DOT NET Framework

Unit – 1 Module – 1

Beginning ASP.NET 4.5 in C#

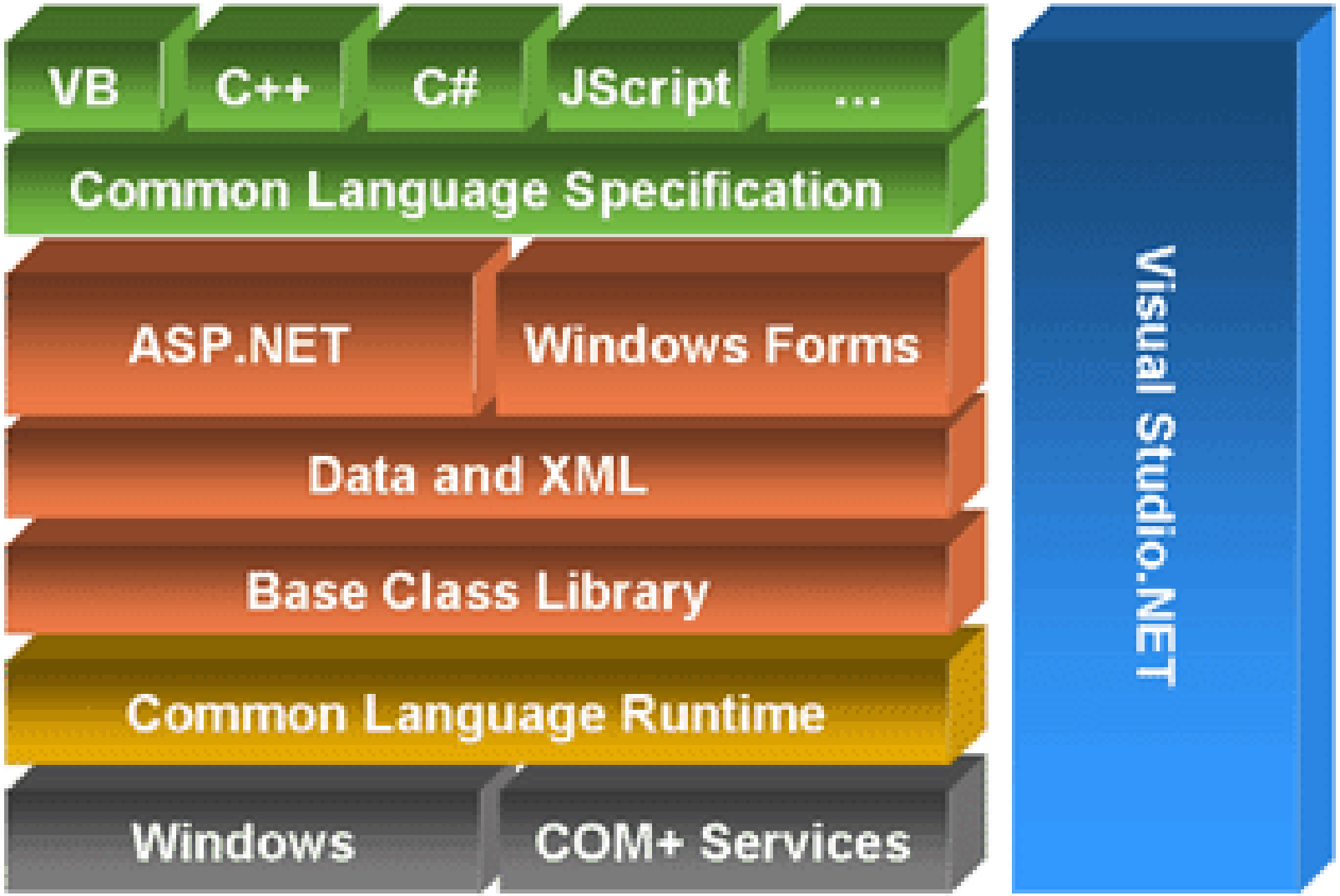
Matthew MacDonald

Pages 9 – 16

Introduction

- **DOT NET Framework** is a software framework developed by Microsoft that runs primarily on Microsoft Windows.
- It includes the following
  - ***Class library named Framework Class Library (FCL)***
  - ***Programs written for .NET Framework execute in a software environment named Common Language Runtime(CLR), an application that provides services such as security, memory management, and exception handling.***

