or key. Unlike other collection types, `Indexer` of `SortedList` requires key and returns value for that key.

However, please make sure that key exists in the SortedList, otherwise it will throw `KeyNotFoundException`.

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
Console.WriteLine(sortedList2["one"]);
Console.WriteLine(sortedList2["two"]);
Console.WriteLine(sortedList2["three"]);
//Following will throw runtime exception:
    KeyNotFoundExceptionConsol
e.WriteLine(sortedList2["ten"]);
```

## foreach

The foreach statement in C# can be used to access the SortedList collection. SortedList element includes both key and value pair. so, the type of element would be `KeyValuePair` structure rather than type of key or value.

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
foreach(KeyValuePair<string,int> kvp in sortedList2 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value
);
```

## Access value of key:

If you are not sure that particular key exists or not than use `TryGetValue` method to retrieve the value of specified key. If key doesn't exists than it will return false instead of throwing exception.

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
int val;
if (sortedList2.TryGetValue("ten",out val))
    Console.WriteLine("value: {0}", val);
```

```

else
    Console.WriteLine("Key is not valid.");
if (sortedList2.TryGetValue("one", out val))
    Console.WriteLine("value: {0}", val);

```

## Remove Elements from SortedList<TKey, TValue>:

Use the Remove(key) and RemoveAt(index) methods to remove values from a SortedList.

```

SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

sortedList2.Remove("one");//removes the element whose key is 'one'
sortedList2.RemoveAt(0);//removes the element at zero index i.e first element: four
foreach(KeyValuePair<string,int> kvp in sortedList2 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value
);

```

## Contains(), ContainsKey() and ContainsValue():

The Contains() & ContainsKey() methods are used to determine whether the specified key exists in the SortedList collection or not.

The ContainsValue() method determines whether the specified value exists in the SortedList or not.

```

SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);

sortedList.Contains(2); // returns true
sortedList.Contains(4); // returns true
sortedList.Contains(6); // returns false

sortedList.ContainsKey(2); // returns true
sortedList.ContainsKey(6); // returns false

```

```
sortedList.ContainsValue("One"); // returns true
sortedList.ContainsValue("Ten"); // returns false
```

## LINQ

You can use LINQ query syntax or method syntax to access SortedList collection using different criterias.

```
SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);
var result = sortedList.Where(kvp => kvp.Key == "two").FirstOrDefault();
Console.WriteLine("key: {0}, value: {1}",
    result.Key, result.Value);
```

```
SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);
var query = from kvp in sortedList
            where kvp.Key == "two"
            select kvp;
var result = query.FirstOrDefault();
Console.WriteLine("key: {0}, value: {1}", result.Key, result.Value);
```

## 37-Dictionary

Dictionary in C# is same as English dictionary. English dictionary is a collection of words and their definitions, often listed alphabetically in one or more specific languages. In the same way, the Dictionary in C# is a collection of Keys and Values, where key is like word and value is like definition.

Dictionary<TKey, TValue> is a generic collection included in the System.Collection.Generics namespace. TKey denotes the type of key and TValue is the type of TValue.

### Dictionary Initialization:

```
IDictionary<int, string> dict = new Dictionary<int, string>();  
//or  
Dictionary<int, string> dict = new Dictionary<int, string>();
```

### Important Properties and Methods of IDictionary<TKey, TValue>:

Property	Description
Count	Gets the total number of elements exists in the Dictionary<TKey,TValue>.
IsReadOnly	Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only.
Item	Gets or sets the element with the specified key in the Dictionary<TKey,TValue>.
Keys	Returns collection of keys of Dictionary<TKey,TValue>.
Values	Returns collection of values in Dictionary<TKey,TValue>.

Method	Description
<code>void Add(T)</code>	Adds an item to the Dictionary collection.

<code>void Add(TKey key, TValue value)</code>	Add key-value pairs in Dictionary<TKey, TValue> collection.
<code>void Remove(T item)</code>	Removes the first occurrence of specified item from the Dictionary<TKey, TValue>.
<code>void Remove(TKey)</code>	Removes the element with the specified key.
<code>bool ContainsKey(TKey key)</code>	Checks whether the specified key exists in Dictionary<TKey, TValue>.
<code>bool ContainsValue(TValue value)</code>	Checks whether the specified key exists in Dictionary<TKey, TValue>.
<code>void Clear()</code>	Removes all the elements from Dictionary<TKey, TValue>.
<code>bool TryGetValue(TKey key, out TValue value)</code>	Returns true and assigns the value with specified key, if key does not exists then return false.

## Add Elements into Dictionary:

Use Add() method to add the key-value pair in dictionary.

