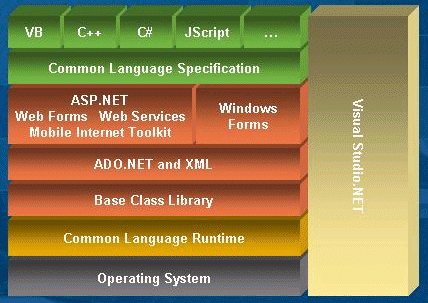
|  |
| --- |
| - |
| .Net Framework & C# |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |

**Understanding the .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



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

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. It also enforces strict type safety and ensures code accuracy in order to provide security and robustness to the applications. 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.

**CLR Features**

1. CLR manages memory, thread execution, code execution, compilation code safety verification and other system services.

2. For security reasons, managed code is assigned varying degrees of trust based on origin. This prevents or allows the managed component from performing file access operations, registry access operations or other sensitive functions even within the same active application.

3. 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. This enables the managed code to consume other managed types and enforce strict type fidelity and type safety.

4. CLR eliminates many common software issues like handling of object layout, references to objects and garbage clearance. This type of memory management prevents memory leaks and invalid memory references.

5. The CLR also accelerates developer productivity. The programmer is free to choose the language of the application without worrying about compatibility and integration issues. He is also enabled to take advantage of the runtime and the class library of the .NET Framework and also harvest components from other applications written in different languages by different developers. This implicitly eases the process of migration.

6. Finally, server side applications can host runtime. High performance servers like Microsoft SQL Server and Internet Information Services can host this CLR and the infrastructure so provided can be used to write business logic while enjoying the best benefits of enterprise server support.

The Class Library is an object oriented collection of reusable types. It is comprehensive and the types can be used to develop command line applications or GUI applications such as Web forms or XML Web services. 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. Third party components can be integrated seamlessly with classes in the .NET framework.

4. It enables a range of common programming tasks such as string management, data collection and file access.

5. It supports a variety of specialized development scenarios such as console application development, Windows GUI applications, ASP.NET Applications, XML Web services.

#### The 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.

The process of converting a value type to a reference type and vice versa is called boxing and unboxing. The implicit conversion of a value type to a reference type is referred to as boxing. The explicit conversion of an object type into a specific value type is referred to as unboxing.

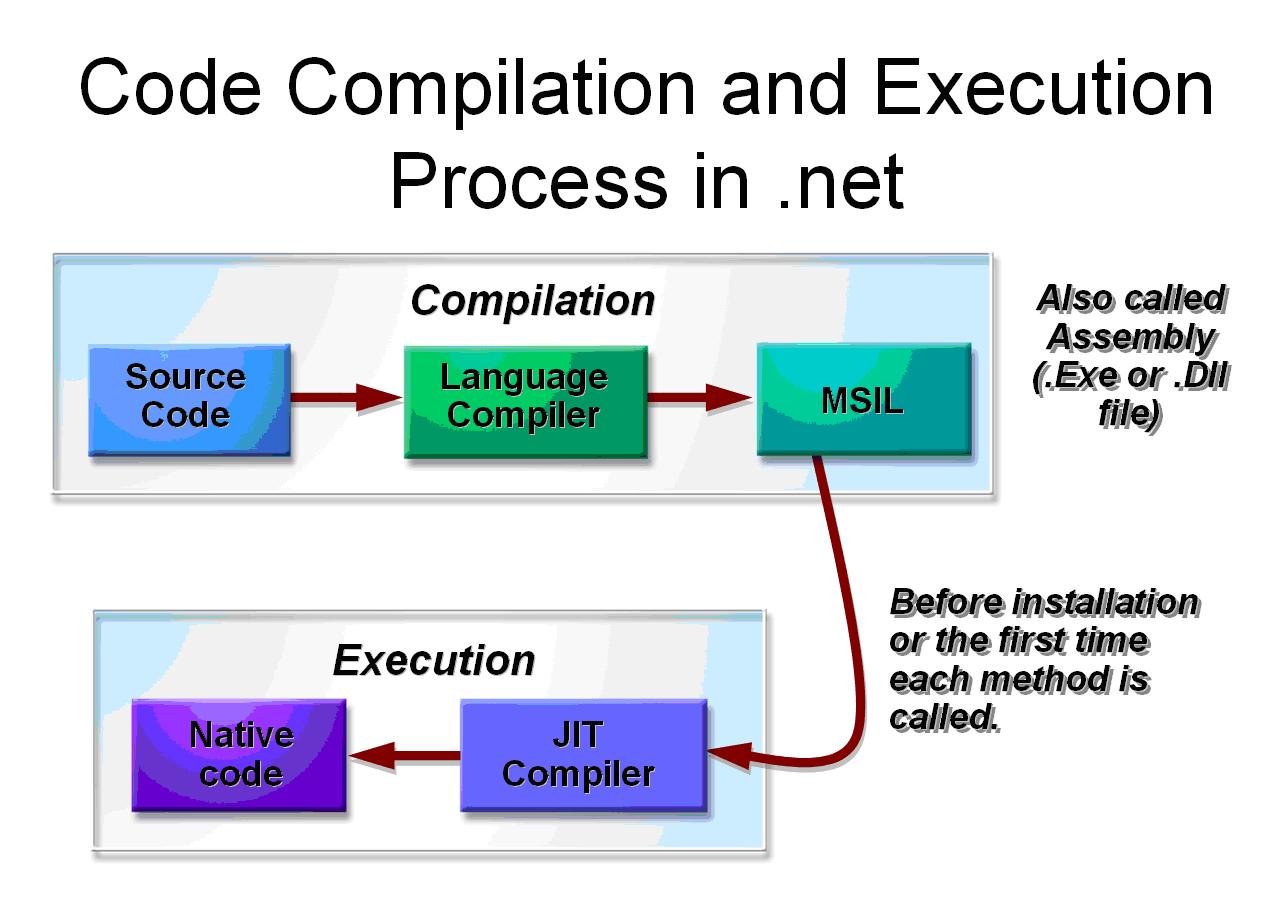
#### The Common Language Infrastructure (CLI)

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. A subset of the Framework Class Library incorporates the base class library, a Network library, a Reflection library, an XML library and Floating point and Extended Array Library. The shared source implementation of the CLI is available for both the FreeBSD and Windows operating Systems.

#### The 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. It provides robust interoperability between the .NET languages and the ability to inherit classes written in one language in any other .NET language. Cross language debugging also becomes a possibility in this scenario. It must be noted that the CLS rules apply only to publicly exposed features of a class.

**Code Execution of .Net**



#### Intermediate language (IL)

This is a processor independent representation of executable code. It is similar to assembly code and specific to the CLR. It is generated by the language compilers that target the CLR. At runtime, the CLR just-in-time compiles the IL to native code for execution. The tool ngen.exe which is part of the .NET framework pre-compiles assemblies to native code at install time and caches the precompiled code to the disk.

#### Managed execution

This refers to code whose execution is managed by the CLR. It includes memory management, access security, cross-language integration for debugging and exception handling etc. These assemblies are required for the creation of metadata on the code and the assemblies so that the CLR can manage the execution of the code.

**C# Data Types**

Data Types in a programming language describes that what type of data a variable can hold. [CSharp](http://csharp.net-informations.com) is a strongly typed language, therefore every variable and object must have a declared type. The CSharp type system contains three Type categories. They **are Value Types & Reference Types.** In CSharp it is possible to convert a value of one type into a value of another type. The operation of Converting a Value Type to a Reference Type is called **Boxing** and the reverse operation is called **Unboxing.**

When we declare a variable, we have to tell the compiler about what type of the data the variable can hold or which data type the variable belongs to.

Syntax: Data Type VariableName

**DataType: The type of data that the variable can hold**

VariableName: the variable we declare for hold the values.

