**C#( C-sharp)**

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. The goal of the language is programmer productivity.

**Object Orientation**

C# is a rich implementation of the object-orientation paradigm, which includes encapsulation, inheritance, and polymorphism. The distinctive features of C# from an object oriented perspective are:

*Unified type system:*

The fundamental building block in C# is an encapsulated unit of data and functions called a type. C# has a unified type system, where all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic set of functionality. For example, any type can be converted to a string by calling its ToString method.

*Classes and interfaces:*

In the pure object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an interface. An interface is like a class except it is only a definition for a type, not an implementation. It’s particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++ and Eiffel, C# does not support multiple inheritance of classes).

Properties, methods, and events:

In the pure object-oriented paradigm, all functions are methods . C#, methods are only one kind of function member, which also includes properties and events. Properties are function members that encapsulate a piece of an object’s state, such as a button’s color or a label’s text. Events are function members that simplify acting on object state changes.

**Type Safety**

C# is primarily a type-safe language, meaning that types can interact only through protocols they define, thereby ensuring each type’s internal consistency. For instance, C# prevents you from interacting with a string type as though it were an integer type. If you assign a float type to a Boolean type, the compiler generates an error.

More specifically, C# supports static typing, meaning that the language enforces type safety at compile time. This is in addition to dynamic type safety, which the .NET CLR enforces at runtime. Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, since it knows for a given variable what type it is, and hence what methods you can call on that variable.

**Memory Management**

C# relies on the runtime to perform automatic memory management. The CLR has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating

the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++.

**Platform Support**

C# is typically used for writing code that runs on Windows platforms. Although Microsoft standardized the C# language and the CLR through ECMA, the total amount of resources dedicated to supporting C# on non-Windows platforms is relatively small. This means that languages such as Java are sensible choices when multiplatform support is of primary concern. Having said this, C# can be used to write cross-platform code in the following scenarios:

• C# code may run on the server and dish up DHTML that can run on any platform.

This is precisely the case for ASP.NET.

• C# code may run on a runtime other than the Microsoft Common Language

Runtime. The most notable example is the Mono project, which has its own

C# compiler and runtime, running on Linux, Solaris, Mac OS X, and Windows.

• C# code may run on a host that supports Microsoft Silverlight (supported for

Windows and Mac OS X). This is a new technology that is analogous to Adobe’s

Flash Player.

**Syntax**

C# syntax is based on C and C++ syntax. C# code is made up of a series of statements, each of which is terminated with a **semicolon.** C# is a **block-structured language**, meaning that all statements are part of a block of code. These blocks which are delimited with curly brackets(**{}**), may contain any number of statements, or none at all. C# code is **case-sensitive.**

**Identifiers**

Identifiers are names that programmers use for their classes, methods, variables, arrays and so on. An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore and subsequent characters may be letter, underscore or number. C# identifiers are case-sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g.,myVariable), and all other identifiers should be in Pascal case (e.g., MyMethod).

**Keywords**

**Keywords** are names reserved by the compiler that you can’t use as identifiers. These are the keywords in our example program: using, class, static, void, int, string and so on

**Variable**

A variable represents a storage location that has a modifiable value. To use variables, programmers have to declare them. This means that programmers have to assign them a name and a type. Once you have declared variables you can use them as storage units for the type of data that you declared them to hold.

Variable declaration

<accessibility\_level><type><variable\_name>;

Eg.int a=2; string myString;

**Constant**

A *constant* is a field that has fixed values and don't change during execution of program. A constant is evaluated statically at compile time and the compiler literally substitutes its value whenever used, rather like a macro in C++. A constant can be any of the built-in numeric types, bool, char,

string, or an enum type. A constant is declared with the **const** keyword and must be initialized with a value. **const** members are static by definition, there is no need to use the static modifier.

Eg. const int myInt=9;

**Data type**

The **type** is a representation of data. A type defines the blueprint for a value. A value is a storage location denoted by a variable or a constant.

Data Types in a programming language describes that what type of data a variable can hold. C sharp is a strongly typed language, therefore every variable and object must have a declared type. The CSharp data types are Value and Reference Type.

1. Value types
2. Reference types

When a value type is converted to object type, it is called boxing and on the other hand, when an object type is converted to a value type, it is called unboxing.

**Value Type**

Value type variables can be assigned a value directly. Value types are stored on stack. When the size of variable is bigger value type is not good. They are derived from the class System.ValueType.

