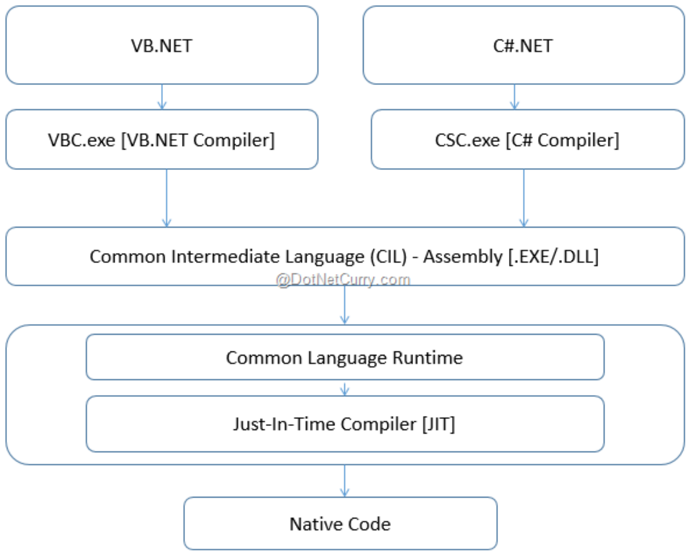
What is .NET?

The **.NET is the technology** from Microsoft, on which all other Microsoft technologies will be depending on in future. It is a major technology change, introduced by Microsoft, to catch the market from the SUN's Java. Few years back, Microsoft had only VC++ and VB to compete with Java, but Java was catching the market very fast. With the world depending more and more on the Internet/ Web and java related tools becoming the best choice for the web applications, Microsoft seemed to be loosing the battle. Thousands of programmers moved to java from VC++ and VB. To recover the .market, .Microsoft announced .NET.

- It is a platform neutral framework.
- It is a layer between the operating system and the programming language.
- It supports many programming languages, including VB.NET, C# etc.
- .NET provides a common set of class libraries, which can be accessed from any .NET based programming language. There will not be separate set of classes and libraries for each language. If you know any one .NET language, you can write code in any .NET language!!
- In future versions of Windows, .NET will be freely distributed as part of operating system and users will never have to install .NET separately.
- .NET is not an operating system.
- .NET is not a programming language.

".NET is a framework"

- We cannot define .NET as a 'single thing'.
- It is a new, easy, and extensive programming platform.
- It is not a programming language, but it supports several programming languages.
- By default .NET comes with few programming languages including C# (C Sharp), VB.NET, J# and managed C++.
- .NET is a common platform for all the supported languages. It gives a common class library, which can be called from any of the supported languages.
- So, developers need not learn many libraries when they switch to a different language. Only the syntax is different for each language.
- When you write code in any language and compile, it will be converted to an 'Intermediate Language' (Microsoft Intermediate Language – MSIL).
- So, your compiled executable contains the IL and not really executable machine language.
- When the .NET application runs, the .NET framework in the target computer take care of the execution. (To run a .NET application, the target computer should have .NET framework installed.)
- The .NET framework converts the calls to .NET class libraries to the corresponding APIs of the Operating system.
- Whether you write code in C# or VB.NET, you are calling methods in the same .NET class libraries.
- The same .NET framework executes the C# and VB.NET applications.
- So, there won't be any performance difference based on the language you write code.


VB.NET
C#.NET
VBC.exe [VB.NET Compiler]
CSC.exe [C# Compiler]
Common Intermediate Language (CIL) - Assembly [.EXE/.DLL]
@DotNetCurry.com
Common Language Runtime
Just-In-Time Compiler [JIT]
Native Code

Is it platform independent?

- The code you write is platform independent, because whatever you write is getting compiled into MSIL.

- There is no native code, which depends on your operating system or CPU. But when you execute the MSIL, the .NET framework in the target system will convert the MSIL into native platform code.
- So, if you run your .NET exe in a Windows machine, the .NET framework for Windows will convert it into Windows native code and execute.
- If you run your .NET application in Unix or Linux, the .NET framework for Unix/Linux will convert your code into Unix/Linux native code and execute.
- So, your code is purely platform independent and runs anywhere!
- But wait, we said it wrong… there is no .NET framework for UNIX or Linux available now. Microsoft has written the .NET framework only for Windows.
- If you or some one else write a .NET framework for other platforms in future, your code will run there too. So, let us wait until someone write .NET framework for Linux before you run your .NET code in Linux.

Just in Time Compiler(JIT)

Machines cannot run MSIL directly. JIT compiler turns MSIL into native code, which is CPU specific code that runs on the same computer architecture as the JIT compiler. Because the common. Language runtime supplies a JIT compiler for each supported CPU architecture, developers can write a set of MSIL that can be JIT-compiled and run on computers with different architectures.

Common Language Infrastructure (CLI)

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft that describes the executable code and runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.

