



# **.Net Framework**

---

**C# Programming  
& ADO.Net**

---

[www.enosislearning.com](http://www.enosislearning.com)

[courses@enosislearning.com](mailto:courses@enosislearning.com)

## Table of Contents

<b>What is the .NET Platform?</b>	7
<b>What is the .NET Framework?</b>	7
.NET Framework Components	8
.NET Framework Architecture	8
<b>Common Language Runtime (CLR)</b>	9
Responsibilities of CLR	9
CLR Architecture	10
Managed and Unmanaged Code	10
Managed Code	10
Unmanaged (Win32) Code	11
Memory Management	11
<b>Intermediate Language (MSIL, IL, CIL)</b>	11
<b>Compilation and Execution</b>	12
<b>Common Type System (CTS)</b>	12
<b>Common Language Infrastructure (CLI)</b>	13
<b>Common Language Specification (CLS)</b>	13
<b>Framework Class Library (FCL)</b>	13
<b>.NET Applications (Assemblies, Metadata, Applications)</b>	14
.NET Assemblies	14
Metadata in the Assemblies	14
Namespace	15
Global assembly cache (GAC)	15
.NET Applications	15
<b>.NET Languages</b>	15
<b>C# Language</b>	16
<b>C# Data Types</b>	16
<b>Value Type and Reference Type</b>	17
Value Type	17
Pass By Value	17
Reference Type	18
Pass By Reference	18
<b>Datatype conversions</b>	19
Implicit Conversion	19
Explicit Conversion	19
Difference between Parse and TryParse	20
<b>Access Modifier Keywords</b>	22
<b>Operators</b>	23
Arithmetic Operators	23
Relational Operators	23
Logical Operators	24

Bitwise Operators .....	24
Assignment Operators .....	25
Miscellaneous Operators .....	26
<b>Decision Making</b> .....	27
if statement .....	27
if-else Statement .....	27
else if Statement .....	28
Nested if Statements .....	28
Ternary operator ?: .....	29
Nested Ternary operator: .....	29
Switch Case .....	30
Goto in switch: .....	30
<b>Loops</b> .....	31
for loop .....	31
while loop: .....	32
do while .....	33
<b>Array</b> .....	34
Declaring Arrays .....	34
Initializing an Array .....	34
Assigning Values to an Array .....	34
Multi-dimensional Array .....	35
Jagged Array .....	36
<b>Methods</b> .....	37
Calling Methods in C# .....	37
<b>Method Parameters</b> .....	38
Value Parameter .....	38
Reference Parameter .....	38
Output Parameter .....	38
Parameter Arrays .....	39
<b>Enum</b> .....	39
Enum methods: .....	39
<b>Classes</b> .....	41
Members of Class .....	41
Constructor .....	42
Destructor .....	42
Member Functions and Encapsulation .....	42
Static And Instance Class Members .....	43
Static Constructor .....	43
Copy Constructor .....	44
Private Constructor .....	45
<b>Properties</b> .....	45

---

Creating Read-Only Properties .....	46
Creating a Write-Only Property .....	47
Creating Auto-Implemented Properties .....	48
<b>Inheritance</b> .....	50
THIS & NEW KEYWORD .....	51
<b>Interface</b> .....	53
<b>Abstract Class</b> .....	55
Important rules applied to abstract classes .....	56
Abstract class vs. Interface : .....	57
DIFFERENCE BETWEEN INTERFACE & ABSTRACT CLASS .....	57
<b>Struct</b> .....	58
DIFFERENCE BETWEEN STRUCTURE AND CLASS .....	59
<b>Partial Classes</b> .....	60
<b>Sealed Classes</b> .....	61
<b>Static Class</b> .....	63
<b>Extention Methods</b> .....	66
Important Points of Static .....	65
<b>Encapsulation</b> .....	67
<b>Polymorphism</b> .....	67
Method Hiding .....	69
Method Overriding(RunTime Polymorphism) .....	70
Differences Among Method Overriding, Method Hiding (new Keyword) and Method Shadowing in C# .....	71
<b>Abstraction</b> .....	73
Difference between Abstraction and Encapsulation .....	74
<b>Exception in C#</b> .....	74
Exception Handling .....	75
Throwing Exceptions (the throw Construct) .....	76
<b>Delegate</b> .....	77
Multicast Delegate .....	78
Array of Delegates .....	78
Anonymous Methods .....	79
<b>Event</b> .....	80
Event Arguments .....	83
<b>Generics</b> .....	85
Generic Delegates .....	86
Generic Type Parameters .....	87
Constraints on type parameters .....	87
<b>Attribute</b> .....	88
Predefined Attributes .....	89
AttributeUsage .....	89
Conditional .....	90

Obsolete.....	91
Creating Custom Attributes .....	91
<b>Reflection.....</b>	<b>92</b>
<b>Collection .....</b>	<b>94</b>
ArrayList.....	95
Properties and Methods of ArrayList .....	95
SortedList.....	96
Properties and Methods of SortedList.....	96
Stack.....	96
Properties and Methods of Stack: .....	97
Queue .....	97
Properties and Methods of Queue:.....	97
Hashtable .....	97
Propertis and Methods of Hashtable .....	98
<b>Multithreading.....</b>	<b>98</b>
Features and Benefits of Threads .....	98
Properties and Methods of the Thread Class .....	99
<b>File Handling .....</b>	<b>104</b>
Working with Directories.....	104
DirectoryInfo Class .....	105
Directory Class .....	105
Reading and Writing to Files.....	106
Stream .....	107
FileStream .....	107
BinaryReader and BinaryWriter.....	108
StringReader and StringWriter .....	109
<b>Introduction to ADO.Net.....</b>	<b>111</b>
What are .NET Data Providers? .....	112
<b>SqlConnection in ADO.NET .....</b>	<b>113</b>
Connection Strings in web.config configuration file.....	115
<b>SqlCommand in ado.net.....</b>	<b>117</b>
<b>Sql injection .....</b>	<b>119</b>
Sql Injection prevention using “Parametrized Queries” .....	120
Sql Injection prevention using “Stored Procedure” .....	120
Calling a stored procedure with output parameters.....	121
<b>SqlDataReader object in ADO.NET.....</b>	<b>122</b>
Read() Method of SqlDataReader.....	123
NextResult() method Of SqlDataReader.....	124
<b>SqlDataAdapter in ADO.NET .....</b>	<b>125</b>
<b>DataSet in ADO.Net.....</b>	<b>127</b>
<b>SqlCommandBuilder.....</b>	<b>132</b>



## What is the .NET Platform?

---

- ◆ The .NET platform
  - Microsoft's platform for software development
  - Unified technology for development of almost any kind of applications
    - GUI / Web / RIA / mobile / server / cloud / etc.
- ◆ .NET platform versions
  - .NET Framework
  - Silverlight / Windows Phone 7
  - .NET Compact Framework

## What is the .NET Framework?

---

- ◆ .NET Framework
  - An environment for developing and executing .NET applications
  - Unified programming model, set of languages, class libraries, infrastructure, components and tools for application development
  - Environment for controlled execution of managed code
- ◆ It is commonly assumed that

.NET platform == .NET Framework

.NET framework is a Windows Component that supports the building and running of windows applications, web and XML Web services. The purpose of the component is to provide the user with a consistent object oriented programming environment whether the code is stored locally or remotely.

It aims to minimize software deployment and versioning conflicts and also promote safe execution of code including codes executed by trusted third parties. It is directed towards eliminating performance problems of scripted or interpreted environments. The effort is to make developer experience consistent across a variety of applications and platforms and create communication standards that help .NET framework applications integrate with all other web based applications

## .NET Framework Components

The .NET framework has two major components-- The Common Runtime (CLR) and the Class Library

- ◆ **Common Language Runtime (CLR)**

The CLR is the foundation upon which the .NET Framework has been built. The runtime manages code at execution time and provides all the core services such as memory management, thread management and remoting. This capability to manage code at runtime is the distinguishing feature of the CLR. All code that is managed by the CLR is known as managed code while other codes are known as unmanaged code.

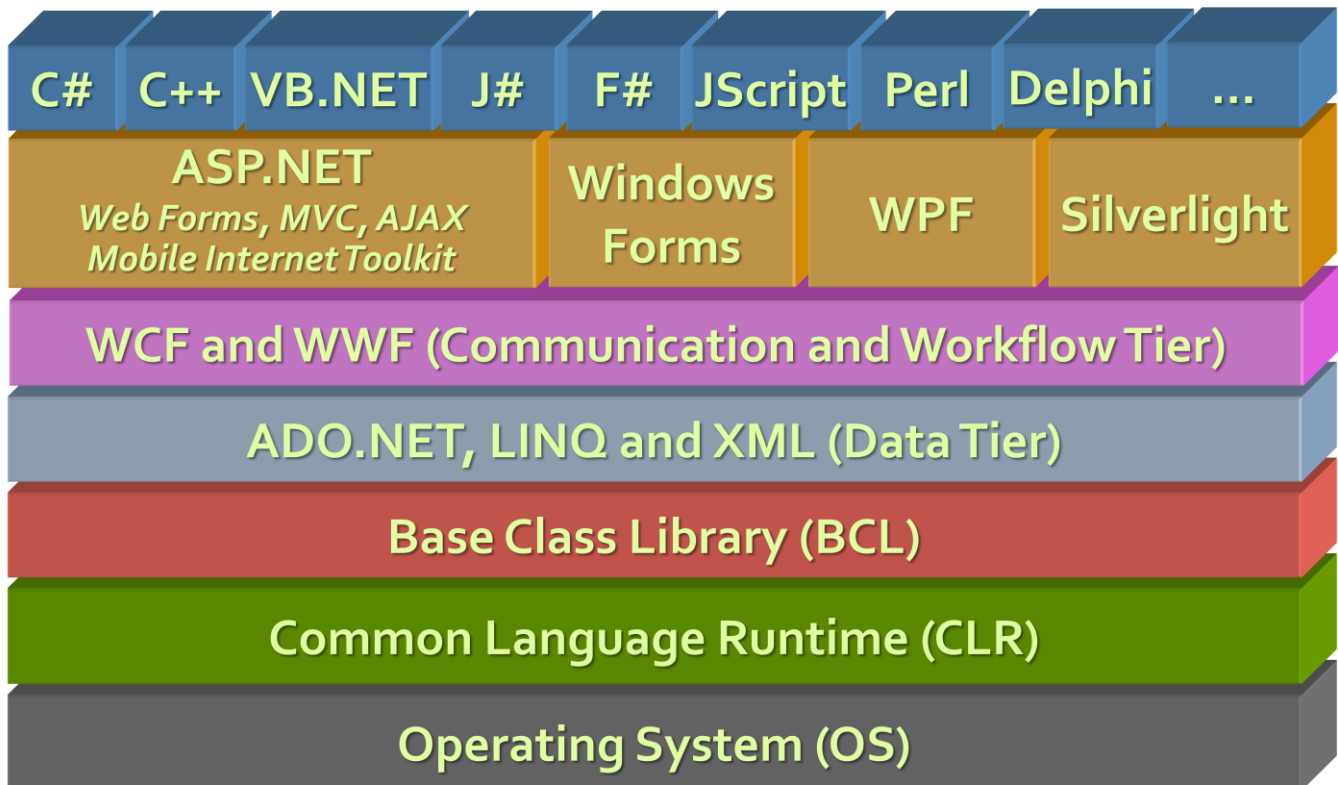
- ◆ **Framework Class Library (FCL)**

Standard class library for .NET development

Delivers basic functionality for developing: XML, ADO.NET, LINQ, ASP.NET, WPF, WCF, WWF, Silverlight, Web services, Windows Forms etc.

- ◆ **SDK, compilers and tools**

## .NET Framework Architecture





- The OS manages the resources, the processes and the users of the machine
- Provides to the applications some services (threads, I/O, GDI+, DirectX, COM, COM+, MSMQ, IIS, WMI, ...)
- CLR is a separate process in the OS
- CLR manages the execution of the .NET code
- Manages the memory, concurrency, security,
- Rich object-oriented library with fundamental classes
- Input-output, collections, text processing, networking, security, multi-threading, ...
- Database access
- ADO.NET, LINQ, LINQ-to-SQL and Entity Framework
- Strong XML support
- Windows Communication Foundation (WCF) and Windows Workflow Foundation (WWF) for the SOA world
- User interface technologies: Web based, Windows GUI, WPF, Silverlight, mobile, ...
- Programming language on your flavor!

## Common Language Runtime (CLR)

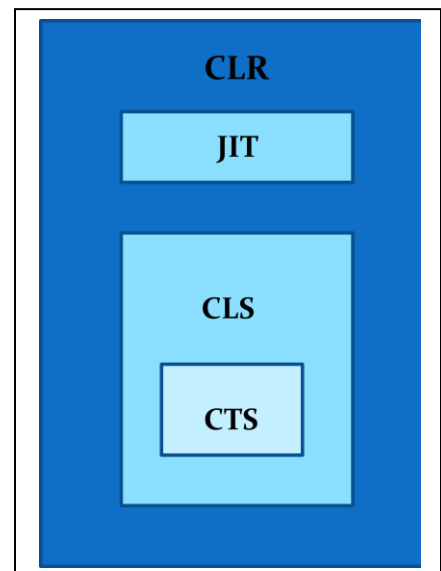
---

(The Heart of .NET Framework)

- ◆ **Managed execution environment**  
Controls the execution of managed .NET programming code
- ◆ **Something like virtual machine**  
Like the Java Virtual Machine (JVM)
- ◆ **Not an interpreter**  
Compilation on-demand is used  
Known as Just In Time (JIT) compilation
- ◆ **Possible compilation in advance (Ngen)**

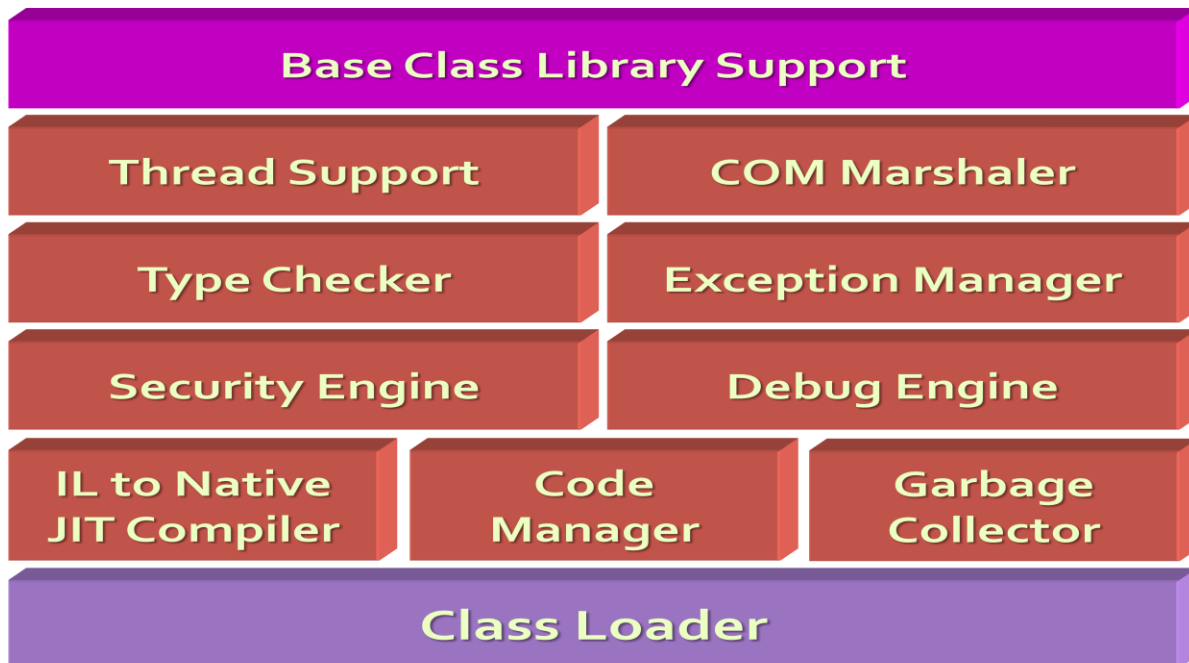
### Responsibilities of CLR

- ◆ Execution of the IL code and the JIT compilation
- ◆ Managing memory and application resources
- ◆ Ensuring type safety
- ◆ Interaction with the OS
- ◆ Managing security
  - Code access security
  - Role-based security
- ◆ Managing exceptions



- ◆ Managing concurrency – controlling the parallel execution of application threads
- ◆ Managing application domains and their isolation
- ◆ Interaction with unmanaged code
- ◆ Supporting debug / profile of .NET code

### CLR Architecture



### Managed and Unmanaged Code

**What is the Difference?**

#### Managed Code

- ◆ CLR executed code is called managed code
- ◆ Represents programming code in the low level language MSIL (MS Intermediate Language)
- ◆ Contains metadata
  - Description of classes, interfaces, properties, fields, methods, parameters, etc.
- ◆ Programs, written in any .NET language are
  - Compiled to managed code (MSIL)
  - Packaged as assemblies (.exe or .dll files)
- ◆ Object-oriented
- ◆ Secure
- ◆ Reliable
  - Protected from irregular use of types (type-safe)
- ◆ Allows integration between components and data types of different programming languages

- ◆ Portable between different platforms  
Windows, Linux, Max OS X, etc.

