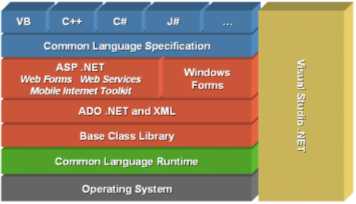
A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services.

The .NET Framework contains different parts:



**Components and their Description**

(1) **Common Language Runtime or CLR:** It performs memory management, exception handling, debugging, security checking, thread execution, code execution, code safety, verification, and compilation. The code that is directly managed by the CLR is called the managed code. When the managed code is compiled, the compiler converts the source code into a CPU independent intermediate language (IL) code. A Just In- Time (JIT) compiler compiles the IL code into native code, which is CPU specific.

(2) **.Net Framework Class Library:** It contains a huge library of reusable types such as classes, interfaces, structures, and enumerated values, which are collectively called types.

(3) **Common Language Specification:** It contains the specifications for the .Net supported languages and implementation of language integration.

(4) **Common Type System:** It provides guidelines for declaring, using, and managing types at runtime, and cross-language communication.

(5) **Metadata and Assemblies:** Metadata is the binary information describing the program, which is either stored in a portable executable file (PE) or in the memory. Assembly is a logical unit consisting of the assembly manifest, type metadata, IL code, and a set of resources like image files.

(6) **Windows Forms:** Windows Forms contain the graphical representation of any window displayed in the application.

(7) **ASP.NET and ASP.NET AJAX:** ASP.NET is the web development model and AJAX is an extension of ASP.NET for developing and implementing AJAX functionality. ASP.NET AJAX contains the components that allow the developer to update data on a website without a complete reload of the page.

(8) **ADO.NET:** It is the technology used for working with data and databases. It provides access to data sources like SQL server, OLE DB, XML etc. The ADO.NET allows connection to data sources for retrieving, manipulating, and updating data.

(9) **Windows Workflow Foundation (WF):** It helps in building workflow-based applications in Windows. It contains activities, workflow runtime, workflow designer, and a rules engine.

(10) **Windows Presentation Foundation:** It provides a separation between the user interface and the business logic. It helps in developing visually stunning interfaces using documents, media, two and three dimensional graphics, animations, and more.

(11) **Windows Communication Foundation (WCF):** It is the technology used for building and executing connected systems.

(12) **Windows CardSpace:** It provides safety for accessing resources and sharing personal information on the internet.

(13) **LINQ:** It imparts data querying capabilities to .Net languages using a syntax which is similar to the tradition query language SQL.

ASP.NET compared with classic ASP

ASP.NET Web Forms simplifies developers' transition from Windows application development to Web development by offering the ability to build pages composed of controls similar to a Windows user interface. A Web control, such as a button or label, functions in very much the same way as its Windows counterparts: code can assign its properties and respond to its events. Controls know how to render themselves: whereas Windows controls draw themselves to the screen, Web controls produce segments of HTML and JavaScript which form parts of the resulting page sent to the end-user's browser.

ASP.NET Web Forms encourages the programmer to develop applications using an event-driven GUI model, rather than in conventional Web-scripting environments like ASP and PHP. The framework combines existing technologies such as JavaScript with internal components like "View-State" to bring persistent (inter-request) state to the inherently stateless Web environment.

**Other differences compared to classic ASP are:**

Compiled code means applications run faster with more design-time errors trapped at the development stage.

Significantly improved run-time error handling, making use of exception handling using try-catch blocks.

Similar metaphors to Microsoft Windows applications such as controls and events.

An extensive set of controls and class libraries, as well as user-defined controls, allow the rapid building of applications. Layout of these controls on a page is easier because most of it can be done visually in most editors.

ASP.NET uses the multi-language abilities of the .NET Common Language Runtime, allowing Web pages to be coded in VB.NET, C#, J#, Delphi.NET etc.

Ability to cache the whole page or just parts of it to improve performance.

Ability to use the code-behind development model to separate business logic from presentation.

Ability to use true object-oriented design for programming pages and controls

If an ASP.NET application leaks memory, the ASP.NET runtime unloads the App-Domain hosting the erring application and reloads the application in a new App-Domain.

Session state in ASP.NET can be saved in a Microsoft SQL Server database or in a separate process running on the same machine as the Web server or on a different machine. That way session values are not lost when the Web server is reset or the ASP.NET worker process is recycled.