```
IDictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "One");
dict.Add(2, "Two");
dict.Add(3, "Three");
```

The IDictionary type instance has one more overload for the Add() method. It accepts a `KeyValuePair<TKey, TValue>` struct as a parameter.

```
IDictionary<int, string> dict = new Dictionary<int, string>();

dict.Add(new KeyValuePair<int, string>(1, "One"));
dict.Add(new KeyValuePair<int, string>(2, "Two"));
//The following is also valid
dict.Add(3, "Three");
```

```
IDictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};
```



## Access Dictionary Elements:

Dictionary elements can be accessed by many ways e.g. foreach, for loop or indexer.

Use foreach or for loop to iterate access all the elements of dictionary. The dictionary stores key-value pairs. So you can use a KeyValuePair<TKey, TValue> type or an implicitly typed variable var in foreach loop as shown below.

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

foreach (KeyValuePair<int, string> item in dict)
{
    Console.WriteLine("Key: {0}, Value: {1}", item.Key, item.Value);
}
```

Use for loop to access all the elements. Use Count property of dictionary to get the total number of elements in the dictionary.

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

for (int i = 0; i < dict.Count; i++)
{
    Console.WriteLine("Key: {0}, Value: {1}",
        dict.Keys.ElementAt(i),
        dict[ dict.Keys.ElementAt(i)]);
}
```

Dictionary can be used like an array to access its individual elements. Specify key (not index) to get a value from a dictionary using indexer like an array.

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
}
```

```

};
Console.WriteLine(dict[1]); //returns One
Console.WriteLine(dict[2]);
// returns Two

```

If you are not sure about the key then use the TryGetValue() method. The TryGetValue() method will return false if it could not find keys instead of throwing an exception.

```

Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

string result;
if(dict.TryGetValue(4, out result))
{
    Console.WriteLine(result);
}
else
{
    Console.WriteLine("Could not find the specified key.");
}

```

## *Check for existing elements:*

Dictionary includes various methods to determine whether a dictionary contains specified elements or keys. Use the ContainsKey() method to check whether a specified key exists in the dictionary or not.

Use the Contains() method to check whether a specified Key and Value pair exists in the dictionary or not.

```

Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

dict.ContainsKey(1); // returns true
dict.ContainsKey(4); // returns false

```

```
dict.Contains(new KeyValuePair<int,string>(1,"One")); //returns true
```

Another overload of the Contains() method takes IEqualityComparer as a second parameter. An instance of IEqualityComparer is used when you want to customize the equality comparison. For example, consider the following example of a dictionary that stores a Student objects.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
class StudentDictionaryComparer :
    IEqualityComparer<KeyValuePair<int,Student>>
{
    public bool Equals(KeyValuePair<int, Student> x,
        KeyValuePair<int, Student> y)
    {
        if (x.Key == y.Key && (x.Value.StudentID == y.Value.StudentID)
            && (x.Value.StudentName == y.Value.StudentName))
            return true;

        return false;
    }

    public int GetHashCode(KeyValuePair<int, Student> obj)
    {
        return obj.Key.GetHashCode();
    }
}
class Program
{
    static void Main(string[] args)
    {
        IDictionary<int, Student> studentDict =
        new Dictionary<int, Student>()
        {
            { 1, new Student(){ StudentID =1, StudentName = "Bill"}},
            { 2, new Student(){ StudentID =2, StudentName = "Steve"}},
            { 3, new Student(){ StudentID =3, StudentName = "Ram"}}
        };

