Unit 1
The Philosophy of .NET

1.1 Understanding the Previous State of Affairs:

Life As a C/Win32 API Programmer:

- Developing software for the Windows family of operating systems involved using the C programming language in conjunction with the Windows application programming interface.
- C is a very terse language, in developer's does manual memory management, ugly pointer arithmetic, and ugly syntactical constructs.
- C is a structured language; it lacks the benefits provided by the object-oriented approach.

Life As a C++/MFC Programmer:

- C++ can be thought of as an object-oriented *layer* on top of C. Thus, even though C++ programmers benefit from the famed "pillars of OOP" (encapsulation, inheritance, and polymorphism), they are still at the mercy of the painful aspects of the C language (e.g., manual memory management, ugly pointer arithmetic, and ugly syntactical constructs).
- The Microsoft Foundation Classes (MFC) provides the developer with a set of C++ classes that facilitate the construction of Win32 applications. The main role of MFC is to wrap a "sane subset" of the raw Win32 API behind a number of classes, magic macros, and numerous code-generation tools (aka *wizards*).
- C++ programming remains a difficult and error-prone experience, given its historical roots in C.

Life As a Visual Basic 6.0 Programmer:

- VB6 is popular due to its ability to build complex user interfaces, code libraries (e.g., COM servers), and data access logic with minimal fuss and bother.
- Even more than MFC, VB6 hides the complexities of the raw Win32 API from view using a number of integrated code wizards, intrinsic data types, classes, and VB-specific functions
- VB6 is not a fully object-oriented language; rather, it is "object aware."
- For example, VB6 does not allow the programmer to establish "is-a" relationships between types (i.e., no classical inheritance) and has no intrinsic support for parameterized class construction.
- VB6 doesn't provide the ability to build multithreaded applications unless you are willing to drop down to low-level Win32 API calls.

Life As a Java/J2EE Programmer

- The Java programming language is (almost) completely object oriented and has its syntactic roots in C++.

- Its support for platform independence.
- Java (as a language) cleans up many unsavory syntactical aspects of C++. Java (as a platform) provides programmers with a large number of predefined "packages" that contain various type definitions.
- Using these types, Java programmers are able to build "100% Pure Java" applications complete with database connectivity, messaging support, web-enabled front ends, and a rich user interface.
- Java is a very elegant language; one potential problem is you must use Java front-to-back during the development cycle.

Life As a COM Programmer:

- The Component Object Model (COM) was Microsoft's previous application development framework.
- COM is an architecture that says in effect, "If you build your classes in accordance with the rules of COM, you end up with a block of ***reusable binary code.***"
- The beauty of a binary COM server is that it can be accessed in a language-independent manner.
- C++ programmers can build COM classes that can be used by VB6. Delphi programmers can use COM classes built using C.
- COM is its location-transparent nature Using constructs such as application identifiers (AppIDs), stubs, proxies, and the COM runtime environment, programmers can avoid the need to work with raw sockets, RPC calls, and other low-level details.

Life As a Windows DNA Programmer:

- Building a web application using COM-based Windows **Distributed interNet Applications Architecture (DNA)** is also quite complex.
- Some of this complexity is due to the simple fact that Windows DNA requires the use of numerous technologies and languages (ASP, HTML, XML, JavaScript, VBScript, and COM(+), as well as a data access API such as ADO). One problem is that many of these technologies are completely unrelated from a syntactic point of view.

1.2 The .NET Solution

The .NET Framework is a completely new model for building systems on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as Mac OS X and various Unix/Linux distributions.

Some core features provided courtesy of .NET:

1. ***Full interoperability with existing code***: This is (of course) a good thing. Existing COM binaries can commingle (i.e., interop) with newer .NET binaries and vice versa. Also, Platform Invocation Services (PInvoke) allows you to call C-based libraries from .NET code.
2. ***Complete and total language integration***: Unlike COM, .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging.
3. ***A common runtime engine shared by all .NET-aware languages***: One aspect of this engine is a well-defined set of types that each .NET-aware language "understands."
4. ***A base class library***: This library provides shelter from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.
5. ***No more COM plumbing***: IClassFactory, IUnknown, IDispatch, IDL code, and the evil VARIANTcompliant data types (BSTR, SAFEARRAY, and so forth) have no place in a native .NET binary.
6. ***A truly simplified deployment model***: Under .NET, there is no need to register a binary unit into the system registry. Furthermore, .NET allows multiple versions of the same ***.dll** to exist in harmony on a single machine.