Versions of ASP.NET prior to 2.0 were criticized for their lack of standards compliance. The generated HTML and JavaScript sent to the client browser would not always validate against W3C/ECMA standards. In addition, the framework's browser detection feature sometimes incorrectly identified Web browsers other than Microsoft's own Internet Explorer as "down-level" and returned HTML/JavaScript to these clients with some of the features removed, or sometimes crippled or broken. In version 2.0 however, all controls generate valid HTML 4.0, XHTML 1.0 (the default) or XHTML 1.1 output, depending on the site configuration. Detection of standards-compliant Web browsers is more robust and support for Cascading Style Sheets is more extensive.

Web Server Controls: these are controls introduced by ASP.NET Web-Forms for providing the UI for the Web form. These controls are state managed controls and are [WYSIWYG](https://en.wikipedia.org/wiki/WYSIWYG) controls.

**C # Tutorial**

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg. This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.

**Sample c# program (first console program):**

**using System;**

**namespace HelloWorldApplication**

**{**

**class HelloWorld**

**{**

**static void Main(string[] args)**

**{**

**/\* my first program in C# \*/**

**Console.WriteLine("Hello World");**

**Console.ReadKey();**

**}**

**}**

**}**

**Array declaration**

**Initializing array**:

double[] balance = new double[10];

balance[0] = 4500.0;

**sample array program:**

**using System;**

**namespace ArrayApplication**

**{**

**class MyArray**

**{**

**static void Main(string[] args)**

**{**

**int [] n = new int[10]; /\* n is an array of 10 integers \*/**

**int j;**

**/\* output each array element's value \*/**

**for (j = 0; j < 10; j++ )**

**{**

**Console.WriteLine("Element[{0}] = {1}", j, n[j]);**

**}**

**Console.ReadKey();**

**}**

**}**

**}**

**C# Class**

**using System;**

**namespace BoxApplication**

**{**

**class Box**

**{**

**private double length; // Length of a box**

**private double breadth; // Breadth of a box**

**private double height; // Height of a box**

**public void setLength( double len )**

**{**

**length = len;**

**}**

**public void setBreadth( double bre )**

**{**

**breadth = bre;**

**}**

**public void setHeight( double hei )**

**{**

**height = hei;**

**}**

**public double getVolume()**

**{**

**return length \* breadth \* height;**

**}**

**}**

**class Boxtester**

**{**

**static void Main(string[] args)**

**{**

**Box Box1 = new Box(); // Declare Box1 of type Box**

**Box Box2 = new Box();**

**double volume;**

**// Declare Box2 of type Box**

**// box 1 specification**

**Box1.setLength(6.0);**

**Box1.setBreadth(7.0);**

**Box1.setHeight(5.0);**

**// box 2 specification**

**Box2.setLength(12.0);**

**Box2.setBreadth(13.0);**

**Box2.setHeight(10.0);**

**// volume of box 1**

**volume = Box1.getVolume();**

**Console.WriteLine("Volume of Box1 : {0}" ,volume);**

**// volume of box 2**

**volume = Box2.getVolume();**

**Console.WriteLine("Volume of Box2 : {0}", volume);**

**Console.ReadKey();**

**}**

**}**

**}**

**C # constructor**

**using System;**

**namespace LineApplication**

**{**

**class Line**

**{**

**private double length; // Length of a line**

**public Line()**

**{**

**Console.WriteLine("Object is being created");**

**}**

**public void setLength( double len )**

**{**

**length = len;**

**}**

**public double getLength()**

**{**

**return length;**

**}**

**static void Main(string[] args)**

**{**

**Line line = new Line();**

**// set line length**

**line.setLength(6.0);**

**Console.WriteLine("Length of line : {0}", line.getLength());**

**Console.ReadKey();**

**}**

**}**

**}**

**Overloading of Constructors and Methods**

It is possible to have more than one method with the same name and return type but with different number and types of arguments (parameters). This is called method overloading. For example, in the following class MathUtil, we have defined two methods with the name plus. Both of them have same return type but the first one has two parameters whereas the second one has 3 parameters. When plus() is called, the compiler will decide, on the basis of the types and number of parameters being passed, which one of these two to actually call, for example, plus(2,3) calls the first one and plus(2,3,4) calls the second one.