The common language Infrastructure (CLI) is a theoretical model of a development platform that provides a device and language independent way to express data and behavior of applications.

The CLI specification describes the following four aspects: –

The Common Type System (CTS)

The language interoperability and .NET Class Framework are not possible without all the language sharing the same data type. CTS is an important part of the runtime support for cross-language integration. The CTS performs the following functions:

- Establishes a framework that enables cross-language integration, type safety and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.

Common Language Specification (CLS)

A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.

Advantages of the .NET Framework

- Consistent programming model
- Multi-platform applications
- Multi-Language integration
- Automatic Resource Management
- Ease of deployment

MEANING OF PUBLIC STATIC VOID MAIN (STRING [] ARGS)

V

OOPS & C#

The skeleton of object – oriented programming is of course the concepts of class. The C# on OOPS explains classes and their importance in implementation of object oriented principles.

Any language can be called object oriented if it has data and method that use data encapsulated in items named objects. An object oriented programming method has many advantages; some of them are flexibility and code reusability.

Key Concepts of Object Orientation

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction is the ability to generalize an object as a data type that has a specific set of characteristics and is able to perform a set of actions. Object-oriented languages provide abstraction via classes. Classes define the properties and methods of an object type.

Examples:
You can create an abstraction of a dog with characteristics, such as color, height, and weight, and actions such as run and bite. The characteristics are called properties, and the actions are called methods.
A Recordset object is an abstract representation of a set of data.
Classes are blueprints for Object.
Objects are instance of classes.

Class and Objects

Classes

A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behavior of a type. If the class is not declared as static, client code can use it by creating objects or instances which are assigned to a variable. The variable remains in memory until all references to it go out of scope. At that time, the CLR marks it as eligible for garbage collection. If the class is declared as static, then only one copy exists in memory and client code can only access it through the class itself, not an instance variable.

Declaring Class

```
public class Customer

{

   //Fields, properties, methods and events go here…

}
```

The class keyword is preceded by the access level. Because public is used in this case, anyone can create objects from this class. The name of the class follows the class keyword.

Objects

An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection.

Creating Objects

A class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

Customer object1 = new Customer();

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, object1 is a reference to an object that is based on Customer.

Constructors

Whenever a class or struct is created, its constructor is called. A class or struct may have multiple constructors that take different arguments.

Constructors allow the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

- Constructor is used to initialize an object (instance) of a class.
- Constructor is a like a method without any return type.
- Constructor has same name as class name.
- Constructor follows the access scope (Can be private, protected, public, Internal and external).
- Constructor can be overloaded.

Constructors generally following types:

- Default Constructor
- Parameterized constructor
- Private Constructor
- Static Constructor
- Copy Constructor

Default Constructor

A constructor that takes no parameters is called a default constructor.

When a class is initiated default constructor is called which provides default values to different data members of the class.

You need not to define default constructor it is implicitly defined.

Example: –

class Program

```
{
    class C1
    {
        int a, b;
        public C1()
        {
            this.a = 10;
            this.b = 20;
        }
        public void display()
        {
            Console.WriteLine(“Value of a: {0}”, a);
            Console.WriteLine(“Value of b: {0}”, b);
        }
    }
    static void Main(string[] args)
    {
        C1 ob1 = new C1();
        ob1.display();
        Console.ReadLine();
    }
}
```

Output: – Value of a: 10

```
Value of b: 20
```

Parameterized constructor

Constructor that accepts arguments is known as parameterized constructor. There may be situations, where it is necessary to initialize various data members of different objects with different values when they are created. Parameterized constructors help in doing that task.

```
class Program

  {

      class C1

      {

          int a, b;

          public C1(int x, int y)

          {

              this.a = x;

              this.b = y;

          }

          public void display()

          {

              Console.WriteLine(“Value of a: {0}”, a);

              Console.WriteLine(“Value of b: {0}”, b);

          }

      }

      static void Main(string[] args)

      { // Here when you create instance of the class

  //  parameterized constructor will be called

 C1 ob1 = new C1(10,20);

          ob1.display();

          Console.ReadLine();

      }
```

```
   }
```

Output: – Value of a: 10

```
             Value of b: 20
```

Private Constructor

Private constructors are used to restrict the instantiation of object using ‘new’ operator. A private constructor is a special instance constructor. It is commonly used in classes that contain static members only.

- If you don’t want the class to be inherited we declare its constructor private.
- We can’t initialize the class outside the class or the instance of class can’t be created outside if its constructor is declared private.
- We have to take help of nested class (Inner Class) or **static** method to initialize a class having private constructor.