**Example:**

int count;

int : is the data type

Count: is the variable name

**bool :** The bool keyword is an alias of System.Boolean. It is used to declare variables to store the Boolean values, true and false. In C#, there is no conversion between the bool type and other types.

|  |
| --- |
| **C# Runtime type: System.Boolean**  **CSharp declaration : bool flag;**  **CSharp Initialization : flag = true; CSharp default initialization value : false** |

**int :** int variables are stored signed 32 bit integer values in the range of -2,147,483,648 to +2,147,483,647

|  |
| --- |
| **C# Runtime type : System.Int32**  **CSharp declaration : int count;**  **CSharp Initialization : count = 100; // CSharp default initialization value : 0** |

**decimal :** The decimal keyword denotes a 128-bit data type. The approximate range and precision for the decimal type are -1.0 X 10-28 to 7.9 X 1028

C# Runtime type : System.Decimal

CSharp declaration : decimal val;

CSharp Initialization : val = 0.12;

CSharp default initialization value : 0.0M

**string :** The string type represents a string of Unicode characters. string variables are stored any number of alphabetic, numerical, and special characters .

C# Runtime type : System.String

CSharp declaration : string str;

CSharp Initialization : str = "csharp string";

**Reading value from console**

**Using Console.ReadLine**

Here we see how you can use the string result variable from the Console.ReadLine parameterless method as an **integer** value. You can invoke the int.TryParse method to see if the result string is an integer representation, and it will return the value of that integer if possible.

class Program

{

static void Main()

{

Console.WriteLine("Type an integer:");

string line = **Console.ReadLine**(); *// Read string from console*

int value;

if (int.TryParse(line, out value)) *// Try to parse the string as an integer*

{

Console.Write("Multiply integer by 10: ");

Console.WriteLine(value \* 10); *// Multiply the integer and display it*

}

else

{

Console.WriteLine("Not an integer!");

}

}

}

**~~~ Output of the program ~~~**

Type an integer:

4356

Multiply integer by 10: 43560

**Parsing line as integer.** The program text contains the Main entry point, which prompts the user for an input. The string variable with the identifier 'line' is then assigned to the reference of the string data allocated by Console.ReadLine and filled with the user's input. The int.TryParse static method then tests for a numeric value, and if this test succeeds we can then use the integer.

**Pausing before exit**

You can insert a Console.ReadLine() method call at the end of the Main method (or even in a finally block in the Main method). This will ensure the terminal window is never dismissed by Windows immediately on program completion.

**LOOPING STATEMENTS**

**For Loop**

**for(initialization;condition;icrementation)**

**{**

**Statements; //Multiple statements can be there**

**}**

**Example :**

**for(int i=1;i<11;i++)**

**{**

**Console.WriteLine(i);**

**}**

**Console.ReadLine();**

**OutPut : 1 2 3 4 5 6 7 8 9 10**

**While Loop**

**Syntax :**

**Initialization;**

**While(condition)**

**{**

**Statements; //Multiple statements can be there**

**Incrementation;**

**}**

**Example :**

**int i=1;**

**While(i<11)**

**{**

**Console.WriteLine(i);**

**}**

**Console.ReadLine();**

**OutPut : 1 2 3 4 5 6 7 8 9 10**

**Do While Loop**

**Initialization;**

**do**

**{**

**Statements; //Multiple statements can be there**

**}while(Condition);**

**Example :**

**int i=1;**

**do**

**{**

**Console.WriteLine(i);**

**}While(i<10);**

**CONDITIONAL STATEMENTS**

**IF \_ ELSE**

**if(Condition) //condition is any type of conditional statements**

**{**

**Statements;**

**}**

**else**

**{**

**Statements;**

**}**

**Example :**

          string Gender=””;

Console.WriteLine(“Enter your Gender”);

Gender = Console.ReadLine();

If (Gender == “MALE”)

{

Console.WriteLine(“Gender is Male”);

}   
 else

{

Console.WriteLine(“Gender is Female”);

}  
**IF \_ ELSE IF \_ELSE**

**if(Condition) //condition is any type of conditional statements**

**{**

**Statements;**

**}**

**else if(condition)**

**{**

**Statements;**

**}**

**else**

**{**

**Statements;**

**}**

#### Switch Case Statement

#### 

#### A Switch Statement for the Equals button

In between the round brackets after the word switch, we've typed the name of our variable (**theOperator**). We want to check what is inside of this variable. It will be one of four options: +, -, \*, /. So after the first case, we type a plus symbol. It's in between double quotes because it's text. You end a case line with a colon:

**case "+" :**

The code to add up goes on a new line. After the code, the break word is used. So what you're saying is:

**"If it's the case that theOperator holds a + symbol, then execute some code"**

We have three more case parts to the switch statement, one for each of the math symbols. Notice the addition of this, though:

default :  
//DEFAULT CODE HERE  
break;

You use **default** instead **case** just "in case" none of the options you've thought of are what is inside of your variable. You do this so that your programme won't crash!

#### Classes

A blueprint of an object is called a class. All definitions of haw a particular object will be instantiated at runtime, its properties and methods and storage structures are defined in the class. Classes are used by developers for creating instances of the class at runtime using the keyword “New”.

**Classes**

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

Every *class* has a constructor, which is called automatically any time an instance of a *class* is created.

The purpose of constructors is to initialize *class* members when an instance of the *class* is created.

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

**Listing 7-1. Example C# Classes: Classes.cs**

// helper class  
class OutputClass   
{  
    string myString;  
    // Constructor  
    public OutputClass(string inputString)   
    {  
        myString = inputString;  
    }  
    // Instance Method  
    public void printString()   
    {  
        Console.WriteLine("{0}", myString);  
    }  
    // Destructor  
    ~OutputClass()   
    {  
        // Some resource cleanup routines  
    }  
}  
// Program start class  
class ExampleClass   
{  
    // Main begins program execution.  
    public static void Main()   
    {  
        // Instance of OutputClass  
        OutputClass outCl = new OutputClass("This is printed by the output class.");  
        // Call Output class' method  
        outCl.printString();   
    }  
}

Listing 7-1 shows two classes. The top *class*, *OutputClass*, has a constructor, instance method, and a destructor. It also had a field named *myString*.

Notice how the *OutputClass* constructor is used to initialize data members of the *class*. In this case, the *OutputClass* constructor accepts a *string* argument, *inputString*. This *string* is copied to the *class* field *myString*.

Constructors are not mandatory, as indicated by the implementation of *ExampleClass*. In this case, a default constructor is provided. A default constructor is simply a constructor with no arguments

The example above illustrates how a class can have multiple constructors. The specific constructor called depends on the number of parameters and the type of each parameter.

In C#, there are two types of *class* members, instance and *static*.

Instance *class* members belong to a specific occurrence of a *class*.

Every time you declare an object of a certain *class*, you create a new instance of that *class*.

The *ExampleClass* *Main()* method creates an instance of the *OutputClass* named *outCl*. You can create multiple instances of *OutputClass* with different names.

Each of these instances are separate and stand alone. For example, if you create two *OutputClass* instances as follows:

    OutputClass oc1 = new OutputClass("OutputClass1");  
    OutputClass oc2 = new OutputClass("OutputClass2");

You create two separate instances of *OutputClass* with separate *myString* fields and separate *printString()* methods.

On the other hand, if a *class* member is *static*, you can access it simply by using the syntax *<classname>.<static class member>*. The instance names are *oc1* and *oc2*.

Suppose *OutputClass* had the following *static* method:

    public static void staticPrinter()   
    {  
        Console.WriteLine("There is only one of me.");  
    }

Then you could call that function from *Main()* like this:

OutputClass.staticPrinter();

You must call *static* class members through their *class* name and not their instance name. This means that you don't need to instantiate a class to use its *static* members. There is only ever one copy of a *static* class member. A good use of *static* members is when there is a function to be performed and no intermediate state is required, such as math calculations.

Matter of fact, the .NET Frameworks Base Class Library includes a *Math* *class* that makes extensive use of *static* members.

*OutputClass* also has a destructor. Destructors look just like constructors, except they have a tilde, "~", in front of them. They don't 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.