Class MathUtil

{

public static int plus(int n1, int n2)

{

return(n1+n2);

}

public static int plus(int n1, int n2, int n3)

{

return (n1 + n2 +n3);

}

Static void Main()

{

plus(2,3);

plus(2,3,4);

}

}

Methods are overloaded depending on the parameter list and not on the return type. The WriteLine() method of Console class in the System namespace has 19 different overloaded forms.

Similarly, constructors can be overload. For example

public class Employee

{ CSC-360 Net Centric Computing Prepared by: Nav Raj Vinady, Patan Mulitple Campus, Lalitpur 15

public int salary;

public Employee(int annualSalary)

{

salary = annualSalary;

}

public Employee(int weeklySalary, int numberOfWeeks)

{

salary = weeklySalary \* numberOfWeeks;

}

}

If we create an object like: Employee emp1=new Employee(1000);

the first constructor will be called initializing the salary to 1000. If we create an object like

Employee emp2=new Employee(1000,5);

the second constructor will be called which takes two arguments to calculate the salary.

Overloading methods and constructors gives program a lot of flexibility and reduces a lot of complexity that would otherwise be produced if we had to use different name for these methods.

**Inheritance:**

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class. The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

**using System;**

**namespace InheritanceApplicati**

**{**

**class Shape**

**{**

**public void setWidth(int w)**

**{**

**width = w;**

**}**

**public void setHeight(int h)**

**{**

**height = h;**

**}**

**protected int width;**

**protected int height;**

**}**

**// Derived class**

**class Rectangle: Shape**

**{**

**public int getArea()**

**{**

**return (width \* height);**

**}**

**}**

**class RectangleTester**

**{**

**static void Main(string[] args)**

**{**

**Rectangle Rect = new Rectangle();**

**Rect.setWidth(5);**

**Rect.setHeight(7);**

**// Print the area of the object.**

**Console.WriteLine("Total area: {0}", Rect.getArea());**

**Console.ReadKey();**

**}**

**}**

**}**

**Sealed Class**

Sealed classes are used to prevent a class from being used as a base class. It is primarily useful to prevent unintended derivations.

sealed class TestClass

{

public TestClass()

{

}

}

class NewClass: TestClass //fails

{

}

**Sealed Method**

A class member, method, field, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the **sealed** keyword before the override keyword in the class member declaration. For example:

Public class C { public virtual void DoWork(){}; }

public class D : C { public sealed override void DoWork() { } }

**Abstract Class**

When we want to define a base class but do not want to instantiate it, then we define abstract class using the keyword abstract. Other classes can be derived from abstract classes. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract method does not have to contain abstract members, but if a class contains an abstract member, the class itself must be declared as abstract. Derived classes of the abstract class must implement all abstract methods. Abstract classes are useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. They also version well, because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

public abstract Class A

{

public abstract void DoWork(int i);

}

When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

public abstract Class D

{

public virtual void DoWork(int i)

{

// original implementation

}

}

public abstract class E:D

{

public abstract override void DoWork(int i);

}

public class F:E

{

public override void DoWork(int i)

{

//New implementation

}

}

Polymorphism has two distinct aspects:

1>At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.

2>Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

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. To override the virtual base member, the **override** keyword is used.

public class ClassA

{

public void myMethod()

{

System.Console.WriteLine("Parent class method");

}

}

public class ClassB : ClassA

{

public void myMethod()

{

System.Console.WriteLine("Child Class Method");

}

}

public class MyClass

{

static void Main()

{

ClassB b = new ClassB();

b.myMethod(); //output: child class method

ClassA b1 = new ClassB();

b1.myMethod(); //output: parent class method

}}

**Overriding the methods - Virtual and Override Keywords**

public class classA

{

public virtual void Method()

{

System.Console.WriteLine("ClassA Method");

}

}

public class classB : classA

{

public override void Method()

{

System.Console.WriteLine("ClassB Method");

}

}

public class MyClass

{

static void Main()

{

classA a = new classB();

a.Method(); //output: classB Method

}

}

As we have overridden the method of classA in classB and since the method is marked virtual in classA, the compiler will no longer see the apparent (or reference) type to call the static early or compile time object binding rather it will apply dynamic, late or runtime object binding and will see the object type at the runtime to decide which method it should call. This procedure is called polymorphism, where we have different implementations of a method with the same name and signature in the base-class and sub-classes.

When such a method is called using a base-type-reference, the compiler uses the actual object type reference by the base type reference to decide which of the method to call.