        Student std = new Student(){
            StudentID = 1,
            StudentName = "Bill"
        };
    }
}
```

```

KeyValuePair<int, Student> elementToFind =
    new KeyValuePair<int, Student>(1, std);

bool result = studentDict.Contains(elementToFind,
    new StudentDictionaryComparer()); // returns true

    Console.WriteLine(result);
}
}

```

## Remove Elements in Dictionary:

Use the Remove() method to remove an existing item from the dictionary. Remove() has two overloads, one overload method accepts a key and the other overload method accepts a KeyValuePair<> as a parameter.

```

Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

dict.Remove(1); // removes the item which has 1 as a key

```

# Partial Class

Each class in C# resides in a separate physical file with a .cs extension. C# provides the ability to have a single class implementation in multiple .cs files using the ***partial*** modifier keyword. The *partial* modifier can be applied to a class, method, interface or structure.

For example, the following MyPartialClass splits into two files, PartialClassFile1.cs and PartialClassFile2.cs:

```
public partial class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }
}
```

```
public partial class MyPartialClass
{
    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

## Partial Class Requirements:

- All the partial class definitions must be in the same assembly and namespace.
- All the parts must have the same accessibility like public or private, etc.
- If any part is declared abstract, sealed or base type then the whole class is declared of the same type.
- Different parts can have different base types and so the final class will inherit all the base types.
- The Partial modifier can only appear immediately before the keywords class, struct, or interface.
- Nested partial types are allowed.

## *Advantages of Partial class:*

- Multiple developers can work simultaneously with a single class in separate files.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

## *Partial Methods:*

A partial class or struct may contain partial methods. A partial method must be declared in one of the partial classes. A partial method may or may not have an implementation. If the partial method doesn't have an implementation in any part then the compiler will not generate that method in the final class. For example, consider the following partial method with a partial keyword:

```
public partial class MyPartialClass
{
    partial void PartialMethod(int val);

    public MyPartialClass()
    {
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

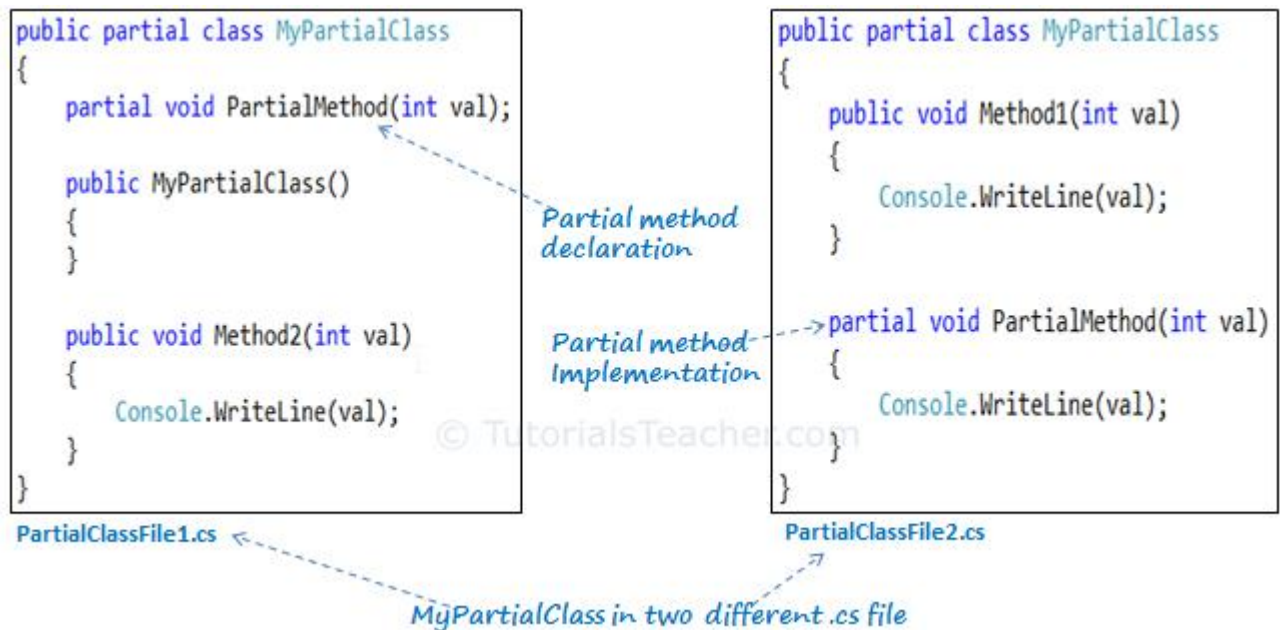
```
public partial class MyPartialClass
{
    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    partial void PartialMethod(int val)
    {
        Console.WriteLine(val);
    }
}
```

Partial method requirements:

- The partial method declaration must begin with the partial modifier.
- The partial method can have a ref but not an out parameter.
- Partial methods are implicitly private methods.
- Partial methods can be static methods.
- Partial methods can be generic.

The following image illustrates partial class and partial method:



### Partial Method

The compiler combines the two partial classes into a single final class:

```

public partial class MyPartialClass
{
    partial void PartialMethod(int val);

    public MyPartialClass()
    {
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}

```

PartialClassFile1.cs

```

public partial class MyPartialClass
{
    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    partial void PartialMethod(int val)
    {
        Console.WriteLine(val);
    }
}

```

PartialClassFile2.cs

*Compiles as a single class*

```

public class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }

    private void PartialMethod(int val)
    {
        Console.WriteLine(val);
    }
}

```



# Static

C# includes "static" keyword just like other programming languages such as C++, Java, etc. The **Static** keyword can be applied on classes, variables, methods, properties, operators, events and constructors. However, it cannot be used with indexers, destructors or types other than classes.

```
public static class MyStaticClass
{
    public static int myStaticVariable = 0;

    public static void MyStaticMethod()
    {
        Console.WriteLine("This is a static method.");
    }

    public static int MyStaticProperty { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyStaticClass.myStaticVariable);

        MyStaticClass.MyStaticMethod();

        MyStaticClass.MyStaticProperty = 100;

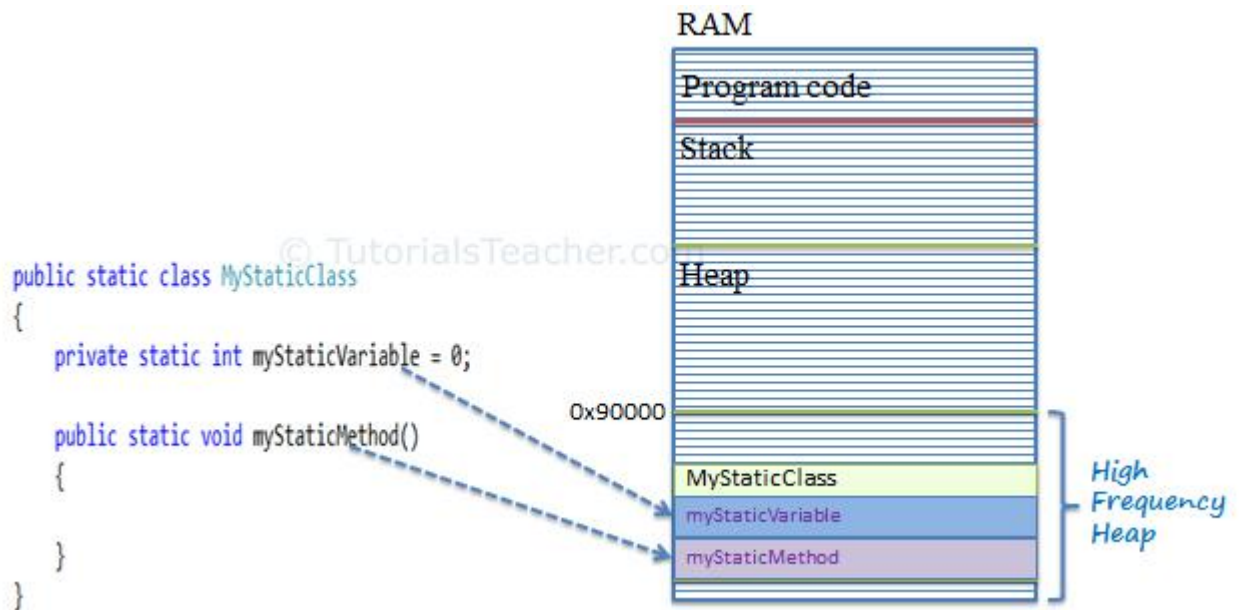
        Console.WriteLine(MyStaticClass.MyStaticProperty);
    }
}
```

## *Memory allocation of static items:*

As you know, the main parts of an application's memory are stack and heap. Static members are stored in a special area inside the heap called High Frequency Heap. Static members of non-static classes are also stored in the heap and shared across all the instances of the class. So the changes done by one instance will be reflected in all the other instances.

As mentioned earlier, a static member can only contain or access other static members, because static members are invoked without creating an instance and so they cannot access non-static members.

The following image illustrates how static members are stored in memory:



## Anonymous Method:

As the name suggests, an anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type.