## Access Specifier or Modifier in C#

**Visibility Control**:- When we want to  implement inheritance, it is important to understand ,how to establish visibility levels for our classes and their members. There are four types of a accessibility modifiers which may be applied to classes and members to specify their level of visibility.

1. Public
2. private
3. protected
4. internal

**Class Member Visibility:-**  One of the goal of object oriented programming is **data hiding.** That is a class may be designed to hide its members from outside accessibility. C#  provides a set of **'access modifiers'** It specify the scope of type and its member up to which level they can be access.

There are five access specifier in C#.

1. **Private:-** Private member can be access only within the block { },where they have been declared. By default the class member are private.
2. **Protected:-** protected member can be access within containing classes and Derived classes. Protected member is visible only to its own class and its derived classes.

**3. Internal:-** Internal member can be access within Containing classes and Containing program.

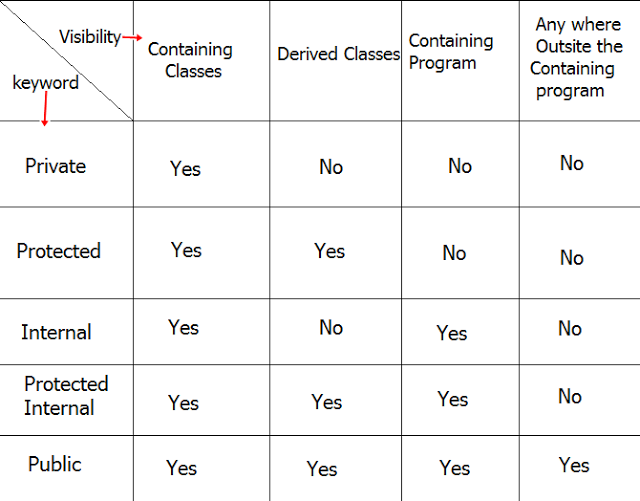
**4.** **Protected Internal:-** Protected Internal member is access within containing classes, Derived classes.and containing program.It is available in the containing program or assembly and in the Derived classes.

**5. Public** :-Public member is access  within containing classes,Derived classes, containing program, anywhere outside the containing program.  Public member is accessible from anywhere outside the the class as well. It is also accessible in Derived classes.

When no modifier is specified, it default to **'private'** accessibility. It is important to remember that the accessibility of a member is never larger than that of the class containing it.

You can easily understand all modifiers with the help of Diagram which is given below.

See it:- **Visibility of class 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

**Properties**

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field. Now, lets see how to use properties.

**Encapsulating Type State with Properties**

**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.

**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. Listing 10-4 shows you how to create and use write-only properties.

**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.

**Auto-Impemented 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 & Polymorphism**

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");  }  }; |

Now try this code out.

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

We have an object of type Animal, but it references an object of type Dog. Thus you can see the base class constructor getting called first followed by the derived class constructor. Now we call Talk() and find that the method that's executed is the base class method. That's not surprising when you consider that the object was declared to be of the base type which in our case is Animal. Now when we call Sing(), we find that the derived class method has got called. This is because in the base class the method is prototyped as public virtual void Sing() and in the derived class we have overridden it by using public override void Sing(). In C#, we need to explicitly use the override keyword as opposed to C++ where we didn't have to do that. And finally when we call Greet() the base class method gets called and this is not confusing at all specially since the derived class has not even implemented the method.

Now try the following code out.

Dog d1 = new Dog();

d1.Talk();

d1.Sing();

d1.Greet();

//Output

Animal constructor

Dog constructor

Dog talk

Dog song

Animal says Hello

Okay, here everything came out as expected. No rude surprises there. The fact that we could invoke the Greet() method is proof of inheritance in C#, not that anyone had any doubts to begin with I guess. Now take a look at this new class we'll be using as a base class for some other classes.

class Color

{

public virtual void Fill()

{

Console.WriteLine("Fill me up with color");

}

public void Fill(string s)

{

Console.WriteLine("Fill me up with {0}",s);

}

};

Now run this code out.

Color c1 = new Color();

c1.Fill();

c1.Fill("red");

**//Output**

Fill me up with color

Fill me up with red

Okay, that went fine, I'd say. Now let's derive a class from this class.

class Green : Color

{

public override void Fill()

{

Console.WriteLine("Fill me up with green");

}

};

Now let's try this code out.

Green g1 = new Green();

g1.Fill();

g1.Fill("violet");

//Output

Fill me up with green

Fill me up with violet

Well, that went fine too. Thus if you have overloaded methods, you can mark some of them as virtual and override them in the derived class. It's not required that you have to override all the overloads. Now I want to demonstrate some stuff on overloaded constructors. For that we'll use the following base class.

class Software

{

public Software()

{

m\_x = 100;

}

public Software(int y)

{

m\_x = y;

}

protected int m\_x;

};

Now we'll derive a class from the above class.

class MicrosoftSoftware : Software

{

public MicrosoftSoftware()

{

Console.WriteLine(m\_x);

}

};

Now try this code out

MicrosoftSoftware m1 = new MicrosoftSoftware();

//MicrosoftSoftware m2 = new MicrosoftSoftware(300); //won't compile

//Output

100

The base class had two overloaded constructors. One that took zero arguments and one that took an int. In the derived class we only have the zero argument constructor. Constructors are not inherited by derived classes. Thus we cannot instantiate a derived class object using the constructor that takes an int as parameter. As you will deduce from the output we got, the base class constructor that called was the default parameter-less constructor. Now take a look at this second derived class.

class DundasSoftware : Software

{

//Here I am telling the compiler which

//overload of the base constructor to call

public DundasSoftware(int y) : base(y)

{

Console.WriteLine(m\_x);

}

//Here we are telling the compiler to first

//call the other overload of the constructor

public DundasSoftware(string s, int f) : this(f)

{

Console.WriteLine(s);

}

};

Here we have two constructors, one that takes an int and one that takes a string and an int. Now lets try some code out.

DundasSoftware du1 = new DundasSoftware(50);

//Output

50

DundasSoftware du2 = new DundasSoftware("test",75);

//Output

75

test

There, now that you've seen how it came out, things are a lot clearer I bet. You can use this and base access keywords on other methods too, and not just on constructors.

**Interfaces**

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. Listing 13-1 shows how to define an interface:

**Listing 13-1. Defining an Interface: MyInterface.cs**

|  |
| --- |
| **interface IMyInterface** {     void MethodToImplement(); } |

Listing 13-1 defines an *interface* named *IMyInterface*. A common naming convention is to prefix all *interface* names with a capital "I". 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.

Listing 13-2 shows how this *interface* could be used.

**Listing 13-2. Using an Interface: InterfaceImplementer.cs**

|  |
| --- |
| class InterfaceImplementer : IMyInterface {     static void Main()     {         InterfaceImplementer iImp = new InterfaceImplementer();         iImp.MethodToImplement();     }     public void MethodToImplement()     {         Console.WriteLine("MethodToImplement() called.");     } } |

The *InterfaceImplementer* *class* in Listing 13.2 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. Any difference between the method signature in the interface and the method signature in the implementing *class* or *struct* will cause a compiler error. Additionally, a *class* or *struct* that inherits an interface must include all interface members; You will receive a compiler error if you don't implement all interface members.

Interfaces may also inherit other interfaces. Listing 13-3 shows how inherited interfaces are implemented.

**Interface Inheritance: InterfaceInheritance.cs**

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

The code in listing 13.3 contains two *interfaces*: *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. Since the *InterfaceImplementer* *class* in Listing 13-3 inherits from *IMyInterface*, it also inherits *IParentInterface*. Therefore, the *InterfaceImplementer* class must implement the *MethodToImplement()* method specified in the *IMyInterface* interface and the *ParentInterfaceMethod()* method specified in the *IParentInterface*

**ABSTRACT CLASS**

Abstract classes are one of the essential behaviors provided by .NET. Commonly, you would like to make classes that only represent base classes, and don’t want anyone to create objects of these class types. You can make use of abstract classes to implement such functionality in C# using the modifier 'abstract'.