1.3 Introducing the Building Blocks of the .NET Platform (CLR, CTS, and CLS)

Some of the benefits provided by .NET, let's preview three key (and interrelated) entities that make it all possible: the CLR, CTS, and CLS.

Common Language Runtime*, or *CLR:

- The primary role of the CLR is to locate, load, and manage .NET types on your behalf. The CLR also takes care of a number of low-level details such as memory management and performing security checks.

***Common Type System* or *CTS*:**

- The CTS specification fully describes all possible data types and programming constructs supported by the runtime, specifies how these entities can interact with each other, and details how they are represented in the .NET metadata format.

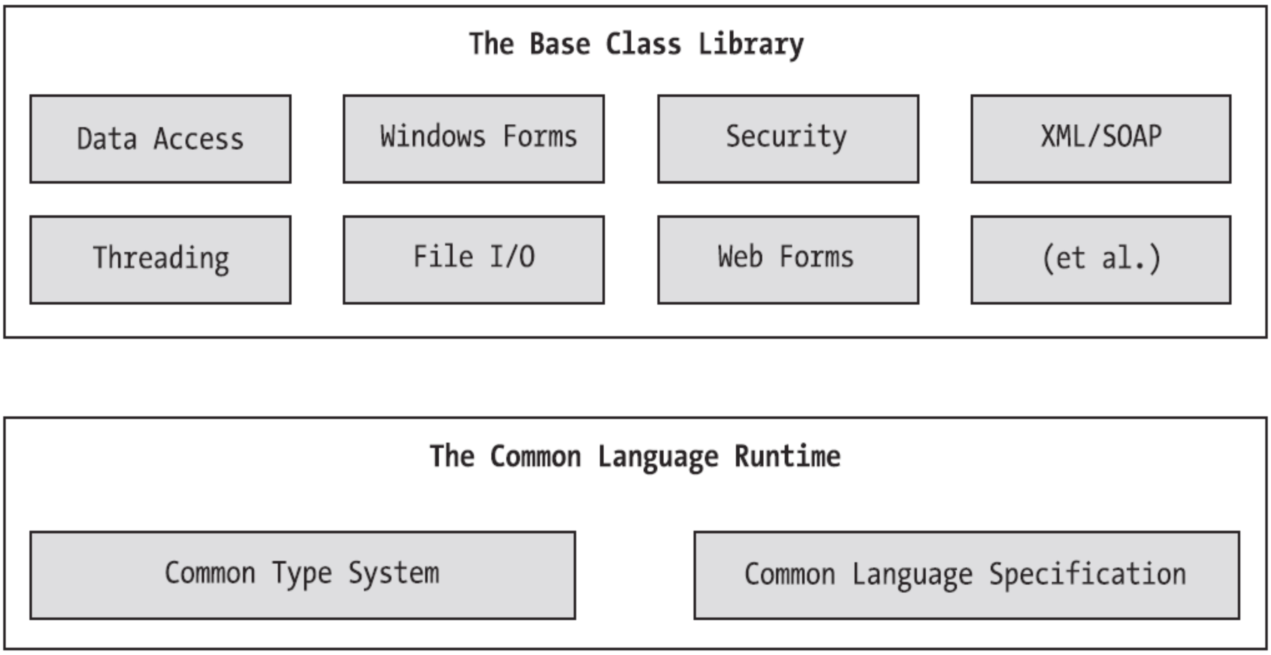
Common Language Specification or CLS:

- The CLS is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on.
- Thus, if you build .NET types that only expose CLS-compliant features, you can rest assured that all .NET-aware languages can consume them.

The Role of the Base Class Libraries

- The .NET platform provides a base class library that is available to all .NET programming languages.
- Base class library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.
- For example, the base class libraries define types that facilitate database access, XML manipulation, programmatic security, and the construction of web-enabled front ends.

The relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure.