The value types directly contain data. Some examples are int, char, and float.

**Reference Type**

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables. they refer to a memory location. They are stored in heap. When the size of variable is bigger reference type is good

Example of built-in reference types are: object, dynamic, and string.

object obj;

obj = 100; // this is boxing

**Dynamic Type**

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is:

dynamic <variable\_name> = value;

For example,

dynamic d = 20;

The predefined types in C# are:

*Value types*

• Numeric

— Signed integer (sbyte, short, int, long)

— Unsigned integer (byte, ushort, uint, ulong)

— Real number (float, double, decimal)

• Logical (bool)

• Character (char)

*Reference types*

• String (string)

• Object (object)

Predefined types in C# alias Framework types in the System namespace. There is only a syntactic difference between these two statements:

int i = 5; System.Int32 i =5

The set of predefined *value* types excluding decimal are known as *primitive types* in the CLR. Primitive types are so called because they are supported directly via instructions in compiled code, and this usually translates to direct support on the

underlying processor. For example:

**Boolean Type and Operators**

C#’s bool type (aliasing the System.Boolean type) is a logical value that can be assignedthe literal true or false.

Although a Boolean value requires only one bit of storage, the runtime will use onebyte of memory, since this is theminimum chunk that the runtime and processorcan efficiently work with. To avoid space inefficiency in the case of arrays, theFramework provides a BitArray class in the System.Collections namespace that isdesigned to use just one bit per Boolean value.

**Bool Conversions**

No conversions can be made from the bool type to numeric types or vice versa.

**Strings and Characters**

C#’s char type (aliasing the System.Char type) represents a Unicode character and occupies two bytes. A char literal is specified inside single quotes:

char c = 'A'; // Simple character

*Escape sequences* express characters that cannot be expressed or interpreted literally.

An escape sequence is a backslash followed by a character with a special meaning.

For example:

char newLine = '\n';

char backSlash = '\\';

**String Type**

C#’s string type (aliasing the System.String type) represents an immutable sequence of Unicode characters. String objects are immutable: they cannot be changed after they have been created. All of the string methods and C# operators that appear to modify a string actually return the results in a new string object. A string literal is specified inside double quotes:

string a = "Hello";

string is a reference type, rather than a value type. Its equality operators, however, follow value-type semantics:

string a = "test";

string b = "test";

Console.Write (a == b); // True

**String concatenation**

The + operator concatenates two strings:

string s = "a" + "b";

**Class**

A class is the most common kind of reference type. A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or members variables), operations that can be performed on data(methods) and accessors to data(properties). A class in C# is declared using the keyword **class** and its members are enclosed in parenthesis. A more complex class optionally has the following: **abstract, sealed, static and partial**

**class** MyClass

{

//fields, properties and operations

}

**Fields**

A field is a variable that is a member of a class and can hold data of the class. For example:

**class** Student

{

string name;

int age = 10;

}

**Properties**

Provide access to a class attribute (a field). Useful for exposing fields in components. A property is declared like a field, but with a **get/set** block added.

public **class** Student

{

string firstName;

public string FirstName

{

get{return firstName;}

set{firstName=value;}

}

}

**Methods**

Methods implement some action that can be performed by an object. Methods are the operations performed on the data.

A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type. A method can specify a void return type, indicating that it doesn’t return any value to its caller. A method can also output data back to the caller via ref/out parameters. A method’s signature must be unique within the type. A method’s signature comprises its name and parameter types (but not the parameter names, nor the return type).

**<accessibilityLevel><returnType><functionName>(paramType paramName, . . . . . . )**

{

Logic here

}

public int Sum(int a, int b)

{

return a+b;

}

**class** MyClass

{

public int Sum(int a, int b)

{

return a+b;

}

}

**Constructor:** A constructor is a method whose name is the same as the name of its class. It executes when object is created. It can initialize data member of new object. It has no return type.

In C#, default constructor is automatically created.

public class Student

{

stringfirstName;

public Student()

{

firstName=fName;

}

public Student(string fName)

{

firstName=fName;

}

}

Example of class with fields, properties and method(or function)

public **class** Student

{

string firstName;

string lastName;

public string FirstName

{

get{return firstName;}

set{firstName=value;}

}

public string LastName

{

get{return lastName;}

set{lastName =value;}

}

public string FullName()

{

return FirstName+” ”+LastName

}

}

**Objects**

An object is the concrete realization or instance built on the model specified by the class. An object is created in the memory using the keyword **new** and is referenced by an identifier called a “reference”.