```

public delegate void Print(int value);

static void Main(string[] args)
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}", val);
    };

    print(100);
}

```

Anonymous methods can access variables defined in an outer function.

```

public delegate void Print(int value);

```

```

static void Main(string[] args)
{
    int i = 10;

    Print prnt = delegate(int val) {
        val += i;
        Console.WriteLine("Anonymous method: {0}", val);
    };

    prnt(100);
}

```

Anonymous methods can also be passed to a method that accepts the delegate as a parameter.

In the following example, PrintHelperMethod() takes the first parameters of the Print delegate:

```

public delegate void Print(int value);

class Program
{
    public static void PrintHelperMethod(Print printDel, int val)
    {
        val += 10;
        printDel(val);
    }

    static void Main(string[] args)
    {

```

```
        PrintHelperMethod(delegate(int val) { Console.WriteLine("Anonymous method: {0}", val); }, 100);  
  
    }  
}
```

Anonymous methods can be used as event handlers:

```
saveButton.Click += delegate(Object o, EventArgs e)  
  
    {  
  
        System.Windows.Forms.MessageBox.Show("Save Successfully!");  
  
    };
```

## *Anonymous method limitations:*

- It cannot contain jump statement like goto, break or continue.
- It cannot access ref or out parameter of an outer method.
- It cannot have or access unsafe code.
- It cannot be used on the left side of the is operator.

## Nullable Type in C#:

As you know, a value type cannot be assigned a null value. For example, *int i = null* will give you a compile time error.

C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using `Nullable<t>` where T is a type.

```
Nullable<int> i = null;
```

## Covariance and Contravariance in C#:

Covariance and contravariance allow us to be flexible when dealing with class hierarchy.

Consider the following class hierarchy before we learn about covariance and contravariance:

```
class Small
{
}
}class Big: Small
{
}
}class Bigger : Big
{
}
}
```

As per the above example classes, small is a base class for big and big is a base class for bigger. The point to remember here is that a derived class will always have something more than a base class, so the base class is relatively smaller than the derived class.

`Small smlCls1 = new Small();` ✓

`Small smlCls2 = new Big();` ✓

`Small smlCls3 = new Bigger();` ✓

`Big bigCls1 = new Bigger();` ✓

`Big bigCls2 = new Small();` ✗

# var – Implicit typed local variable in C#:

C# 3.0 introduced the implicit typed local variable "var". Var can only be defined in a method as a local variable. The compiler will infer its type based on the value to the right of the "=" operator.