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;  }  }  } |

In the above sample, you can see that the abstract class absClass contains two methods AddTwoNumbers and MultiplyTwoNumbers. AddTwoNumbers is a non-abstract method which contains implementation and MultiplyTwoNumbers is an abstract method that does not contain implementation.

The class absDerived is derived from absClass and the MultiplyTwoNumbers is implemented on absDerived. Within the Main, an instance (calculate) of the absDerived is created, and calls AddTwoNumbers and MultiplyTwoNumbers. You can derive an abstract class from another abstract class. In that case, in the child class it is optional to make the implementation of the abstract methods of the parent class.

**Example**

|  |
| --- |
| //Abstract Class1  **abstract class absClass1**  {  public abstract int AddTwoNumbers(int Num1, int Num2);  public abstract int MultiplyTwoNumbers(int Num1, int Num2);  }  //Abstract Class2  **abstract class absClass2:absClass1**  {  //Implementing AddTwoNumbers  public override int AddTwoNumbers(int Num1, int Num2)  {  return Num1+Num2;  }  }  //Derived class from absClass2  **class absDerived:absClass2**  {  //Implementing MultiplyTwoNumbers  public override int MultiplyTwoNumbers(int Num1, int Num2)  {  return Num1\*Num2;  }  } |

In the above example, absClass1 contains two abstract methods AddTwoNumbers and MultiplyTwoNumbers. The AddTwoNumbers is implemented in the derived class absClass2. The class absDerived is derived from absClass2 and the MultiplyTwoNumbers is implemented there.

**Abstract properties**

Following is an example of implementing abstract properties in a class.

|  |
| --- |
| //Abstract Class with abstract properties  **abstract class absClass**  {  protected int myNumber;  public abstract int numbers  {  get;  set;  }  }  **class absDerived:absClass**  {  //Implementing abstract properties  public override int numbers  {  Get {  return myNumber;  }  Set {  myNumber = value;  }  }  } |

In the above example, there is a protected member declared in the abstract class. The get/set properties for the member variable myNumber is defined in the derived class absDerived.

**Important rules applied to abstract classes**

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

//Incorrect

abstract sealed class absClass

{

}

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.

//Incorrect

**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. In an interface, creation of additional functions will have an effect on its child classes, due to the necessary implementation of interface methods to classes.

The selection of interface or abstract class depends on the need and design of your project. You can make an abstract class, interface or combination of both depending on your needs.

* 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.

**Enums**

Enums are basically a set of named constants.

They are declared in C# using the enum keyword. Every enum type automatically derives from System.Enum and thus we can use System.

Enum methods on our Enums.

Enums are value types and are created on the stack and not on the heap. You don't have to use new to create an enum type. Declaring an enum is a little like setting the members of an array as shown below.

enum Rating {Poor, Average, Okay, Good, Excellent}

You can pass enums to member functions just as if they were normal objects. And you can perform arithmetic on enums too. For example we can write two functions, one to increment our  enum and the other to decrement our enum.

Rating IncrementRating(Rating r)

{

if(r == Rating.Excellent)

return r;

else

return r+1;

}

Rating DecrementRating(Rating r)

{

if(r == Rating.Poor)

return r;

else

return r-1;

}

Both functions take a Rating object as argument and return back a Rating object. Now we can simply call these functions from elsewhere.

for (Rating r1 = Rating.Poor;

r1 < Rating.Excellent ;

r1 = IncrementRating(r1))

{

Console.WriteLine(r1);

}

Console.WriteLine();

for (Rating r2 = Rating.Excellent;

r2 > Rating.Poor;

r2 = DecrementRating(r2))

{

Console.WriteLine(r2);

}

And here is a sample code snippet showing how you can call System.Enum methods on our Enum object. We call the GetNames method which retrieves an array of the names of the constants in the enumeration.

foreach(string s in Rating.GetNames(typeof(Rating)))

Console.WriteLine(s);

**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.

**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. Of all the silly features this one must win the annual contest by a long way. I see no sane reason why this must be so. 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.

**When to use structs**

Because structs are value types they would be easier to handle and more efficient that classes. When you find that you are using a class mostly for storing a set of values, you must replace those classes with structs. When you declare arrays of structs because they are created on the heap, efficiency again improves. Because if they were classes each class object would need to have memory allocated on the heap and their references would be stored. In fact lots of classes within the .NET framework are actually structs. For example System.Drawing.Point is actually a struct and not a 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. |

#### Namespaces

This is the key part of the .NET Framework. It provides scope for both preinstalled framework classes and custom developed classes. Vb.NET uses the “Imports” keyword to enable the use of member names from the namespace declared. C# uses the “using” keyword. In both cases the System Namespace is also imported so that the Console window can be written without explicitly referring to the System.Console.

#### Assemblies

Assemblies are also known as managed DLLs. They are the fundamental unit of deployment for the .NET platform. The .NET framework itself is made of a number of assemblies. An assembly contains the Intermediate language generated by the language compiler, an assembly manifest, type metadata and resources. They can be private or public. They are self describing and hence different versions of the same assembly can be run simultaneously

**Partial class**

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.

If you remove the partial modifier, you will get an error containing this text: [The namespace '<global namespace>' already contains a definition for 'A'].

Compiled result of A1.cs and A2.cs: C#

|  |
| --- |
| **internal class A**  {  // Methods  public static void A1()  {  Console.WriteLine("A1");  }  public static void A2()  {  Console.WriteLine("A2");  }  } |

**Sealed class**

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

{

}

In the following code, 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"); }  } |

**Why Sealed Classes?**   
We just saw how to create and use a sealed class. The main purpose of a sealed class is to take away the inheritance feature from the user so they cannot derive a class from a sealed class. One of the best usage of sealed classes is when you have a class with static members. For example, the "Pens" and "Brushes" classes of the "System.Drawing" namespace.

The Pens class represents the pens for standard colors. This class has only static members. For example, "Pens.Blue" represents a pen with the blue color. Similarly, the "Brushes" class represents standard brushes. "Brushes.Blue" represents a brush with blue color.

So when you're designing your application, you may keep in mind that you have sealed classes to seal the user's boundaries.

**Static Class**

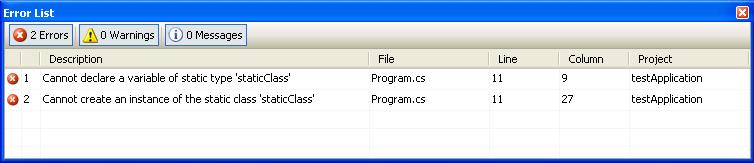
If a class declaration includes the keyword static then that class is termed as static class. Static class is a specialized class that cannot have any objects.

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

public static class staticClass  
 {

}  
 public class testStaticClass  
 {  
     public static void Main()  
     {  
         staticClass obj = new staticClass();  
     }  
 }

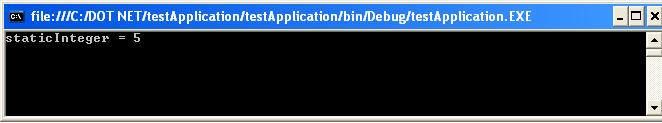
In this example, the class staticClass is a static class because it has the keyword static in its declaration. Hence staticClass cannot be instantiated. But inside Main method of testStaticClass, staticClass is instantiated with the object obj. This is incorrect leading to error:



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.

**Example#2 for Static Class in C#:**

|  |
| --- |
| public static class staticClass  {      static int staticInteger;      public static void setStaticInteger(int tmpData)      {          staticInteger = tmpData;      }      public static int getStaticInteger()      {          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);**

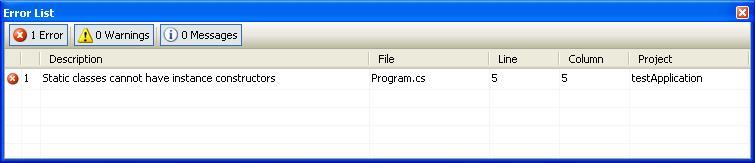
**Characteristics of Static Class in C#**

* **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.