.NET Framework Architecture



DOT NET Languages

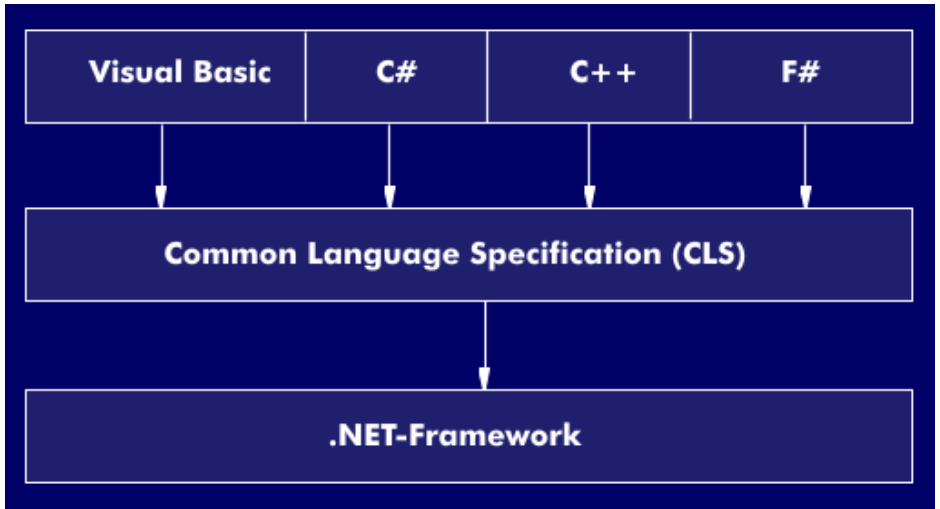
- Both ***VB and C#*** use the .NET class library and are supported by the CLR. In fact, almost any block of C#code can be translated, line by line, into an equivalent block of VB code (and vice versa).
- A developer who has learned one .NET language can move quickly and efficiently to another.
- In short, both VB and C# are elegant, modern languages that are ideal for creating the next generation of web applications.

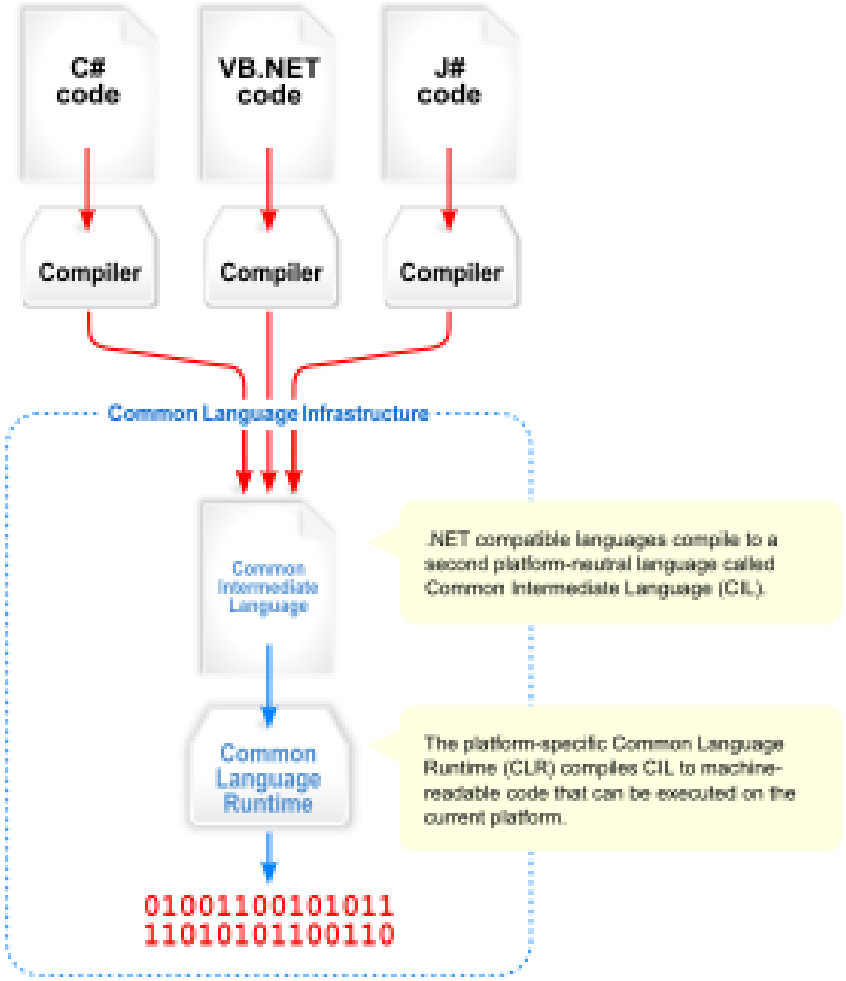
Intermediate Language

- All the .NET languages are compiled into another lower-level language before the code is executed. This lower-level language is the Common Intermediate Language (CIL, or just IL).
- The CLR, the engine of .NET, uses only IL code. Because all .NET languages are designed based on IL, they all have profound similarities. This is the reason that the VB and C# languages provide essentially the same features and performance.
- The .NET Framework formalizes this compatibility with something called the Common Language Specification (CLS). Essentially, the CLS is a contract that, if respected, guarantees that a component written in one .NET language can be used in all the others. One part of the CLS is the common type system (CTS), which defines the rules for data types such as strings, numbers, and arrays that are shared in all .NET languages. The CLS also defines object oriented ingredients such as classes, methods, events, and quite a bit more.
- For the most part, .NET developers don’t need to think about how the CLS works, even though they rely on it every day.
- Every EXE or DLL file that you build with a .NET language contains IL code. This is the file you deploy to other computers.In the case of a web application, you deploy your compiled code to a live web server.