Eg.

Student objStudent=new Student();

In above, we made an object of type Student which is referenced by an identifier objStudent. Objects are reference types. Also objects are created at the heap.

**Class Types**

**Partial classes**

Many developers need access to the same class, then having the class in multiple files can be beneficial. The partial keywords allow a class to span multiple source files.

There are some rules for defining a partial class as in the following;

* A partial type must have the same accessibility.
* Each partial type is preceded with the "partial" keyword.
* If the partial type is sealed or abstract then the entire class will be sealed and abstract.

Eg.

public partial class partialclassDemo  
{

}

**Static classes**  
A static class is declared using the "static" keyword. If the class is declared as static then the compiler never creates an instance of the class. All the member fields, properties and functions must be declared as static and they are accessed by the class name directly not by a class instance object.

public static class staticDemo  
{  
        //static fields  
        public static int a=10,b=15,sum;

        //static method  
       public static void Add()  
        {  
            sum =a+b;  
        }  
}

//function calling directly  
            staticDemo.Add();

**Abstract Classes**  
C# allows both classes and functions to be declared abstract using the **abstract** keyword. We can't create an instance of an abstract class. An abstract member has a signature but no function body and they must be overridden in any non-abstract derived class.//abstract class  
    public abstract class Employess  
    {  
        //abstract method with no implementation  
        public abstract void displayData();  
    }

    //derived class  
    public class test :Employess  
    {  
        //abstract class method implementation  
        public override void displayData()  
        {  
            Console.WriteLine("Abstract class method");  
        }  
    }  
**Sealed Classes**  
Sealed classes cannot be inherited. You can create an instance of a sealed class. A sealed class is used to prevent further refinement through inheritance.

sealed class SealedClass  
{  
        void myfunv();  
}

 public class test :SealedClass//wrong. will give compilation error  
{  
}

**Method types**

**static:** accessible through class name. not the object instance.

**virtual:**Method may be overridden in derived class.

**abstract:**Method must be overridden in non-abstract derived class(permitted only on abstract class).

**override:** Method overrides a base class method.

**String**

A C# string (== System.String) is an immutable (unchangeable) sequence of characters.

**Constructing strings:** The simplest way to construct a string is to assign a literal.

string s1 = "Hello";

**Null and empty strings:**An empty string has a length of zero. To create an empty string, we can useeither a literal or the static *string.Empty*field;

string empty = ""; and string str=string.Empty.

string nullString = null;

**Searching within strings**

The simplest methods for searching within strings are Contains, StartsWith, andEndsWith. These all return true or false:

Console.WriteLine ("quick brown fox".Contains ("brown")); // True

Console.WriteLine ("quick brown fox".EndsWith ("fox")); // True

IndexOf is more powerful: it returns the first position of a given character or substring(or −1 if the substring isn’t found):

Console.WriteLine ("abcde".IndexOf ("cd")); // 2

**String Manipulation**

Eg.

string myString==”This is dotnet lab”;

char[ ] separator={‘ ’};

string[ ] myWords=myString.Split(separator);

string takeSubString = myString.Substring(0,10);

stringmyString=”A string”;

char[ ] myChars=myString.ToCharArray();

stringconvertToLower=myString.ToLower();

int length=myString.Length;

**Arrays**

An array represents a fixed number of elements of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access. An array is denoted with square brackets after the element type. For example:

char[] vowels = new char[5]; // Declare an array of 5 characters

Square brackets also index the array, accessing a particular element by position:

vowels [0] = 'a';

vowels [1] = 'e';

vowels [2] = 'i';

vowels [3] = 'o';

vowels [4] = 'u';

Console.WriteLine (vowels [1]); // e

This prints “e” because array indexes start at 0. We can use a for loop statement toiterate through each element in the array. The for loop in this example cycles theinteger i from 0 to 4:

for (int i = 0; i <vowels.Length; i++)

Console.Write (vowels [i]); // aeiou

The Length property of an array returns the number of elements in the array. Oncean array has been created, its length cannot be changed. The System.Collectionnamespace and subnamespaces provide higher-level data structures, such as dynamicallysized arrays and dictionaries.