Example: –

```
   class Program

   {

       class C1

       {

           int a, b;

           public C1(int x, int y)

           {

               this.a = x;

               this.b = y;

}

           public static C1 create_instance()

           { return new C1(12, 20); }

           public void display()

           {
               Console.WriteLine(“Value of a: {0}”, a);

               Console.WriteLine(“Value of b: {0}”, b);

               int z = a + b;

               Console.WriteLine(z);

           }

       }

       static void Main(string[] args)

       { // Here the class is initiated using a static method of the

class than only you can use private constructor

C1 ob1 = C1.create_instance();

           ob1.display();

           Console.ReadLine();

       }

   }
```

Static Constructors

C# supports two types of constructor, a class constructor static constructor and an instance constructor (non-static constructor).

Static constructors might be convenient, but they are slow. The runtime is not smart enough to optimize them in the same way it can optimize inline assignments. Non-static constructors are inline and are faster.

Static constructors are used to initializing class static data members.

Point to be remembered while creating static constructor:

1. There can be only one static constructor in the class.

2. The static constructor should be without parameters.

3. It can only access the static members of the class.

4. There should be no access modifier in static constructor definition.

Static members are preloaded in the memory. While instance members are post loaded into memory.

Static methods can only use static data members.

Example:

```
class Program

   {

       public class test

       {

           static string name;

           static int age;

           static test()

           {

 Console.WriteLine(“Using static constructor to initialize

                 static data members”);

               name = “John Sena”;

               age = 23;

           }

           public static void display()

           {

               Console.WriteLine(“Using static function”);

               Console.WriteLine(name);

               Console.WriteLine(age);

           }

       }

       static void Main(string[] args)

       {
           test.display();

           Console.ReadLine();

       }}
```

Output:

```
Using static constructor to initialize static data members

Using static function

John Sena

23
```

Copy Constructor

If you create a new object and want to copy the values from an existing object, you use copy constructor.

This constructor takes a single argument: a reference to the object to be copied.

Example:

```
class Program

    {

        class c1

        {

            int a, b;

            public c1(int x, int y)

            {

                this.a = x;

                this.b = y;

            }

            // Copy construtor

            public c1(c1 a)

            {
                this.a = a.a;

                this.b = a.b;

            }

            public void display()

            {

                int z = a + b;

                Console.WriteLine(z);

            }

        }

        static void Main(string[] args)

        {

            c1 ob1 = new c1(10, 20);

            ob1.display();

      // Here we are using copy constructor. Copy constructor is

using the values already defined with ob1

            c1 ob2 = new c1(ob1);

            ob2.display();

            Console.ReadLine();

        }

    }
```

Output:

```
30

30
```

Destructors

The .NET framework has an in built mechanism called Garbage Collection to de-allocate memory occupied by the un-used objects. The destructor implements the statements to be executed during the garbage collection process. A destructor is a function with the same name as the name of the class but starting with the character ~.

Example:

```
class Complex

{

public Complex()

{

// constructor

}

~Complex()

{

// Destructor

}

}
```

- Remember that a destructor can’t have any modifiers like private, public etc. If we declare a destructor with a modifier, the compiler will show an error.
- Also destructor will come in only one form, without any arguments.
- There is no parameterized destructor in C#.

Destructors are invoked automatically and can’t be invoked explicitly. An object becomes eligible for garbage collection, when it is no longer used by the active part of the program. Execution of destructor may occur at any time after the instance or object becomes eligible for destruction.

7. Object References

When we work with an object we are using a reference to that object. On the other hand, when we are working with simple data types such as Integer, we are working with the actual value rather than a reference.

When we create a new object using the New keyword, we store a reference to that object in a variable. For instance:

Draw MyDraw = new Draw;

This code creates a new instance of Draw. We gain access to this new object via the MyDraw variable. This variable holds a reference to the object.

Now we have a second variable, which also has a reference to that same object. We can use either variable interchangeably, since they both reference the exact same object. The thing we need to remember is that the variable we have is not the object itself but, rather, is just a reference or pointer to the object itself.

Early binding means that our code directly interacts with the object, by directly calling its methods. Since the compiler knows the object’s data type ahead of time, it can directly compile code to invoke the methods on the object. Early binding also allows the IDE to use IntelliSense to aid our development efforts; it allows the compiler to ensure that we are referencing methods that do exist and that we are providing the proper parameter values.

Late binding means that our code interacts with an object dynamically at run-time. This provides a great deal of flexibility since our code literally doesn’t care what type of object it is interacting with as long as the object supports the methods we want to call. Because the type of the object isn’t known by the IDE or compiler, neither IntelliSense nor compile-time syntax checking is possible but we get unprecedented flexibility in exchange.
If we enable strict type checking by using Option Strict On at the top of our code modules, then the IDE and compiler will enforce early binding behavior. By default, Option Strict is turned off and so we have easy access to the use of late binding within our code.