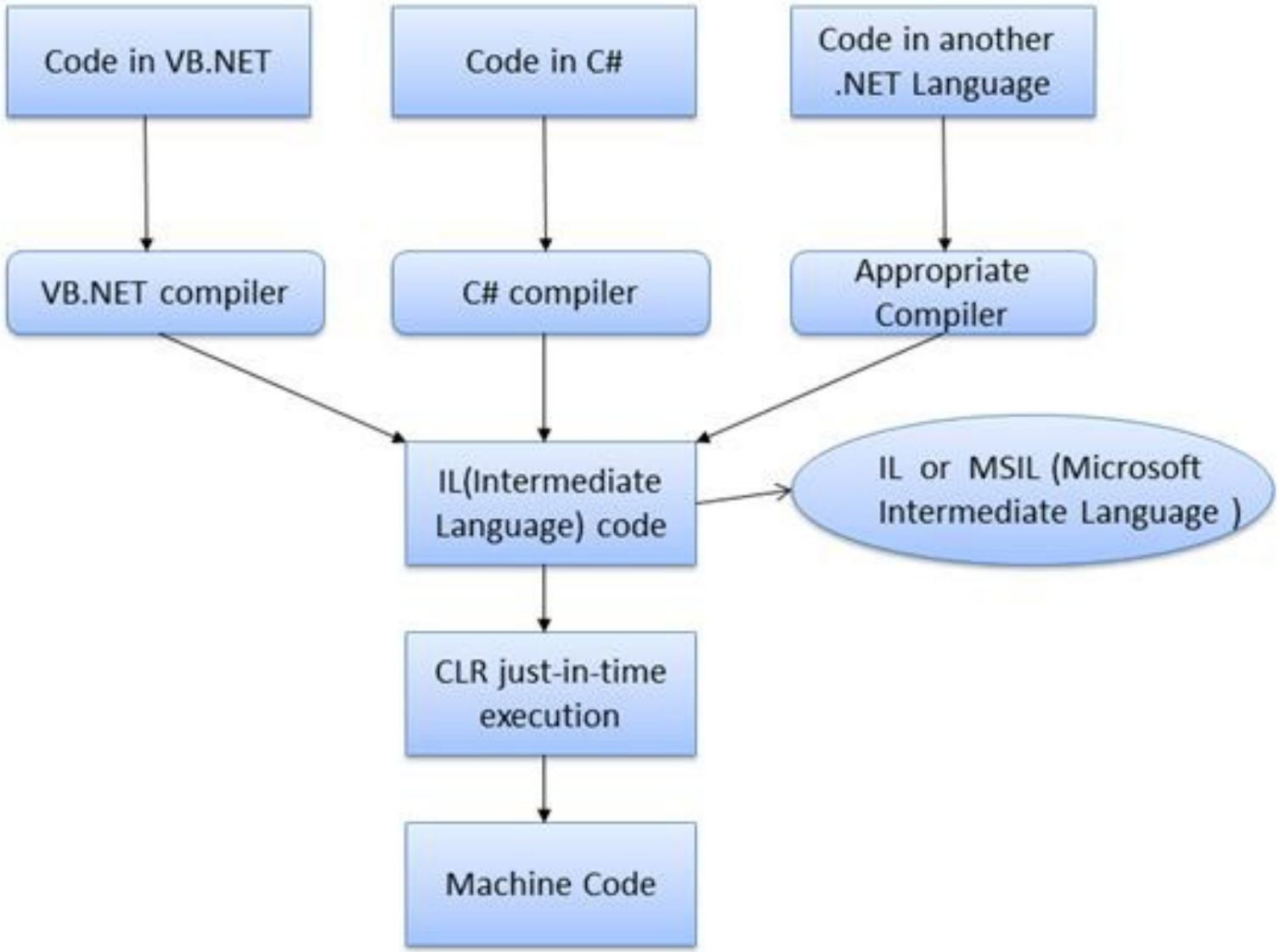
Common Language Specification

- A **Common Language Specification** (**CLS**) is a document that says how computer programs can be turned into Common Intermediate Language (CIL) code. When several languages use the same bytecode, different parts of a program can be written in different languages.
- Microsoft uses a Common Language Specification for their .NET Framework.
- Microsoft has defined CLS which are nothing but guidelines for languages to follow so that it can communicate with other .NET languages in a seamless manner.