An array initialization expression specifies each element of an array. For example:

char[] vowels = new char[] {'a','e','i','o','u'};

or simply:

char[] vowels = {'a','e','i','o','u'};

All arrays inherit from the System.Array class, providing common services for all arrays. These members include methods to get and set elements regardless of the array type

**System Collection**

The System.Collections namespace in the .NET Framework Class Library is the primary source for *nongeneric collections*. These classes provide standard implementations of many of the data structures with collections that store references of type object. classes are ArrayList, Stack and Hashtable

The .NET Framework provides a standard set of types for storing and managing collections of objects. These include resizable lists, linked lists, and sorted and unsorted dictionaries, as well as arrays.

The types in the Framework for collections can be divided into the following categories:

• Interfaces that define standard collection protocols

• Ready-to-use collection classes (lists, dictionaries, etc.)

• Base classes for writing application-specific collections

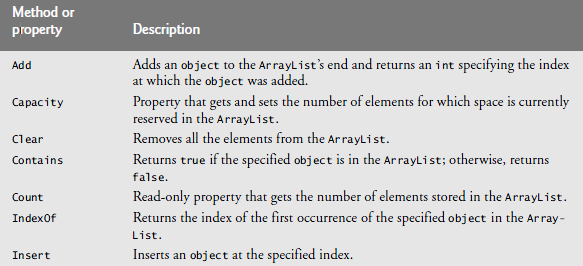
The collection namespaces are System.Collections, System.Collections.Generic etc.

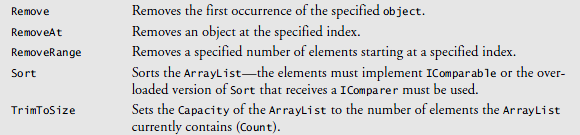
**Class ArrayList**

conventional arrays have a fixed size—they cannot be changed dynamically to conform to an app’s execution-time memory requirements. In some apps, this fixed-size limitation presents a problem. You must choose between using fixed-size arrays that are large enough to store the maximum number of elements the app may require and using dynamic data structures that can grow and shrink the amount of memory required to store data in response to the changing requirements of an app at execution time.

The .NET Framework’s ArrayList collection class mimics the functionality of conventional arrays and provides dynamic resizing of the collection through the class’smethods.ArrayLists store references to objects. All classes derive from class object, so an ArrayList can contain objects of any type.

Some useful methods and properties of class ArrayList.





**Add:** It appends a new element object to the end of the ArrayList.

ArrayList list = new ArrayList();

list.Add("One");

**AddRange:** This add one ArrayList to another. Internally, AddRange uses the Array.Copy or CopyTo methods , which have better performance than some loops.

ArrayList list = new ArrayList();

list.Add(5);

ArrayList list2 = new ArrayList();

list2.Add(10);

list.AddRange(list2); *// Add second ArrayList to first.*

**Count Property:** Gets the number of elements actually contained in the ArrayList. Eg. list.Count

**Clear:** Removes all elements from the ArrayList. Internally, this calls the Array.Clear method.

list.Clear();

ArrayList list = new ArrayList();

list.Add("Cat");

list.Add("Zebra");

list.Add("Dog");

**Insert:** Inserts an element into the ArrayList at the specified index. Eg. list.Insert(0, "Cow");

**Remove:** Removes the first occurrence of a specific object from the ArrayList.

list.Remove("Zebra ");

**RemoveAt:** Removes the element at the specified index of the ArrayList.

list.RemoveAt(1);

**RemoveRange**

list.RemoveRange(0, 2);

**Sort:** Sorts the elements in the entire ArrayList.

ArrayList list = new ArrayList();

list.Add("Cat");

list.Add("Zebra");

list.Add("Dog");

list.Sort();

**Reverse:**Reverses the order of the elements in the entire ArrayList.

list.Reverse();

**Contains:**Determines whether an element is in the ArrayList.

list.Contains("Cow ")

**IndexOf(object):** Searches for the specified object and returns the zero-based index of the first occurrence within the entire ArrayList.

list.IndexOf("Cow ")

**Enumeration**

The ability to traverse the contents of the collection is an almost universal need. The Framework supports this need via a

pair of interfaces (IEnumerable, IEnumerator, and their generic counterparts) that allow different data structures to expose a common traversal API. These are part of a larger set of collection interfaces.