8. Access Modifiers
Access Modifiers are keywords used to specify the declared accessibility of a member of a type.

- **Public** is visible to everyone. A public member can be accessed using an instance of a class, by a class’s internal code, and by any descendants of a class.
- **Private** is hidden and usable only by the class itself. No code using a class instance can access a private member directly and neither can a descendant class.
- **Protected** members are similar to private ones in that they are accessible only by the containing class. However, protected members also may be used by a descendant class. So members that are likely to be needed by a descendant class should be marked protected.
- **Internal/Friend** is public to the entire application but private to any outside applications. Internal is useful when you want to allow a class to be used by other applications but reserve special functionality for the application that contains the class. Internal is used by C# and Friend by VB .NET.
- **Protected Internal** may be accessed only by a descendant class that’s contained in the same application as its base class. You use protected internal in situations where you want to deny access to parts of a class functionality to any descendant classes found in other applications.

9. Inheritance is the idea that one class, called a subclass, can be based on another class, called a base class. Inheritance provides a mechanism for creating hierarchies of objects.

Inheritance is an important object-oriented concept. It allows you to build a hierarchy of related classes, and to reuse functionality defined in existing classes.

Inheritance is the ability to apply another class's interface and code to your own class.

Normal base classes may be instantiated themselves, or inherited. Derived classes can inherit base class members marked with protected or greater access. The derived class is specialized to provide more functionality, in addition to what its base class provides. Inheriting base class members in derived class is not mandatory.
C# supports two types of Inheritance mechanisms: –
1) Implementation Inheritance
2) Interface Inheritance

What is Implementation Inheritance?
– When a class (type) is derived from another class(type) such that it inherits all the members of the base type it is Implementation Inheritance

What is Interface Inheritance?
– When a type (class or a struct) inherits only the signatures of the functions from another type it is Interface Inheritance
In general, Classes can be derived from another class, hence support Implementation inheritance At the same time Classes can also be derived from one or more interfaces Hence they support Interface inheritance Structs can derive from one more interface, hence support Interface Inheritance Structs cannot be derived from another class they are always derived from SystemValueType

Types of Inheritance
1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance (Implementation is possible through **Interface**)
4. Hierarchical Inheritance

<u>Multilevel Inheritance</u>
Class B
Class B
<u>Single Inheritance</u>
Class A
Class D
Class C
Class A
<u>Hierarchical Inheritance</u>
Class A
Class B
Class C
Class A
Class B
Class C
<u>Multiple Inheritance</u>

Example: –
Single Inheritance: – **Multilevel Inheritance**: – **Hierarchical Inheritance**: –
public class A public class A public class A

```
{   }
public class B : A
{   }
```

```
{   }
public class B : A
{   }
public class C : B
{   }
```

```
{   }
public class B : A
{   }
public class C : A
{   }
public class D : A
{   }
```

Multiple Inheritance: –

```
public class A
{   }
public class B
{   }
public class C : A, B
{   }
```

10. Polymorphism

Polymorphism is the ability to define a method or property in a set of derived classes with matching method signatures but provide different implementations and then distinguish the objects' matching interface from one another at runtime when you call the method on the base class.

It is a feature to use one name in many forms. It can be achieved in following ways: –

- Method Overloading
- Method Overriding
- Method Hiding

Method overriding and hiding makes use of the following three method keywords –

1. new
2. virtual
3. override

1. When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. To change the data and behavior of a base class, you have two choices: you can replace the base member with a **new** derived member, or you can override a virtual base member.

Replacing a member of a base class with a new derived member requires the **new** keyword. If a base class defines a method, field, or property, the new keyword is used to create a **new** definition of that method, field, or property on a derived class. The **new** keyword is placed before the return type of a class member that is being replaced. For example:

```
public class BaseClass
{
   public void DoWork() { }
   public int WorkField;
   public int WorkProperty
   {
      get { return 0; }
   }
}
public class DerivedClass : BaseClass
{
   public new void DoWork() { }
   public new int WorkField;
   public new int WorkProperty
   {
        get { return 0; }
    }
}
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.
BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

In order for an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as **virtual**. This is accomplished by adding the **virtual** keyword before the return type of the member. A derived class then has the option of using the **override** keyword, instead of new, to replace the base class implementation with its own.

For example:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.
BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Remarks about Virtual

- When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

- By default, methods are non-virtual. You cannot override a non-virtual method.

- You cannot use the virtual modifier with the static, abstract, private or override modifiers.

- Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the virtual modifier on a static property.

- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

Remarks about Override

- The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

- An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.

- You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

- An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

- You cannot use the modifiers new, static, virtual, or abstract to modify an override method.

- An overriding property declaration must specify the exact same access modifier, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

Namespace:

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code.

In Microsoft .Net, Namespace is like containers of objects. They may contain unions, classes, structures, interfaces, enumerators and delegates. Main goal of using namespace in .Net is for creating a hierarchical organization of program. In this case, you need not to worry about the **naming conflicts** of classes, functions, variables etc., inside a project.

In Microsoft .Net, every program is created with a default namespace. This default namespace is called as global namespace. But the program itself can declare any number of namespaces, each of them with a unique name. The advantage is that every namespace can contain any number of classes, functions, variables and also namespaces etc., whose names are unique only inside the namespace. The members with the same name can be created in some other namespace without any compiler complaints from Microsoft .Net.

To declare namespace C# .Net has a reserved keyword namespace. If a new project is created in Visual Studio .NET it automatically adds some global namespaces. These namespaces can be different in different projects. But each of them should be placed under the base namespace System. The names space must be added and used through the using operator, if used in a different project.

A namespace has the following properties:

- They organize large code projects.
- They are delimited with the **.** operator.
- The using directive means you do not need to specify the name of the namespace for every class.
- The global namespace is the “root” namespace: global**::**system will always refer to the .NET Framework namespace System.

Now have a look at the example of declaring some namespace:

```
namespace SampleNamespace

{

   class SampleClass{}

   interface SampleInterface{}

   struct SampleStruct{}

   enum SampleEnum{a,b}

   delegate void SampleDelegate(int i);

   namespace SampleNamespace.Nested

   {

      class SampleClass2{}

   }

}
```

Within a namespace, you can declare one or more of the following types:

- another namespace
- class
- interface
- struct
- enum
- delegate

Interface

An Interface is a reference type and it contains only abstract members. Interface’s members can be Events, Methods, Properties and Indexers. But the interface contains only declaration for its members. Any implementation must be placed in class that realizes them. The interface can not contain constants, data fields, constructors, destructors and static members. All the member declarations inside interface are implicitly public and they cannot include any access modifiers.

An interface has the following properties:

- An interface is like an abstract base class: any non-abstract type that implements the interface must implement all its members.
- An interface cannot be instantiated directly.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- Classes and structs can implement more than one interface.
- An interface itself can inherit from multiple interfaces.

```
interface IPoint

{

   int x

   { get; set; }

   int y

   { get; set; }

}

namespace Ex_Interface

{

   class MyPoint:IPoint
```

```
{
    private int myX;
    private int myY;
    public MyPoint(int x, int y)
    {
        myX= x;
        myY=y;
    }
    public int x
    {
        get
        {
            return myX;
        }
        set
        {
            myX=value;
        }
    }
    public int y
    {
        get
        {
            return myY;
        }
```

```
            set
            {
                myY=value;
            }
        }
    }
    class Program
    {
        private static void PrintPoint(IPoint P)
        {
            Console.WriteLine(“x={0}, y={1}”,P.x,P.y);
        }
        static void Main(string[] args)
        {
            MyPoint P = new MyPoint(2, 3);
            Console.Write(“My Point::”);
            PrintPoint(P);
            Console.Read();
        }
    }
}
```

Exception Handling

Exceptions are error conditions that arise when the normal flow of a code path-that is, a series of method calls on the call stack-is impractical. Exception handling is an in built mechanism in .NET framework to detect and handle run time errors. The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) do not provide a mechanism to handle these anomalies, the .NET run time environment provide a default mechanism, which terminates the program execution. C# provides three keywords try, catch and finally to do exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean up process.

The general form try-catch-finally in C# is shown below:

```
try

{

// Statement which can cause an exception.

}

catch(Type x)

{

// Statements for handling the exception

}

finally

{

//Any cleanup code

}
```

If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.

But in C#, both catch and finally blocks are optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always. Note that it is an error to transfer control out of a finally block by using break, continue, return or goto.

In C#, exceptions are nothing but objects of the type Exception. The Exception is the ultimate base class for any exceptions in C#. The C# itself provides couple of standard exceptions. Or even the user can create their own exception classes, provided that this should inherit from either Exception class or one of the standard derived classes of Exception class like DivideByZeroExcpetion ot ArgumentException etc.

The modified form of the above program with exception handling mechanism is as follows: –

```
//C#: Exception Handling
```

```
using System;

class MyClient

{

public static void Main()

{

int x = 0; int div = 0;

try

{           div = 100/x;

Console.WriteLine("Not executed line");

}

catch(DivideByZeroException de)

{ Console.WriteLine("Exception occured"); }

finally

{ Console.WriteLine("Finally Block"); }

Console.WriteLine("Result is {0}",div);