Compilation in .NET



The Common Language Runtime

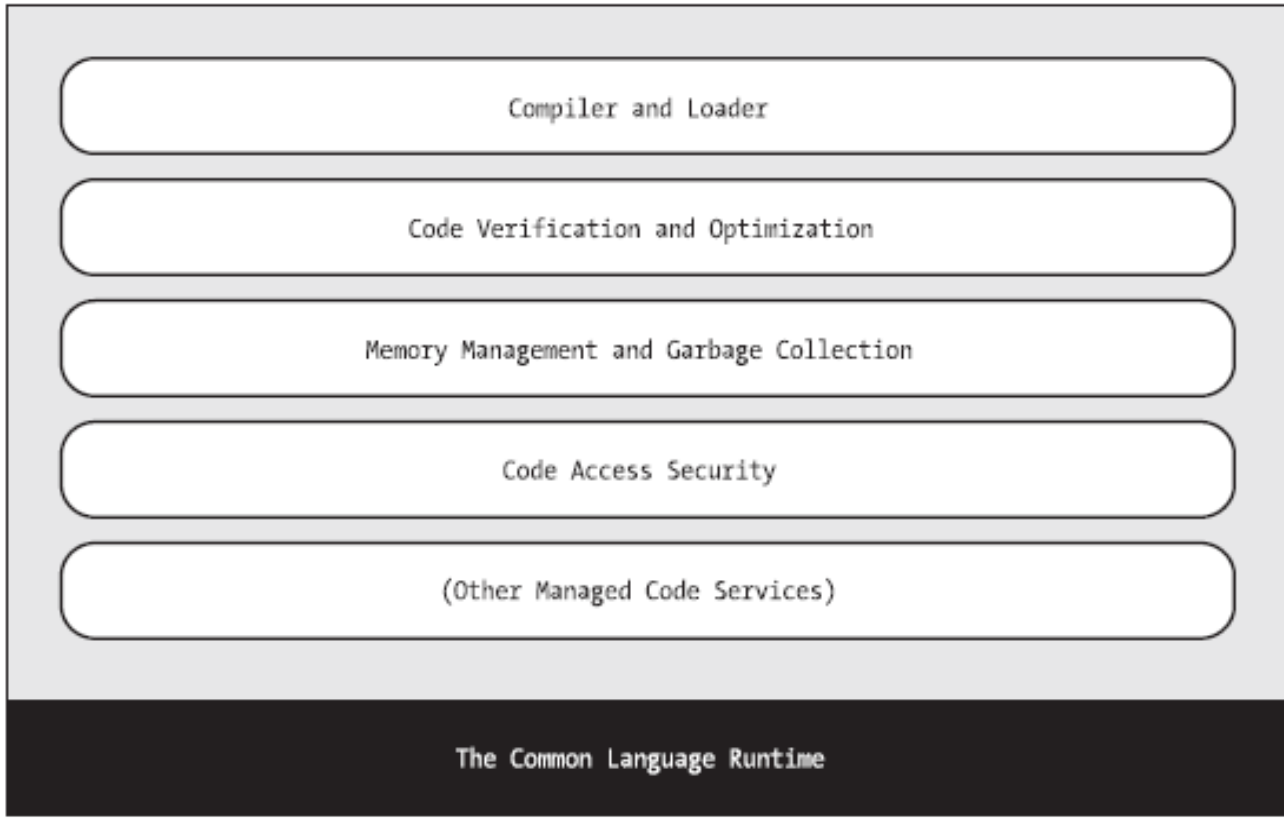
The CLR is the engine that supports all the .NET languages. Many modern languages use runtimes.

These runtimes may provide libraries used by the language, or they may have the additional responsibility of executing the code (as with Java).

Not only does the CLR execute code, it also provides a whole set of related services such as code verification, optimization, and object management.

All .NET code runs inside the CLR - running a Windows application or a web service.

For example, when a client requests an ASP.NET web page, the ASP.NET service runs inside the CLR environment, executes your code, and creates a final HTML page to send to the client.



The implications of the CLR are wide-ranging:
Deep language integration: VB and C#, like all .NET languages, compile to IL. This is far more than mere language compatibility; it's language *integration*.

Side-by-side execution: The CLR also has the ability to load more than one version of a component at a time. In other words, you can update a component many times, and the correct version will be loaded and used for each application. As a side effect, multiple versions of the .NET Framework can be installed