**IEnumerator**

An IEnumerator object that can be used to iterate(enumerate) through the collection. Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection. The IEnumerator interface defines the basic low-level protocol by which elements in a collection are traversed—or enumerated—in a forward-only manner.

|  |  |
| --- | --- |
| Current | Gets the element in the collection at the current position of the enumerator. |
| MoveNext() | Advances the enumerator to the next element of the collection. |
| Reset() | Sets the enumerator to its initial position, which is before the first element in the collection. |

**IEnumerable**

IEnumerable is an interface that defines one method GetEnumerator which returns an IEnumerator interface, it allows readonly access to a collection. A collection that implements IEnumerable can be used with a **foreach** statement. GetEnumerator() methods returns an enumerator that iterates through a collection.

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

**Implementing the Enumeration Interfaces**

We might want to implement IEnumerable or IEnumerable<T>for one or more of the following reasons:

• To support the foreach statement

• To interoperate with anything expecting a standard collection

• As part of implementing a more sophisticated collection interface

• To support collection initializers

To implement IEnumerable/IEnumerable<T>, you must provide an enumerator. You can do this in one of three ways:

• If the class is “wrapping” another collection, by returning the wrapped collection’s enumerator

• Via an iterator using yield return

• By instantiating your own IEnumerator/IEnumerator<T>implementation

**The ICollection and IList Interfaces**

Although the enumeration interfaces provide a protocol for forward-only iteration over a collection, they don’t provide a mechanism to determine the size of the collection, access a member by index, search, or modify the collection. For such functionality,

the .NET Framework defines the ICollection, IList, and IDictionary interfaces. Each comes in both generic and nongeneric versions; however, the nongeneric versions exist mostly for legacy.

**Summary**

IEnumerable<T>*(and* IEnumerable*)*

Provides minimum functionality (enumeration only)

ICollection<T>*(and* ICollection*)*

Provides medium functionality (e.g., the Count property)

IList<T>*/*IDictionary<K,V>*and their nongeneric versions*

Provide maximum functionality (including “random” access by index/key)

**Delegates**

A delegate is a type that enables us to store references to functions. Delegates are declared much like functions, but with no function body and using the delegate keyword. The delegate declaration specifies a function signature consisting of a return type and the parameter list. A delegate dynamically wires up a method caller to its target method. There are two aspects to a delegate: *type* and *instance*. A *delegate type* defines a *protocol* to which the caller and target will conform, comprising a list of parameter types and a return type. A *delegate instance* is an object that refers to one (or more) target methods conforming to that protocol.

Eg.

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

public int Substract(int a, int b)

{

return a - b;

}

public delegate int MyDelegate(int p, int q);

public int Operation(string opName, int p, int q)

{

MyDelegate obj;

if (opName == "Add")

obj = new MyDelegate(Add);

else if (opName == "Substract")

obj = new MyDelegate(Substract);

return obj(p, q);

}

}

Calculator c=new Calculator();

c. Operation(“Add”,5,3);

**Event**

An event is a mechanism via which a class can notify its clients when something happens. For example when you click a button, a button-click-event notification is sent system. Events are declared using delegates. Events are certain actions that happen during execution of program that the application wishes to be notified about, so it can respond. Event can be mouse click.

Two emergent roles commonly appear: *broadcaster* and *subscriber*.

The broadcaster is a type that contains a delegate field. The broadcaster decides when to broadcast, by invoking the delegate.

The subscribers are the method target recipients. A subscriber decides when to start and stop listening, by calling += and −= on the broadcaster’s delegate. A subscriber does not know about, or interfere with, other subscribers. Events are a language feature that formalizes this pattern. An event is a construct that exposes just the subset of delegate features required for the broadcaster/ subscriber model. The main purpose of events is to prevent subscribers from interfering with each other.

Eg.

public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);

public class Stock

{

string symbol;

decimal price;

public Stock (string symbol) { this.symbol = symbol; }

**public event PriceChangedHandlerPriceChanged;**

public decimal Price

{

get { return price; }

set

{

if (price == value) return; // Exit if nothing has changed

if (PriceChanged != null) // If invocation list not empty,

**PriceChanged (price, value);** // fire event.

price = value;

}

}

}

**Indexers**

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but areaccessed via an index argument rather than a property name.

class Sentence

{

string[] words =new string[5];

public string this [intwordNum] // indexer

{

get { return words [wordNum]; }

set { words [wordNum] = value; }

}

}

Here’s how we could use this indexer:

Sentence s = new Sentence();

s[3] = "kangaroo";

Console.WriteLine (s[3]);

**Versioning**

Versioning is the process of evolving a component over time in a compatible manner. A new version of a component is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. In contrast, a new version of a component is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Most languages do not support binary compatibility at all, and many do little to facilitate source compatibility. In fact, some languages contain flaws that make it impossible, in general, to evolve a class over time without breaking at least some client code.