```
int i = 100; // explicitly typed var i = 100; // implicitly type
```

```
static void Main(string[] args)
{
    var i = 10;
    Console.WriteLine("Type of i is {0}", i.GetType().ToString());

    var str = "Hello World!!";
    Console.WriteLine("Type of str is {0}", str.GetType().ToString());

    var d = 100.50d;
    Console.WriteLine("Type of d is {0}", d.GetType().ToString());

    var b = true;
    Console.WriteLine("Type of b is {0}", b.GetType().ToString());
}
```

# Func in c#

We have learned in the previous section, that a [delegates](#) can be defined as shown below.

```
public delegate int SomeOperation(int i, int j);
class Program
{
    static int Sum(int x, int y)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        SomeOperation add = Sum;

        int result = add(10, 10);

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

C# 3.0 includes built-in generic delegate types **Func** and **Action**, so that you don't need to define custom delegates as above.

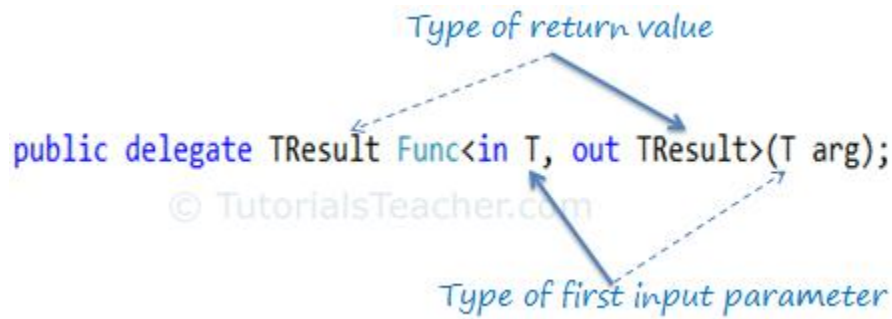
Func is a generic delegate included in the *System* namespace. It has zero or more **input** parameters and one **out** parameter. The last parameter is considered as an out parameter.

For example, a Func delegate that takes one input parameter and one out parameter is defined in the System namespace as below:

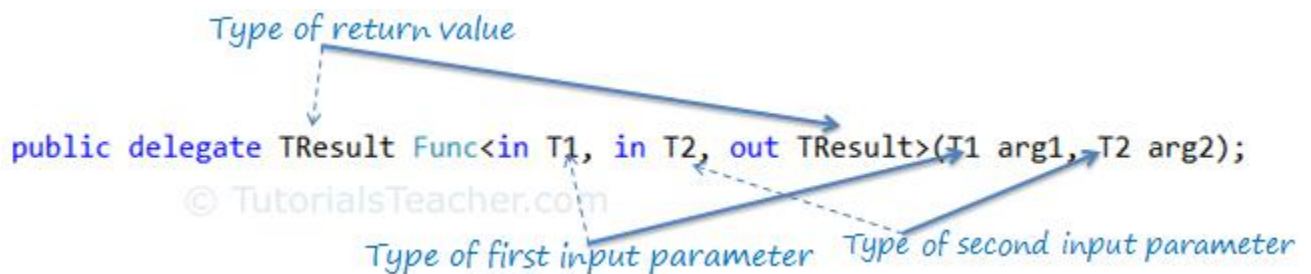
```
namespace System
{
    public delegate TResult Func<in T, out TResult>(T arg);
}
```

The last parameter in the angle brackets <> is considered as the return type and remaining parameters are considered as input parameter types as shown in the following figure.





A Func delegate with two input parameters and one out parameters will be represent as below.



## Func with an Anonymous method:

You can assign an anonymous method to the Func delegate by using the delegate keyword.

```
Func<int> getRandomNumber = delegate()
{
    Random rnd = new Random();
    return rnd.Next(1, 100);
};
```

## Func with lambda expression:

```
Func<int> getRandomNumber = () => new Random().Next(1, 100);
//Or
Func<int, int, int> Sum = (x, y) => x + y;
```