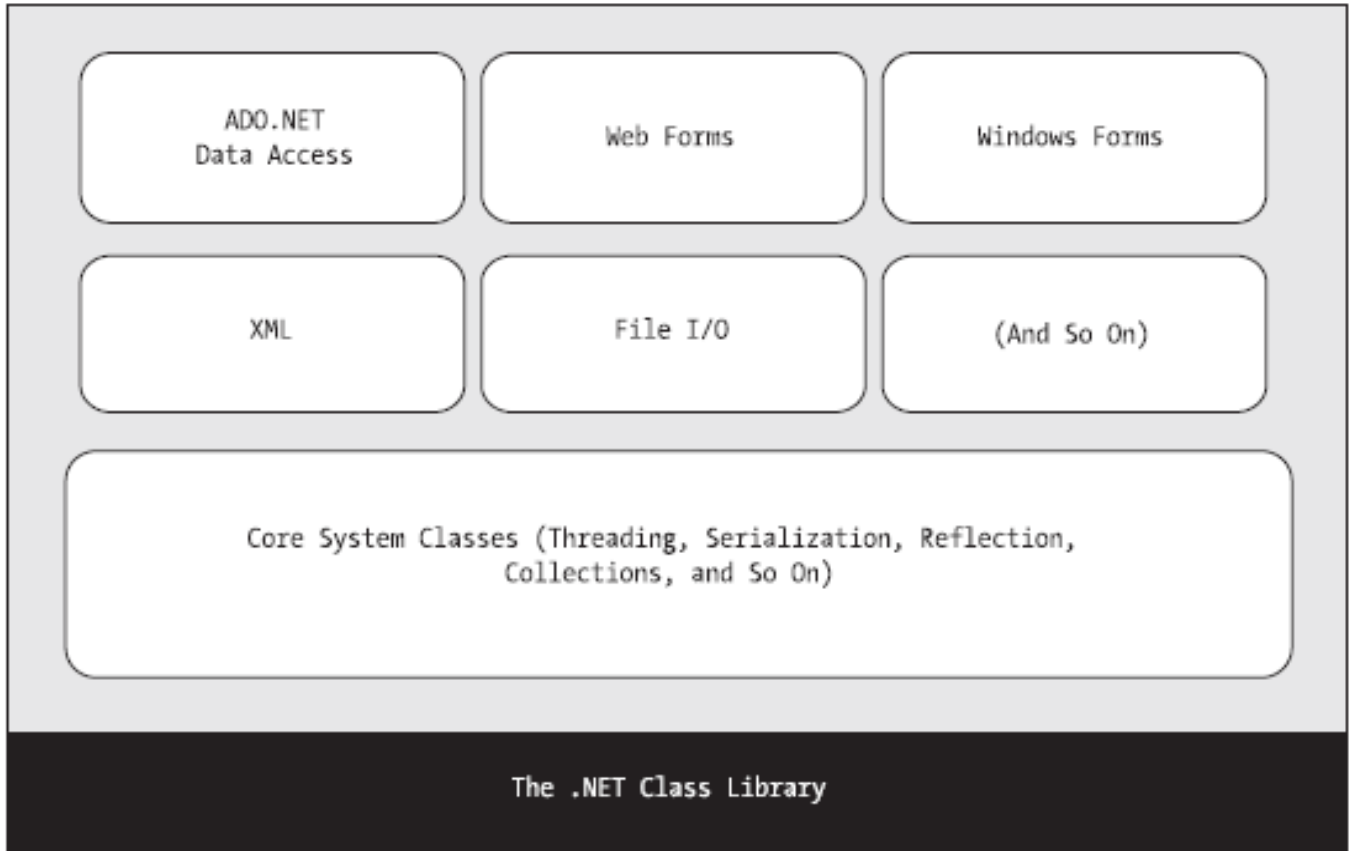
Fewer errors: Whole categories of errors are impossible with the CLR. For example, the CLR prevents many memory mistakes that are possible with lower-level languages such as C++.

Performance: A typical ASP.NET application is much faster than a comparable ASP application, because ASP.NET code is compiled to machine code before it's executed. With high-volume web applications, the potential bottlenecks are rarely processor-related but are usually tied to the speed of an external resource such as a database or the web server's file system.

Code transparency If you distribute a compiled application or component, other programmers may have an easier time determining how your code works. This isn't much of an issue for ASP.NET applications, which aren't distributed but are hosted on a secure web server.

Questionable cross-platform support: No one is entirely sure whether .NET will ever be adopted for use on other operating systems and platforms. However, .NET will probably never have the wide reach of a language such as Java because it incorporates too many different platform-specific and operating system–specific technologies and features.

The .NET Class Library



The .NET class library is a giant repository of classes that provide prefabricated functionality
for everything from reading an XML file to sending an e-mail message.

Any .NET language can use the .NET class library's features by interacting with the right objects. This helps encourage consistency among different .NET languages and removes the need to install numerous components on your computer or web server.

Some parts of the class library are meant for desktop applications with the Windows interface and some are targeted directly at web development.

There are some classes that can be used in various programming scenarios and aren't specific to web or Windows development. These include the base set of classes that define common variable types and the classes for data access.

DOT NET Framework

Assignment – 1

Refer https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview and answer the following questions

- Objectives of DOT NET Framework
- Write a note on the following terms
  - Class Library
  - CLS-Compliant
- Features of CLR

Refer https://docs.microsoft.com/en-us/dotnet/standard/common-type-system and answer the following questions

- What are the activities do CTS perform ?
- What are the types and categories of data types supported by CTS ?

