INTRODUCTION TO .NET FRAMEWORK

NET Framework

.NET Framework is a complete environment that allows developers to develop, run, and deploy the following applications:

- Console applications
- Windows Forms applications
- Windows Presentation Foundation (WPF) applications
- Web applications (ASP.NET applications)
- Web services
- Windows services
- Service-oriented applications using Windows Communication Foundation (WCF)
- Workflow-enabled applications using Windows Workflow Foundation (WF)

.NET Framework also enables a developer to create sharable components to be used in distributed computing architecture. NET Framework supports the object-oriented programming model for multiple languages, such as Visual Basic, Visual C#, and Visual C++. .NET Framework supports multiple programming languages in a manner that allows language interoperability. This implies that each language can use the code written in some other language.

The main components of .NET Framework?

The following are the key components of .NET Framework:

- .NET Framework Class Library
- Common Language Runtime
- Dynamic Language Runtimes (DLR)
- Application Domains
- Runtime Host
- Common Type System
- Metadata and Self-Describing Components
- Cross-Language Interoperability
- .NET Framework Security
- Profiling
- Side-by-Side Execution

Microsoft Intermediate Language (MSIL)

The .NET Framework is shipped with compilers of all .NET programming languages to develop programs. Each .NET compiler produces an intermediate code after compiling the source code.

The intermediate code is common for all languages and is understandable only to .NET environment. This intermediate code is known as MSIL.

IL

Intermediate Language is also known as MSIL (Microsoft Intermediate Language) or CIL (Common Intermediate Language). All .NET source code is compiled to IL. IL is then converted to machine code at the point where the software is installed, or at run-time by a Just-In-Time (JIT) compiler.

The roles of CLR in .NET Framework.

CLR provides an environment to execute .NET applications on target machines. CLR is also a common runtime environment for all .NET .

CLR also provides various services to execute processes, such as memory management service and security services. CLR performs various tasks to manage the execution process of .NET applications.

The responsibilities of CLR are listed as follows:

- Automatic memory management
- Garbage Collection
- Code Access Security
- Code verification
- JIT compilation of .NET code

Differentiate between managed and unmanaged code

Managed code is the code that is executed directly by the CLR instead of the operating system. The code compiler first compiles the managed code to intermediate language (IL) code, also called as MSIL code. This code doesn't depend on machine configurations and can be executed on different machines.

Unmanaged code is the code that is executed directly by the operating system outside the CLR environment. It is directly compiled to native machine code which depends on the machine configuration.
In the managed code, since the execution of the code is governed by CLR, the runtime provides different services, such as garbage collection, type checking, exception handling, and security support. In the unmanaged code, the allocation of memory, type safety, and security is required to be taken care of by the developer. If the unmanaged code is not properly handled, it may result in memory leak. Examples of unmanaged code are ActiveX components and Win32 APIs that execute beyond the scope of native CLR.

The JIT compiler in .NET Framework

The JIT compiler is an important element of CLR, which loads MSIL on target machines for execution. The MSIL is stored in .NET assemblies after the developer has compiled the code written in any .NET-compliant programming language, such as Visual Basic and C#.

OBJECTS AND TYPES

Properties Overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set accessor is used to assign a new value.
- The value keyword is used to define the value being assigned by the set accessor.
- Properties that do not implement a set accessor are read only.
- For simple properties that require no custom accessor code, consider the option of using auto-implemented properties.

Accessing Class Fields With Properties

```
using System;
public class Customer
{
    private int m_id = -1;

    public int ID
    {
        get
        {
            return m_id;
        }
        set
        {
            m_id = value;
        }
    }

public class CustomerManagerWithProperties
{
    public static void Main()
    {
        Customer cust = new Customer();

        cust.ID = 1;

        Console.WriteLine(
            "ID: {0}",cust.ID);

        Console.ReadKey();
    }
}
```

Read-Only Properties

```
using System;
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();
    }
}
```

Write-Only Properties

```
using System;
public class Customer
{
    private int m_id = -1;

    public int ID
    {
        set
        {
            m_id = value;
        }
    }
    public void DisplayCustomerData()
    {
        Console.WriteLine("ID: {0}", m_id);
    }
}

public class WriteOnlyCustomerManager
{
    public static void Main()
    {
        Customer cust = new Customer();
        cust.ID = 1;
        cust.DisplayCustomerData();
        Console.ReadKey();
    }
}
```

Auto-Implemented Properties

```
using System;
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();
    }
}
```

readonly modifier

The readonly keyword is a modifier that you can use on fields. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class.

```
    class Age
    {
        readonly int _year;
        Age(int year)
        {
            _year = year;
        }
        void ChangeYear()
        {
            //_year = 1967; // Compile error if uncommented.
        }
    }
```

When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

static modifier

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}



class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object:
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information:
        Console.WriteLine("Name: {0}", e.name);
        Console.WriteLine("ID:   {0}", e.id);
        Console.WriteLine("New Number of Employees: {0}",
                      Employee4.employeeCounter);
    }
}
```

const keyword

You use the const keyword to declare a constant field or a constant local.