### Unmanaged (Win32) Code

- ◆ No protection of memory and type-safety
  - Reliability problems
  - Safety problems
- ◆ Doesn't contain metadata
  - Needs additional overhead like (e.g. use COM)
- ◆ Compiled to machine-dependent code
  - Need of different versions for different platforms
  - Hard to be ported to other platforms

### Memory Management

- ◆ CLR manages memory automatically
  - Dynamically loaded objects are stored in the managed heap
  - Unusable objects are automatically cleaned up by the garbage collector
- ◆ Some of the big problems are solved
  - Memory leaks
  - Access to freed or unallocated memory
- ◆ Objects are accessed through a reference

### Intermediate Language (MSIL, IL, CIL)

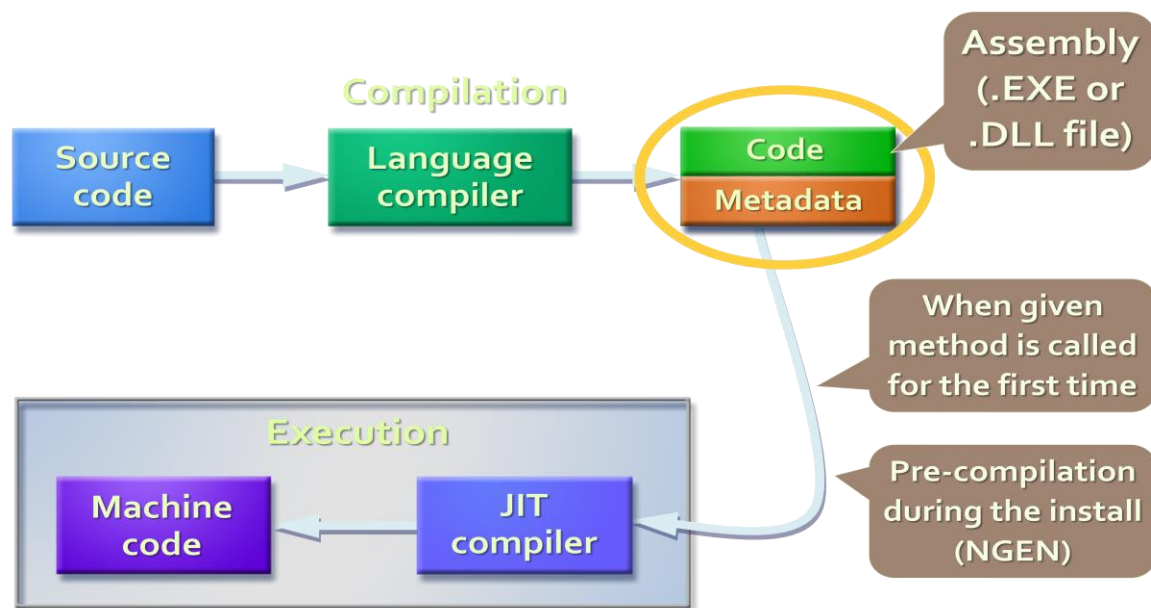
---

- ◆ Low level language (machine language) for the .NET CLR
- ◆ Has independent set of CPU instructions
  - Loading and storing data, calling methods
  - Arithmetic and logical operations
  - Exception handling
- ◆ MSIL is converted to instructions for the current physical CPU by the JIT compiler

Sample MSIL Code :

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size          11 (0xb)
    .maxstack 8
    ldstr      "Hello, world!"
    call       void
               [mscorlib]System.Console::WriteLine(string)
    ret
} // end of method HelloWorld::Main
```

## Compilation and Execution



## Common Type System (CTS)

A number of types are supported by the CLR and are described by the CTS. Both value types are supported—primitive data types and reference types. The primitive data types include Byte, Int16, Double and Boolean while Reference types include arrays, classes and object and string types. Reference types are types that store a reference to the location of their values. The value is stored as part of a defined class and is referenced through a class member on the instance of a class.

User defined value types and enumerations are derived from the value types mentioned above.

Language compilers implement types using their own terminology.

- ◆ CTS defines the CLR supported types of data and the operations over them
- ◆ Ensures data level compatibility between different .NET languages

E.g. string in C# is the same like String in VB.NET and in J#

- ◆ Value types and reference types

All types derive from System.Object

The Runtime enforces code robustness by implementing strict type and code verification infrastructure called Common type System (CTS). The CTS ensures that all managed code is self describing and all Microsoft or third party language compiler generated codes conform to CTS.

---

## Common Language Infrastructure (CLI)

A subset of the .NET framework is the CLI. The CLI includes the functionality of the Common Language Runtime and specifications for the Common Type System, metadata and Intermediate language.

---

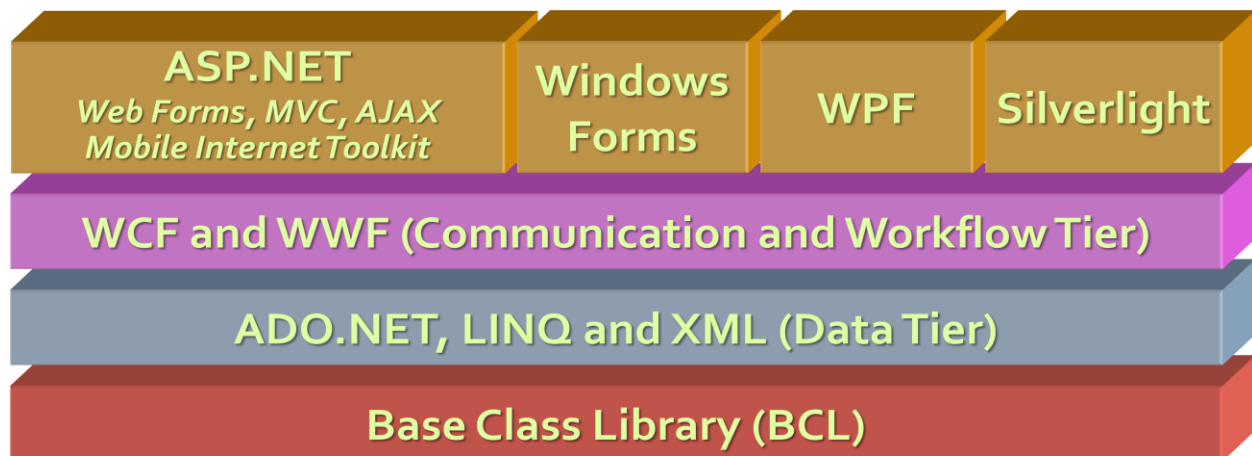
## Common Language Specification (CLS)

The CLR supports the CLS which is a subset of it. Additionally the CLR supports a set of rules that language and compiler designers follow.

---

## Framework Class Library (FCL)

- ◆ Framework Class Library is the standard .NET Framework library of out-of-the-box reusable classes and components (APIs)



The Class Library is an object oriented collection of reusable types. Unmanaged components that load CLR into their processes can be hosted by the .NET Framework to initiate the execution of managed code. This creates a software environment that exploits both the managed and unmanaged codes. The .NET Framework also provides a number of runtime hosts and supports third party runtime hosts

### Class Library Features :

1. The class library is a collection of reusable types that integrate with the CLR.
2. It is object oriented and provides types from which user defined types can derive functionality. This makes for ease of use and is time saving.
3. It supports a variety of specialized development scenarios such as console application development, Windows GUI applications, ASP.NET Applications, XML Web services.

## **.NET Applications (Assemblies, Metadata, Applications)**

### **.NET Assemblies**

Assembly is a single deployable unit that contains all the information about the classes, structure, and interface. Assemblies store all the information about itself. This information can be called metadata. It is physical grouping of logical units, . Namespace can span multiple assemblies. Assemblies can be private & public

- ◆ .NET assemblies:

- Self-containing .NET components

- Stored in .DLL and .EXE files

- Contain list of classes, types and resources

- Smallest deployment unit in CLR

- Have unique version number

- ◆ .NET deployment model

- No version conflicts (forget the "DLL hell")

- Supports side-by-side execution of different versions of the same assembly

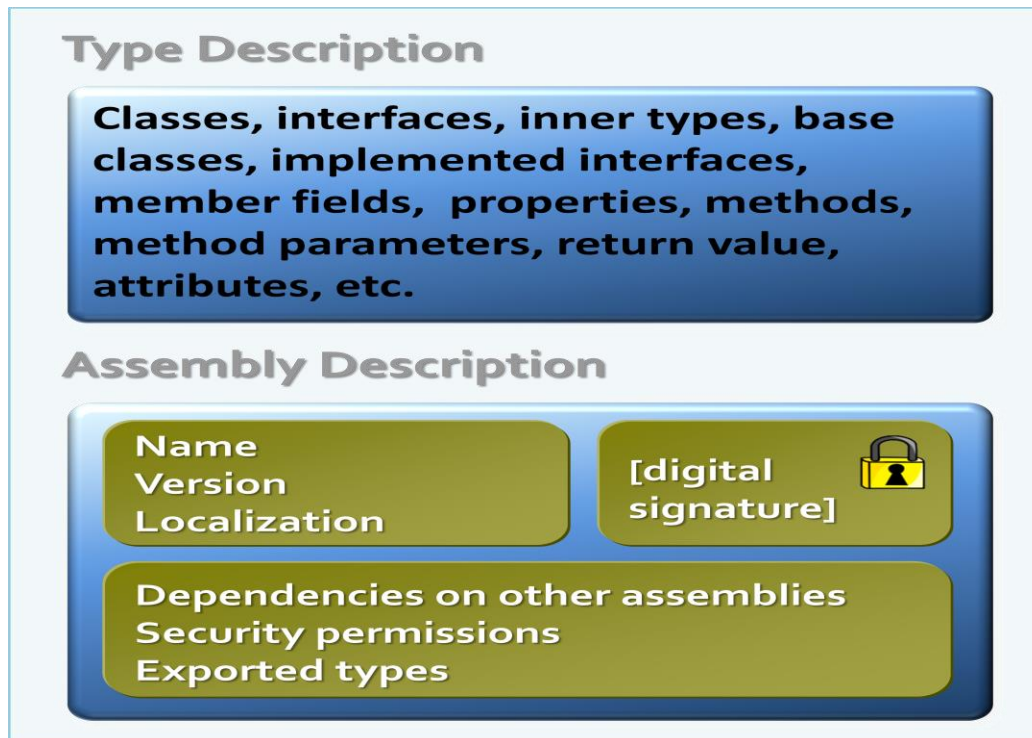
### **Metadata in the Assemblies**

- Data about data contained in the assembly

- Integral part of the assembly

- Generated by the .NET languages compiler

- Describes all classes, their class members, versions, resources, etc.



### Namespace

It is logically group classes. It avoid any naming conflicts between classes which have same name. It allows u to organize your classes so that they can be easily accessed in other application.

### Global assembly cache (GAC)

Is where all assemblies resides

If the app. has to be shared among several applications this is in the same comp

### .NET Applications

- ◆ Configurable executable .NET units
- ◆ Consist of one or more assemblies
- ◆ Installed by "copy / paste"
  - No complex registration of components
- ◆ Different applications use different versions of common assemblies
  - No conflicts due to their "strong name"

Easy installation, un-installation and update

### .NET Languages

- ◆ .NET languages by Microsoft
  - C#, VB.NET, Managed C++, J#, F#, JScript
- ◆ .NET languages by third parties

Object Pascal, Perl, Python, COBOL, Haskell, Oberon, Scheme, Smalltalk...

- ◆ Different languages can be mixed in a single application

Cross-language inheritance of types and exception handling

## C# Language

---

- ◆ C# is mixture between C++, Java and Delphi
  - Fully object-oriented by design
- ◆ Component-oriented programming model
  - Components, properties and events
  - No header files like C/C++
  - Suitable for GUI and Web applications
  - XML based documentation
- ◆ In C# all data types are objects

Example: `5.ToString()` is a valid call

## C# Data Types

---

The data type tells the 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, etc.

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



Alias	.NET Type	Type	Size (bits)	Range (values)
bool	Boolean	Logical Boolean type	8	True or False
object	Object	Base type of all other types		
string	String	A sequence of characters		
decimal	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
DateTime	DateTime	Represents date and time		0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

Example :

```
class Program
{
    static void Main(string[] args)
    {
        string stringVar = "Hello World!!";
        int intVar = 100;
        float floatVar = 10.2f;
        char charVar = 'A';
        bool boolVar = true;
    }
}
```

## Value Type and Reference Type

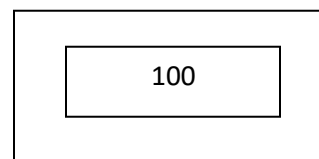
Data types are further classified as *value type* or *reference type*, depending on whether a variable of a particular type stores its own data or a pointer to the data in the memory.

### Value Type:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.

For example, consider integer variable `int i = 100;`

`int i = 100;`



### Pass By Value :

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example :

```
static void ChangeValue(int x)
{
    x = 200;

    Console.WriteLine(x);
}

static void Main(string[] args)
{
    int i = 100;

    Console.WriteLine(i);

    ChangeValue(i);

    Console.WriteLine(i);
}
```

Output :

```
100
200
100
```

In the above example, variable `i` in `Main()` method remains unchanged even after we pass it to the `ChangeValue()` method and change its value there.

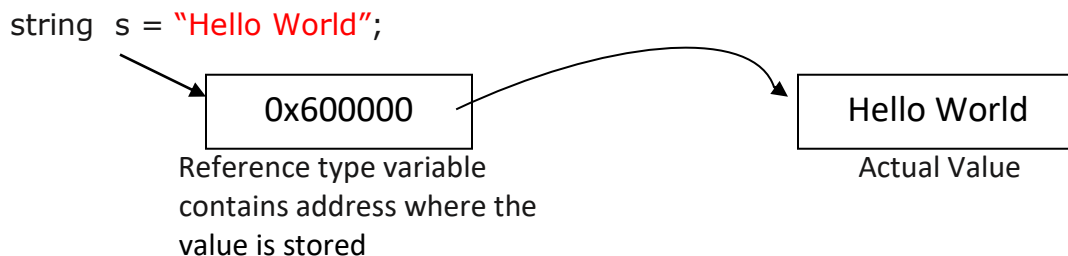
### Reference Type :

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider following string variable:

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

The following image shows how the system allocates the memory for the above string variable.



### Pass By Reference :

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example :

```
Static void ChangeReferenceType(Student std2)
{
    std2.StudentName = "Steve";
}

static void Main(string[] args)
{
    Student std1 = new Student();
    std1.StudentName = "Bill";

    ChangeReferenceType(std1);
}
```

Output :

```
Steve
```

### Points to Remember :

1. Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.
2. Primitive data types and struct are of the 'Value' type. Class objects, string, array, delegates are reference types.
3. Value type passes by val by default. Reference type passes byref by default.
4. Value types and reference types stored in Stack and Heap in the memory depends on the scope of the variable.

---

## Datatype conversions

1. Implicit Conversion
2. Explicit Conversion

### Implicit Conversion

**Implicit conversion is done by the compiler:**

1. When there is no loss of information if the conversion is done
2. If there is no possibility of throwing exceptions during the conversion

**Example:** Converting an **int** to a **float** will not loose any data and no exception will be thrown, hence an implicit conversion can be done.

### Explicit Conversion

Where as when converting a float to an int, we loose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required. For explicit conversion we can use cast operator or the convert class in c#. Eplicit Conversion can be done using following ways :

1. Type cast
2. Convert Class

3. Parse() Method
4. TryParse() Method

#### Implicit Conversion Example

```
using System;
class Program
{
    public static void Main()
    {
        int i = 100;
        float f = i;
        Console.WriteLine(f);
    }
}
```

#### Explicit Conversion Example

```
using System;
class Program
{
    public static void Main()
    {
        float f = 100.25F;
        int i = (int)f;           // explicit conversion by type casting
        int i = Convert.ToInt32(f); // explicit conversion by Convert class
        Console.WriteLine(i);
    }
}
```

#### Difference between Parse and TryParse

1. If the number is in a string format you have 2 options - Parse() and TryParse()
2. Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.
3. Use Parse() if you are sure the value will be valid, otherwise use TryParse()

#### Parse() Method:

```
using System;
class Program
{
    public static void Main()
    {
        string s = "Hello World";

        int i = int.Parse(s); // explicit conversion by Parse Method
        Console.WriteLine(i);
    }
}
```

**Tryparse() Method:**

```
using System;
class Program
{
    public static void Main()
    {
        string value = "200t";
        int r=0;
        bool number = int.TryParse(value, out r);
        if (number)
        {
            Console.WriteLine( r);
        }
    }
}
```

## 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
public	The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members.
private	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.
internal	The Internal modifier allows other program code in the same assembly to access the type or its members. This is default access modifiers if no modifier is specified.
protected	The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members.

The following six accessibility levels can be specified using the access modifiers:

- [public](#): Access is not restricted.
- [protected](#): Access is limited to the containing class or types derived from the containing class.
- [internal](#): Access is limited to the current assembly.
- [protected internal](#): Access is limited to the current assembly or types derived from the containing class.
- [private](#): Access is limited to the containing class.
- [private protected](#): Access is limited to the containing class or types derived from the containing class within the current assembly.

## Operators :

C# has rich set of built-in operators and provides the following type of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

### Arithmetic Operators

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

### Relational Operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the	(A <= B) is

	value of right operand, if yes then condition becomes true.	true.
--	---	-------

## Logical Operators

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

## Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows –

P	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; then in the binary format they are as follows –

A = 0011 1100

B = 0000 1101

-----

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then –

Show Examples



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A   B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

### Assignment Operators

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

	with the left operand and assign the result to left operand	equivalent to $C = C * A$
<code>/=</code>	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	$C \% = A$ is equivalent to $C = C \% A$
<code>&lt;&lt;=</code>	Left shift AND assignment operator	$C <<= 2$ is same as $C = C << 2$
<code>&gt;&gt;=</code>	Right shift AND assignment operator	$C >>= 2$ is same as $C = C >> 2$
<code>&amp;=</code>	Bitwise AND assignment operator	$C \&= 2$ is same as $C = C \& 2$
<code>^=</code>	bitwise exclusive OR and assignment operator	$C \wedge= 2$ is same as $C = C \wedge 2$
<code> =</code>	bitwise inclusive OR and assignment operator	$C  = 2$ is same as $C = C   2$

### Miscellaneous Operators

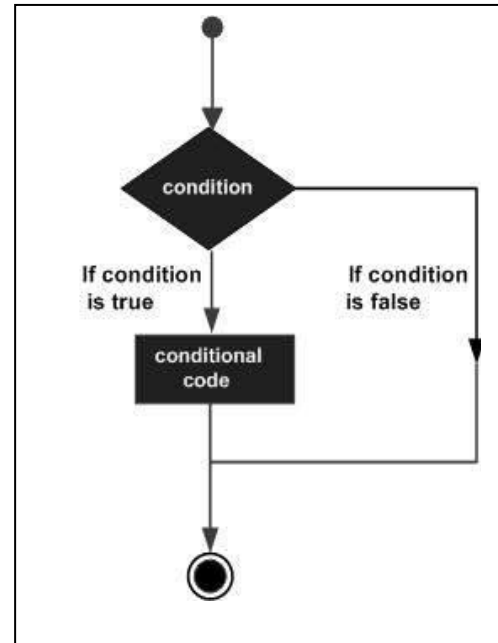
Operator	Description	Example
<code>sizeof()</code>	Returns the size of a data type.	<code>sizeof(int)</code> , returns 4.
<code>typeof()</code>	Returns the type of a class.	<code>typeof(StreamReader);</code>
<code>&amp;</code>	Returns the address of an variable.	<code>&amp;a</code> ; returns actual address of the variable.
<code>*</code>	Pointer to a variable.	<code>*a</code> ; creates pointer named 'a' to a variable.
<code>? :</code>	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
<code>is</code>	Determines whether an object is of a certain type.	If( Ford is Car) // checks if Ford is an object of the Car class.
<code>as</code>	Cast without raising an exception if the cast fails.	Object obj = new StreamReader("Hello"); StreamReader r = obj as StreamReader;

## Decision Making

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

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. if-else statement
3. switch statement
4. Ternary operator :?
5. Switch case



### if statement

The if statement contains boolean expression inside brackets followed by a single or multi line code block. At runtime, if a boolean expression is evaluates to true then the code block will be executed. Consider the following example where the if condition contains true as an expression.

Example :

```
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.

Example :

```
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.

Example :

```
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");
}
```

### Nested if Statements

Example :

```
int i = 10;
if (i > 0)
{
    if (i <= 100)
    {
        Console.WriteLine("i is positive number less than 100");
    }
    else
    {
        Console.WriteLine("i is positive number greater than 100");
    }
}
```

#### **Points to Remember :**

1. if-else statement controls the flow of program based on the evaluation of the boolean expression.
  2. It should start from the if statement followed by else or else-if statements.
  3. Only one else statement is allowed in the if-else chain.
  4. Multiple else-if statements are allowed in a single if-else chain.
- Nested if-else statement is allowed.

### Ternary operator ?:

The if-else statement can be replaced by ternary operator.

Syntax :

```
Boolean Expression ? First Statement : Second Statement
```

As you can see in the above syntax, ternary operator includes three parts. First part (before ?) includes conditional expression that returns boolean value true or false. Second part (after ? and before :) contains a statement which will be returned if the conditional expression in the first part evaluates to true. The third part includes another statement which will be returned if the conditional expression returns false.

Example :

```
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);
```

Output:

```
x is greater than y
```

The ternary operator can return a value of any data type. So it is advisable to store it in [implicitly typed variable - var](#).

Example :

```
int x = 20, y = 10;

var result = x > y ? x : y;

Console.WriteLine(result);
```

Output :

```
20
```

### Nested Ternary operator:

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

```
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";

Console.WriteLine(result);
```

```
x is less than y
```

## Switch Case

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

Switch statement contains an expression into brackets. It also includes multiple case labels, where each case represents a particular literal value. The switch cases are separated by a break keyword which stops the execution of a particular case. Also, the switch can include a default case to execute if no case value satisfies the expression.

Example :

```
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;
}
```

Output :

```
Value of x is 10
```

## Goto in switch:

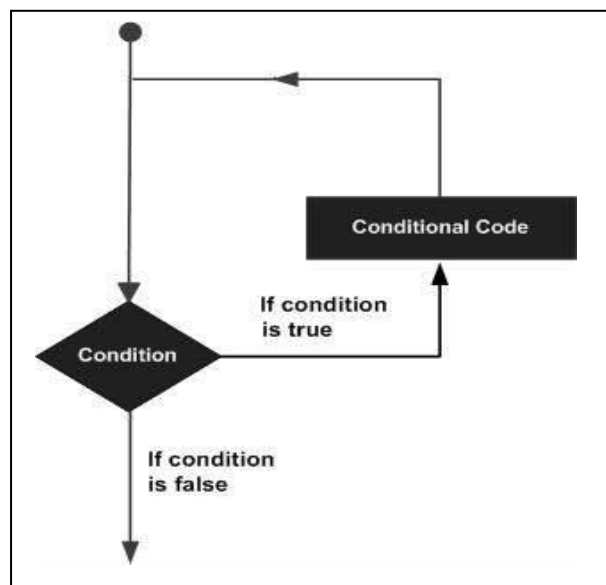
The switch case can use goto to jump over a different case.

Example :

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

## Loops

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



### for loop

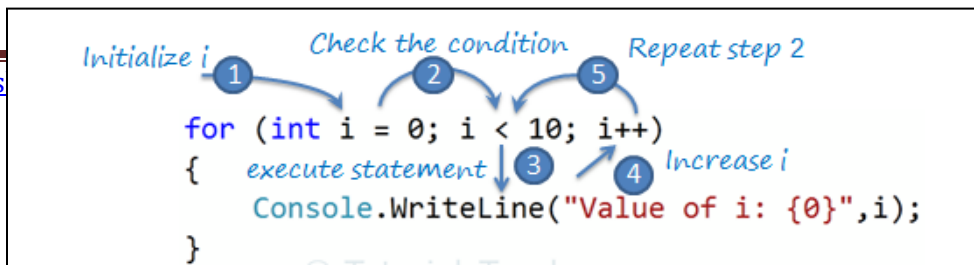
The **for** keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false.

Syntax :

```
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. condition: The condition is a boolean expression which will return either true or false.
3. steps: The steps defines the incremental or decremental part



### while loop:

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

Syntax:

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

As per the while loop syntax, the while loop includes a boolean expression as a condition which will return true or false. It executes the code block, as long as the specified conditional expression returns true. Here, the initialization should be done before the loop starts and increment or decrement steps should be inside the loop.