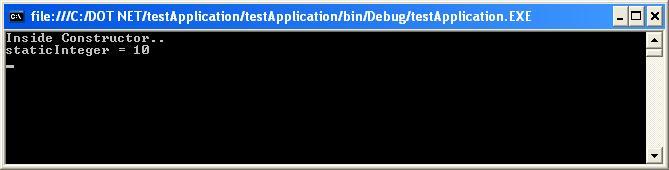
|  |
| --- |
| public static class staticClass  {      staticClass()      {          Console.WriteLine("Inside Constructor..");      }  } |

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.

**Static Class is Implicitly Sealed**

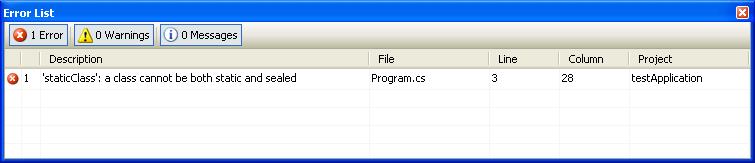
When the sealed keyword is used in any class declaration then the class cannot be inherited. Static class is implicitly sealed which means that static class cannot be inherited as well.

**Static Class Cannot Be Declared as Sealed**

Since the static class is implicitly sealed, static class should not be explicitly declared as Sealed as done in the following example:

public **sealed** static class staticClass  
 {  
   
 }

In the above example, the static class staticClass is marked as sealed. This will lead to the following error:

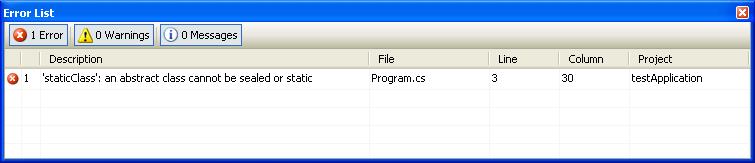


**Static Class Cannot be Declared as Abstract**

Abstract keyword in a class declaration means that the abstract class cannot be instantiated but it has to be inherited. But static class cannot be inherited. Hence static class cannot be abstract. This is proven in the below example:

using System;  
   
 public **abstract** static class staticClass  
 {  
 }

The above code on execution leads to the following error:



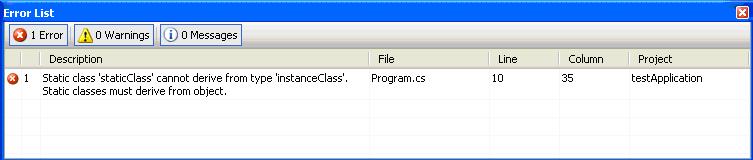
**Static Class Should Inherit Only Object**

**Static class cannot inherit any other static class / concrete class.**

**Static class is allowed to inherit only the Object. Example:**

|  |
| --- |
| public class instanceClass  {      public static void Main()  {          Console.WriteLine("Testing Inheritance by Static Class");    }  }  public static class staticClass :  instanceClass  {  } |

In this example, the static class named staticClass inherits a non static class called instanceClass. This is not possible.

 All the classes are implicitly derived from Object class. Static classes are no exception to it. If a static class has to explicitly participate in inheritance, then static class can derive only from object class as shown below:

|  |
| --- |
| public class instanceClass  {     public static void Main()      {        Console.WriteLine("Testing Inheritance by Static Class");        Console.Read();      }  }  public static class staticClass : Object  {  } |

Now this code will work fine and produce the following output:

**Static Class Cannot be Inherited**

**Threading**

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

**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.

**How do they work**

A multitasking operation system divides the available processor time among the processes and threads that need it. A thread is executed in the given time slice, and then it is suspended and execution starts for next thread/process in the queue. When the OS switches from one thread to another, it saves thread context for preempted thread and loads the thread context for the thread to execute.

The length of time slice that is allocated for a thread depends on the OS, the processor, as also on the priority of the task itself.

**Working with threads**

In .NET framework, System. Threading namespace provides classes and interfaces that enable multi-threaded programming. This namespace provides:

* ThreadPool class for managing group of threads,
* Timer class to enable calling of delegates after a certain amount of time,
* A Mutex class for synchronizing mutually exclusive threads, along with classes for scheduling the threads, sending wait notifications and deadlock resolutions.

Information on this namespace is available in the help documentations in the Framework SDK.

**Defining and Calling threads**

To get a feel of how Threading works, run the below code:

|  |
| --- |
| 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 InstranceMethod.Running on Thread A”);  Console.WriteLine("Thread A Going to Sleep Zzzzzzzz”);  // Pause for a moment to provide a delay to make threads more apparent.  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 Zzzzzzzz");  Thread.Sleep(5000);  Console.WriteLine("You are 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.

An important property of this class (which is also settable) is Priority.

**Scheduling Threads**

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of ThreadPriority.Normal. Threads created outside the runtime retain the priority they had before they entered the managed environment. You can get or set the priority of any thread with the Thread.Priority property.

**Pausing and Resuming threads**

After you have started a thread, you often want to pause that thread for a fixed period of time. Calling [Thread.Sleep](http://www.codeproject.com/Articles/8694/_target) causes the current thread to immediately block for the number of milliseconds you pass to Sleep, yielding the remainder of its time slice to another thread. One thread cannot call Sleep on another thread. Calling Thread.Sleep(Timeout.Infinite) causes a thread to sleep until it is interrupted by another thread that calls [Thread.Interrupt](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassInterruptTopic.asp) or is aborted by [Thread.Abort](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassAbortTopic.asp).

**Thread Safety**

When we are working in a multi threaded environment, we need to maintain that no thread leaves the object in an invalid state when it gets suspended. Thread safety basically means the members of an object always maintain a valid state when used concurrently by multiple threads.

There are multiple ways of achieving this – The Mutex class or the Monitor classes of the Framework enable this, and more information on both is available in the Framework SDK documentation. What we are going to look at here is the use of locks.

You put a lock on a block of code – which means that that block has to be executed at one go and that at any given time, only one thread could be executing that block.

The syntax for the lock would be as follows:

using System.Threading;

//define the namespace, class etc.

...

public somemethod(...)

{

...

lock(this)

{

Console.WriteLine(“Inside the lock now”);

...

}

}

In the above code sample, the code block following the lock statement will be executed as one unit of execution, and only one thread would be able to execute it at any given time. So, once a thread enters that block, no other thread can enter the block till the first thread has exited it.

This becomes necessary in the kind of database transactions required in banking applications and reservations systems etc.

# What is multi-threading?

It's basically trying to do more than one thing at a time within a process.

So, what is a thread? A thread (or "thread of execution") is a sort of context in which code is running. Any one thread follows program flow for wherever it is in the code, in the obvious way. Before multi-threading, effectively there was always one thread running for each process in an operating system (and in many systems, there was only one process running anyway). If you think of processes running in parallel in an operating system (e.g. a browser downloading a file and a word processor allowing you to type, both "at the same time"), then apply the same kind of thinking within a single process, that's a reasonable way to visualize threading.

**How does multi-threading work in .NET?**

.NET has been designed from the start to support multi-threaded operation. There are two main ways of multi-threading which .NET encourages: starting your own threads with ThreadStart delegates, and using the ThreadPool class either directly (using ThreadPool.QueueUserWorkItem) or indirectly using asynchronous methods (such as Stream.BeginRead, or calling BeginInvoke on any delegate).

In general, you should create a new thread "manually" for long-running tasks, and use the thread pool only for brief jobs. The thread pool can only run so many jobs at once, and some framework classes use it internally, so you don't want to block it with a lot of tasks which need to block for other things. The examples in this article mostly use manual thread creation. On the other hand, for short-running tasks, particularly those created often, the thread pool is an excellent choice.