The most basic approach to Version is the 3 component format MAJOR.MINOR.PATCH, where:

**MAJOR** version is incremented when incompatible API changes. There is a significant code change that might be incompatible with previous versions, such as a fundamental change of framework.

**MINOR** version is incremented when significant bug fixes are implemented, or a new feature is added.

**RRVISION (PATCH)** is incremented when minor bug fixes are implemented.

When versions are identified numerically, the numbers are assigned in increasing order.

For example, version **1.3.22** would be major version 1, minor version 3 and revision 22

Versioning in C# through the use of the **override** and **new** keywords. Versioning maintains compatibility between base and derived classes as they evolve.

public class MyBase

{

public virtual string Meth1()

{

return "MyBase-Meth1";

}

public virtual string Meth2()

{

return "MyBase-Meth2";

}

public virtual string Meth3()

{

return "MyBase-Meth3";

}

}

class MyDerived :MyBase

{

// Overrides the virtual method Meth1 using the override keyword:

public override string Meth1()

{

return "MyDerived-Meth1";

}

// Explicitly hide the virtual method Meth2 using the new keyword:

public new string Meth2()

{

return "MyDerived-Meth2";

}

// Because no keyword is specified in the following declaration a warning will be issued to alert the programmer that

// the method hides the inherited member MyBase.Meth3():

public string Meth3()

{

return "MyDerived-Meth3";

}

public static void Main()

{

MyDerived mD = new MyDerived();

MyBase mB = (MyBase) mD;

System.Console.WriteLine(mB.Meth1());

System.Console.WriteLine(mB.Meth2());

System.Console.WriteLine(mB.Meth3());

}

}

Output

MyDerived-Meth1

MyBase-Meth1

MyBase-Meth2

**I/O**

In the .NET Framework, the System.IO namespace defines classes for reading and writing files and data streams . It contains types that provide basic file and directory support. The System.IO namespace, which resides in the mscorlib.dll assembly, provides classes for working with the system I/O and with streams.

**Namespaces**

A namespace is a domain within which type names must be unique. Types are typically organized into hierarchical namespaces—both to avoid naming conflicts and to make type names easier to find. A Namespace is simply a logical collection of related classes in c#. Collectively, .NET’s namespaces are referred to as the **.NET Framework Class Library**.

Namespaces are independent of assemblies, which are units of deployment such as an *.exe* or *.dll.*

Namespaces also have no impact on member visibility—public, internal, private, and so on.

Eg.

namepspace SchoolManagement

{

public class Student

{

}

}

SchoolManagement.Studentobjs=new SchoolManagement.Student ();

**Windows Form**

We can create a userinterface by dragging and dropping controls from a toolbox on windows form

Windows Forms is a rich client API that’s as old as the .NET Framework. Compared to WPF, Windows Forms is a relatively simple technology that provides most of the features you need in writing a typical Windows application. It also has significant relevancy in maintaining legacy applications. It has a number of drawbacks, though, compared to WPF:

• Controls are positioned and sized in pixels, making it easy to write applications that break on clients whose DPI settings differ from the developer’s.

• The API for drawing nonstandard controls is GDI+, which, although reasonably flexible, is slow in rendering large areas (and without double buffering, flickers horribly.

• Controls lack true transparency.

• Dynamic layout is difficult to get right reliably.

The last point is an excellent reason to favor WPF over Windows Forms—even ifyou’re writing a business application that needs just a user interface and not a “userexperience.” The layout elements in WPF, such as Grid, make it easy to assemblelabels and text boxes such that they always align—even after language changinglocalization—without messy logic and without any flickering. Further, you don’thave to bow to the lowest common denominator in screen resolution—WPF layoutelements have been designed from the outset to adapt properly to resizing.On the subject of speed, it was originally thought that graphics card manufacturerswould incorporate GDI+ hardware accelerators. This never happened; their focuswas instead on DirectX. Consequently, GDI+ is considerably slower than even itspredecessor, GDI, let alone WPF.On the positive side, Windows Forms is relatively simple to learn and has a wealthof support in third-party controls.

The Windows Forms types are in the namespaces System.Windows.Forms (inSystem.Windows.Forms.dll) and System.Drawing (in System.Drawing.dll). The latteralso contains the GDI+ types for drawing custom controls.

Eg.