Example :

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

Output :

```
Value of i: 0
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
Value of i: 6
Value of i: 7
Value of i: 8
Value of i: 9
```

while loop includes an expression `i < 10`. Inside while loop, value of `i` increased to 1 (using `i++`). So, the above while loop will be executed till the value of `i` will be 10.

Use the *break* keyword to exit from a while loop as shown below.

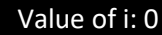
---

```
www.enosis.com int i = 0;
while (true)
{
    Console.WriteLine("Value of i: {0}", i);
    i++;
}
```



Example :

Output :



Value of i: 0

## 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.

Syntax :

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

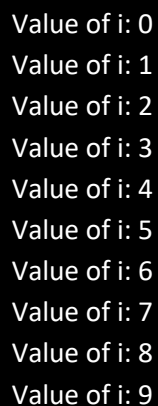
As per the syntax above, do-while loop starts with the 'do' keyword followed by a code block and boolean expression with 'while'.

Example:

```
int i = 0;
do
{
    Console.WriteLine("Value of i: {0}", i);

    i++;
} while (i < 10);
```

Output :



Value of i: 0  
Value of i: 1  
Value of i: 2  
Value of i: 3  
Value of i: 4  
Value of i: 5  
Value of i: 6  
Value of i: 7  
Value of i: 8  
Value of i: 9

## Array

---

**“An array is a collection of similar data types.”**

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

### Declaring Arrays

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

```
datatype[] arrayName;
```

where,

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

For example,

```
double[] balance;
```

### Initializing an Array

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

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

```
double[] balance = new double[10];
```

### Assigning Values to an Array

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

```
double[] balance = new double[10];  
balance[0] = 1734.4;
```

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

```
double[] bal = { 240.3, 583.69, 631.0};
```

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

```
int [] items = new int[5] { 39, 17, 9, 72, 25};
```

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

```
int [] marks = new int[] { 39, 17, 9, 72, 25};
```

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

```
int [] marks = new int[] { 39, 17, 9, 72, 25};  
int[] score = marks;
```

**NOTE :** An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

## Multi-dimensional Array

C# also supports multi-dimensional arrays. A multi-dimensional array is a two dimensional series like rows and columns.

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

The values of a multi-dimensional array can be accessed using two indexes. The first index is for the row and the second index is for the column. Both the indexes start from zero.

No of Rows  
No of Columns

```
int[,]. intArray = new int[3, 2] {  
    Row 0 { 1, 1 },  
    Row 1 { 1, 2 },  
    Row 2 { 1, 3 }  
};
```

Example :

```
int[,]. intArray = new int[3,2]{  
    {1, 2},  
    {3, 4},  
    {5, 6}  
};  
  
intArray[0,0]; //Output: 1  
intArray[0,1]; // 2  
  
intArray[1,0]; // 3  
intArray[1,1]; // 4  
  
intArray[2,0]; // 5  
intArray[2,1]; // 6
```

## 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]); // 1  
Console.WriteLine(intJaggedArray[0][2]); // 3  
Console.WriteLine(intJaggedArray[1][1]); // 5
```

## Methods

Methods are also called as functions. Methods make the maintenance of your application easier. Methods are extremely useful because they allow you to define your logic once, and use it at many times.

Syntax :

```
[attributes]
Access-modifier Return-type Method-name (Parameter-list)
{
    //Method-body
}
```

Following are the various elements of a method –

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

## Calling Methods in C#

You can call a method using the name of the method.

Example :

```
Class Test {
public int compare(int num1, int num2) {
    /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
}
```

```
public static void Main(string[] args)
{
    Test t1 = new Test();
    int result = t1.compare(20,10);
    Console.WriteLine(result);
}
```

## Method Parameters

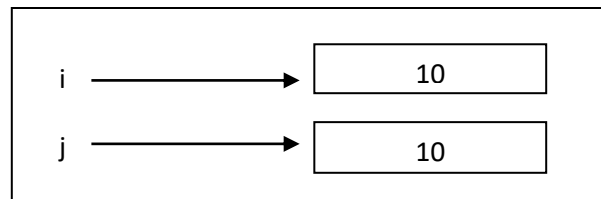
There are 4 different types of parameters a method can have :

1. Value Parameter
2. Reference Parameter (ref)
3. Output Parameter (out)
4. Parameter Arrays

### Value Parameter

Creates a copy of the parameter passed, so modifications does not affect each other.

```
static void disp(int j)
{
    J = 15;
}
static void Main(string[] args)
{
    int i = 10;
    disp(i);
    Console.WriteLine(i);
}
```

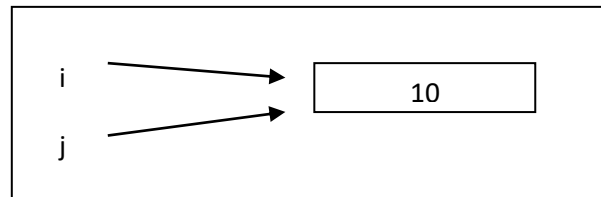


i and j are pointing to different memory locations. Operations on one variable will not affect the value of the other variable.

### Reference Parameter

The ref keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

```
static void disp(ref int j)
{
    J = 15;
}
static void Main(string[] args)
{
    int i = 10;
    disp(ref i);
    Console.WriteLine(i);
}
```



i and j are pointing to the same memory location. Operations on one variable will affect the value of the other variable.

### Output Parameter

The out keyword is used when you want a method to return more than one value.

## Parameter Arrays

The `params` keyword lets you specify a method parameter that takes a variable number of arguments. You can send a comma-separated list of arguments, or an array, or no arguments. `Params` keyword should be the last one in the parameter list in method declaration, and only one `params` keyword is permitted in a method declaration.

---

## 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.

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

Example :

```
enum WeekDays
{
    Monday = 0,
    Tuesday = 1,
    Wednesday = 2,
    Thursday = 3,
    Friday = 4,
    Saturday = 5,
    Sunday = 6
}
static void Main(String[] args)
{
    Console.WriteLine(WeekDays.Friday);
    Console.WriteLine((int)WeekDays.Friday);
}
```

Output:



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

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

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

### Enum methods:

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

Enum method	Description
Format	Converts the specified value of <code>enum</code> type to the specified string format.

Enum method	Description
GetName	Returns the name of the constant of the specified value of specified enum.
GetNames	Returns an array of string name of all the constant of specified enum.
GetValues	Returns an array of the values of all the constants of specified enum.
object Parse(type, string)	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.
bool TryParse(string, out TEnum)	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object. The return value indicates whether the conversion succeeded.

Example :

```
enum WeekDays
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

Console.WriteLine(Enum.GetName(typeof(WeekDays), 4));
Console.WriteLine("WeekDays constant names:");
foreach (string str in Enum.GetNames(typeof(WeekDays)))
    Console.WriteLine(str);
Console.WriteLine("Enum.TryParse():");
WeekDays wdEnum;
Enum.TryParse<WeekDays>("1", out wdEnum);
Console.WriteLine(wdEnum);
```

Output :

```
Friday
WeekDays constant names:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Enum.TryParse():
Tuesday
```

**Where to use enums**



Quite often we have situations where a class method takes as an argument a custom option. Let's say we have some kind of file access class and there is a file open method that has a parameter that might be one of read-mode, write-mode, read-write-mode, create-mode and append-mode. Now you might think of adding five static member fields to your class for these modes. Wrong approach! Declare and use an enumeration which is a whole lot more efficient and is better programming practice in my opinion.

---

## Classes

The Class consists of data and behavior. Class data (Member Variables) is represented by its fields and behavior (Member functions) is represented by its methods.

When you define a class, you define a blueprint for a data type. This does not actually define any data, but it does define what the class name means. That is, what an object of the class consists of and what operations can be performed on that object. Objects are instances of a class.

If you want to create a complex custom data types, then we can make use of classes.

Classes are declared by using the keyword **class** followed by the *class* name and a set of *class* members surrounded by curly braces.

### Note –

- Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

## Members of Class

So far, the only class members you've seen are Fields, Methods, Constructors, and Destructors. Here is a complete list of the types of members you can have in your classes:

- Constructors
- Destructors
- Fields
- Methods
- Properties
- Indexers
- Delegates
- Events

- Nested Classes

## Constructor

The purpose of class constructor is to initialize class fields. A class constructor is automatically called when an instance of a class is created.

Constructors always have the same name as the class and do not have any return values.

Constructors are not mandatory. If you do not provide a constructor, a default parameter-less constructor is automatically provided.

Constructors can be overloaded by the number and type of parameters.

## Destructor

Destructors have the same name as the class with tilt (~) symbol in front of them.

They do not take any parameters and do not return a value.

Destructors are places where you could put code to release any resources your class was holding during its lifetime. They are normally called when the C# garbage collector decides to clean your object from memory.

## Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are the attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Example :

```
// helper class
public class Test
{
    string str;
    // Constructor
    public Test(string s)
    {
        str = s;
    }
    // Instance Method
    public void printString()
    {
        Console.WriteLine("{0}", str);
    }
    // Destructor
    ~Test()
    {
        // Some resource cleanup routines
    }
}
// Program start class
class ExampleClass
{
    // Main begins program execution.
    public static void Main()
    {
        // Instance of Test
        Test obj = new Test("This is Test Class content");
```

## Static And Instance Class Members

When a class member includes a static modifier, the member is called as **static member**. When no static modifier is present the member is called as **non-static member** or **instance member**.

Static members are invoked using class name, where as instance members are invoked using instances (objects) of the class.

An instance member belongs to specific instance (object) of the class. If 3 objects are there of a class, then there will be 3 sets of instance members in the memory, where as there will be only one copy of a static member, no matter how many instances of a class are created.

## Static Constructor

Static constructors are used to initialize static fields in the class.

Static constructors are declared using the keyword static in front of the constructor name.

Static constructor is called only once, no matter how many instances you create.

Static constructors are called before instance constructors.

Example :

```
public class Circle
{
    public int radius;
    public static double pi;
    // Static Constructor
    public static Circle()
    {
        Console.WriteLine("Static Constructor invoked");
        pi = 3.14;
    }
    // Constructor
    public Circle(int r)
    {
        Console.WriteLine("Instance Constructor invoked");
        radius = r;
    }
    // instance Method
    public void area()
    {
        Console.WriteLine("Area of circle = {0}", pi * radius * radius);
    }
}
// Program start class
class ExampleClass
{
    public static void Main()
```

### Copy Constructor

**A parameterized constructor that contains a parameter of same class type is called as copy constructor.**  
Main purpose of copy constructor is to initialize new instance to the values of an existing instance.

```
class Test2
{
    int A, B;
    public Test2(int X, int Y)
    {
        A = X;
        B = Y;
    }
    //Copy Constructor
    public Test2(Test2 T)
    {
        A = T.A;
        B = T.B;
    }
    public void Print()
    {
        Console.WriteLine("A = {0}\tB = {1}", A, B);
    }
}
class CopyConstructor
{
    static void Main()
    {
        Test2 T2 = new Test2(80, 90);
        //Invoking copy constructor
        Test2 T3 = new Test2(T2);
        Test2 T3 = Test2(T2);

        T2.Print();
        T3.Print();
        Console.Read();
    }
}
```

## Private Constructor

You can also create a constructor as private. When a class contains at least one private constructor, then it is not possible to create an instance for the class. Private constructor is used to restrict the class from being instantiated when it contains every member as static.

- Private constructor a constructor with "private" access modifier in a class.
- A class with private constructor cannot be inherited.
- We cannot create an object of the class which has private constructor.
- A long with private constructor we can also have public constructor (overloaded constructor).
- To access the methods or properties in a class with private constructor we need to assign methods or properties with "static" keyword.

---

## Properties

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. C# properties enable this type of protection while also letting you access the property just like it was a field.

### Accessing Class Fields with Properties

```
Public class Customer
{
    Private int m_id = -1;
    Public int ID
    {
        Get { Return m_id;    }
        Set {
            m_id = value;
        }
    }
    Private string m_name = string.Empty;
    Public string Name
    {
        Get { Return m_name;    }
        Set {
            m_name = value;
        }
    }
}
Public class CustomerManagerWithProperties
```

```
{  
    Public static void Main()  
    {  
        Customer cust = new Customer();  
        cust.ID = 1;  
        cust.Name = "Amelio Rosales";  
        Console.WriteLine( "ID: {0}, Name: {1}", cust.ID, cust.Name);  
    }  
}
```

The *Customer* class has the *ID* and *Name* property implementations.

There are also private fields named *m\_id* and *m\_name*; which *ID* and *Name*, respectively, encapsulate. Each property has two accessors, *get* and *set*.

The accessor returns the value of a field. The *set* accessor sets the value of a field with the contents of *value*, which is the value being assigned by calling code. The *value* shown in the accessor is a C# reserved word.

This was a read/write property, but you can also create read-only properties, which you'll learn about next.

### Creating Read-Only Properties

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation.

#### **Example of Read-Only Properties**

```
Public class Customer
{
    Private int m_id = -1;
    Private string m_name = string.Empty;

    public Customer(int id, string name)
    {
        m_id = id;
        m_name = name;
    }
    Public int ID
    {
        get { Return m_id;    }
    }
    Public string Name
    {
        get { Return m_name;  }
    }
}
Public class ReadOnlyCustomerManager
{
    Public static void Main()
    {
        Customer cust = new Customer(1, "Amelio Rosales");
        Console.WriteLine( "ID: {0}, Name: {1}", cust.ID, cust.Name);
        Console.ReadKey();
    }
}
```

The *Customer* class in Listing 10-3 has two read-only properties, *ID* and *Name*. You can tell that each property is read-only because they only have *get*accessors. At some time, values for the *m\_id* and *m\_name* must be assigned, which is the role of the constructor in this example.

### Creating a Write-Only Property

You can assign values to, but not read from, a write-only property. A write-only property only has a *set*accessor.

#### **Example of Write-Only Properties**

```
Public class Customer
{
    Private int m_id = -1;
```

```
Public int ID
{
    set {
        m_id = value;
    }
}
Private string m_name = string.Empty;
Public string Name
{
    set {
        m_name = value;
    }
}
Public void DisplayCustomerData()
{
    Console.WriteLine("ID: {0}, Name: {1}", m_id, m_name);
}
}
Public class WriteOnlyCustomerManager
{
    Public static void Main()
    {
        Customer cust = new Customer();
        cust.ID = 1;
        cust.Name = "Amelio Rosales";
        cust.DisplayCustomerData();
        Console.ReadKey();
    }
}
```

This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class, shown in Listing 10-1.

The *set* accessors have been added, assigning *value* to the backing store fields, *m\_id* and *m\_name*.

### Creating Auto-Implemented Properties

The patterns you see here, where a property encapsulates a property with *get* and *set* accessors, without any other logic is common. C# 3.0 introduced a new syntax for a property, called an *auto-implemented property*, which allows you to create properties without *get* and *set* accessor implementations.

#### **Example of Auto-Implemented Properties**

```
Public class Customer
{
    Public int ID { get; set; }
```



```
Public string Name { get; set; }  
}  
Public class AutoImplementedCustomerManager  
{  
    Static void Main()  
    {  
        Customer cust = new Customer();  
        cust.ID = 1;  
        cust.Name = "Amelio Rosales";  
        Console.WriteLine( "ID: {0}, Name: {1}", cust.ID, cust.Name);  
        Console.ReadKey();  
    }  
}
```

Notice how the *get* and *set* accessors in Listing 10-5 do not have implementations. **In an auto-implemented property, the C# compiler creates the backing store field behind the scenes, giving the same logic that exists with traditional properties, but saving you from having to use all of the syntax of the traditional property.** As you can see in the *Main* method, the usage of an auto-implemented property is exactly the same as traditional properties, which you learned about in previous sections.

## Inheritance

---

Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

Inheritance is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the **base class**. The class that inherits the members of the base class is called the **derived class**.

When you derive a class from a base class, the derived class will inherit all members of the base class except constructors, though whether the derived class would be able to access those members would depend upon the accessibility of those members in the base class. C# gives us polymorphism through inheritance. Inheritance-based polymorphism allows us to define methods in a base class and override them with derived class implementations.

Thus if you have a base class object that might be holding one of several derived class objects, polymorphism when properly used allows you to call a method that will work differently according to the type of derived class the object belongs to.

Consider the following class which we'll use as a base class.

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("Animal constructor");
    }
    public void Greet()
    {
        Console.WriteLine("Animal says Hello");
    }
    public void Talk()
    {
        Console.WriteLine("Animal talk");
    }
    public virtual void Sing()
    {
        Console.WriteLine("Animal song");
    }
}
```

**Now see how we derive another class from this base class.**

```
class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("Dog constructor");
    }
    public new void Talk()
    {
        Console.WriteLine("Dog talk");
    }
    public override void Sing()
    {
        Console.WriteLine("Dog song");
    }
};
```

```
Animal a1 = new Animal();
a1.Talk();
a1.Sing();
a1.Greet();
```

```
//Output
Animal constructor
Animal talk
Animal song
Animal says Hello
```

Okay, that came out just as expected. Now try this code out.

```
Animal a2 = new Dog();
a2.Talk();
a2.Sing();
a2.Greet();
```

```
//Output
Animal constructor
Dog constructor
Animal talk
Dog song
Animal says Hello
```

## THIS & NEW KEYWORD

```
namespace InheritExamples
{
    public class Maths
    {
        protected int n1;
        protected int n2;
        protected int total;
```

```
public Maths()
{
    Console.WriteLine("maths is called");
}
public Maths(int n1,int n2)
{
    this.n1 = n1;
    this.n2 = n2;
    //Console.WriteLine("maths is called");
}
public void add()
{
    total = n1 + n2;
    Console.WriteLine("add=" + total);
}
}
public class Calculator : Maths
{
    protected new int n2;

    public Calculator()
    {
        Console.WriteLine("calculator is called");
    }
    public Calculator(int nn1, int nn2) : base(nn1,nn2)
    {
        Console.WriteLine("parameterized calculator is called");
    }
    public void sub()
    {
        total = n1 - n2;
        Console.WriteLine("sub=" + total);
    }

}
public class Scientific : Calculator
{
    public Scientific()
    {
        Console.WriteLine("Scientific is called");
    }

}
}
```

```
}
```

## Interface

---

An interface looks like a class, but has no implementation. The only thing it contains are definitions of events, indexers, methods and/or properties. The reason interfaces only provide definitions is because they are inherited by classes and structs, which must provide an implementation for each interface member defined.