The CLR, CTS, CLS, and base class library relationship

1.4 What C# Brings to the Table?

- C#'s syntactic constructs are modeled after various aspects of VB 6.0 and C++.
- For example, like VB6, C# supports the notion of formal type properties and the ability to declare methods taking varying number of arguments (via parameter arrays).
- Like C++, C# allows you to overload operators, to create structures, enumerations, and callback functions (via delegates).
- C# is a hybrid of numerous languages, syntactically clean, provides power and flexibility.

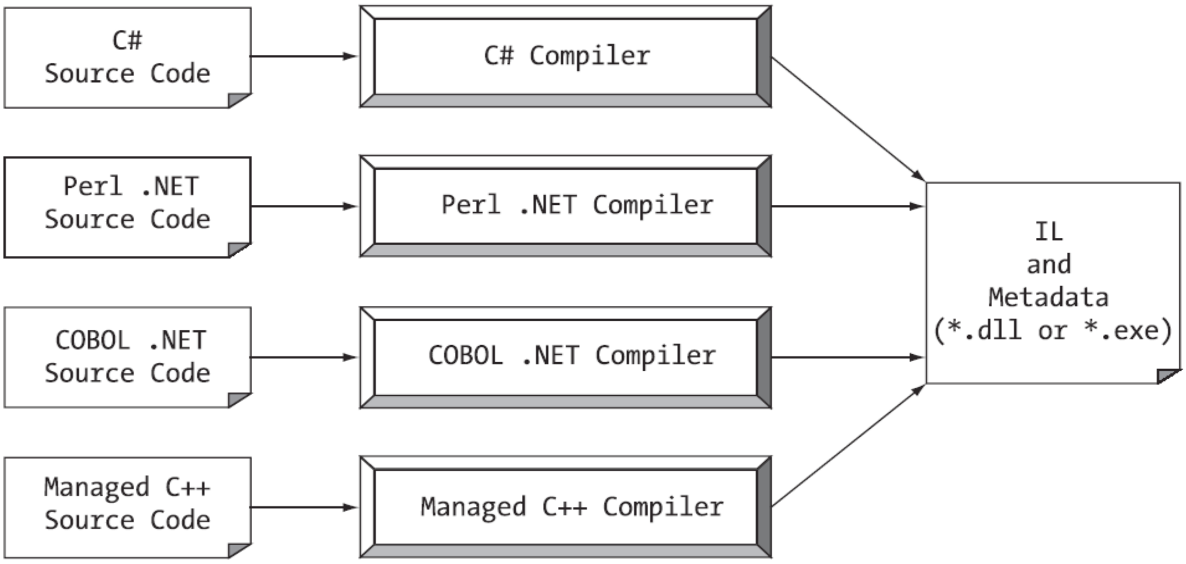
The **C# language offers the following features** :

1. No pointers required! C# programs typically have no need for direct pointer manipulation
2. Automatic memory management through garbage collection, So C# does not support a delete keyword.

3. Formal syntactic constructs for enumerations, structures, and class properties.
4. The C++-like ability to overload operators for a custom type, without the complexity.
5. The ability to build generic types and generic members using a syntax very similar to C++ templates.
6. Full support for interface-based programming techniques.
7. Full support for aspect-oriented programming (AOP) techniques via attributes.

1.5 An Overview of .NET Assemblies

- ***.dll** .NET binaries do not export methods to facilitate communications with the COM
- .NET binaries are not described using COM type libraries and are not registered into the system registry.
- .NET binaries do not contain platform-specific instructions, but rather **platform-agnostic *intermediate language* (*IL*) and type metadata.**
- When a *.dll or *.exe has been created using a .NET-aware compiler, the resulting module is bundled into an ***assembly***.
- An assembly contains CIL code, which is conceptually similar to Java bytecode in that it is not compiled to platform-specific instructions until absolutely necessary.



All .NET-aware compilers emit IL instructions and metadata.

- Assemblies also contain *metadata.* .NET metadata is a dramatic improvement to COM type metadata. .NET metadata is always present and is automatically generated by a given .NET-aware compiler.
- Assemblies themselves are also described using metadata, which is officially termed a ***manifest***

- The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources), and a list of all externally referenced assemblies that are required for proper execution.

Single-File and Multifile Assemblies

Single-file assembly If an assembly is composed of a single *.dll or *.exe module, called as *single-file assembly*. Single-file assemblies contain all the necessary CIL, metadata, and associated manifest in an autonomous, single, well-defined package.