}

}
```

Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

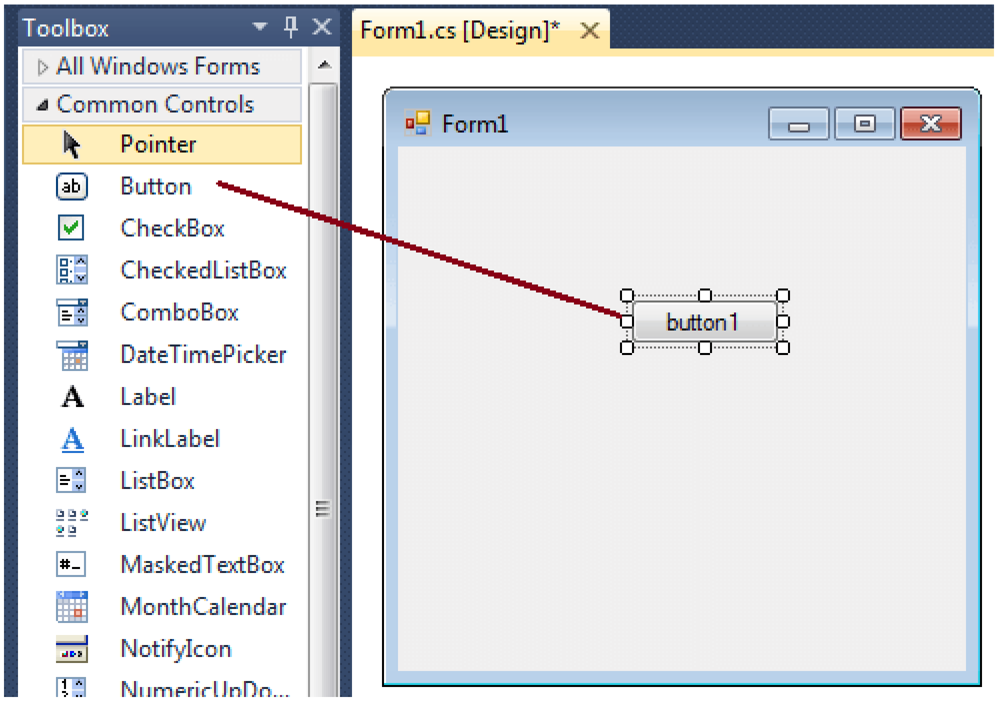
```
//C#: Exception Handling: Multiple catch

using System;

class MyClient

{

public static void Main()
{
int x = 0;
int div = 0;
try
{
div = 100/x;
Console.WriteLine("Not executed line");
}
catch(DivideByZeroException de)
{ Console.WriteLine("DivideByZeroException" ); }
catch(Exception ee)
{ Console.WriteLine("Exception" ); }
finally
{ Console.WriteLine("Finally Block"); }
Console.WriteLine("Result is {0}",div);
}
}
```

Exception Handling Example:

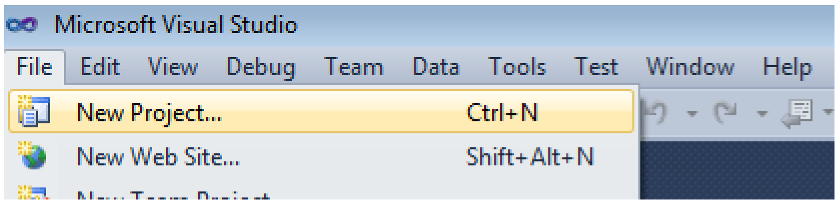
```
using System;
class MyClient
{
public static void Main()
{
try
{       throw new DivideByZeroException("Invalid Division");}
catch(DivideByZeroException e)

{ Console.WriteLine("Exception" ); }

Console.WriteLine("LAST STATEMENT");