Because interfaces must be implemented by derived classes and structs, they define a contract.

### Defining an Interface: MyInterface.cs

```
Interface IMyInterface
{
    void MethodToImplement();
}
```

This *interface* has a single method named *MethodToImplement()*.

This could have been any type of method declaration with different parameters and return types. I just chose to declare this method with no parameters and a *void* return type to make the example easy.

Notice that this method does not have an implementation (instructions between curly braces - {}), but instead ends with a semi-colon, ";". This is because the *interface* only specifies the signature of methods that an inheriting *class* or *struct* must implement.

### Using an Interface: InterfaceImplementer.cs

```
Class InterfaceImplementer : IMyInterface
{
    static void Main()
    {
        InterfaceImplementer ilmp = new InterfaceImplementer();
        ilmp.MethodToImplement();
    }
    public void MethodToImplement()
    {
        Console.WriteLine("MethodToImplement() called.");
    }
}
```

The *InterfaceImplementer* class implements the *IMyInterface* interface. Indicating that a *class* inherits an *interface* is the same as inheriting a *class*. In this case, the following syntax is used:

### Class InterfaceImplementer : IMyInterface

Now that this *class* inherits the *IMyInterface* interface, it must implement its members. It does this by implementing the *MethodToImplement()* method. Notice that this method implementation has the exact same signature, parameters and method name, as defined in the *IMyInterface* interface. Interfaces may also inherit other interfaces.

#### **Interface Inheritance: InterfaceInheritance.cs**

```
Interface IParentInterface
{
    void ParentInterfaceMethod();
}
interface IMyInterface : IParentInterface
{
    void MethodToImplement();
}
class InterfaceImplementer : IMyInterface
{
    static void Main()
    {
        InterfaceImplementer ilmp = new InterfaceImplementer();
        ilmp.MethodToImplement();
        ilmp.ParentInterfaceMethod();
    }
    public void MethodToImplement()
    {
        Console.WriteLine("MethodToImplement() called.");
    }
    public void ParentInterfaceMethod()
    {
        Console.WriteLine("ParentInterfaceMethod() called.");
    }
}
```

*IMyInterface* and the *interface* it inherits, *IParentInterface*. When one *interface* inherits another, any implementing *class* or *struct* must implement every *interface* member in the entire inheritance chain.

## Abstract Class

---

An abstract class means that, no object of this class can be instantiated, but can make derivations of this.

**An example of an abstract class declaration is:**

```
abstract class absClass
{
}
```

An abstract class can contain either abstract methods or non abstract methods. Abstract members do not have any implementation in the abstract class, but the same has to be provided in its derived class.

An example of an abstract method:

```
abstract class absClass
{
    public abstract void abstractMethod();
}
```

Also, note that an abstract class does not mean that it should contain abstract members. Even we can have an abstract class only with non abstract members. For example:

```
abstract class absClass
{
    public void NonAbstractMethod()
    {
        Console.WriteLine("NonAbstract Method");
    }
}
```

**A sample program that explains abstract classes:**

```
//Creating an Abstract Class
abstract class absClass
{
    //A Non abstract method
    public int AddTwoNumbers(int Num1, int Num2)
    {
        return Num1 + Num2;
    }
    //An abstract method, to be
    //overridden in derived class
    public abstract int MultiplyTwoNumbers(int Num1, int Num2);
}
```

```
}  
//A Child Class of absClass  
class absDerived:absClass  
{  
    [STAThread]  
    static void Main(string[] args)  
    {  
        //You can create an  
        //instance of the derived class  
        absDerived calculate = new absDerived();  
        int added = calculate.AddTwoNumbers(10,20);  
        int multiplied = calculate.MultiplyTwoNumbers(10,20);  
        Console.WriteLine(<span class='cpp-string'>"Added : {0},  
            Multiplied : {1}"</span>, added, multiplied);  
    }  
    //using override keyword,  
    //implementing the abstract method  
    //MultiplyTwoNumbers  
    public override int MultiplyTwoNumbers(int Num1, int Num2)  
    {  
        return Num1 * Num2;  
    }  
}  
}
```

### Important rules applied to abstract classes

An abstract class cannot be a sealed class. I.e. the following declaration is incorrect.

Declaration of abstract methods are only allowed in abstract classes.

**An abstract method cannot be private.**

//Incorrect **private abstract int MultiplyTwoNumbers();**

The access modifier of the abstract method should be same in both the abstract class and its derived class. If you declare an abstract method as protected, it should be protected in its derived class. Otherwise, the compiler will raise an error.

An abstract method cannot have the modifier virtual. Because an abstract method is implicitly virtual.//Incorrect : **public abstract virtual int MultiplyTwoNumbers();**

An abstract member cannot be static. **public abstract static int MultiplyTwoNumbers();**



### **Abstract class vs. Interface :**

An abstract class can have abstract members as well non abstract members. But in an interface all the members are implicitly abstract and all the members of the interface must override to its derived class.

An example of interface:

```
interface ISampleInterface
{
    //All methods are automaticall abstract
    int AddNumbers(int Num1, int Num2);
    int MultiplyNumbers(int Num1, int Num2);
}
```

- Defining an abstract class with abstract members has the same effect to defining an interface.
- The members of the interface are public with no implementation. Abstract classes can have protected parts, static methods, etc.
- A class can inherit one or more interfaces, but only one abstract class.
- Abstract classes can add more functionality without destroying the child classes that were using the old version.

### **DIFFERENCE BETWEEN INTERFACE & ABSTRACT CLASS**

INTERFACE	ABSTRACT CLASS
An Interface cannot implement methods.	An abstract class can implement methods.
An Interface can only inherit from another Interface.	An abstract class can inherit from a class and one or more interfaces.
An Interface cannot contain fields.	An abstract class can contain fields.
An Interface can contain property definitions.	An abstract class can implement a property.
An Interface cannot contain constructors or destructors.	An abstract class can contain constructors or destructors.
An Interface can be inherited from by structures.	An abstract class cannot be inherited from by structures.
An Interface can support multiple inheritance.	An abstract class cannot support multiple inheritance.

## Struct

In C++ a struct is just about the same as a class for all purposes except in the default access modifier for methods. In C# a struct are a pale puny version of a class. I am not sure why this was done so, but perhaps they decided to have a clear distinction between structs and classes. Here are some of the drastic areas where classes and structs differ in functionality.

- **structs are stack objects and however much you try you cannot create them on the heap**
- **structs cannot inherit from other structs though they can derive from interfaces**
- **You cannot declare a default constructor for a struct, your constructors must have parameters**
- **The constructor is called only if you create your struct using new, if you simply declare the struct just as in declaring a native type like int, you must explicitly set each member's value before you can use the struct**

```
struct Student : IGrade
{
    public int maths;
    public int english;
    public int csharp;

    //public member function
    public int GetTot()
    {
        return maths+english+csharp;
    }

    //We have a constructor that takes an int as argument
    public Student(int y)
    {
        maths = english = csharp = y;
    }
    //This method is implemented because we derive
    //from the IGrade interface
    public string GetGrade()
    {
        if(GetTot() > 240 )
            return "Brilliant";
        if(GetTot() > 140 )
            return "Passed";
        return "Failed";
    }
}
```

```
interface IGrade
{
    string GetGrade();
}
```

Well, now let's take a look at how we can use our struct.

```
Student s1 = new Student();
Console.WriteLine(s1.GetTot());
Console.WriteLine(s1.GetGrade());
```

```
//Output
0
Failed
```

Here the default constructor gets called. This is automatically implemented for us and we cannot have our own default parameter-less constructor. The default parameter-less constructor simply initializes all values to their zero-equivalents. This is why we get a 0 as the total.

```
Student s2;
s2.maths = s2.english = s2.csharp = 50;
Console.WriteLine(s2.GetTot());
Console.WriteLine(s2.GetGrade());
```

```
//Output
150
Passed
```

Because we haven't used new, the constructor does not get called. Anyway you have to initialize all the member fields. If you comment out the line that does the initialization you will get a compiler error :-*Use of unassigned local variable 's2'*

```
Student s3 = new Student(90);
Console.WriteLine(s3.GetTot());
Console.WriteLine(s3.GetGrade());
```

```
//Output
270
Brilliant
```

This time we use our custom constructor that takes an int as argument.

## DIFFERENCE BETWEEN STRUCTURE AND CLASS

**The struct is value type in C# and it inherits from System.ValueType**

The class is reference type in C# and it inherits from the System.Object Type

**The struct value will be stored on the stack memory.**

The class object is stored on the heap memory. The object will be under garbage collection and automatically removed when there is no reference to the created objects.

**The struct use the array type and it's good to use for read only and light weight object.**

The class uses the collection object type and it can perform all the operations and designed for

complex data type storage.

**The struct can't be base type to the classes and also to the other structure.**

The class can inherit another class, interface and it can be base class to another class.

**The struct can only inherit the interfaces**

The class can inherit the interfaces, abstract classes.

**The struct can have only constructor.**

The class can have the constructor and destructor.

**The struct can instantiated without using the new keyword.**

The new keyword should be used to create the object for the class

**The struct can't have the default constructor**

The class will have the default constructor

**The struct is by default sealed class hence it will not allow to inherit. It can't use the abstract, sealed, base keyword.**

The class can be declared as abstract, sealed class

**The struct can't use the protected or protected internal modifier.**

The class can use all the access modifiers.

**The struct can't initialize at the time of declaration.**

The class can have the initializes fields.

---

## Partial Classes

Partial classes span multiple files. How can you use the partial modifier on a C# class declaration? With partial, you can physically separate a class into multiple files. This is often done by code generators.

**Program that uses partial class: C#**

```
class Program
{
    static void Main()
    {
        A.A1();
        A.A2();
    }
}

Contents of file A1.cs: C#
partial class A
{
    public static void A1()
    {
        Console.WriteLine("A1");
    }
}
```

**Contents of file A2.cs: C#**

```
partial class A
{
    public static void A2()
    {
        Console.WriteLine("A2");
    }
}
```

**Output**

A1  
A2

Partial is required here.

## Sealed Classes

---

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a **sealed class**, the class cannot be inherited.

In C#, the sealed modifier is used to define a class as **sealed**. In Visual Basic .NET the **NotInheritable** keyword serves the purpose of sealed. If a class is derived from a sealed class then the compiler throws an error. If you have ever noticed, structs are sealed. You cannot derive a class from a struct.

The following class definition defines a sealed class in C#: *// Sealed class*

```
Sealed class SealedClass
{
}
```

I create a sealed class **SealedClass** and use it from Class1. If you run this code then it will work fine. But if you try to derive a class from the SealedClass, you will get an error.

```
Class Class1
{
    Static void Main(string[] args)
    {
        SealedClass sealedCls = new SealedClass();
        int total = sealedCls.Add(4, 5);
        Console.WriteLine("Total = " + total.ToString());
    }
}
// Sealed class
Sealed class SealedClass
{
    Public int Add(int x, int y)
    {
        return x + y;
    }
}
```

### Sealed Methods and Properties

**You can also use the sealed modifier on a method or a property that overrides a virtual method or property in a base class.** This enables you to allow classes to derive from your class and prevent other developers that are using your classes from overriding specific virtual methods and properties.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("X.F3"); }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F"); }
    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

## Static Class

Static class is a specialized class that cannot have any objects.

### Example#1 for Static Class in C#:

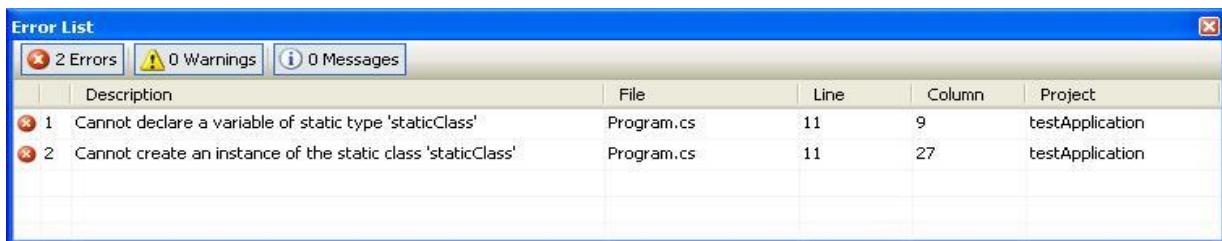
```
public static class staticClass
{
}
public class testStaticClass
{
    public static void Main()
    {
        staticClassobj = new staticClass(); //ERROR
    }
}
```

In this example, the class staticClass is a static class because it has the keyword static in its declaration.

**Hence staticClass cannot be instantiated.**

**But inside Main method of testStaticClass, staticClass is instantiated with the object obj.**

This is incorrect leading to error:



Static class cannot be instantiated. But when static class cannot be instantiated, how can its data and members be accessed? The members of a static class belongs to the class itself and they can only be accessed using the class name.

### Static Class in C#:

```
public static class staticClass
{
    static intstaticInteger;
    public static void setStaticInteger(inttmpData)
    {
        staticInteger = tmpData;
    }
    public static intgetStaticInteger()
    {
        return staticInteger;
    }
}
public class testStaticClass
{
    public static void Main()
    {
        staticClass.setStaticInteger(5);
        Console.WriteLine("staticInteger = "+ staticClass.getStaticInteger());
        Console.ReadLine();
    }
}
```

**Output:**



In this example, the members of staticClass are accessed using syntax as highlighted below:

**staticClass.setStaticInteger(5);**

**Characteristics of Static Class in C#**

- **Members of Static Class Can Only Be Static**
- **Static Class Can Contain Only Static Constructor**

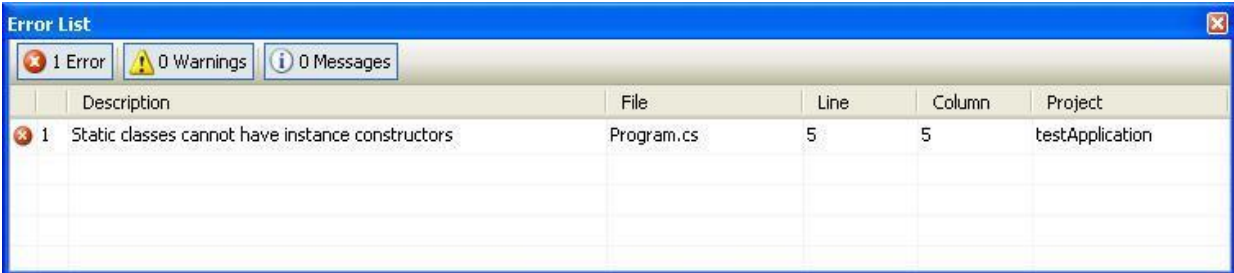
**Static class can contain only static constructors and not instance constructors.**

```
public static class staticClass
{
    static Class()
    {
```



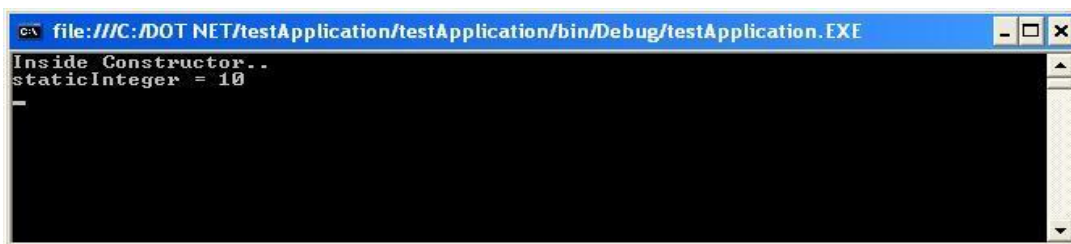
```
        Console.WriteLine("Inside Constructor..");  
    }  
}
```

The above code has a non-static constructor inside static class. This is incorrect leading to error:



The above code will work if the constructor includes the keyword static as shown below:

```
public static class staticClass  
{  
    static staticClass()  
    {  
        Console.WriteLine("Inside Constructor..");  
    }  
    public static int staticInteger;  
}  
public class testStaticClass  
{  
    public static void Main()  
    {  
        staticClass.staticInteger = 10;  
        Console.WriteLine("staticInteger = " + staticClass.staticInteger);  
        Console.ReadLine();  
    }  
}
```



Output:

**In the above example, constructor of staticClass is marked with the keyword static.**

**This is legal.** The static constructor will get invoked before any members of the static class get invoked. This is proven from the above output.

### Important Points of Static

- Static Class is Implicitly Sealed
- Static Class Cannot Be Declared as Sealed
- Static Class Cannot be Declared as Abstract
- Static Class Should Inherit Only Object
- Static class cannot inherit any other static class / concrete class.
- Static class is allowed to inherit only the Object.

## Extention Methods

---

Extension methods enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

An extension method is a special kind of static method, but they are called as if they were instance methods on the extended type.

### How to use extension methods?

An extension method is a static method of a static class, where the "this" modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.

Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

```
public static class ExtensionMethods
{
    public static string UppercaseFirstLetter(this string value)
    {
        // Uppercase the first letter in the string.
        if (value.Length > 0)
        {
            char[] array = value.ToCharArray();
            array[0] = char.ToUpper(array[0]);
            return new string(array);
        }
        return value;
    }
}
```

```
class Program
{
    static void Main()
    {
        // Use the string extension method on this value.
        string value = "dot net perls";
        value = value.UppercaseFirstLetter();
        Console.WriteLine(value);
    }
}
```

## Encapsulation

---

Encapsulation means where we consider the related group properties, methods and other members as a single unit. This is usually called a class. This specifies access levels for each properties, methods and for members of the class. In C # you can manage encapsulation using access modifiers.

Abstraction allows you to hide the implementation while encapsulation helps to provide different levels of access to the members and methods.

**In c#, Encapsulation is implemented using the [access modifier](#) keywords.**

Encapsulation is a process of binding the data members and member functions into a single unit.

- **Encapsulation means hiding the internal details of an object, i.e. how an object does something.**
- **Encapsulation prevents clients from seeing its inside view, where the behaviour of the abstraction is implemented.**
- **Encapsulation is a technique used to protect the information in an object from the other object.**
- **Hide the data for security such as making the variables as private, and expose the property to access the private data which would be public.**

## Polymorphism

---

It is ability to take more than one form. An operation may exhibit different behavior in different situations.

**Ex:-** An addition operation involving two numeric values will produce sum and the same addition operation will produce a string. If we pass parameter numeric value then method return sum(numeric value) and if we pass parameter String then same method is return string value .this called Polymorphism.

Polymorphism is one of the important concepts in object oriented programming.

**This means taking different forms i.e same operation can have different behavior's. In a real world example take the operation "move". A bird will "Fly" and a dog will "walk" but both moves. In programming we can take overriding and overloading as examples for polymorphism.**

## Method overloading

---

Method will have same name but different argument passed. This is a compile time polymorphism.

*Example*

I have called the method "**CalculateTotalPrice**" with different arguments. Depending on the arguments related function is called.