Multifile assemblies are composed of numerous .NET binaries, each of which is termed a *module*. When building a multifile assembly, one of these modules (termed the *primary module*) must contain the assembly manifest (and possibly CIL instructions and metadata for various types). The other related modules contain a module level manifest, CIL, and type metadata.

The Role of the Common Intermediate Language

CIL is a language that sits above any particular platform-specific instruction set. Regardless of which .NET-aware language you choose, the associated compiler emits CIL instructions.

For example,

```
// Calc.cs
using System;
namespace CalculatorExample
{
// This class contains the app's entry point.
public class CalcApp
{
static void Main()
{
Calc c = new Calc();
int ans = c.Add(10, 84);
Console.WriteLine("10 + 84 is {0}.", ans);
// Wait for user to press the Enter key before shutting down.
Console.ReadLine();
}
}
// The C# calculator.
public class Calc
```

```
{
public int Add(int x, int y)
{ return x + y; }
}
}
```

Once the C# compiler (csc.exe) compiles this source code file, you end up with a single-file *.exe assembly that contains a manifest, CIL instructions, and metadata describing each aspect of the Calc and CalcApp classes.

For example, if you were to open this assembly using ildasm.exe, you would find that the Add() method is represented using CIL such as the following:

```
.method public hidebysig instance int32 Add(int32 x, int32 y) cil managed
{
// Code size 8 (0x8)
.maxstack 2
.locals init ([0] int32 CS$1$0000)
IL_0000: ldarg.1
IL_0001: ldarg.2
IL_0002: add
IL_0003: stloc.0
IL_0004: br.s IL_0006
IL_0006: ldloc.0
IL_0007: ret
} // end of method Calc::Add
```

The C# compiler emits CIL, not platform-specific instructions.

Benefits of CIL:

- Each .NET-aware compiler produces nearly identical CIL instructions. Therefore, all languages are able to interact within a well-defined binary arena.
- CIL is platform-agnostic, the .NET Framework itself is platform-agnostic, providing the same benefits Java developers have grown accustomed to (i.e., a single code base running on numerous operating systems).

Compiling CIL to Platform-Specific Instructions:

Due to the fact that assemblies contain CIL instructions, rather than platform-specific instructions, CIL code must be compiled before use.

- The entity that compiles CIL code into meaningful CPU instructions is termed a *just-in-time* (*JIT*) *compiler or Jitter*
- The .NET runtime environment leverages (Use to maximum advantage) a JIT compiler for each CPU targeting the runtime, each optimized for the underlying platform.
- For example,

  1.If you are building a .NET application that is to be deployed to a handheld device (such as a Pocket PC), the corresponding Jitter is well equipped to run within a low memory environment.

  2. If you are deploying your assembly to a back-end server (where memory is seldom an issue), the Jitter will be optimized to function in a high memory environment.
- In this way, developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures.
- Jitter compiles CIL instructions into corresponding machine code, it will cache the results in memory in a manner suited to the target operating system.
- In this way, if a call is made to a method named PrintDocument(), the CIL instructions are compiled into platform specific instructions on the first invocation and retained in memory for later use. Therefore, the next time PrintDocument() is called, there is no need to recompile the CIL.

The Role of .NET Type Metadata

- A .NET assembly contains full, complete, and accurate metadata, which describes each and every type (class, structure, enumeration, and so forth) defined in the binary, as well as the members of each type (properties, methods, events, and so on).
- Because .NET metadata is so wickedly meticulous, assemblies are completely self-describing entities and .NET binaries have no need to be registered into the system registry.

To illustrate the format of .NET type metadata, look the metadata that has been generated for the Add() method of the C# Calc class.

```
TypeDef #2 (02000003)
-------------------------------------------------------
TypDefName: CalculatorExample.Calc (02000003)
Flags : [Public] [AutoLayout] [Class]
[AnsiClass] [BeforeFieldInit] (00100001)
Extends : 01000001 [TypeRef] System.Object
Method #1 (06000003)
-------------------------------------------------------
```

MethodName: Add (06000003)

Flags : [Public] [HideBySig] [ReuseSlot] (00000086)

RVA : 0x00002090

ImplFlags : [IL] [Managed] (00000000)

CallCnvntn: [DEFAULT]

hasThis

ReturnType: I4

2 Arguments

Argument #1: I4

Argument #2: I4

2 Parameters

(1) ParamToken : (08000001) Name : x flags: [none] (00000000)

(2) ParamToken : (08000002) Name : y flags: [none] (00000000)

- Metadata is used by numerous aspects of the .NET runtime environment, as well as by various development tools.
- For example, the **IntelliSense feature** provided by Visual Studio 2005 is made possible by reading an assembly's metadata at design time.
- Metadata is also used by various object browsing utilities, debugging tools, and the C# compiler itself.
- Metadata is the backbone of numerous .NET technologies including remoting, reflection, late binding, XML web services, and object serialization.