# Multi-threaded "Hello, world"

Here is virtually the simplest threading example which actually shows something happening:

|  |  |
| --- | --- |
| using System.Threading;  public class Test  {  static void Main()  {  ThreadStart job = new ThreadStart(ThreadJob);  Thread thread = new Thread(job);  thread.Start();    for (int i=0; i < 5; i++)  {  Console.WriteLine ("Main thread: {0}", i);  Thread.Sleep(1000);  }  }  static void ThreadJob()  {  for (int i=0; i < 10; i++)  {  Console.WriteLine ("Other thread: {0}", i);  Thread.Sleep(500);  }  }  } |  |

The code creates a new thread which runs the ThreadJob method, and starts it. That thread counts from 0 to 9 fairly fast (about twice a second) while the main thread counts from 0 to 4 fairly slowly (about once a second). The way they count at different speeds is by each of them including a call to Thread.Sleep, which just makes the current thread sleep (do nothing) for the specified period of time. Between each count in the main thread we sleep for 1000ms, and between each count in the other thread we sleep for 500ms. Here are the results from one test run on my machine:

|  |
| --- |
| Main thread: 0  Other thread: 0  Other thread: 1  Main thread: 1  Other thread: 2  Other thread: 3  Main thread: 2  Other thread: 4  Other thread: 5  Main thread: 3  Other thread: 6  Other thread: 7  Main thread: 4  Other thread: 8  Other thread: 9  **Synchronization in Threads**  When we have multiple threads that share data, we need to provide synchronized access to the data. We have to deal with synchronization issues related to concurrent access to variables and objects accessible by multiple threads at the same time. This is controlled by giving one thread a chance to acquire a lock on the shared resource at a time. We can think it like a box where the object is available and only one thread can enter into and the other thread is waiting outside the box until the previous one comes out.  using System;  using System.Threading;  namespace CSharpThreadExample  {  class Program  {  static void Main(string[] arg)  {  Console.WriteLine("\*\*\*\*\*Multiple Threads\*\*\*\*\*");  Printer p=new Printer();  Thread[] Threads=new Thread[3];  for(int i=0;i<3;i++)  {  Threads[i]=new Thread(new ThreadStart(p.PrintNumbers));  Threads[i].Name="Child "+i;  }  foreach(Thread t in Threads)  t.Start();  Console.ReadLine();  }  }  class Printer  {  public void PrintNumbers()  {  for (int i = 0; i < 5; i++)  {  Thread.Sleep(100);  Console.Write(i + ",");  }  Console.WriteLine();  }  }  }  In the above example, we have created three threads in the main method and all the threads are trying to use the PrintNumbers() method of the same Printer object to print to the console. Here we get this type of output:  image003.jpg  Now we can see, as the thread scheduler is swapping threads in the background each thread is telling the **Printer** to print the numerical data. We are getting inconsistent output as the access of these threads to the **Printer** object is synchronized. There are various synchronization options which we can use in our programs to enable synchronization of the shared resource among multiple threads.  **Using the Lock Keyword**  In C# we use lock(object) to synchronize the shared object.  Syntax:  lock (objecttobelocked) {  objecttobelocked.somemethod();  }  Here objecttobelocked is the object reference which is used by more than one thread to call the method on that object. The lock keyword requires us to specify a token (an object reference) that must be acquired by a thread to enter within the lock scope. When we are attempting to lock down an instance level method, we can simply pass the reference to that instance. (We can use this keyword to lock the current object) Once the thread enters into a lock scope, the lock token (object reference) is inaccessible by other threads until the lock is released or the lock scope has exited.  If we want to lock down the code in a static method, we need to provide the System.Type of the respective class.  **Converting the Code to Enable Synchronization using the Lock Keyword**  public void PrintNumbers()  {  lock (this)  {  for (int i = 0; i < 5; i++)  {  Thread.Sleep(100);  Console.Write(i + ",");  }  Console.WriteLine();  }  }  **OUTPUT**  image004.jpg  **Using the Monitor Type**  The C# lock keyword is just a notation for using System.Threading.Monitor class type. The lock scope actually resolves to the Monitor class after being processed by the C# compiler.  **Converting the Code to Enable Synchronization using the Monitor Class**  public void PrintNumbers()  {  **Monitor.Enter(this);**  try  {  for (int i = 0; i < 5; i++)  {  Thread.Sleep(100);  Console.Write(i + ",");  }  Console.WriteLine();  }  finally  {  **Monitor.Exit(this);**  }  }  Monitor.Enter() method is the ultimate recipient of the thread token. We need to write all code of the lock scope inside a try block. The finally clause ensures that the thread token is released(using the Monitor.Exit() method), regardless of any runtime exception.  **OUTPUT**  image005.jpg |

# USING .NET LIBRARIES

# System.IO

## Manipulate folders/directories using the System.IO Library:

|  |
| --- |
| **Delete a folder:**  Directory.Delete(“c:\\directory\\subdirectory\\”); //standard delete  Directory.Delete(“c:\\directory\\subdirectory\\”, true); //delete folder and all subdirectories |
| **Create a folder:**  **Directory.CreateDirectory(“c:\\directory\\subdirectory\\”)**  **Check if a folder exists:**  **Directory.Exists(“c:\\directory\\subdirectory\\”); //returns boolean value**  **Move/Rename a folder:**  **Directory.Move(“c:\\directory\\oldlocation\\”, “c:\\directory\\newlocation\\”);**  **Get a list of files in a folder:**  Directory.GetFiles(“c:\\directory\\subfolder\\”); // returns a string array of files in the folder |

|  |
| --- |
| **Delete a file:**  File.Delete(“c:\\directory\\filename.text”);  **Create a file:**  File.Create(“c:\\directory\\filename.text”);  //note, the file will be empty. There are some overloads available that will let you create a file with content in it  **Check if a file exists:**  File.Exists(“c:\\directory\\subdirectory\\filename.text”);  //returns boolean value  **Move/Rename a file:**  File.Move(“c:\\directory\\filename.text”, “c:\\directory\\filenamenew.text”);  **Create a copy of a file:**  File.Copy(“c:\\directory\\filename.text”, “c:\\directory\\filenamenew.text”); |

|  |
| --- |
| **Read text from file:**  string FileContents = File.ReadAllText(“c:\\folder\\filename.text”);  **Write text to a file:**  File.WriteAllText(“c:\\filename.text”, “contents of file”); |

**Getting List of Drives**

|  |
| --- |
| string[] str=Directory.GetLogicalDrives();  Console.WriteLine( "Using C# Directory Class,Available drives are:");  for(int i=0;i< str.Length;i++)  Console.WriteLine(str[i]); |

|  |
| --- |
| using System.IO;  //Reading a File  StreamReader streamReader = new StreamReader(filePath);  string text = streamReader.ReadToEnd();  streamReader.Close();  //Writing to File  using (StreamWriter writer = new StreamWriter("important.txt"))  {  writer.Write("Word ");  writer.WriteLine("word 2");  writer.WriteLine("Line");  } |

## System.Text

Using String Builder

|  |
| --- |
| StringBuilder builder = new StringBuilder();  builder.Append("The list starts here:");  builder.AppendLine();  builder.Append("1 cat").AppendLine();  // Get a reference to the StringBuilder's buffer content.  string innerString = builder.ToString(); |

## CALLING EXTERNAL EXECUTABLE FROM .NET ENVIRONMENT

using System.Diagnostics;

Process p= new Process();  
p.StartInfo.WorkingDirectory = @"C:\whatever";  
p.StartInfo.FileName = @"C:\some.exe";  
p.StartInfo.CreateNoWindow = true;  
p.Start();  
p.WaitForExit();

**//CALLING A NOTEPAD**

|  |
| --- |
| ProcessStartInfo startInfo = new ProcessStartInfo();  startInfo.FileName = "NOTEPAD.EXE";  Process.Start(startInfo);  //PASSING ARGUMENTS TO THE EXTERNAL EXE  ProcessStartInfo startInfo = new ProcessStartInfo(); c  startInfo.WorkingDirectory = "D:\\test";  startInfo.Arguments = "newfile.txt";    startInfo.FileName = "NOTEPAD.EXE";  Process.Start(startInfo); |

**//CALLING A WORD APPLICATION**

|  |
| --- |
| ProcessStartInfo startInfo = new ProcessStartInfo();  startInfo.FileName = "WINWORD.EXE";  Process.Start(startInfo); |

**What is an assembly in .NET?**

An assembly is a fundamental unit of any .NET application. It contains the code that is executed by CLR (common language runtime).