C# Basics

Unit – 1 Module – 2

Beginning ASP.NET 4.5 in C#

Matthew MacDonald

Chapter 2 & 3

Points to Remember

- Case-Sensitive
- Statements are terminated with a semi-colon
- Control Statements and loops should be enclosed with { }
- Comments – Single-Line (//) and Multi-Line (/* */)
- Escaped Characters - \n, \t, \\, \”

Data Types

C# Name	VB Name	.NET Type Name	Contains
byte	Byte	Byte	An integer from 0 to 255.
short	Short	Int16	An integer from –32,768 to 32,767.
int	Integer	Int32	An integer from –2,147,483,648 to 2,147,483,647.
long	Long	Int64	An integer from about –9.2e18 to 9.2e18.
float	Single	Single	A single-precision floating point number from approximately –3.4e38 to 3.4e38 (for big numbers) or –1.5e-45 to 1.5e-45 (for small fractional numbers).
double	Double	Double	A double-precision floating point number from approximately –1.8e308 to 1.8e308 (for big numbers) or –5.0e-324 to 5.0e-324 (for small fractional numbers).
decimal	Decimal	Decimal	A 128-bit fixed-point fractional number that supports up to 28 significant digits.
char	Char	Char	A single 16-bit Unicode character.
string	String	String	A variable-length series of Unicode characters.
bool	Boolean	Boolean	A true or false value.
*	Date	DateTime	Represents any date and time from 12:00:00 AM, January 1 of the year 1 in the Gregorian calendar, to 11:59:59 PM, December 31 of the year 9999. Time values can resolve values to 100 nano-second increments. Internally, this data type is stored as a 64-bit integer.
*	*	TimeSpan	Represents a period of time, as in ten seconds or three days. The smallest possible interval is 1 *tick* (100 nanoseconds).
object	Object	Object	The ultimate base class of all .NET types. Can contain any data type or object.

Arrays and ArrayLists

- 1D Array – string[] stringArray = new string[4];

  string[] stringArray = {"1", "2", "3", "4"};
- 2D Array – int[,] intArray = new int[2, 4];

  int[,] intArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
- ArrayList
- C# arrays do not support redimensioning. This means that once you create an array, you can't change its size. If you need a dynamic array-like list, you can use one of the collection classes provided to all .NET languages
- through the .NET class library. One of the simplest collection classes that .NET offers is the ArrayList, which always allows dynamic resizing. Here's a snippet of C# code that uses an ArrayList:
- ArrayList dynamicList = new ArrayList();
- // The ArrayList is not strongly typed, so you can add any data type
- dynamicList.Add("one");
- dynamicList.Add("two");
- dynamicList.Add("three");
- // Retrieve the first string. Notice that the object must be converted to a string, because there's no way for .NET to be certain what it is.
- string item = Convert.ToString(dynamicList[0]);

TypeCasting & String Type

```
double myValue;
myValue = Math.Sqrt(81);            // myValue = 9.0
myValue = Math.Round(42.889, 2);    // myValue = 42.89
myValue = Math.Abs(-10);            // myValue = 10.0
myValue = Math.Log(24.212);         // myValue = 3.18.. (and so on)
myValue = Math.PI;                  // myValue = 3.14.. (and so on)
```

```
string countString = "10";

// Convert the string "10" to the numeric value 10.
int count = Convert.ToInt32(countString);

// Convert the numeric value 10 into the string "10".
countString = Convert.ToString(count);
```

```
string myString = "This is a test string       ";
myString = myString.Trim();                 // = "This is a test string"
myString = myString.Substring(0, 4);        // = "This"
myString = myString.ToUpper();              // = "THIS"
myString = myString.Replace("IS", "AT");    // = "THAT"

int length = myString.Length;               // = 4
```

Operator	Description	Example
+	Addition	1 + 1 = 2
-	Subtraction (and to indicate negative numbers)	5 - 2 = 3
*	Multiplication	2 * 5 = 10
/	Division	5.0 / 2 = 2.5
%	Gets the remainder left after integer division	7 % 3 = 1

Operator	Description
==	Equal to.
!=	Not equal to.
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
&&	Logical and (evaluates to true only if both expressions are true). If the first expression is false, the second expression is not evaluated.
||	Logical or (evaluates to true if either expression is true). If the first expression is true, the second expression is not evaluated.

Member	Description
Length	Returns the number of characters in the string (as an integer).
ToUpper() and ToLower()	Returns a copy of the string with all the characters changed to uppercase or lowercase characters.
Trim(), TrimEnd(), and TrimStart()	Removes spaces or some other characters from either (or both) ends of a string.
Insert()	Puts another string inside a string at a specified (zero-based) index position. For example, Insert(1, "pre") adds the string *pre* after the first character of the current string.
Remove()	Removes a specified number of strings from a specified position. For example, Remove(0, 1) removes the first character.
Replace()	Replaces a specified substring with another string. For example, Replace("a", "b") changes all *a* characters in a string into *b* characters.
Substring()	Extracts a portion of a string of the specified length at the specified location (as a new string). For example, Substring(0, 2) retrieves the first two characters.
StartsWith() and EndsWith()	Determines whether a string starts or ends with a specified substring. For example, StartsWith("pre") will return either true or false, depending on whether the string begins with the letters *pre* in lowercase.
IndexOf() and LastIndexOf()	Finds the zero-based position of a substring in a string. This returns only the first match and can start at the end or beginning. You can also use overloaded versions of these methods that accept a parameter that specifies the position to start the search.
Split()	Divides a string into an array of substrings delimited by a specific substring. For example, with Split(".") you could chop a paragraph into an array of sentence strings.
Join()	Fuses an array of strings into a new string. You can also specify a separator that will be inserted between each element.