```
const int x = 0;
public const double gravitationalConstant = 6.673e-11;
private const string productName = "Visual C#";
```

The static modifier is not allowed in a constant declaration.
The readonly keyword differs from the const keyword. **A const field can only be initialized at the declaration of the field. A readonly field can be initialized either at the declaration or in a constructor.** Therefore, readonly fields can have different values depending on the constructor used. Also, although a const field is a compile-time constant, the readonly field can be used for run-time constants, as in this line: **public static readonly uint l1 = (uint)DateTime.Now.Ticks;**

Static constructor

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file.
- To split a class definition, use the partial keyword modifier, as shown here:

In the following example, the fields and the constructor of the class, CoOrds, are declared in one partial class definition, and the member, PrintCoOrds, is declared in another partial class definition.

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}
class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Partial Methods

A partial class may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

Static Class

A **static** class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated.
For example, in the .NET Framework Class Library, the static System.Math class contains methods that perform mathematical operations.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
```

Inner Classes (Nested Classes)

In C# an inner (nested) class is called a class that is declared inside the body of another class. Accordingly, the class that encloses the inner class is called an outer class.

The main reason to declare one class into another are:

1. To better organize the code when working with objects in the real world, among which have a special relationship and one cannot exist without the other.
2. To hide a class in another class, so that the inner class cannot be used outside the class wrapped it.

```
public class OuterClass
{
	private string name;

	private OuterClass(string name)
	{
		this.name = name;
	}

	private class NestedClass
	{
		private string name;
		private OuterClass parent;

		public NestedClass(OuterClass parent, string name)
		{
			this.parent = parent;
			this.name = name;
		}

		public void PrintNames()
		{
```

```
                        Console.WriteLine("Nested name: " + this.name);
                        Console.WriteLine("Outer name: " + this.parent.name);
                }
        }

        static void Main()
        {
                OuterClass outerClass = new OuterClass("outer");
                NestedClass nestedClass = new
                        OuterClass.NestedClass(outerClass, "nested");
                nestedClass.PrintNames();
        }
}
```

Inheritance

Classes can inherit from another class.

```
public class A
{
    public A() { }
}

public class B : A
{
    public B() { }
}
public class Employee
{
    public int salary;

    public Employee(int annualSalary)
    {
        salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        salary = weeklySalary * numberOfWeeks;
    }
}
```

This class can be created using either of the following statements:

C#
```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the **base** keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

virtual and override

The **virtual** keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}


class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}
```

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the Main method, declare variables bc, dc, and bcdc.

- bc is of type BaseClass, and its value is of type BaseClass.
- dc is of type DerivedClass, and its value is of type DerivedClass.
- bcdc is of type BaseClass, and its value is of type DerivedClass. This is the variable to pay attention to.

Because bc and bcdc have type BaseClass, they can only directly access Method1, unless you use casting. Variable dc can access both Method1 and Method2. These relationships are shown in the following code.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Next, add the following Method2 method to BaseClass. The signature of this method matches the signature of the Method2 method in DerivedClass.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Because BaseClass now has a Method2 method, a second calling statement can be added for BaseClass variables bc and bcdc, as shown in the following code.

C#
```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

When you build the project, you see that the addition of the Method2 method in BaseClass causes a warning. The warning says that the Method2 method in DerivedClass hides the Method2 method in BaseClass. You are advised to use the new keyword in the Method2 definition if you intend to cause that result. Alternatively, you could rename one of the Method2 methods to resolve the warning, but that is not always practical.

Before adding new, run the program to see the output produced by the additional calling statements. The following results are displayed.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

The new keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type BaseClass continue to access the members of BaseClass, and the variable that has type DerivedClass continues to access members in DerivedClass first, and then to consider members inherited from BaseClass.

To suppress the warning, add the new modifier to the definition of Method2 in DerivedClass, as shown in the following code. The modifier can be added before or after public.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using new, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance.

To contrast this behavior to the effects of using override, add the following method to DerivedClass. The override modifier can be added before or after public.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the virtual modifier to the definition of Method1 in BaseClass. The virtual modifier can be added before or after public.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

C#
```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the override modifier enables bcdc to access the Method1 method that is defined in DerivedClass. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using override to extend the base class method.

Assemblies

Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

Access modifier

The access modifiers, public, protected, internal, or private, is used to specify one of the following declared accessibility levels for members.

Declared accessibility	Meaning
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protected internal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.

The internal keyword is an access modifier for types and type members. Internal types or members are accessible only within files in the same assembly, as in this example:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

Types or members that have access modifier protected internal can be accessed from the current assembly or from types that are derived from the containing class.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

This example contains two files, Assembly1.cs and Assembly1_a.cs. The first file contains an internal base class, BaseClass. In the second file, an attempt to instantiate BaseClass will produce an error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // CS0122
   }
}
```

In the folloing example, use the same files you used in example 1, and change the accessibility level of BaseClass to public. Also change the accessibility level of the member IntM to internal. In this case, you can instantiate the class, but you cannot access the internal member.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}


// Assembly2_a.cs
// Compile with: /reference:Assembly1.dll
public class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters.

In the following example, a generic class is defined and provided with simple get and set accessor methods as a means of assigning and retrieving values. The Program class creates an instance of this class for storing strings.

```
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer, which will allow client code
   // to use [] notation on the class instance itself.
```

```
    // (See line 2 of code in Main below.)
    public T this[int i]
    {
        get
        {
            // This indexer is very simple, and just returns or sets
            // the corresponding element from the internal array.
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}

// This class shows how client code uses the indexer.
class Program
{
    static void Main(string[] args)
    {
        // Declare an instance of the SampleCollection type.
        SampleCollection<string> stringCollection = new SampleCollection<string>();

        // Use [] notation on the type.
        stringCollection[0] = "Hello, World";
        System.Console.WriteLine(stringCollection[0]);
    }
}
```