<a href="#">Class Billing</a>
-------------------------------

```
{
    public double TotalPrice = 0.0;
    public double CalculateTotalPrice(double UnitPrice,int NumberOfItems)
    {
        TotalPrice = UnitPrice * NumberOfItems;
        return TotalPrice;
    }
    public double CalculateTotalPrice(doubleUnitPrice, intNumberOfItems,double
Discount)
    {
        TotalPrice = (UnitPrice * NumberOfItems)-((UnitPrice * NumberOfItems * Discount) /
100);
        return TotalPrice;
    }
    static void Main(string[] args)
    {
        string Unit_Price;
        string Number_Of_Items;
        string Discount_;
        double Total_Price;
        double Total_Price_Discount;

        Console.WriteLine("Enter the Unit Price : ");
        Unit_Price = Console.ReadLine();

        Console.WriteLine("Enter the Number Of Items : ");
        Number_Of_Items = Console.ReadLine();

        Console.WriteLine("Enter the Discout : ");
        Discount_ = Console.ReadLine();

        Billing B = newBilling();

        Total_Price = B.CalculateTotalPrice(Convert.ToDouble(Unit_Price),
Convert.ToInt16(Number_Of_Items));

        Total_Price_Discount = B.CalculateTotalPrice(Convert.ToDouble(Unit_Price),
Convert.ToInt16(Number_Of_Items), Convert.ToDouble(Discount_));

        Console.WriteLine("Total Price : {0}",Total_Price);
        Console.WriteLine("Total Price with Discount {0} : ",Total_Price_Discount);
        Console.WriteLine("\nPress any key to continue...");
        System.ConsoleKeyInfo press = Console.ReadKey(true);
    }
}
```

```
}  
}  
}
```

## Method Hiding

Why did the compiler in the second listing generate a warning? Because C# not only supports method overriding, **but also method hiding**. Simply put, if a method is not overriding the derived method, it is hiding it. A hiding method has to be declared using the **new** keyword. The correct class definition in the second listing is thus:

```
class A  
{  
    public void Foo() { Console.WriteLine("A::Foo()"); }  
}  
class B : A  
{  
    public new void Foo() { Console.WriteLine("B::Foo()"); }  
}  
class Test  
{  
    static void Main(string[] args)  
    {  
        A a;  
        B b;  
        a = new A();  
        b = new B();  
        a.Foo(); // output --> "A::Foo()"  
        b.Foo(); // output --> "B::Foo()"  
        a = new B();  
        a.Foo(); // output --> "A::Foo()"  
    }  
}
```

In other words, "Many forms of a single object is called Polymorphism."

Def :

**When a message can be processed in different ways is called polymorphism. Polymorphism means many forms.**

Polymorphism is one of the fundamental concepts of OOP.

[Polymorphism provides following features:](#)

- It allows you to invoke methods of derived class through base class reference during runtime.
- It has the ability for classes to provide different implementations of methods that are called through the same name.

### Polymorphism is of two types:

1. Compile time polymorphism/Overloading
2. Runtime polymorphism/Overriding

### Compile Time Polymorphism

Compile time polymorphism is method and operators overloading. It is also called early binding. In method overloading method performs the different task at the different input parameters.

### Runtime Time Polymorphism

Runtime time polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, you change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.

**Caution:** Don't confused method overloading with method overriding, they are different, unrelated concepts. But they sound similar.

### Method Overriding(RunTime Polymorphism)

---

Whereas **Overriding** means changing the functionality of a method without changing the signature. We can override a function in base class by creating a similar function in derived class. This is done by using virtual/overrides keywords.

Base class method has to be marked with virtual keyword and we can override it in derived class using override keyword.

Derived class method will completely overrides base class method i.e. when we refer base class object created by casting derived class object a method in derived class will be called.

Example:

```
// Base class
public class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base Class Method");
    }
}

// Derived class
public class DerivedClass : BaseClass
{
}
```

```
public override void Method1()
{
    Console.WriteLine("Derived Class Method");
}
}
// Using base and derived class
public class Sample
{
    public void TestMethod()
    {
        // calling the overridden method
        DerivedClass objDC = new DerivedClass();
        objDC.Method1();
        // calling the baesd class method
        BaseClass objBC = (BaseClass)objDC;
        objDC.Method1();
    }
}
```

**Output : Derived Class Method**  
**Derived Class Method**

### Differences Among Method Overriding, Method Hiding (new Keyword) and Method Shadowing in C#

1. The "override" modifier extends the base class method, and the "new" modifier hides it.
2. The "virtual" keyword modifies a method, property, indexer, or event declared in the base class and allows it to be overridden in the derived class.
3. The "override" keyword extends or modifies a virtual/abstract method, property, indexer, or event of the base class into the derived class.
4. The "new" keyword is used to hide a method, property, indexer, or event of the base class into the derived class.
5. If a method is not overriding the derived method then it is hiding it. A hiding method must be declared using the new keyword.
6. Shadowing is another commonly used term for hiding. The C# specification only uses "hiding" but either is acceptable. Shadowing is a VB concept.

### **What are the differences between method hiding and overriding in C#?**

1. For hiding the base class method from derived class simply declare the derived class method with the new keyword.

Whereas in C#, for overriding the base class method in a derived class, you need to declare the base class method as virtual and the derived class method as overridden.

2. If a method is simply hidden then the implementation to call is based on the compile-time type of the argument "this".  
Whereas if a method is overridden then the implementation to be called is based on the run-time type of the argument "this".
3. New is reference-type specific, overriding is object-type specific.

#### **What are the differences between method hiding and method shadowing?**

1. Shadowing is a VB concept. In C#, this concept is called hiding.
2. The two terms mean the same in C#. Method hiding == shadowing
3. In short, name "hiding" in C# (new modifier) is called shadowing in VB.NET (keyword Shadows).
4. In C# parlance, when you say "hiding" you're usually talking about inheritance, where a more derived method "hides" a base-class method from the normal inherited method call chain.
5. When you say "shadow" you're usually talking about scope; an identifier in an inner scope is "shadowing" an identifier at a higher scope.
6. In other languages, what is called "hiding" in C# is sometimes called "shadowing" as well.



```
namespace ConsoleApplication2
{
    class BaseClass
    {
        public void Method()
        {
            Console.WriteLine("Base - Method");
        }
    }

    class DerivedClass1 : BaseClass
    {
        public new virtual void Method()
        {
            Console.WriteLine("Derived1 - Method");
        }
    }

    class DerivedClass2 : DerivedClass1
    {
        public override void Method()
        {
            Console.WriteLine("Derived2 - Method");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            bc.Method(); //Base - Method

            DerivedClass1 dc1 = new DerivedClass1();
            dc1.Method(); //Derived1 - Method

            DerivedClass2 dc2 = new DerivedClass2();
            dc2.Method(); //Derived2 - Method

            BaseClass bcdcl = new DerivedClass1();
            bcdcl.Method(); //Base - Method

            DerivedClass1 dc1dc2 = new DerivedClass2();
            dc1dc2.Method(); //Derived2 - Method

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

## Abstraction

Abstraction in Object Oriented Concepts means hiding the implementation and shows only the necessary features of the object. For example we can achieve abstraction by creating classes. Sometimes we may not want to know what is inside the class.

- Abstraction is a process of hiding the implementation details and displaying the essential features.
- Abstraction is "To represent the essential feature without representing the background details."
- Abstraction lets you focus on what the object does instead of how it does it.
- Abstraction provides you a generalized view of your classes or object by providing relevant information.

- Abstraction is the process of hiding the working style of an object, and showing the information of an object in an understandable manner.
- Abstraction is "To represent the essential feature without representing the back ground details."
- Abstraction lets you focus on what the object does instead of how it does it.
- Abstraction provides you a generalized view of your classes or object by providing relevant information.
- Abstraction is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

### Difference between Abstraction and Encapsulation

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. <b>Abstraction</b> - Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. <b>Encapsulation</b> - Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

### Exception in C#

An exception is a unforeseen error that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

There are two types of exceptions in .Net, exceptions generated by the executing program and exceptions generated by the CLR. Exceptions provide a programming paradigm for detecting and reacting to unexpected events. When an exception arises, the state of the program is saved, the normal flow is interrupted and the control is passed to an **exception handler** (if such exists in the current context).

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.

```
class ExceptionsDemo
{
    static void Main()
    {
        string fileName = "WrongTextFile.txt";
        ReadFile(fileName);
    }

    static void ReadFile(string fileName)
    {
        TextReader reader = new StreamReader(fileName);
        string line = reader.ReadLine();
        Console.WriteLine(line);
        reader.Close();
    }
}
```

### Exception Handling

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** – A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** – The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** – A program throws an exception when a problem shows up. This is done using a throw keyword.

```
static void ReadFile(string fileName)
{
    // Exceptions could be thrown in the code below
    try
    {
        TextReader reader = new StreamReader(fileName);
        string line = reader.ReadLine();
    }
}
```

```
    Console.WriteLine(line);
    reader.Close();
}
catch (FileNotFoundException fnfe)
{
    // Exception handler for FileNotFoundException
    // We just inform the user that there is no such file
    Console.WriteLine(
        "The file '{0}' is not found.", fileName);
}
catch (IOException ioe)
{
    // Exception handler for other input/output exceptions
    // We just print the stack trace on the console
    Console.WriteLine(ioe.StackTrace);
}
}
```

### Throwing Exceptions (the throw Construct)

Exceptions in C# are thrown using the keyword **throw**. We need to provide an instance of the exception, containing all the necessary information about the error. Exceptions are normal classes and the only requirement is that they inherit directly or indirectly from the **System.Exception** class.

Here is an example:

```
static void Main()
{
    Exception e = new Exception("There was a problem");
    throw e;
}
```

The result from running this program is:

```
Unhandled Exception: System.Exception: There was a problem
at Exceptions.Demo1.Main() in Program.cs:line 11
```

## Delegate

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**.

Syntax for delegate declaration is –

```
delegate <return type> <delegate-name> <parameter list>
```

A delegate is a type safe function pointer. That is, it holds a reference (pointer) to a function. The signature of the delegate must match the signature of the function, the delegate points to, otherwise you get a compiler error. This is the reason delegates are called as type safe function pointers.

A delegate is similar to a class. You can create an instance of it, and when you do so, you pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate will point to. All the delegates are implicitly derived from `System.Delegate` class.

The diagram shows three code snippets with annotations. The first snippet is `public delegate void Print(int value);`. Annotations include: 'Access modifier' pointing to 'public', 'Delegate type' pointing to 'void', and 'Delegate function signature' pointing to 'Print(int value)'. The second snippet is `Print printDel = PrintNumber;`. An annotation 'Function signature must match with delegate signature' points to 'Print' and 'PrintNumber'. The third snippet is `public static void PrintNumber(int num) { Console.WriteLine("Number: {0,-12:N0}",num); }`. An annotation 'Function signature must match with delegate signature' points to 'PrintNumber(int num)'.

Example :

```
class Program
{
    public delegate void DelTest(int value);

    static void Main(string[] args)
    {
        DemoMethod(DispInNumber, 10000);
        DemoMethod(DispInMoney, 10000);
    }
    public static void DemoMethod(DelTest delegateFunc, int number)
    {
        delegateFunc(number);
    }
    public static void DispInNumber(int num)
    {
        Console.WriteLine("Number: {0,-12:N0}",num);
    }
    public static void DispInMoney(int money)
    {
        Console.WriteLine("Money: {0:C}", money);
    }
}
```

## Multicast Delegate

The delegate can point to multiple methods. A delegate that points to multiple methods is called a multicast delegate. The "+" operator adds a function to the delegate object and the "-" operator removes an existing function from a delegate object.

Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate –

**Example :**

```
public delegate void DelTest(int value);
static void Main(string[] args)
{
    DelTest Del = DispNumeric;
    Del += DispHexadecimal;
    Del += DispMoney;
    Del(100000);
}
public static void DispNumber(int num)
{
    Console.WriteLine("Number : {0,-12:N0}", num);
}
public static void DispMoney(int money)
{
    Console.WriteLine("Money : {0:C}", money);
}
public static void DispHexadecimal(int dec)
{
    Console.WriteLine("Hexadecimal : {0:X}", dec);
}
```

Output :

```
Number : 1,00,000
Hexadecimal : 186A0
Money : $ 1,00,000.00
```

## Array of Delegates

Creating an array of delegates is very similar to declaring an array of any type. The following example has a couple of static methods to perform specific math related operations. Then you use delegates to call these methods.

```
namespace Delegates
{
    public class Operation
    {
        public static void Add(int a, int b)
        {
```

```
        Console.WriteLine("Addition={0}", a + b);
    }
    Public static void Multiple(int a, int b)
    {
        Console.WriteLine("Multiply={0}", a * b);
    }
}
Class Program
{
    delegate void DelOp(int x, int y);
    Static void Main(string[] args)
    {
        // Delegate instantiation
        DelOp[] obj =
        {
            new DelOp(Operation.Add),
            new DelOp(Operation.Multiple)
        };
        for (int i = 0; i < obj.Length; i++)
        {
            obj[i](2, 5);
            obj[i](8, 5);
            obj[i](4, 6);
        }
        Console.ReadLine();
    }
}
```

In this code, you instantiate an array of Delop delegates. Each element of the array is initialized to refer to a different operation implemented by the operation class. Then, you loop through the array, apply each operation to three different values. After compiling this code, the output will be as follows;

### Anonymous Methods

**Anonymous methods, as their name implies, are nameless methods.** They prevent creation of separate methods, especially when the functionality can be done without a new method creation. Anonymous methods provide a cleaner and convenient approach while coding.

Define an anonymous method with the delegate keyword and a nameless function body. This code assigns an anonymous method to the delegate. The anonymous method must not have a signature.

The signature and return type is inferred from the delegate type. For example if the delegate has three parameters and return a double type, then the anonymous method would also have the same signature.

```
namespace Delegates
{
    class Program
    {
        // Delegate Definition
        delegate void operation();
        Static void Main(string[] args)
        {
            // Delegate instantiation
            operation obj = delegate
            {
                Console.WriteLine("Anonymous method");
            };
            obj();
            Console.ReadLine();
        }
    }
}
```

The anonymous methods reduce the complexity of code, especially where there are several events defined. With the anonymous method, the code does not perform faster. The compiler still defines methods implicitly.

## Event

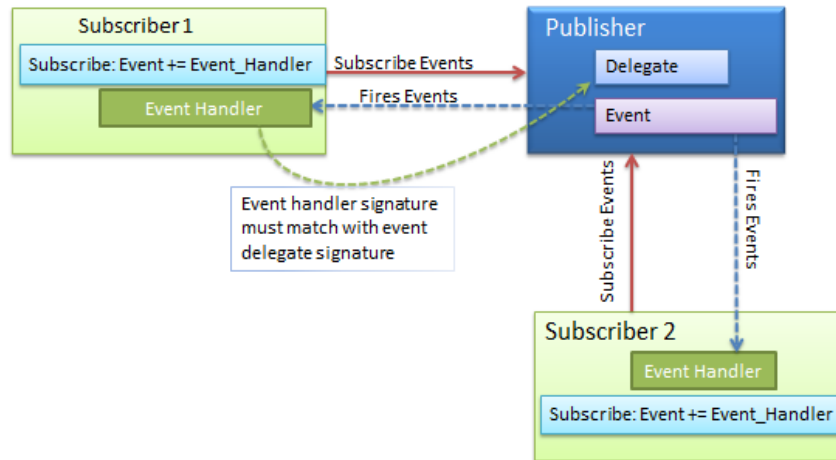
---

Events enable a [class](#) or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

**Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

An event is nothing but an encapsulated delegate. As we have learned in the previous section, a delegate is a reference type data type.





A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

#### Example for Event :

```

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);
    }
}
```

PrintHelper is a publisher class that publishes the beforePrint event. Notice that in each print method, it first checks to see if beforePrintEvent is not null and then it calls *beforePrintEvent()*. beforePrintEvent is an object of type BeforPrint delegate, so it would be null if no class is subscribed to the event and that is why it is necessary to check for null before calling a delegate.

Now, let's create a subscriber. Consider the following simple Number class for example.

```
class Number
{
    private PrintHelper _printHelper;

    public Number(int val)
    {
        _value = val;

        _printHelper = new PrintHelper();
        //subscribe to beforePrintEvent event
        _printHelper.beforePrintEvent += printHelper_beforePrintEvent;
    }
    //beforePrintevent handler
    void printHelper_beforePrintEvent()
    {
        Console.WriteLine("BeforPrintEventHandler: PrintHelper is going to print a value");
    }

    private int _value;

    public int Value
    {
        get { return _value; }
        set { _value = value; }
    }

    public void PrintMoney()
```

```
{
    _printHelper.PrintMoney(_value);
}

public void PrintNumber()
{
    _printHelper.PrintNumber(_value);
}
}
```

All the subscribers must provide a handler function, which is going to be called when a publisher raises an event. In the above example, the Number class creates an instance of PrintHelper and subscribes to the beforePrintEvent with the "+=" operator and gives the name of the function which will handle the event (it will be called when publish fires an event). *printHelper\_beforePrintEvent* is the event handler that has the same signature as the BeforePrint delegate in the PrintHelper class.

So now, create an instance of Number class and call print methods:

```
Number myNumber = new Number(100000);
myNumber.PrintMoney();
myNumber.PrintNumber();
```

**Output:**

```
BeforePrintEventHandler: PrintHelper is going to print value
Money: $ 1,00,000.00
BeforePrintEventHandler: PrintHelper is going to print value
Number: 1,00,000
```

## Event Arguments

Events can also pass data as an argument to their subscribed handler. An event passes arguments to the handler as per the delegate signature.

```
public class PrintHelper
{
    public delegate void BeforePrint(string message);
    public event BeforePrint beforePrintEvent;

    public void PrintNumber(int num)
    {
        if (beforePrintEvent != null)
            beforePrintEvent.Invoke("PrintNumber");
        Console.WriteLine("Number: {0,-12:N0}", num);
    }

    public void PrintDecimal(int dec)
    {
        if (beforePrintEvent != null)
            beforePrintEvent.Invoke("PrintDecimal");
        Console.WriteLine("Decimal: {0:G}", dec);
    }
}
```

```
}

public void PrintMoney(int money)
{
    if (beforePrintEvent != null)
        beforePrintEvent("PrintMoney");
    Console.WriteLine("Money: {0:C}", money);
}

public void PrintTemperature(int num)
{
    if (beforePrintEvent != null)
        beforePrintEvent("PrintTemperature");
    Console.WriteLine("Temperature: {0,4:N1} F", num);
}

public void PrintHexadecimal(int dec)
{
    if (beforePrintEvent != null)
        beforePrintEvent("PrintHexadecimal");
    Console.WriteLine("Hexadecimal: {0:X}", dec);
}
}
```

Now, the subscriber class should have an event handler that has a string parameter. Number class has a printHelper\_beforePrintEvent function with string parameter.

```
class Number
{
    private PrintHelper _printHelper;
    public Number(int val)
    {
        _value = val;
        _printHelper = new PrintHelper();
        //subscribe to beforePrintEvent event
        _printHelper.beforePrintEvent += printHelper_beforePrintEvent;
    }
    //beforePrintEvent handler
    void printHelper_beforePrintEvent(string message)
    {
        Console.WriteLine("BeforePrintEvent fires from {0}",message);
    }

    private int _value;

    public int Value
    {
```

```
    get { return _value; }
    set { _value = value; }
}

public void PrintMoney()
{
    _printHelper.PrintMoney(_value);
}

public void PrintNumber()
{
    _printHelper.PrintNumber(_value);
}
}
```

---

## Generics

Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations.

Generics allow for designing a classes and methods whose types are specified only at the time of declaration and instantiation. This enables development of universal classes and methods that help in improving performance, productivity and type-safety.

Example :

```
public class GenericList<T>
{
    public void Add(T input) { }
}

class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);
        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");
        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}

public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

## Generic Delegates

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl {
    class TestDelegate {
        static int num = 10;

        public static int AddNum(int p) {
            num += p;
            return num;
        }
        public static int MultNum(int q) {
            num *= q;
            return num;
        }
        public static int getNum() {
            return num;
        }
        static void Main(string[] args) {
            //create delegate instances
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

            //calling the methods using the delegate objects
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());

            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

## Generic Type Parameters

In a generic type or method definition, a type parameters is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type.

A generic class, such as `GenericList<T>`, cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();  
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();  
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class will be substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition.