Table 2-4. *Useful DateTime Members*

Member	Description
Now	Gets the current date and time. You can also use the UtcNow property to take the current computer's time zone into account. UtcNow gets the time as a *coordinated universal time* (UTC). Assuming your computer is correctly configured, this corresponds to the current time in the Western European (UTC+0) time zone.
Today	Gets the current date and leaves time set to 00:00:00.
Year, Date, Month, Day, Hour, Minute, Second, and Millisecond	Returns one part of the DateTime object as an integer. For example, Month will return 12 for any day in December.
DayOfWeek	Returns an enumerated value that indicates the day of the week for this DateTime, using the DayOfWeek enumeration. For example, if the date falls on Sunday, this will return DayOfWeek.Sunday.
Add() and Subtract()	Adds or subtracts a TimeSpan from the DateTime.
AddYears(), AddMonths(), AddDays(), AddHours(), AddMinutes(), AddSeconds(), AddMilliseconds()	Adds an integer that represents a number of years, months, and so on, and returns a new DateTime. You can use a negative integer to perform a date subtraction.
DaysInMonth()	Returns the number of days in the specified month in the specified year.
IsLeapYear()	Returns true or false depending on whether the specified year is a leap year.
ToString()	Returns a string representation of the current DateTime object. You can also use an overloaded version of this method that allows you to specify a parameter with a format string.

DateTime

- DateTime dob = **new** DateTime(1974, 7, 10, 7, 10, 24);
- Console.WriteLine("Day:{0}", dob.Day);
- Console.WriteLine("Month:{0}", dob.Month);
- Console.WriteLine("Year:{0}", dob.Year);
- Console.WriteLine("Hour:{0}", dob.Hour);
- Console.WriteLine("Minute:{0}", dob.Minute);
- Console.WriteLine("Second:{0}", dob.Second);
- Console.WriteLine("Millisecond:{0}", dob.Millisecond);
- dob.AddYears(2);
- dob.AddDays(12);
- dob.AddHours(4);
- dob.AddMinutes(15);
- dob.AddSeconds(45);
- dob.AddMilliseconds(200);

DateTime and TimeSpan Objects

```
DateTime dob = new DateTime(2000, 10, 20, 12, 15, 45);
DateTime subDate = new DateTime(2000, 2, 6, 13, 5, 15);
  // TimeSpan with 10 days, 2 hrs, 30 mins, 45 seconds, and 100 milliseconds
TimeSpan ts = new TimeSpan(10, 2, 30, 45, 100);
  // Subtract a DateTime
TimeSpan diff1 = dob.Subtract(subDate);
Console.WriteLine(diff1.ToString());
  // Subtract a TimeSpan
DateTime diff2 = dob.Subtract(ts);
Console.WriteLine(diff2.ToString());
  // Subtract 10 Days
DateTime daysSubtracted = new DateTime(dob.Year, dob.Month, dob.Day - 10);
Console.WriteLine(daysSubtracted.ToString());
  // Subtract hours, minutes, and seconds
DateTime hms = new DateTime(dob.Year, dob.Month, dob.Day, dob.Hour - 1, dob.Minute - 15, dob.Second - 15);
Console.WriteLine(hms.ToString());
```

Comparing two dates

```
DateTime firstDate = new DateTime(2002, 10, 22);
DateTime secondDate = new DateTime(2009, 8, 11);
int result = DateTime.Compare(firstDate, secondDate);

if (result < 0)
    Console.WriteLine("First date is earlier");
else if (result == 0)
    Console.WriteLine("Both dates are same");
else
    Console.WriteLine("First date is later");
```

Date Formatting

- Console.WriteLine(dob.ToString("r"));

```
d: 10/22/2002
D: Tuesday, October 22, 2002
f: Tuesday, October 22, 2002 12:00 AM
F: Tuesday, October 22, 2002 12:00:00 AM
g: 10/22/2002 12:00 AM
G: 10/22/2002 12:00:00 AM
m: October 22
M: October 22
```

```
t: 12:00 AM
T: 12:00:00 AM
```

```
y: October, 2002
Y: October, 2002
```

Table 2-6. *Useful Array Members*

Member	Description
Length	Returns an integer that represents the total number of elements in all dimensions of an array. For example, a 3×3 array has a length of 9.
GetLowerBound() and GetUpperBound()	Determines the dimensions of an array. As with just about everything in .NET, you start counting at zero (which represents the first dimension).
Clear()	Empties part or all of an array's contents, depending on the index values that you supply. The elements revert to their initial empty values (such as 0 for numbers).
IndexOf() and LastIndexOf()	Searches a one-dimensional array for a specified value and returns the index number. You cannot use this with multidimensional arrays.
Sort()	Sorts a one-dimensional array made up of comparable data such as strings or numbers.
Reverse()	Reverses a one-dimensional array so that its elements are backward, from last to first.