}

}
```

C# Windows Forms

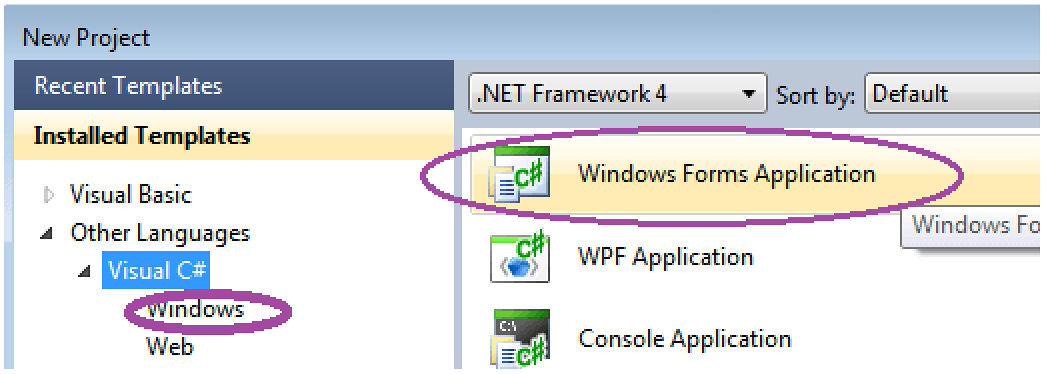
C# programmers have made extensive use of forms to build user interfaces. Each time you create a Windows application, Visual Studio will display a default blank form, onto which you can drag the controls onto your applications main form and adjust their size and position.



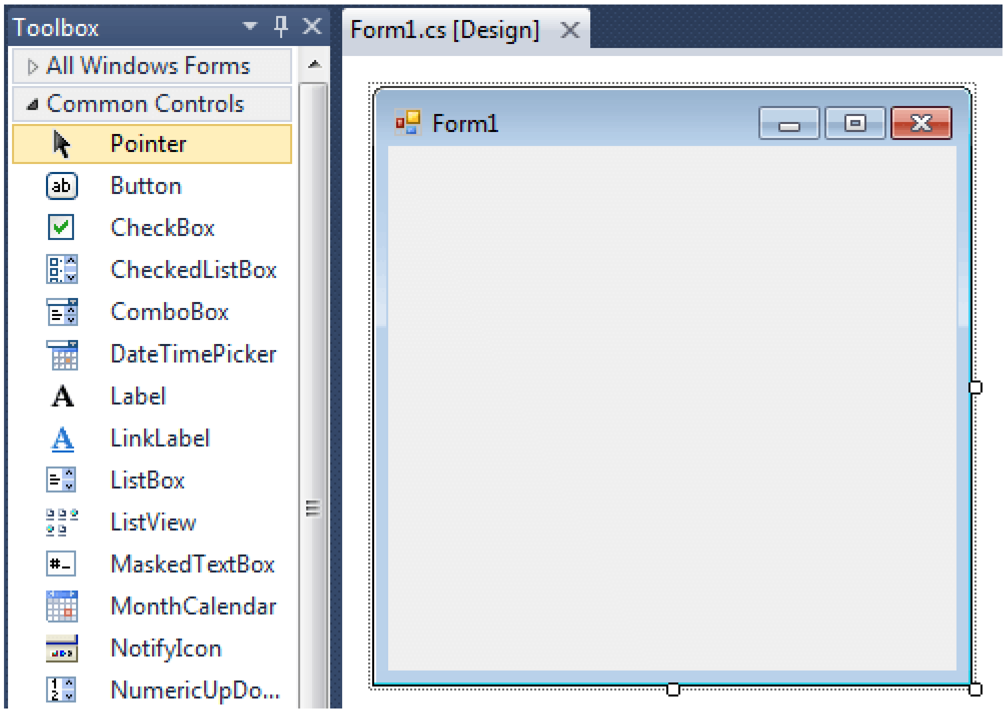
The first step is to start a new project and build a form. Open your Visual Studio and select File->New Project and from the new project dialog box select Other Languages->Visual C# and select Windows Forms Application. Enter a project name at the bottom of the dialouge box and click OK button. The following picture shows how to create a new Form in Visual Studio.


Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
New Project... Ctrl+N
New Web Site... Shift+Alt+N

Select Windows Forms Application from New Project dialog box.

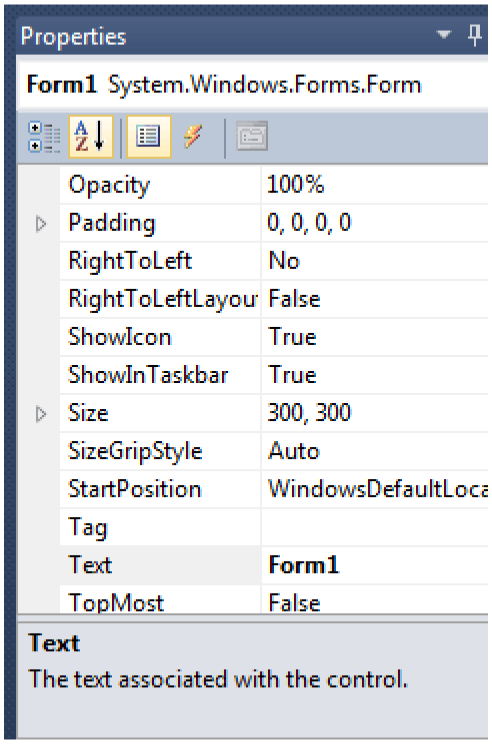


After selecting Windows Forms Application , you can see a default Form (Form1) in your new C# project. The Windows Form you see in Designer view is a visual representation of the window that will open when your application is opened. You can switch between this view and Code view at any time by right-clicking the design surface or code window and then clicking View Code or View Designer. The following picture shows how is the default Form (Form1) looks like.



At the top of the form there is a title bar which displays the forms title. Form1 is the default name, and you can change the name to your convenience . The title bar also includes the control box, which holds the minimize, maximize, and close buttons.

If you want to set any properties of the Form, you can use Visual Studio Property window to change it. If you do not see the Properties window, on the View menu, click Properties window. This window lists the properties of the currently selected Windows Form or control, and its here that you can change the existing values.



Opacity	100%
Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayou	False
ShowIcon	True
ShowInTaskbar	True
Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLoca
Tag	
Text	**Form1**
TopMost	False

Text
The text associated with the control.

CONTROLS/TOOLS

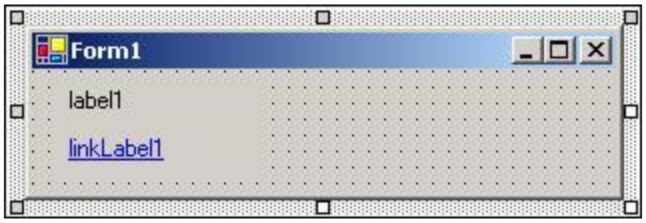
LABEL

The Label control is probably the most used control of them all. Look at any Windows application and you'll see them on just about any dialog you can find. The label is a simple control with one purpose only: to present a caption or short hint to explain something on the form to the user.

Out of the box, Visual Studio.NET includes two label controls that are able to present them selves to the user in two distinct ways:

- Label, the standard Windows label
- LinkLabel, a label like the standard one (and derived from it), but presents itself as an internet link (a hyperlink)

The two controls are found at the top of the control panel on the Window Forms tab. In the picture below, one of each of the two types of Label have been dragged to a to illustrate the difference in appearance between the two:



If you have experience with Visual Basic you may notice that the Text property is used to set the text that is displayed, rather than the Caption property. You will find that all intrinsic .NET controls use the name Text to describe the main text for a control. Before .NET, Caption and Text were used interchangeably.

And that's it **for most uses of the Label control. Usually you need to add no event handling code for a standard Label**. In the case of the LinkLabel, however, some extra code is needed if you want to allow the user to click it and take him or her to the web page shown in the text.

TextBox

Text boxes should be used when you want the user to enter text that you have no knowledge of at design time (for example the name of the user). The primary function of a text box is for the user to enter text, but any characters can be entered, and it is quite possible to force the user to enter numeric values only.

Out of the box .NET comes with two basic controls to take text input from the user: TextBox and RichTextBox (we'll discuss RichTextBox later in this chapter). Both controls are derived from a base class called TextBoxBase which itself is derived from Control.