## Constraints on type parameters

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of [Object](#), which is the ultimate base class for any .NET type.

If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error. Constraints are specified by using the `where` contextual keyword. The following table lists the seven types of constraints:

Constraint	Description
where T : struct	The type argument must be a value type. Any value type except <a href="#">Nullable&lt;T&gt;</a> can be specified. For more information about nullable types, see <a href="#">Nullable types</a> .
where T : class	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type.
where T : unmanaged	The type argument must not be a reference type and must not contain any reference type members at any level of nesting.

Constraint	Description
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last.
where T :<base class name>	The type argument must be or derive from the specified base class.
where T :<interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

---

## Attribute

C# provides a mechanism for defining declarative tags, called attributes, which you can place on certain entities in your source code to specify additional information. The information that attributes contain can be retrieved at runtime through reflection. You can use predefined attributes, or you can define your own custom attributes.

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute. They can also provide a reusable element that can be applied to a variety of targets.

Syntax for specifying an attribute is as follows –

```
[attribute(positional_parameters, name_parameter = value, ...)]  
element
```



Name of the attribute and its values are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

In C#, attributes are classes that inherit from the Attribute base class. Any class that inherits from Attribute can be used as a sort of "tag" on other pieces of code.

Example :

```
[Obsolete("TestClass1 is obsolete. Use TestClass2 instead.")]
public class TestClass1
{
}

public class TestClass2
{
}
```

The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built* attributes.

### Predefined Attributes

The .Net Framework provides three pre-defined attributes –

- AttributeUsage
- Conditional
- Obsolete

#### AttributeUsage

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows –

```
[AttributeUsage (
    validon,
    AllowMultiple = allowmultiple,
    Inherited = inherited
)]
```

Where,

- The parameter *validon* specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator *AttributeTargets*. The default value is *AttributeTargets.All*.
- The parameter *allowmultiple* (optional) provides value for the *AllowMultiple* property of this attribute, a Boolean value. If this is true, the attribute is multiuse. The default is false (single-use).
- The parameter *inherited* (optional) provides value for the *Inherited* property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false (not inherited).

```
[AttributeUsage(
    AttributeTargets.Class |
    AttributeTargets.Constructor |
    AttributeTargets.Field |
    AttributeTargets.Method |
    AttributeTargets.Property,
    AllowMultiple = true)]
```

Example :

## Conditional

This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

It causes conditional compilation of method calls, depending on the specified value such as **Debug** or **Trace**. For example, it displays the values of the variables while debugging a code.

Syntax for specifying this attribute is as follows –

```
[Conditional(  
    conditionalSymbol  
)]
```

For example,

```
#define DEBUG  
using System;  
using System.Diagnostics;  
  
public class Myclass {  
  
    [Conditional("DEBUG")]  
    public static void Message(string msg)  
    {  
        Console.WriteLine(msg);  
    }  
}  
  
class Test {  
    static void function1() {  
        Myclass.Message("In Function 1.");  
        function2();  
    }  
    static void function2() {  
        Myclass.Message("In Function 2.");  
    }  
    public static void Main() {  
        Myclass.Message("In Main function.");  
        function1();  
        Console.ReadKey();  
    }  
}
```

Outout :

```
In Main function  
In Function 1  
In Function 2
```

## Obsolete

This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element. For example, when a new method is being used in a class and if you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.

Syntax for specifying this attribute is as follows –

```
[Obsolete (message)]  
[Obsolete (message, iserror)]
```

Where,

- The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
- The parameter *iserror*, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false (compiler generates a warning).

Example :

```
public class MyClass {  
  
    [Obsolete("Don't use OldMethod, use NewMethod instead", true)]  
    static void OldMethod() {  
        Console.WriteLine("It is the old method");  
    }  
    static void NewMethod() {  
        Console.WriteLine("It is the new method");  
    }  
    public static void Main() {  
        OldMethod();  
    }  
}
```

## Creating Custom Attributes

To create a custom attributes you need to inherit from the attribute class. Creating and using custom attributes involve four steps –

- Declaring a custom attribute
- Constructing the custom attribute
- Apply the custom attribute on a target program element
- Accessing Attributes Through Reflection

Example : Below is a simple “HelpAttribute” which has a “HelpText” property.

```
class HelpAttribute : Attribute  
{  
    public string HelpText { get; set; }  
}
```

```
}

[Help(HelpText="This is a class")]
class Customer
{
    private string _CustomerCode;

    [Help(HelpText = "This is a property")]
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }

    [Help(HelpText = "This is a method")]
    public void Add()
    {
    }
}
```

"HelpAttribute" is applied to the "Customer" as shown in the code below.

---

## Reflection

**Reflection** objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.

Reflection provides objects (of type [Type](#)) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them.

The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.

Reflection has the following applications –

- It allows view attribute information at runtime.
- It allows examining various types in an assembly and instantiate these types.
- It allows late binding to methods and properties
- It allows creating new types at runtime and then performs some tasks using those types.

Using reflection you can view the attribute information. The **MemberInfo** object of the **System.Reflection** class needs to be initialized for discovering the attributes associated with a class. To do this, you define an object of the target class, as –

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

**Example:**

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute {
    public readonly string Url;

    public string Topic // Topic is a named parameter {
        get {
            return topic;
        }
        set {
            topic = value;
        }
    }
    public HelpAttribute(string url) // url is a positional parameter {
        this.Url = url;
    }
    private string topic;
}

[HelpAttribute("Information on the class MyClass")]
class MyClass {

}

namespace AttributeAppl {
    class Program {
        static void Main(string[] args) {
            System.Reflection.MemberInfo info = typeof(MyClass);
            object[] attributes = info.GetCustomAttributes(true);

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

```
        System.Console.WriteLine(attributes[i]);  
    }  
    Console.ReadKey();  
}  
}  
}
```

When it is compiled and run, it displays the name of the custom attributes attached to the class *MyClass* –

HelpAttribute

## 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.

The **System.Collections** namespace includes following non-generic collections.

Non-generic Collections	Usage
<b>ArrayList</b>	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.
<b>SortedList</b>	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

Non-generic Collections	Usage
	off (0).

## ArrayList

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.

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

### Properties and Methods of ArrayList

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.
Method	Description
<u>Add()/AddRange()</u>	Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList.
<u>Insert()/InsertRange()</u>	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.
<u>Remove()/RemoveRange()</u>	Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList.
<u>RemoveAt()</u>	Removes the element at the specified index from the ArrayList.
<u>Sort()</u>	Sorts entire elements of the ArrayList.
<u>Reverse()</u>	Reverses the order of the elements in the entire ArrayList.
<u>Contains</u>	Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false.
Clear	Removes all the elements in ArrayList.
CopyTo	Copies all the elements or range of elements to compitible Array.

Method	Description
GetRange	Returns specified number of elements from specified index from ArrayList.
IndexOf	Search specified element and returns zero based index if found. Returns -1 if element not found.
ToArray	Returns compatible array from an ArrayList.

## SortedList

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.

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

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

## 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.

#### Properties and Methods of Stack:

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

### 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.

#### Properties and Methods of Queue:

Property	Usage
Count	Returns the total count of elements in the Queue.
Method	Usage
Enqueue	Adds an item into the queue.
Dequeue	Removes and returns an item from the beginning of the queue.
Peek	Returns an first item from the queue
Contains	Checks whether an item is in the queue or not
Clear	Removes all the items from the queue.
TrimToSize	Sets the capacity of the queue to the actual number of items in the queue.

### 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.

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

## Multithreading

Multithreading or free-threading is the ability of an operating system to concurrently run programs that have been divided into subcomponents, or threads.

A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control.

### Features and Benefits of Threads

Mutually exclusive tasks, such as gathering user input and background processing can be managed with the use of threads. Threads can also be used as a convenient way to structure a program that performs several similar or identical tasks concurrently.

One of the advantages of using the threads is that you can have multiple activities happening simultaneously. Another advantage is that a developer can make use of threads to achieve faster computations by doing two different computations in two threads instead of serially one after the other.

### Threading Concepts in C#

An AppDomain is a runtime representation of a logical process within a physical process. And a thread is the basic unit to which the OS allocates processor time. To start with, each AppDomain is started with a

single thread. But it is capable of creating other threads from the single thread and from any created thread as well.

The life cycle of a thread starts when an object of the `System.Threading.Thread` class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread –

- **The Unstarted State** – It is the situation when the instance of the thread is created but the `Start` method is not called.
- **The Ready State** – It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State** – A thread is not executable, when
  - `Sleep` method has been called
  - `Wait` method has been called
  - Blocked by I/O operations
- **The Dead State** – It is the situation when the thread completes execution or is aborted.

### Properties and Methods of the Thread Class

The following table shows some most commonly used **properties** of the **Thread** class –

Sr.No.	Property & Description
1	<b>CurrentContext</b> Gets the current context in which the thread is executing.
2	<b>CurrentCulture</b> Gets or sets the culture for the current thread.
3	<b>CurrentPrincipal</b> Gets or sets the thread's current principal (for role-based security).
4	<b>CurrentThread</b> Gets the currently running thread.
5	<b>CurrentUICulture</b> Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
6	<b>ExecutionContext</b> Gets an <code>ExecutionContext</code> object that contains information about the various contexts of the current thread.
7	<b>IsAlive</b> Gets a value indicating the execution status of the current thread.
8	<b>IsBackground</b>

	Gets or sets a value indicating whether or not a thread is a background thread.
9	<b>IsThreadPoolThread</b> Gets a value indicating whether or not a thread belongs to the managed thread pool.
10	<b>ManagedThreadId</b> Gets a unique identifier for the current managed thread.
11	<b>Name</b> Gets or sets the name of the thread.
12	<b>Priority</b> Gets or sets a value indicating the scheduling priority of a thread.
13	<b>ThreadState</b> Gets a value containing the states of the current thread.

The following table shows some of the most commonly used **methods** of the **Thread** class –

Sr.No.	Method & Description
1	<b>public void Abort()</b> Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2	<b>public static LocalDataStoreSlot AllocateDataSlot()</b> Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	<b>public static LocalDataStoreSlot AllocateNamedDataSlot(string name)</b> Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	<b>public static void BeginCriticalRegion()</b> Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.

5	<b>public static void BeginThreadAffinity()</b> Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	<b>public static void EndCriticalRegion()</b> Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	<b>public static void EndThreadAffinity()</b> Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	<b>public static void FreeNamedDataSlot(string name)</b> Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	<b>public static Object GetData(LocalDataStoreSlot slot)</b> Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	<b>public static AppDomain GetDomain()</b> Returns the current domain in which the current thread is running.
11	<b>public static AppDomain GetDomainID()</b> Returns a unique application domain identifier
12	<b>public static LocalDataStoreSlot GetNamedDataSlot(string name)</b> Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
13	<b>public void Interrupt()</b> Interrupts a thread that is in the WaitSleepJoin thread state.
14	<b>public void Join()</b> Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms.
15	<b>public static void MemoryBarrier()</b>

	Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.
16	<b>public static void ResetAbort()</b> Cancels an Abort requested for the current thread.
17	<b>public static void SetData(LocalDataStoreSlot slot, Object data)</b> Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead.
18	<b>public void Start()</b> Starts a thread.
19	<b>public static void Sleep(int millisecondsTimeout)</b> Makes the thread pause for a period of time.
20	<b>public static void SpinWait(int iterations)</b> Causes a thread to wait the number of times defined by the iterations parameter
21	<b>public static byte VolatileRead(ref byte address)</b> <b>public static double VolatileRead(ref double address)</b> <b>public static int VolatileRead(ref int address)</b> <b>public static Object VolatileRead(ref Object address)</b> Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above.
22	<b>public static void VolatileWrite(ref byte address,byte value)</b> <b>public static void VolatileWrite(ref double address, double value)</b> <b>public static void VolatileWrite(ref int address, int value)</b> <b>public static void VolatileWrite(ref Object address, Object value)</b> Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above.
23	<b>public static bool Yield()</b> Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

## Defining and Calling threads

```
using System.Threading;
public class ServerClass
{
    // The method that will be called when the thread is started.
    public void Instance Method()
    {
        Console.WriteLine("You are in InstanceMethod.Running on Thread A");
        Console.WriteLine("Thread A Going to Sleep Zzzzzzz");

        // Pause for a moment to provide a delay
        Thread.Sleep(3000);
        Console.WriteLine ("You are Back in InstanceMethod.Running on Thread A");
    }
    public static void StaticMethod()
    {
        Console.WriteLine("You are in StaticMethod. Running on Thread B.");
        // Pause for a moment to provide a delay to make threads more apparent.
        Console.WriteLine("Thread B Going to Sleep Zzzzzzz");

        Thread.Sleep(5000);
        Console.WriteLine("back in static method. Running on Thread B");
    }
}
public class Simple
{
    public static int Main(String[] args)
    {
        Console.WriteLine ("Thread Simple Sample");
        ServerClass serverObject = new ServerClass();
        // Create the thread object, passing in the
        // serverObject.InstanceMethod method using a ThreadStart delegate.
        Thread InstanceCaller = new Thread(new ThreadStart(serverObject.InstanceMethod));

        // Start the thread.
        InstanceCaller.Start();
        Console.WriteLine("The Main() thread calls this " +
            "after starting the new InstanceCaller thread.");

        // Create the thread object, passing in the
        // serverObject.StaticMethod method using a ThreadStart delegate.
        Thread StaticCaller = new Thread(new
            ThreadStart(ServerClass.StaticMethod));
```

```
// Start the thread.  
StaticCaller.Start();  
Console.WriteLine("The Main () thread calls this " +  
    "after starting the new StaticCaller threads.");  
return 0;  
}  
}
```

If the code in this example is compiled and executed, you would notice how processor time is allocated between the two method calls. If not for threading, you would have to wait till the first method slept for 3000 secs for the next method to be called. Try disabling threading in the above code and notice how they work. Nevertheless, execution time for both would be the same.

## File Handling

In the .NET framework, the System.IO namespace is the region of the base class libraries devoted to file based input and output services. Like any namespace, the System.IO namespace defines a set of classes, interfaces, enumerations, structures and delegates. The following table outlines the core members of this namespace:

Class Types	Description
Directory/ DirectoryInfo	These classes support the manipulation of the system directory structure.
DriveInfo	This class provides detailed information regarding the drives that a given machine has.
FileStream	This gets you random file access with data represented as a stream of bytes.
File/FileInfo	These sets of classes manipulate a computer's files.
Path	It performs operations on System.String types that contain file or directory path information in a platform-neutral manner.
BinaryReader/ BinaryWriter	These classes allow you to store and retrieve primitive data types as binary values.
StreamReader/StreamWriter	Used to store textual information to a file.
StringReader/StringWriter	These classes also work with textual information. However, the underlying storage is a string buffer rather than a physical file.
BufferedStream	This class provides temp storage for a stream of bytes that you can commit to storage at a later time.

### Working with Directories

The .NET framework provides the two rudimentary classes, DirectoryInfo and Directory, to do directory-related operations such as creation and deletion.



### DirectoryInfo Class

The DirectoryInfo class contains a set of members for the creation, deletion, moving and enumeration over directories and subdirectories. Here, in the following code sample, display the information related to temp directory.

```
1. DirectoryInfo di=new DirectoryInfo(@"D:\temp");
2. Console.WriteLine("*****Direcotry Informations*****\n\n");
3. Console.WriteLine("Full Name={0}",di.FullName);
4. Console.WriteLine("Root={0}",di.Root);
5. Console.WriteLine("Attributes={0}", di.Attributes);
6. Console.WriteLine("Creation Time={0}", di.CreationTime);
7. Console.WriteLine("Name={0}", di.Name);
8. Console.WriteLine("Parent={0}", di.Parent);
```

The previous code produces the information related to the temp directory located in the D drive as in the following:



Typically, we make the assumption that the path passed in the constructor of the DirectoryInfo class physically exists. However, if you attempt to interact with a nonexistent directory then the CLR will throw an exception. So, we need to create a directory first to handle the exceptions that occur as in the following.

```
9. DirectoryInfo di=new DirectoryInfo(@"D:\temp\xyz");
10. di.Create();
```

We can also programmatically extends a directory structure using the CreateSubdirectory() method. The following code sample first creates a sub directory in D drive then in D:\ajay\ as in the following:

```
11. DirectoryInfo di=new DirectoryInfo(@"D:\");
12. di.CreateSubdirectory("ajay");
13. di.CreateSubdirectory(@"ajay\ajay11");
```

### Directory Class

The Directory class provides nearly the same functionality as DirecotryInfo. The Directory class typically

returns string data rather than strongly typed DirectoryInfo objects. The following sample deletes the directory and subdirectory in the D drive.

```
14. static void Main(string[] args)
15. {
16.     DirectoryInfo di = new DirectoryInfo(@"d:\abc");
17.     Console.WriteLine("Name:{0}",di.FullName);
18.
19.     Console.Write("Are you sure to Delete:");
20.     string str=Console.ReadLine();
21.     if (str == "y")
22.     {
23.         Directory.Delete(@"d:\abc", true);
24.     }
25.     Console.Write("Deleted.....");
26. }
```

### Reading and Writing to Files

Reading and writing operations are done using a File object. The following code snippet reads a text file located in the machine somewhere.

```
1. private void button1_Click(object sender, EventArgs e)
2. {
3.     try
4.     {
5.         textBox2.Text = File.ReadAllText(txtPath.Text);
6.     }
7.     catch (FileNotFoundException)
8.     {
9.         MessageBox.Show("File not Found....");
10.    }
11. }
```

Here, first the user interface asks the user to enter the path of the file that he wanted to display. Later that path is passed to the File method ReadAllText() method that reads all the text integrated in the file and displays it over the text box.

Besides reading a file, we can write some contents over an existing text file by the File class WriteAllText() method as in the following:

```
1. File.WriteAllText(@"d:\test.txt", textBox2.Text);
```

It takes a path to save the file and content input method medium such as a text box or any other control.

## Stream

The .NET provides many objects such as FileStream, StreamReader/Writer, BinaryReader/Writer to read from and write data to a file. A stream basically represents a chunk of data flowing between a source and a destination. Stream provides a common way to interact with a sequence of bytes regardless of what kind of devices store or display the bytes. The following table provides common stream member functions:

Methods	Description
Read()/ ReadByte()	Read a sequence of bytes from the current stream.
Write()/WriteByte()	Write a sequence of bytes to the current stream.
Seek()	Sets the position in the current stream.
Position()	Determine the current position in the current stream.
Length()	Return the length of the stream in bytes.
Flush()	Updates the underlying data source with the current state of the buffer and then clears the buffer.
Close()	Closes the current stream and releases any associated stream resources.

## FileStream

A FileStream instance is used to read or write data to or from a file. In order to construct a FileStream, first we need a file that we want to access. Second, the mode that indicates how we want to open the file. Third, the access that indicates how we want to access a file. And finally, the share access that specifies whether you want exclusive access to the file.

Enumeration	Values
<b>FileMode</b>	Create, Append, Open, CreateNew, Truncate, OpenOrCreate
<b>FileAccess</b>	Read, Write, ReadWrite
<b>FileShare</b>	Inheritable, Read, None, Write, ReadWrite

The FileStream can read or write only a single byte or an array of bytes. You will be required to encode the System.String type into a corresponding byte array. The System.Text namespace defines a type named encoding that provides members that encode and decode strings to an array of bytes. Once encoded, the byte array is persisted to a file with the FileStream.Write() method. To read the bytes back into memory, you must reset the internal position of the stream and call the ReadByte() method. Finally, you display the raw byte array and the decoded string to the console.