* An assembly contains name, version, types (classes and others) created in it and details about other assemblies it references.
* An assembly may be either an executable file - .EXE or a dynamic link library - .DLL

**Structure of an Assembly**

The following is the content of an assembly. Each assembly contains first three parts. Fourth part may not be present in all assemblies. It is used primarily for localization - using resources according to the country or region.

* Assembly Metadata or Manifest
* Type Metadata
* MSIL Code
* Resources

**Assembly Metadata or Manifest**

This contains information about the assembly. Remember, assemblies in .NET are self-describing. They contain all the information that .NET needs to use them. Assembly metadata contains the following details of an assembly:

* Assembly name
* Version number of the assembly, which has four numbers in the format ***major.minor.revison.build***
* Culture - language assembly supports
* Strong name - required only for global assemblies
* List of files in the assembly. An assembly can be made up of multiple files
* Type reference information - informs which type is in which file of the assembly
* Information about referenced assemblies - Contains list of other assemblies referenced by this assembly. For each assembly referenced we have assembly name, version, culture and public key (if assembly is a global assembly)

**Type metadata**

This section of an assembly contains information about all classes, structure etc. created in the assembly.

**MSIL Code**

MSIL code of the assembly is placed in third part of the assembly. This MSIL is converted to native code by CLR at runtime.

**Resource**

This section contains messages and pictures used by assembly.

**How to create an assembly in C#**

Steps to create a private assembly (by default all assemblies are private) using Visual C# 2005 Express Edition.

1. Select **File->New Project**
2. From Templates, select **Class Library**
3. Enter name **CounterLibrary**
4. A class library is created using a single class **Class1**
5. Rename class to **Counter** and add the following code.
6. namespace CounterLibrary
7. {
8. public class Counter
9. {
10. protected int v = 0;
11. public Counter(int v)
12. {
13. this.v = v;
14. }
15. public int Value
16. {
17. get
18. {
19. return v;
20. }
21. }
22. }
23. }

Save project using **File->Save All**. When prompted to enter location for project, select the folder where you want to save your project. I use **c:\csharp**. Do not select checkbox for *Create directory for solution*

Build (not run) the project using **Build->Build Solution**

After the above process, we get C**ounterLibrary.dll** assembly placed in **c:\csharp\counterlibrary\bin\release** directory.

The next step is to use this assembly in a console application. As a matter of fact, once a source program is converted to MSIL, it can be used anywhere in .NET, irrespective of language and type of application.

**Using a private assembly in a console application developed in C#**

Now, let us use the class library created in C# in a console application. Though I am using a console application in C#, you can use any language supported by .NET.

1. Start **Visual C# 2005 Express Edition**
2. Create a new console application using **File -> New Project**
3. From template select **Console Application** as type of project
4. Give name **UseCounter** for application.   
   A new application is created with a single class with **Main**() method.
5. Go to **Solution Explorer** and select project
6. Right click on it and select **Add References** from the context menu.
7. From dialog box, select Browse tab and select **c:\csharp\counterlibrary\bin\release\counterlibrary.dll**
8. Solution explorer displays **counterlibrary** as one of the references under references node in solution explorer
9. Add the following code in Main() method of **Program.cs**
10. using System;
11. namespace UseCounter
12. {
13. class Program
14. {
15. static void Main(string[] args)
16. {
17. **counterlibrary**.**Counter** c = new counterlibrary.Counter(100);
18. c.Inc();
19. Console.WriteLine(c.Value);
20. }
21. }
22. }

As you do the above, you can notice that a copy of **counterlibrary.dll** is copied into BIN directory of **UseCounter** application. This is the case with any private library. Whenever an application makes a reference to it, a copy of private assembly is copied into it's bin directory.  
If you do not see .DLL file that is copied to BIN directory of console application (UseCounter), close the application and reopen it.

**Making a private assembly a global assembly**

A global assembly is a public assembly that is shared by multiple applications. Unlike private assembly, a global assembly is not copied to bin directory of each application that references it. Global assembly instead is placed in **GAC** (Global Assembly Cache) and it can be referenced anywhere within the system. So only one copy is stored, but many applications can use that single copy.

In order to convert a private assembly to global assembly, we have to take the following steps.

* Create a strong name
* Associate strong name with assembly
* Place assembly in GAC

**Creating a strong name**

Any assembly that is to be placed in GAC, must have a strong name. Strong name is a combination of **public key and private key**. The relationship between public and private keys are such, given one you cannot get the other, but any data that is encrypted with private key can be decrypted only with the corresponding public key.

Take the following steps to invoke **SN** (Strong Name) tool to create strong name.

1. Go to command prompt using **Microsoft .NET Framework SDK v2.0 -> SDK Command prompt**
2. Go to **c:\csharp\counterlibrary** folder and enter the following command.
3. sn -k srikanth.key
4. The above command writes private and public key pair into **srikanth.key** file.

**Associate strong name with assembly**

Once private and public keys are generated using SN tool, use the following procedure to sign **counterlibrary** with the key file.

1. Open **counterlibrary** project.
2. Select project properties using **Project -> counterlibrary properties**
3. Select **Signing** tab in project properties window
4. Check **Sign the assembly** check box
5. Select **srikanth.key** file using **Choose a strong name key file** combo box
6. Close properties window
7. Build the solution again using **Build->Build Solution**

Now, **counterlibrary.dll** is associated with a public key and also digitally signed with private key. This ensures no one can modify this assembly as any change to assembly should re-sign the assembly with private key of the user who created it first. This protects the assembly from getting tampered with by others. A global assembly needs this projection as it is placed in common place.

You can verify whether the assembly is associated with public key using **ILDASM** (IL Disassembler) program provided by .NET Framework.

* Start **ILDASM** using **.NET Framework SDK v2.0->Tools->MSIL Disassembler**
* Select **counterlibrary.dll** using **File->Open**
* Once assembly is opened, double click on **Manifest** section of the assembly to see the public key associated with the assembly.

**Place assembly in GAC**

In order to make an assembly a global assembly, the assembly must be associated with a strong name and then placed in **Global** **Assembly** **Cache** (GAC).

GAC is a folder with name **Assembly** in **windows** folder of your system. So, place **counterlibrary.dll** in GAC using **GACUTIL** tool as follows.

c:\csharp\counterlibrary\bin\Release>gacutil -i counterlibrary.dll

After you install global assembly into GAC, you can see **counterlibrary.dll** in **windows/assembly** folder.

Once, you place an assembly in GAC, any reference to the assembly will not create a copy of the assembly in BIN directory of the application.

Instead all application that reference the assembly use the same copy that is placed in GAC.

**Indexers**

Indexers allow your class to be used just like an array. On the inside of a class, you manage a collection of values any way you want. These objects could be a finite set of class members, another array, or some complex data structure. Regardless of the internal implementation of the class, its data can be obtained consistently through the use of indexers. Here's an example.

|  |
| --- |
| Listing 11-1. An Example of An Indexer: IntIndexer.cs using System; class IntIndexer {     private string[] myData;      public IntIndexer(int size)     {         myData = new string[size];          for (int i=0; i < size; i++)         {             myData[i] = "empty";         }     }     public string this[int pos]     {         get        {             return myData[pos];         }         set        {             myData[pos] = value;         }     }     static void Main(string[] args)     {         int size = 10;         IntIndexer myInd = new IntIndexer(size);         myInd[9] = "Some Value";         myInd[3] = "Another Value";         myInd[5] = "Any Value";          Console.WriteLine("\nIndexer Output\n");         for (int i=0; i < size; i++)         {             Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);         }     } } |

**Working with Nullable Types**

Working with value types and data can sometimes be challenging because a value type doesn't normally hold a null value. This lesson shows you how to overcome this limitation with C# nullable types. Here's what you'll learn.