TextBoxBase provides the base functionality for text manipulation in a text box, such as selecting text, cutting to and pasting from the Clipboard, and a wide range of events. We'll not focus so much now on what is derived from where, but instead look at the simpler of the two controls first – TextBox. We'll build one example that demonstrates the TextBox properties and build on that to demonstrate the RichTextBox control later.

The RadioButton and CheckBox Controls

As mentioned earlier, the RadioButton and CheckBox controls share their base class with the button control, though their appearance and use differs substantially from the button.

Radio buttons traditionally displays themselves as a label with a dot to the left of it, which can be either selected or not. You should use the radio buttons when you want to give the user a choice between several mutually exclusive options. An example of this could be, if you want to ask for the gender of the user.

To group radio boxes together so that they create one logical unit you must use a GroupBox control. By first placing a group box on a form, and then placing the RadioButton controls you need within the borders of the group box, the RadioButton controls will know to change their state to reflect that only one within the group box can be selected. If you do not place them within a group box, only one RadioButton on the form can be selected at any given time.

A CheckBox traditionally displays itself as a label with a small box with a checkmark to the left of it. You should use the check box when you want to allow the user to choose one or more options. An example could be a questionnaire asking which operating systems the user has tried (for example, Windows 95, Windows 98, Linux, Max OS X, and so on.)

<u>The GroupBox Control</u>

Before we move on to the example, we'll look at the group box control. This control is often used in conjunction with the RadioButton and CheckBox controls to display a frame around, and a caption above, a series of controls that are logically linked in some way.

Using the group box is as simple as dragging it onto a form, and then dragging the controls it should contain onto it (but not the other way round – you can't lay a group box over some pre-exisiting controls). The effect of this is that the parent of the controls becomes the group box, rather than the form, and it is therefore possible to have more than one RadioButton selected at any given time. Within the group box, however, only one RadioButton can be selected.

The relationship between parent and child probably need to be explained a bit more. When a control is placed on a form, the form is said to become the parent of the control, and hence the control is the child of the form. When you place a GroupBox on a form, it becomes a child of a form. As a group box can itself contain controls, it becomes the parent of these controls. The effect of this is that moving the GroupBox will move all of the controls placed on it.

Another effect of placing controls on a group box, is that it allows you to change certain properties of all the controls it contains simply by setting the corresponding property on the group box. For instance, if you want to disable all the controls within a group box, you can simply set the Enabled property of the group box to false.

<u>The ListBox and CheckedListBox Controls</u>

List boxes are used to show a list of strings from which one or more can be selected at a time. Just like check boxes and radio buttons, the list box provides a means of asking the user to make one or more selections. You should use a list box when at design time you don't know the actual number of values the user can choose from (an example could be a list of co-workers). Even if you know all the possible values at design time, you should consider using a list box if there are a great number of values.

The ListBox class is derived from the ListControl class, which provides the basic functionality for the two list-type controls that come out-of-the-box with Visual Studio.NET. The other control, the ComboBox, is discussed later in this chapter.

Another kind of list box is provided with Visual Studio.NET. This is called CheckedListBox and is derived from the ListBox class. It provides a list just like the ListBox, but in addition to the text strings it provide a check for each item in the list.