Control Statements and Loops

```
if (myNumber > 10)
{
    // Do something.
}
else if (myString == "hello")
{
    // Do something.
}
else
{
    // Do something.
}
```

```
switch (myNumber)
{
    case 1:
        // Do something.
        break;
    case 2:
        // Do something.
        break;
    default:
        // Do something.
        break;
}
```

```
for (int i = 0; i < 10; i++)
{
    // This code executes ten times.
    System.Diagnostics.Debug.Write(i);
}
```

```
int i = 0;
while (i < 10)
{
    i += 1;
    // This code executes ten times.
}
```

```
string[] stringArray = {"one", "two", "three"};

foreach (string element in stringArray)
{
    // This code loops three times, with the element variable set to
    // "one", then "two", and then "three".
    System.Diagnostics.Debug.Write(element + " ");
}
```

```
int i = 0;
do
{
    i += 1;
    // This code executes ten times.
}
while (i < 10);
```

Methods

```
// This method doesn't return any information.
void MyMethodNoReturnedData()
{
    // Code goes here.
}

// This method returns an integer.
int MyMethodReturnsData()
{
    // As an example, return the number 10.
    return 10;
}
```

```
private int AddNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

```
private decimal GetProductPrice(int ID)
{
    // Code here.
}

private decimal GetProductPrice(string name)
{
    // Code here.
}

// And so on...
```

```
// Get price by product ID (the first version).
price = GetProductPrice(1001);

// Get price by product name (the second version).
price = GetProductPrice("DVD Player");
```

Classes

```
public class MyClass
{
    // Class code goes here.
}
```

```
public class Product
{
    private string name;
    private decimal price;
    private string imageUrl;
}
```

```
Product saleProduct = new Product();

// Optionally you could do this in two steps:
// Product saleProduct;
// saleProduct = new Product();

// Now release the object from memory.
saleProduct = null;
```

Table 3-1. *Accessibility Keywords*

Keyword	Accessibility
public	Can be accessed by any class
private	Can be accessed only by members inside the current class
internal	Can be accessed by members in any of the classes in the current assembly (the compiled code file)
protected	Can be accessed by members in the current class or in any class that inherits from this class
protected internal	Can be accessed by members in the current application (as with internal) *and* by the members in any class that inherits from this class

```
using System;
namespace demo
{ class Student
{
private string code = "N.A";
private string name = "not known";
private int age = 0;
// Declare a Code property of type string:
public string Code
{ get
        { return code; }
  set
        { code = value; }
}
// Declare a Name property of type string:
public string Name
{ get
        { return name; }
  set { name = value; }
}
// Declare a Age property of type int:
public int Age
{ get
        { return age; }
  set
        { age = value; }
}
public override string ToString()
{ return "Code = " + Code +", Name = " + Name + ", Age
= " + Age; }
}
```

```
class ExampleDemo
{ public static void Main()
{
// Create a new Student object:
Student s = new Student();
// Setting code, name and the age of the student
s.Code = "001";
s.Name = "Zara";
s.Age = 9;
Console.WriteLine("Student Info: {0}", s);
//let us increase age
s.Age += 1;
Console.WriteLine("Student Info: {0}", s);
Console.ReadKey();
}
}
}
```

Adding Properties

```
public class Product
{
    private string name;
    private decimal price;
    private string imageUrl;
    public string Name
    {
        get
        { return name; }
        set
        { name = value; }
    }

    public decimal Price
    {
        get
        { return price; }
        set
        { price = value; }
    }

    public string ImageUrl
    {
        get
        { return imageUrl; }
        set
        { imageUrl = value; }
    }
}
```

```
Product saleProduct = new Product();
saleProduct.Name = "Kitchen Garbage";
saleProduct.Price = 49.99M;
saleProduct.ImageUrl = "http://mysite/garbage.png";
```

```
public class Product
{
    // (Variables and properties omitted for clarity.)

    public string GetHtml()
    {
        string htmlString;
        htmlString = "<h1>" + name + "</h1><br />";
        htmlString += "<h3>Costs: " + price.ToString() + "</h3><br />";
        htmlString += "<img src='" + imageUrl + "' />";
        return htmlString;
    }
}
```

```
public Product(string name, decimal price)
{
    // Set the two properties in the class.
    Name = name;
    Price = price;
}
```