By marking *virtual*, we allow a method to be overridden and be used polymorphically. In the same way, we make the method in the sub-class as override when it is overriding the virtual method in the base class.

**Interfaces**

Interfaces are a special kind of type in c# used to define the specifications that should be followed by its sub-types. An interface can be defined using the *interface* keyword.

An interface has the following properties:

1. An interface is like an abstract class which can not be instantiated.

2. Any class implementing the interface must implement all of its members.

3. Interface can contain signatures for methods, properties, events.

4. Interface contain no implementation of methods

5. Classes can implement more than one interface

6. An interface itself can inherit from multiple interfaces.

7. Interface can not contain constants, fields, instance constructors, destructors or types

8. All members are public and abstract by default

Interfaces provide a way to achieve runtime polymorphism.

interface ISampleInterface

{

void SampleMethod();

}

class TestClass : ISampleInterface

{

public void SampleMethod()

{

System.Console.WriteLine("Interface method implementation");

}

static void Main()

{

ISampleInterface obj = new TestClass();

obj.SampleMethod();

}

}

The following example demonstrates interface implementation. In this example, the interface IPoint contains the property declaration, which is responsible for setting and getting the values of the fields. The class Point contains the property implementation.

**Partial Class**

It is possible to split the definition of a class or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled.

**Benefits**

1. More than one developer can simultaneously work on the same class

2. When working with automatically generated source code, code can be added to the class without having to re-create the source file. Visual studio uses this approach when it creates windows forms etc. You can create code that uses these classes without having to modify the file created by visual studio.

To split a class definition, use the partial keyword modifier

public partial class Employee

{

public void DoWork()

{

}

}

public partial class Employee

{

public void MyWork()

{

}

}

**Handling Exceptions:**

**using System;**

**namespace ErrorHandlingApplication**

**{**

**class DivNumbers**

**{**

**int result;**

**DivNumbers()**

**{**

**result = 0;**

**}**

**public void division(int num1, int num2)**

**{**

**try**

**{**

**result = num1 / num2;**

**}**

**catch (DivideByZeroException e)**

**{**

**Console.WriteLine("Exception caught: {0}", e);**

**}**

**finally**

**{**

**.WriteLine("Result: {0}", result);**

**}**

**}**

**static void Main(string[] args)**

**{**

**DivNumbers d = new DivNumbers();**

**d.division(25, 0);**

**Console.ReadKey();**

**}**

**}**

**}**

**OUTPUT:** **Exception caught: System.DivideByZeroException: Attempted to divide by zero.**

**Result: 0**

**FILE IO:**

**using System;**

**using System.IO;**

**namespace FileIOApplication**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);**

**for (int i = 1; i <= 20; i++)**

**{**

**F.WriteByte((byte)i);**

**}**

**F.Position = 0;**

**for (int i = 0; i <= 20; i++)**

**{**

**Console.Write(F.ReadByte() + " ");**

**}**

**F.Close();**

**Console.ReadKey();**

**}**

**}**

**}**

**Collections**

**Collections are the enumerable data structures that can be accessed using indexes or keys. Closely**

**related data can be handled more efficiently when grouped together into a collection. Instead of writing**

**separate code to handle each individual object. We can use the same code to process all the elements of**

**a collection.**

**To manage a collection, the array class and the System.Collections classes are used to add, remove and**

**modify either individual elements of the collection or a range of elements.**

**The collection classes provide support for stacks, queues, lists and hash tables. Most collection classes**

**implement the same interfaces, and these interfaces may be inherited to create new collection classes**

**that fit more specialized data storage needs.**

**Example:**

**ArrayList list=new ArrayList()**

**list.Add(10);**

**list.Add("John");**

**list.Add('R');**

**Generics**

Generic classes and methods combine re-usability, type safety and efficiency to their counterpart collections. Generics are most commonly used with collections and the method that operate on them. .NET Framework 2.0 class library provide a new namespace System.Collections. Generic which contains several new generic based collection classes. Generics are checked at compile time.

List is a generic form of a ArrayList.

**List<string> names = new List<string> ();**

**names.Add(―John‖);**

**names.Add(―Danny‖);**

**static void Main(string[] args)**

**foreach (string name in names)**

**{**

**System.Console.WriteLine(name);**

**}**

**names.Insert(2, "Rafael");**

**names.Remove("Hari");**