1. <code>using(FileStream fs=new FileStream(@"d:\ajay123.doc",FileMode.Create))</code>
--

```

2. {
3.     string msg = "first program";
4.     byte[] byteArray = Encoding.Default.GetBytes(msg);
5.     fs.Write(byteArray, 0, byteArray.Length);
6.     fs.Position = 0;
7.
8.     byte[] rFile = new byte[byteArray.Length];
9.
10.    for (int i = 0; i < byteArray.Length; i++)
11.    {
12.        rFile[i] = (byte)fs.ReadByte();
13.        Console.WriteLine(rFile[i]);
14.    }
15.
16.    Console.WriteLine(Encoding.Default.GetString(rFile));
17. }

```

### BinaryReader and BinaryWriter

The BinaryReader and Writer class allows you to read and write discrete data types to an underlying stream in a compact binary format. The BinaryWriter class defines a highly overloaded Write method to place a data type in the underlying stream.

Members	Description	Class
Write	Write the value to current stream	BinaryWriter
Seek	Set the position in the current stream	BinaryWriter
Close	Close the binary reader	BinaryWriter
Flush	Flush the binary stream	BinaryWriter
PeekChar	Return the next available character without advancing the position in the stream	BinaryReader
Read	Read a specified set of bytes or characters and store them in the incoming array.	BinaryReader

The following sample first writes a number of data contents to a new champu.dat file using BinaryWriter. Later, to read the data, the BinaryReader class employs a number of methods.

```

1. class Program
2. {
3.     static void Main(string[] args)

```

```
4.  {
5.      // writing
6.      FileInfo fi = new FileInfo("champu.dat");
7.      using (BinaryWriter bw = new BinaryWriter(fi.OpenWrite()))
8.      {
9.          int x = 007;
10.         string str = "hello champu ,one day you will become doomkatu";
11.
12.         bw.Write(x);
13.         bw.Write(str);
14.     }
15.
16.     //Reading
17.     FileInfo f = new FileInfo("champu.dat");
18.     using (BinaryReader br = new BinaryReader(fi.OpenRead()))
19.     {
20.         Console.WriteLine(br.ReadInt32());
21.         Console.WriteLine(br.ReadString());
22.     }
23.     Console.ReadLine();
24.
25. }
26. }
```

### StringReader and StringWriter

We can use StringWriter and StringReader to treat textual information as a stream of in-memory characters. This can prove helpful when you wish to append character-based information to an underlying buffer. The following code sample illustrates this by writing a block of string data to a StringWriter object, rather than to a file on the local hard drive:

```
1.  static void Main(string[] args)
2.  {
3.      // writing
4.      using (StringWriter sw = new StringWriter())
5.      {
6.          sw.WriteLine("hellooooooooooooooooooooo");
7.
8.          // Reading
9.          using (StringReader sr = new StringReader(sw.ToString()))
10.         {
11.             string input = null;
12.             while((input = sr.ReadLine())!=null)
13.             {
```

```
14.         Console.WriteLine(input);  
15.     }  
16. }  
17. }  
18. }
```





# ADO.Net

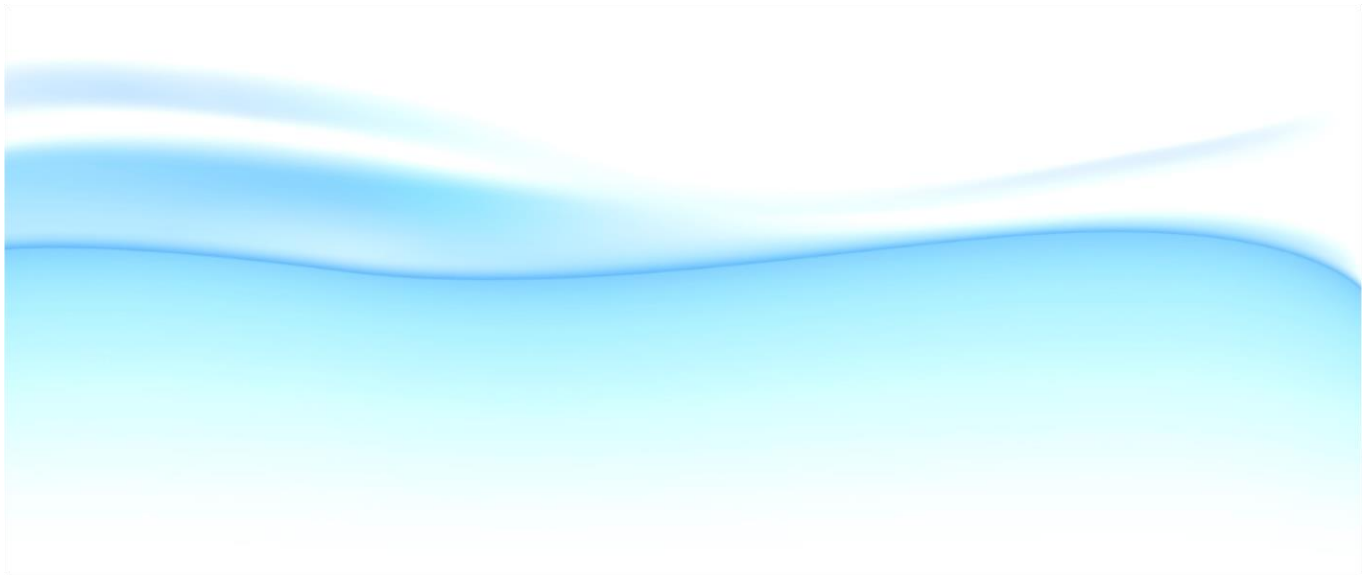
---

## C# Programming

---

[www.enosislearning.com](http://www.enosislearning.com)

[courses@enosislearning.com](mailto:courses@enosislearning.com)



---

### Introduction to ADO.Net

---

#### What is ADO.NET?

**ADO.NET is not a different technology.** In simple terms, you can think of ADO.NET, as a set of classes (Framework), that can be used to interact with data sources like Databases and XML files. This data can, then be consumed in any .NET application. ADO stands for Microsoft ActiveX Data Objects.

ADO.NET provides a bridge between the front end controls and the back end database. The ADO.NET

objects encapsulate all the data access operations and the controls interact with these objects to display data, thus hiding the details of movement of data.

### What are .NET Data Providers?

Databases only understand SQL. If a .NET application (Web, Windows, Console etc..) has to retrieve data, then the application needs to

1. Connect to the Database
2. Prepare an SQL Command
3. Execute the Command
4. Retrieve the results and display in the application

**Sample ADO.NET code to connect to SQL Server Database and retrieve data.** Notice that we are using **SqlConnection**, **SqlCommand** and **SqlDataReader** classes . All the objects are prefixed with the word **SQL**. All these classes are present in **System.Data.SqlClient** namespace. So, we can say that the **.NET data provider for SQL Server is System.Data.SqlClient.**

```
SqlConnection con = new SqlConnection("data source=.; database=Sample; integrated security=SSPI");  
SqlCommand cmd = new SqlCommand("Select * from tblProduct", con);  
con.Open();  
SqlDataReader rdr = cmd.ExecuteReader();  
GridView1.DataSource = rdr;  
GridView1.DataBind();  
con.Close();
```

**If we want to connect to OLEDB datasources like Excel, Access etc, we can** use **OleDbConnection**, **OleDbCommand** and **OleDbDataReader** classes. So, .NET data provider for OLEDB is System.Data.OleDb.

### Different .NET Data Providers

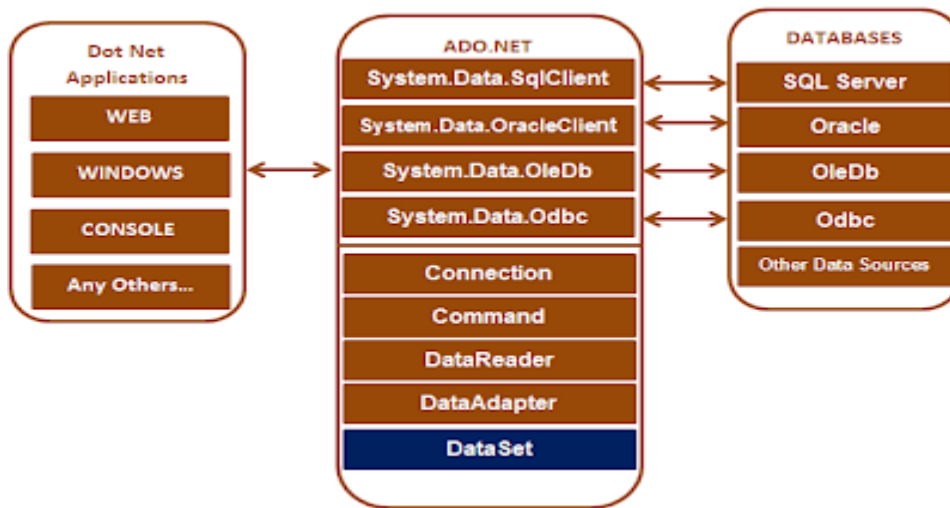
Data Provider for SQL Server - System.Data.SqlClient

Data Provider for Oracle - System.Data.OracleClient

Data Provider for OLEDB - System.Data.OleDb



### Data Provider for ODBC - System.Data.Odbc



**Please note that, depending on the provider,** the following ADO.NET objects have a different prefix

1. Connection - SqlConnection, OracleConnection, OleDbConnection, OdbcConnection etc
2. Command - SqlCommand, OracleCommand, OleDbCommand, OdbcCommand etc
3. DataReader - SqlDataReader, OracleDataReader, OleDbDataReader, OdbcDataReader etc
4. DataAdapter - SqlDataAdapter, OracleDataAdapter, OleDbDataAdapter, OdbcDataAdapter etc

**The DataSet object is not provider specific.** Once we connect to a Database, execute command, and retrieve data into .NET application. The data can then be stored in a DataSet and work independently of the database.

## SqlConnection in ADO.NET

**The first thing that we will have to do,** when working with databases is to create a connection object. There are 2 ways to create an instance of **SqlConnection** class as shown below.

**Create an instance of SqlConnection class using the constructor that takes connectionString parameter**

```
SqlConnection connection = new SqlConnection("data source=.; database=SampleDB; integrated security=SSPI");
```

```
//First create an instance of SqlConnection class using the parameter-less constructor
```

```
SqlConnection connection = new SqlConnection();
```

```
//Then set the ConnectionString property of the connection object
```

```
connection.ConnectionString = "data source=.; database=SampleDB; integrated security=SSPI";
```

**The ConnectionString parameter** is a string made up of Key/Value pairs that has the information required to create a connection object.

**To create a connection object with windows authentication**

```
string ConnectionString = "data source=.; database=SampleDB; integrated security=SSPI";
```

**To create a connection object with SQL Server authentication**

```
string ConnectionString = "data source=.; database=SampleDB; user id=MyUserName;  
password=MyPassword";
```

The **"data source"** is the **name or IP Address** of the **SQL Server** that we want to connect to. If you are working with a local instance of sql server, you can just specify DOT(.). If the server is on a network, then use Name or IP address.

**Sample ADO.NET code that**

1. Creates a connection
2. The created connection object is then passed to the command object, so that the command object knows on which sql server connection to execute this command.
3. Execute the command, and set the command results, as the data source for the gridview control.
4. Call the DataBind() method
5. Close the connection in the finally block. Connections are limited and are very valuable. Connections must be closed properly, for better performance and scalability.

**Note:** Connections should be opened as late as possible, and should be closed as early as possible.

```
protected void Page_Load(object sender, EventArgs e)
{
    //Create the connection object
    SqlConnection connection = new SqlConnection("data source=.;  
database=Sample_Test_DB; integrated security=SSPI");
    try
    {
        // Pass the connection to the command object, so the command object knows on  
which
        // connection to execute the command
        SqlCommand cmd = new SqlCommand("Select * from tblProductInventory",  
connection);
        // Open the connection. Otherwise you get a runtime error. An open connection is  
// required to execute the command
        connection.Open();
        GridView1.DataSource = cmd.ExecuteReader();
        GridView1.DataBind();
    }
    catch (Exception ex)
    {
        // Handle Exceptions, if any
    }
}
```

```
finally
{
    // The finally block is guaranteed to execute even if there is an exception.
    // This ensures connections are always properly closed.
    connection.Close();
}
```

We can also use **"using"** statement to properly close the connection as shown below. We don't have to explicitly call **Close()** method, when **using** is used. The connection will be automatically closed for us.

```
protected void Page_Load(object sender, EventArgs e)
{
    using (SqlConnection connection = new SqlConnection("data source=.;
    database=Sample_Test_DB; integrated security=SSPI"))
    {
        SqlCommand cmd = new SqlCommand("Select * from tblProductInventory",
        connection);
        connection.Open();
        GridView1.DataSource = cmd.ExecuteReader();
        GridView1.DataBind();
    }
}
```

**Common Interview Question: What are the 2 uses of an using statement in C#?**

1. To import a namespace. Example: using System;
2. To close connections properly as shown in the example above

### [Connection Strings in web.config configuration file](#)

**There are 2 issues with hard coding the connection strings in application code**

1. For some reason, if we want to point our application to a different database server, we will have to change the application code. If you change application code, the application requires a re-build and a re-deployment which is a time waster.
2. All the pages that has the connection string hard coded needs to change. This adds to the maintenance overhead and is also error prone.

**In real time, we may point our applications from time to time, from Development database to testing database to UAT database.**

**Because of these issues, the best practice is to store the connection in the configuration file**, from which all the pages can read and use it. This way we have only one place to change, and we don't have to re-build and re-deploy our application. This saves a lot of time.

**In an asp.net web application, the configuration strings can be stored in web.config file**, as shown below. Give a meaningful name to your connection string. Since we are working with sql server, the provider name is **System.Data.SqlClient**.

```
<connectionStrings>
  <add name="DatabaseConnectionString"
        connectionString="data source=.; database=Sample_Test_DB; Integrated Security=SSPI"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

#### How to read the connection string from web.config file?

Use the **ConnectionStrings** property of the **ConfigurationManager** class to retrieve the connection string value from **web.config**. **ConfigurationManager** class is present in **System.Configuration** namespace.

```
protected void Page_Load(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection(ConnectionString))
    {
        SqlCommand cmd = new SqlCommand("Select * from tblProductInventory", connection);
        connection.Open();
        GridView1.DataSource = cmd.ExecuteReader();
        GridView1.DataBind();
    }
}
```

The configuration file in a windows application is **App.config**. Storing connection strings in **App.config** is similar to **web.config**. The same **ConfigurationManager** class can be used to read connection string from **App.config** file. The example below, shows how to read connection strings from **App.config** file, and bind the data to a **DataGridview** control in a windows application.

```
private void Form1_Load(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection(ConnectionString))
    {
        SqlCommand cmd = new SqlCommand("Select * from tblProductInventory", connection);
        connection.Open();
        BindingSource source = new BindingSource();
        source.DataSource = cmd.ExecuteReader();
        dataGridView1.DataSource = source;
        dataGridView1.DataBind();
    }
}
```

## SqlCommand in ado.net

SqlCommand class is used to prepare an SQL statement or StoredProcedure that we want to execute on a SQL Server database. In this session, we will discuss about executing Transact-SQL statements on a SQL Server.

**The following are the most commonly used methods of the SqlCommand class.**

**ExecuteReader** - Use when the T-SQL statement returns more than a single value. For example, if the query returns rows of data.

**ExecuteNonQuery** - Use when you want to perform an Insert, Update or Delete operation

**ExecuteScalar** - Use when the query returns a single(scalar) value. For example, queries that return the total number of rows in a table.

**The sample code below**, executes a T-SQL statement, that returns multiple rows of data using **ExecuteReader()** method.

```
protected void Page_Load(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection("data source=.;
    database=Sample_Test_DB; integrated security=SSPI"))
    {
        //Create an instance of SqlCommand class, specifying the T-SQL command that
        //we want to execute, and the connection object.
        SqlCommand cmd = new SqlCommand("Select Id,ProductName,QuantityAvailable from
        tblProductInventory", connection);
        connection.Open();
        //As the T-SQL statement that we want to execute return multiple rows of data,
        //use ExecuteReader() method of the command object.
        GridView1.DataSource = cmd.ExecuteReader();
        GridView1.DataBind();
    }
}
```

**In the example below, we are using ExecuteScalar() method**, as the T-SQL statement returns a single value.

```
protected void Page_Load(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection("data source=.;
    database=Sample; integrated security=SSPI"))
```

```
{
    //Create an instance of SqlCommand class, specifying the T-SQL command
    //that we want to execute, and the connection object.
    SqlCommand cmd = new SqlCommand("Select Count(Id) from tblProductInventory",
connection);
    connection.Open();
    //As the T-SQL statement that we want to execute return a single value,
    //use ExecuteScalar() method of the command object.
    //Since the return type of ExecuteScalar() is object, we are type casting to int datatype
    int TotalRows = (int)cmd.ExecuteScalar();
    Response.Write("Total Rows = " + TotalRows.ToString());
}
}
```

**The following example performs an Insert, Update and Delete operations** on a SQL server database using the ExecuteNonQuery() method of the SqlCommand object.

```
protected void Page_Load(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection("data source=.; database=Sample_Test_DB;
integrated security=SSPI"))
    {
        //Create an instance of SqlCommand class, specifying the T-SQL command
        //that we want to execute, and the connection object.
        SqlCommand cmd = new SqlCommand("insert into tblProductInventory values (103, 'Apple Laptops',
100)", connection);
        connection.Open();
        //Since we are performing an insert operation, use ExecuteNonQuery()
        //method of the command object. ExecuteNonQuery() method returns an
        //integer, which specifies the number of rows inserted
        int rowsAffected = cmd.ExecuteNonQuery();
        Response.Write("Inserted Rows = " + rowsAffected.ToString() + "<br/>");

        //Set to CommandText to the update query. We are reusing the command object,
        //instead of creating a new command object
        cmd.CommandText = "update tblProductInventory set QuantityAvailable = 101 where Id = 101";
        //use ExecuteNonQuery() method to execute the update statement on the database
        rowsAffected = cmd.ExecuteNonQuery();
        Response.Write("Updated Rows = " + rowsAffected.ToString() + "<br/>");

        //Set to CommandText to the delete query. We are reusing the command object,
```

```
//instead of creating a new command object
cmd.CommandText = "Delete from tblProductInventory where Id = 102";
//use ExecuteNonQuery() method to delete the row from the database
rowsAffected = cmd.ExecuteNonQuery();
Response.Write("Deleted Rows = " + rowsAffected.ToString() + "<br/>");
}
}
```

## Sql injection

In this example, we are building the query dynamically by concatenating the strings that the user has typed into the textbox. This is extremely dangerous, as it is vulnerable to SQL injection attacks.

```
protected void GetProductsButton_Click(object sender, EventArgs e)
{
    string ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;
    using (SqlConnection connection = new SqlConnection("DatabaseConnectionString"))
    {
        //Build the query dynamically, by concatenating the text, that the user has
        //typed into the ProductNameTextBox. This is a bad way of constructing
        //queries. This line of code will open doors for sql injection attack
        SqlCommand cmd = new SqlCommand("Select * from tblProductInventory where ProductName like '" +
        ProductNameTextBox.Text + "%'", connection);
        connection.Open();
        ProductsGridView.DataSource = cmd.ExecuteReader();
        ProductsGridView.DataBind();
    }
}
```

To give you a flavour of that, just imagine what could happen if the user types the following into the TextBox, and clicks Get Products button.

**i'; Delete from tblProductInventory --**

**Now execute the following select query on the database**

**Select \* from tblProductInventory**

**The entire data from tblProductInventory table is deleted.** This is called **SQL injection** attack. I have seen a lot of new developers building queries dynamically by concatenating the strings, that end users enter into user interface controls like textboxes. Just imagine the extent of damage that can happen as a result of sql injection.

**However, sql injection can be easily avoided, by using parameterized queries or stored procedures.**

### Sql Injection prevention using “Parametrized Queries”

The above code can be easily re-written using parameterized queries to prevent sql injection attack. The re-written code is shown below. Notice, that the query now uses parameter - **@ProductName**. The value for this parameter is then provided using the **AddWithValue()** method. The parameter is associated with the command object using **Parameters** collection property of the **command** object.