* Understand the problem that nullable types solve
* See how to declare a nullable type
* Learn how to use nullable types

**Understanding the Problem with Value Types and Null Values**

default value of a struct (value type) is some form of 0. This is another difference between reference types and value types. The default value of a reference type is *null*. If you're just writing C# code and managing your own data source, such as a file that holds data for your application, the default values for structs works fine.

In reality, most applications work with databases, which have their own type systems. The implications of working with database type systems is that you don't have a one-to-one mapping between C# and database types. One glaring difference is that database types can be set to *null*. A database has no knowledge of reference and value types, which are C# language (.NET Platform) concepts. This means that C# value type equivalents in the database, such as *int*, *decimal*, and *DateTime*, can be set to *null*.

Since a type in the database can be *null*, but your C# value type can't be *null*, you have to find some way to provide a translation in your C# code to account for *null* values. Effectively, the scheme you use will often be inconsistent from one program to another; something you often don't have a choice about. For example, what if you wanted to handle a *null* *DateTime* from SQL Server as the minimum *DateTime* value in C#. After that project, your next task would be to read data from a legacy Foxpro database, whose minimum *DateTime* value is different from SQL Server. Because of this lack of constency and potential confusion, C# 2.0 added nullable types, which are more elegant and natural for working with *null* data.

**Declaring Nullable Types**

To declare a value type as nullable, append a question mark, *?*, to the type name. Here's how to declare a *DateTime* variable as a nullable type:

DateTime? startDate;

A *DateTime* can't normally hold a *null* value, but the declaration above enables *startDate* to hold *null*, as well as any legal *DateTime* value. The proper terminology is to refer to the type of *startDate* as a nullable *DateTime*.

You can assign a normal value to *startDate* like this:

startDate = DateTime.Now;

or you can assign *null*, like this:

startDate = null;

Here's another example that declares and initializes a nullable *int*:

int? unitsInStock = 5;

The *unitsInStock* in the example above can be assigned a value of *null* also.

**Working with Nullable Types**

When you have nullable types, you'll want to check them to see if they're *null*. Here's an example that shows how you can check for a *null* value:

bool isNull = startDate == null;

Console.WriteLine("isNull: " + isNull);

The example above shows that you only need to use the equals operator to check for *null*. You could also make the equality check as part of an *if* statement, like this:

int availableUnits;

if (unitsInStock == null)

{

availableUnits = 0;

}

else

{

availableUnits = (int)unitsInStock;

}

**Note:** Notice the cast operator in the else clause above. An explicit conversion is required when assigning from nullable to non-nullable types.

That's several lines of code for something that appears to be such a common operation. Fortunately, there's a better way to perform the same task, using the coalesce operator, *??*, shown below:

int availableUnits = unitsInStock ?? 0;

The coalesce operator works like this: if the first value (left hand side) is *null*, then C# evaluates the second expression (right hand side).

**Extension Methods**

Extension methods are a new feature in C# 3.0. An extension method enables us to add methods to existing types without creating a new derived type, recompiling, or modify the original types. We can say that it extends the functionality of an existing type in .NET. An extension method is a static method to the existing static class. We call an extension method in the same general way; there is no difference in calling.

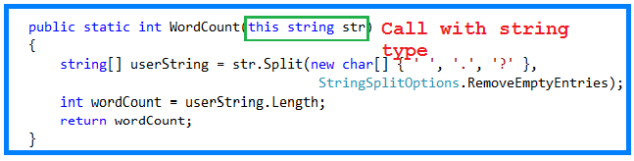
**Feature and Property of Extension Methods**

The following list contains basic features and properties of extension methods:

1. It is a static method.
2. It must be located in a static class.
3. It uses the "this" keyword as the first parameter with a type in .NET and this method will be called by a given type instance on the client side.
4. It also shown by VS intellisense. When we press the dot (.) after a type instance, then it comes in VS intellisense.
5. An extension method should be in the same namespace as it is used or you need to import the namespace of the class by a using statement.
6. You can give any name for the class that has an extension method but the class should be static.
7. If you want to add new methods to a type and you don't have the source code for it, then the solution is to use and implement extension methods of that type.
8. If you create extension methods that have the same signature methods as the type you are extending, then the extension methods will never be called.

**Using the Code**

We create an extension method for a string type so string will be specified as a parameter for this extension method and that method will be called by a string instance using the dot operator.



In the above method WordCount(), we are passing a string type with this so it will be called by the string type variable, in other words a string instance.

Now we create a static class and two static methods, one for the total word count in a string and another for the total number of characters in a string without a space.

|  |
| --- |
| using System;  namespace ExtensionMethodsExample  {  public static class Extension  {  public static int WordCount(this string str)  {  string[] userString = str.Split(new char[] { ' ', '.', '?' },  StringSplitOptions.RemoveEmptyEntries);  int wordCount = userString.Length;  return wordCount;  }  public static int TotalCharWithoutSpace(this string str)  {  int totalCharWithoutSpace = 0;  string[] userString = str.Split(' ');  foreach (string stringValue in userString)  {  totalCharWithoutSpace += stringValue.Length;  }  return totalCharWithoutSpace;  }  }  } |

Now we create an executable program that has a string as an input and uses an extension method to count the total words in that string and the total number of characters in that string then show the result in a console screen.

using System;

namespace ExtensionMethodsExample

{

class Program

{

static void Main(string[] args)

{

string userSentance = string.Empty;

int totalWords = 0;

int totalCharWithoutSpace = 0;

Console.WriteLine("Enter the your sentance");

userSentance = Console.ReadLine();

//calling Extension Method WordCount

totalWords = userSentance.WordCount();

Console.WriteLine("Total number of words is :"+ totalWords);

//calling Extension Method to count character

totalCharWithoutSpace = userSentance.TotalCharWithoutSpace();

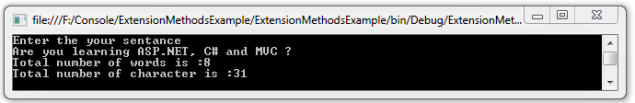
Console.WriteLine("Total number of character is :"+totalCharWithoutSpace);

Console.ReadKey();

}

}

}



# Object Initializers With C#

# Object Initializers With C# 3.0, initializing both objects and collections have become much easier. Consider this simple Car class, where we use the automatic properties described in a previous chapter:

# class Car

# {

# public string Name { get; set; }

# public Color Color { get; set; }

# }

# Now, in C# 2.0, we would have to write a piece of code like this to create a Car instance and set its properties:

# Car car = new Car();

# car.Name = "Chevrolet Corvette";

# car.Color = Color.Yellow;

# It's just fine really, but with C# 3.0, it can be done a bit more cleanly, thanks to the new object initializer syntax:

# Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow };

# As you can see, we use a set of curly brackets after instantiating a new Car object, and within them, we have access to all the public properties of the Car class. This saves a bit of typing, and a bit of space as well. The cool part is that it can be nested too. Consider the following example, where we add a new complex property to the Car class, like this:

# class Car

# {

# public string Name { get; set; }

# public Color Color { get; set; }

# public CarManufacturer Manufacturer { get; set; }

# }

# class CarManufacturer

# {

# public string Name { get; set; }

# public string Country { get; set; }

# }

# To initialize a new car with C# 2.0, we would have to do something like this:

# Car car = new Car();

# car.Name = "Corvette";

# car.Color = Color.Yellow;

# car.Manufacturer = new CarManufacturer();

# car.Manufacturer.Name = "Chevrolet";

# car.Manufacturer.Country = "USA";

# With C# 3.0, we can do it like this instead:

# Car car = new Car {

# Name = "Chevrolet Corvette",

# Color = Color.Yellow,

# Manufacturer = new CarManufacturer {

# Name = "Chevrolet",

# Country = "USA"

# }

# };

# Or in case you're not too worried about readability, like this:

# Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow, Manufacturer = new CarManufacturer { Name = "Chevrolet", Country = "USA" } };

Just like with the automatic properties, this is syntactical sugar - you can either use it, or just stick with the old, fashioned way of doing things.