```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    // Parameterized query. @ProductName is the parameter
    string Command = "Select * from tblProductInventory where ProductName like @ProductName" ;
    SqlCommand cmd = new SqlCommand(Command, con);
    // Provide the value for the parameter
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");
    con.Open();
    GridView1.DataSource = cmd.ExecuteReader();
    GridView1.DataBind();
}
```

### Sql Injection prevention using “Stored Procedure”

Sql injection can also be prevented using stored procedures. So, first let's write a stored procedure, that returns the list of products. This stored procedure takes an input parameter **@ProductName**.

```
Create Procedure spGetProductsByName
@ProductName nvarchar(50)
as
Begin
    Select * from tblProductInventory
    where ProductName like @ProductName + '%'
End
```

Now, let's re-write the code, to use stored procedure **spGetProductsByName**.

```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
```



```
// The command, that we want to execute is a stored procedure,  
// so specify the name of the procedure as cmdText  
SqlCommand cmd = new SqlCommand("spGetProductsByName", con);  
// Specify that the T-SQL command is a stored procedure  
cmd.CommandType = System.Data.CommandType.StoredProcedure;  
// Associate the parameter and it's value with the command object  
cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");  
con.Open();  
GridView1.DataSource = cmd.ExecuteReader();  
GridView1.DataBind();  
}
```

### Calling a stored procedure with output parameters

Example :

```
Create Procedure spAddEmployee  
@Name nvarchar(50),  
@Gender nvarchar(20),  
@Salary int,  
@EmployeeId int Out  
as  
Begin  
Insert into tblEmployees values(@Name, @Gender, @Salary)  
Select @EmployeeId = SCOPE_IDENTITY()  
End
```

The design of the webform, should be as shown below.

Employee Name	<input type="text"/>
Gender	<input type="text" value="Male"/> ▼
Salary	<input type="text"/>
<input type="button" value="Submit"/>	

```
protected void btnSubmit_Click(object sender, EventArgs e)  
{  
    //Read the connection string from Web.Config file  
    string ConnectionString = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
    using (SqlConnection con = new SqlConnection(ConnectionString))  
    {  
        //Create the SqlCommand object  
        SqlCommand cmd = new SqlCommand("spAddEmployee", con);
```

```
//Specify that the SqlCommand is a stored procedure
cmd.CommandType = System.Data.CommandType.StoredProcedure;

//Add the input parameters to the command object
cmd.Parameters.AddWithValue("@Name", txtEmployeeName.Text);
cmd.Parameters.AddWithValue("@Gender", ddlGender.SelectedValue);
cmd.Parameters.AddWithValue("@Salary", txtSalary.Text);

//Add the output parameter to the command object
SqlParameter outPutParameter = new SqlParameter();
outPutParameter.ParameterName = "@EmployeeId";
outPutParameter.SqlDbType = System.Data.SqlDbType.Int;
outPutParameter.Direction = System.Data.ParameterDirection.Output;
cmd.Parameters.Add(outPutParameter);

//Open the connection and execute the query
con.Open();
cmd.ExecuteNonQuery();

//Retrieve the value of the output parameter
string EmployeeId = outPutParameter.Value.ToString();
lblMessage.Text = "Employee Id = " + EmployeeId;
}
}
```

**Note:** Please make sure to add the following **using** declarations at the top of the code behind page.

```
using System.Data.SqlClient;
using System.Configuration;
```

**Now, run the application. Fill in the employee details and click Submit. The Employee row gets added to the database, and the generated EmployeeId is shown on the screen.**

## SqlDataReader object in ADO.NET

**SqlDataReader** reads data in the most efficient manner possible. SqlDataReader is read-only and forward only, meaning once you read a record and go to the next record, there is no way to go back to the previous record. It is also not possible to change the data using SqlDataReader. SqlDataReader is connection oriented, meaning it requires an active connection to the data source, while reading data. The forward-only nature of SqlDataReader is what makes it an efficient choice to read data.

**You cannot create an instance of SqlDataReader using the new operator** as shown below. If you try to

new up SqlDataReader, you will get a compilation error stating - **The type 'System.Data.SqlClient.SqlDataReader' has no constructors defined.**  
`SqlDataReader rd = new SqlDataReader();`

**The SqlCommand object's ExecuteReader() method creates and returns an instance of SqlDataReader.**

```
SqlCommand command = new SqlCommand("Select * from tblProductInventory", connection);  
SqlDataReader reader = command.ExecuteReader();
```

**Another important point to keep in mind is that, SqlDataReader is connection oriented** and the connection needs to be opened explicitly, by calling the Open() method on the connection object, before calling the ExecuteReader() method of the command object.

**The simplest way to bind a SqlDataReader object to a GridView** (Data-bound control), is to assign it to the DataSource property of the GridView control and then call the DataBind() method as shown below. Notice that, just like the SqlConnection object, SqlDataReader is wrapped in an using block. This will ensure that the SqlDataReader is closed in a timely fashion, and that we don't run out of available connections to the database.

```
string ConnectionString  
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
using (SqlConnection connection = new SqlConnection(ConnectionString))  
{  
    connection.Open();  
    SqlCommand command = new SqlCommand("Select * from tblProductInventory", connection);  
    using (SqlDataReader reader = command.ExecuteReader())  
    {  
        ProductsGridView.DataSource = reader;  
        ProductsGridView.DataBind();  
    }  
}
```

### **Read() Method of SqlDataReader**

**If for some reason, you want to loop thru each row in the SqlDataReader object**, then use the Read() method, which returns **true** as long as there are rows to read. If there are no more rows to read, then this method will return false. In the following example, we loop thru each row in the **SqlDataReader** and then compute the **10% discounted price**.

```
string ConnectionString  
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
using (SqlConnection connection = new SqlConnection(ConnectionString))
```

```
{
    connection.Open();
    SqlCommand command = new SqlCommand("Select * from tblProductInventory",connection);
    using (SqlDataReader reader = command.ExecuteReader())
    {
        // Create the DataTable and columns. This will
        // be used as the datasource for the GridView
        DataTable sourceTable = new DataTable();
        sourceTable.Columns.Add("ID");
        sourceTable.Columns.Add("Name");
        sourceTable.Columns.Add("Price");
        sourceTable.Columns.Add("DiscountedPrice");

        while (reader.Read())
        {
            //Calculate the 10% discounted price
            int OriginalPrice = Convert.ToInt32(reader["UnitPrice"]);
            double DiscountedPrice = OriginalPrice * 0.9;

            // Populate datatable column values from the SqlDataReader
            DataRow datarow = sourceTable.NewRow();
            datarow["ID"] = reader["ProductId"];
            datarow["Name"] = reader["ProductName"];
            datarow["Price"] = OriginalPrice;
            datarow["DiscountedPrice"] = DiscountedPrice;

            //Add the DataRow to the DataTable
            sourceTable.Rows.Add(datarow);
        }

        // Set sourceTable as the DataSource for the GridView
        ProductsGridView.DataSource = sourceTable;
        ProductsGridView.DataBind();
    }
}
```

### NextResult() method Of SqlDataReader

The following SqlCommand object returns two result-sets, **one from - tblProductInventory** and the **other from tblProductCategories**.

```
SqlCommand command = new SqlCommand("select * from tblProductInventory; select * from
tblProductCategories", connection);
```

To retrieve the second result-set from **SqlDataReader** object, use the **NextResult()** as shown in the code snippet below. The **NextResult()** method returns true and advances to the next result-set.

```
string connectionString
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    SqlCommand command = new SqlCommand("select * from tblProductInventory; select * from
tblProductCategories", connection);
    using (SqlDataReader reader = command.ExecuteReader())
    {
        ProductsGridView.DataSource = reader;
        ProductsGridView.DataBind();

        while (reader.NextResult())
        {
            CategoriesGridView.DataSource = reader;
            CategoriesGridView.DataBind();
        }
    }
}
```

The **SqlDataReader** object's **Read()** method is used to loop thru the rows in a given result set, where as the **NextResult()** method is used to loop thru multiple result sets.

## SqlDataAdapter in ADO.NET

---

**SqlDataReader** is connection oriented, meaning it requires an active and open connection to the data source. **SqlDataAdapter** and **DataSet** provides us with disconnected data access model

**When creating an instance of the SqlDataAdapter, we specify**

1. The sql command that we want to execute
2. The connection on which we want to execute the command

**Example :**

1. Creates an **instance of SqlDataAdapter**, passing in the required parameters (**SqlCommandText** and the **Connection** object)
2. Creates an instance of **DataSet** object. A **DataSet** is an in-memory data store, that can store tables, just like a database.
3. The **Fill()** method of the **SqlDataAdapter** class is then invoked. This method does most of the work. It opens the connection to the database, executes the sql command, fills the dataset with the data, and closes the connection. Opening and closing connections is handled for us. The connection is kept open only as long as it is needed.

4. The **dataset** object, is then set as the **datasource** of the **GridView1** control
5. Finally the **DataBind()** method is called, which binds the data to the control.

```
string connectionString
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
{
    // Create an instance of SqlDataAdapter. Specify the command and the connection
    SqlDataAdapter dataAdapter = new SqlDataAdapter("select * from tblProductInventory", connection);
    // Create an instance of DataSet, which is an in-memory datastore for storing tables
    DataSet dataset = new DataSet();
    // Call the Fill() methods, which automatically opens the connection, executes the command
    // and fills the dataset with data, and finally closes the connection.
    dataAdapter.Fill(dataset);

    GridView1.DataSource = dataset;
    GridView1.DataBind();
}
```

#### Executing a stored procedure using SqlDataAdapter:

```
Create procedure spGetProductInventory
as
Begin
Select ProductId, ProductName, UnitPrice
from tblProductInventory
End
```

If you want to execute stored procedure **spGetProductInventory**, using the **SqlDataAdapter**, just specify the name of the procedure instead of the in-line sql statement.

```
SqlDataAdapter dataAdapter = new SqlDataAdapter("spGetProductInventory", connection);
dataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
```

#### Executing a stored procedure with parameters using SqlDataAdapter:

```
Create procedure spGetProductInventoryById
@ProductId int
as
Begin
Select ProductId, ProductName, UnitPrice
from tblProductInventory
where ProductId = @ProductId
End
```

To execute stored procedure **spGetProductInventoryById**, we need to associate parameter **@ProductId** to the **SqlDataAdapter** object's **SelectCommand** as shown below.

```
string ConnectionString
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
using (SqlConnection connection = new SqlConnection(ConnectionString))
{
    // Create an instance of SqlDataAdapter, specifying the stored procedure
    // and the connection object to use
    SqlDataAdapter dataAdapter = new SqlDataAdapter("spGetProductInventoryById", connection);
    // Specify the command type is an SP
    dataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
    // Associate the parameter with the stored procedure
    dataAdapter.SelectCommand.Parameters.AddWithValue("@ProductId", 1);
    DataSet dataset = new DataSet();
    dataAdapter.Fill(dataset);

    GridView1.DataSource = dataset;
    GridView1.DataBind();
}
```

## DataSet in ADO.Net

---

When the following stored procedure is executed, we get 2 result-sets

Create procedure **spGetProductAndCategoriesData**

as

Begin

Select ProductId, ProductName, UnitPrice

from tblProductInventory

Select CategoryId, CategoryName

from tblProductCategories

End

Drag and drop 2 **GridView** controls onto the webform and change the ID to **GridViewProducts** and **GridViewCategories**. The HTML in the aspx page should be as shown below.

```
<asp:GridView ID="GridViewProducts" runat="server">
</asp:GridView>
<br />
<asp:GridView ID="GridViewCategories" runat="server">
</asp:GridView>
```

To specify the specific DataTable, that you want to bind to a gridview control, use the Tables collection property of the dataset object, as shown below.

```
string connectionString
= ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlDataAdapter dataAdapter = new SqlDataAdapter("spGetProductAndCategoriesData", connection);
    dataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
    DataSet dataset = new DataSet();
    dataAdapter.Fill(dataset);

    GridViewProducts.DataSource = dataset.Tables[0];
    GridViewProducts.DataBind();

    GridViewCategories.DataSource = dataset.Tables[1];
    GridViewCategories.DataBind();
}
```

**By default the tables in the DataSet will have table names as Table, Table1, Table2 etc.** So if you want to give the tables in the DataSet a meaningful name, use the TableName property as shown below.

```
dataset.Tables[0].TableName = "Products";
dataset.Tables[1].TableName = "Categories";
```

**These table names can then be used when binding to a GridView control,** instead of using the integral indexer, which makes your code more readable, and maintainable.

```
GridViewProducts.DataSource = dataset.Tables["Products"];
GridViewProducts.DataBind();

GridViewCategories.DataSource = dataset.Tables["Categories"];
GridViewCategories.DataBind();
```

## Insert Data into Excel

A Example of Inserting Data Into Excel

```
string connectionString = @"Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=D:\Sample1.xlsx;Extended Properties=""Excel 12.0;HDR=YES;""";
// if you don't want to show the header row (first row) in the grid
// use 'HDR=NO' in the string

string strSQL = "SELECT * FROM [Sheet1$]";
OleDbConnection excelConnection = new OleDbConnection(connectionString);
```



```
excelConnection.Open();
OleDbCommand command = new OleDbCommand();

command.Connection = excelConnection;
command.CommandText = "INSERT INTO [Sheet1$](ID ,Name )
VALUES(10,'Kapil')";

command.ExecuteNonQuery();
excelConnection.Close();

Console.WriteLine("Record Inserted Successfully");
```

---

#### Reading Data from Excel Sheet

---

```
string connectionString =
@"Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=D:\Customers.xlsx;Extended Properties=""Excel
12.0;HDR=YES;""";
// if you don't want to show the header row (first row)
in the grid // use 'HDR=NO' in the string

string strSQL = "SELECT * FROM [Sheet1$]";
OleDbConnection excelConnection = new
OleDbConnection(connectionString);
excelConnection.Open(); // this will open an Excel file
OleDbCommand dbCommand = new OleDbCommand(strSQL,
excelConnection);
OleDbDataAdapter dataAdapter = new
OleDbDataAdapter(dbCommand);

// create data table
DataTable dTable = new DataTable();

dataAdapter.Fill(dTable);

ReadData(dTable);
```

```
//dTable.Dispose()
dataAdapter.Dispose();
dbCommand.Dispose();
excelConnection.Close();
excelConnection.Dispose();
```

## Convert Dataset to XML

### Examples

```
SqlConnection sconn = new SqlConnection("Data
Source=dell1;Initial Catalog=nikhil_db;Integrated Security=true");
//OleDbConnection con = new OleDbConnection();
//con.ConnectionString =
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\\Northwind.mdb";
// create a data adapter
//OleDbDataAdapter da = new OleDbDataAdapter("Select *
from Customers", con);
// create a new dataset
SqlDataAdapter adap = new SqlDataAdapter("SELECT * FROM
CUSTOMER", sconn);
DataSet ds = new DataSet();
// fill dataset
adap.Fill(ds);
// write dataset contents to an xml file by calling
WriteXml method
ds.WriteXml(@"D:\3.0 ENOSISLEARNING\1.0
NetBasics\Files\OutputCustomer.xml");

Console.WriteLine("Record added successfully");
```

## Convert XML to Dataset

---

### Examples

```
public static void ReadXMLToSqlServer()
{
    SqlConnection sconn = new SqlConnection("Data
Source=DELL1;Initial Catalog=nikhil_db;Integrated Security=true");
    //XmlReader xmlFile;
    //xmlFile = XmlReader.Create("Product.xml", new
XmlReaderSettings());
    DataSet ds = new DataSet();
    // ds.ReadXml("D:\\Products.xml");

    ds.ReadXml(@"E:\1.ENOSISLEARNING\1.0
NetBasics\Files\OutputCustomer.xml");
    int i = 0;
    for (int r = 0; r <= ds.Tables[0].Rows.Count - 1; r++)
    {
        int id =
Convert.ToInt32(ds.Tables[0].Rows[r][0].ToString());
        string name = ds.Tables[0].Rows[r][1].ToString();
        DateTime enqdate =
Convert.ToDateTime(ds.Tables[0].Rows[r][2]);

        string query = "insert into customer_new values(" +
id + ", '" + name + "', '" + enqdate + "')";
        SqlCommand scomm = new SqlCommand(query, sconn);
        sconn.Open();
        scomm.ExecuteNonQuery();
        sconn.Close();

        //MessageBox.Show(ds.Tables[0].Rows[i].ItemArray[2].ToString());
    }
}
```

## SqlCommandBuilder

---

SqlCommandBuilder automatically generates INSERT, UPDATE and DELETE sql statements based on the SELECT statement for a single table.

For the Transact-SQL statements to be generated using SqlCommandBuilder, there are 2 steps

**Step 1.** Set the "SelectCommand" property of the SqlDataAdapter object

```
SqlDataAdapter dataAdapter = new SqlDataAdapter();  
dataAdapter.SelectCommand = new SqlCommand("SELECT_Query", con);
```

**Step 2.** Create an instance of SqlCommandBuilder class and associate the SqlDataAdapter object created above using DataAdapter property of the SqlCommandBuilder object

```
SqlCommandBuilder builder = new SqlCommandBuilder();  
builder.DataAdapter = dataAdapter;
```

**Please Note:** Step 2, can also be done in single line as shown below. Here, we are passing the SqlDataAdapter instance as an argument to SqlCommandBuilder class constructor

```
SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapter);
```

```
public partial class WebForm1 : System.Web.UI.Page  
{  
    protected void Page_Load(object sender, EventArgs e)  
    {  
    }  
  
    protected void btnGetStudent_Click(object sender, EventArgs e)  
    {  
        string connectionString =  
            ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
        SqlConnection connection = new SqlConnection(connectionString);  
    }  
}
```

```
string selectQuery = "Select * from tblStudents where ID = " +
    txtStudentID.Text;
SqlDataAdapter dataAdapter = new SqlDataAdapter(selectQuery, connection);

DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet, "Students");

// Store DataSet and the select query in ViewState, so they can be used
// later to generate the T-SQL commands using SqlCommandBuilder class
ViewState["DATASET"] = dataSet;
ViewState["SELECT_QUERY"] = selectQuery;

if (dataSet.Tables["Students"].Rows.Count > 0)
{
    DataRow dataRow = dataSet.Tables["Students"].Rows[0];
    txtStudentName.Text = dataRow["Name"].ToString();
    txtTotalMarks.Text = dataRow["TotalMarks"].ToString();
    ddlGender.SelectedValue = dataRow["Gender"].ToString();
    lblStatus.Text = "";
}
else
{
    lblStatus.ForeColor = System.Drawing.Color.Red;
    lblStatus.Text = "No record with ID = " + txtStudentID.Text;
}
}

protected void btnUpdate_Click(object sender, EventArgs e)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
    SqlConnection con = new SqlConnection(connectionString);

    SqlDataAdapter dataAdapter = new SqlDataAdapter();
    // Retrieve the Select query from ViewState and use it to build
    // SqlCommand command object, which will then be set as the
    // SelectCommand of the SqlDataAdapter object
    dataAdapter.SelectCommand =
```

```

        new SqlCommand((string)ViewState["SELECT_QUERY"], con);

// Associate SqlDataAdapter object with SqlCommandBuilder. At this point
// SqlCommandBuilder should generate T-SQL statements automatically
SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapter);

DataSet ds = (DataSet)ViewState["DATASET"];
DataRow dr = ds.Tables["Students"].Rows[0];
dr["Name"] = txtStudentName.Text;
dr["Gender"] = ddlGender.SelectedValue;
dr["TotalMarks"] = txtTotalMarks.Text;
dr["Id"] = txtStudentID.Text;

int rowsUpdated = dataAdapter.Update(ds, "Students");
if (rowsUpdated == 0)
{
    lblStatus.ForeColor = System.Drawing.Color.Red;
    lblStatus.Text = "No rows updated";
}
else
{
    lblStatus.ForeColor = System.Drawing.Color.Green;
    lblStatus.Text = rowsUpdated.ToString() + " row(s) updated";
}
}
}

```