The Role of the Assembly Manifest

- .NET assembly also contains metadata that describes the assembly itself (technically termed a *manifest*).
- The manifest documents all external assemblies required by the current assembly to function correctly, the assembly's version number, copyright information, and so forth. Like type metadata, it is always the job of the compiler to generate the assembly's manifest.

For example,

CSharpCalculator.exe manifest:

.assembly extern mscorlib

{

.publickeytoken = (B7 7A 5C 56 19 34 E0 89)

.ver 2:0:0:0

```
}
.assembly CSharpCalculator
{
.hash algorithm 0x00008004
.ver 0:0:0:0
}
.module CSharpCalculator.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
```

Understanding the Common Type System

Common Type System (CTS) is a formal specification that documents how types must be defined in order to be hosted by the CLR.

There are five types defined by the CTS in their language.

1. **CTS Class Types**
2. **CTS Structure Types**
3. **CTS Interface Types**
4. **CTS Enumeration Types**
5. **CTS Delegate Types**

CTS Class Types

- Every .NET-aware language supports, at the very least, the notion of a *class type*, which is the cornerstone of object-oriented programming (OOP).
- A class may be composed of any number of members (such as properties, methods, and events) and data points (fields).
- In C#, classes are declared using the class keyword:

```
// A C# class type.
public class Calc
{
public int Add(int x, int y)
{ return x + y; }
}
```

CTS Class Characteristics

1. ***Sealed classes*** cannot function as a base class to other classes.
2. The CTS allows a class to implement any number of **interfaces**._ an ***interface*** is a collection of abstract members that provide a contract between the object and object user.
3. ***Abstract*** **classes** cannot be directly created, but are intended to define common behaviors for derived types. ***Concrete*** **classes** can be created directly.
4. Each class must be configured with a ***visibility attribute***. Basically, this trait defines if the class may be used by external assemblies, or only from within the defining assembly.

CTS Structure Types

- A *structure* can be thought of as a lightweight class type having value-based semantics
- Structures are best suited for modeling geometric and mathematical data, and are created in C# using the struct keyword:

```
// A C# structure type.
struct Point
{
// Structures can contain fields.
public int xPos, yPos;
// Structures can contain parameterized constructors.
public Point(int x, int y)
{ xPos = x; yPos = y;}
// Structures may define methods.
public void Display()
{
Console.WriteLine("({0}, {1}", xPos, yPos);
}
}
```

CTS Interface Types

- ***Interfaces*** are nothing more than a named collection of abstract member definitions, which may be supported (i.e., implemented) by a given class or structure.
- In C#, interface types are defined using the **interface keyword.**

For example:

```
// A C# interface type.
public interface IDraw
{
void Draw();
}
```

On their own, interfaces are of little use. However, when a class or structure implements a given interface in its unique way, you are able to request access to the supplied functionality using an interface reference in a polymorphic manner.

CTS Enumeration Types

- ***Enumerations*** are a handy programming construct that allows you to group name/value pairs.
- For example, assume you are creating a video-game application that allows the player to select one of three character categories (Wizard, Fighter, or Thief).
- Rather than keeping track of raw numerical values to represent each possibility, you could build a custom enumeration using the **enum keyword**:

```
// A C# enumeration type.
public enum CharacterType
{
Wizard = 100,
Fighter = 200,
Thief = 300
}
```

- By default, the storage used to hold each item is a 32-bit integer; however, it is possible to alter this storage slot if need be (e.g., when programming for a low-memory device such as a Pocket PC).
- Also, the CTS demands that enumerated types derive from a common base class, System.Enum.

CTS Delegate Types

- *Delegates* are the .NET equivalent of a type-safe C-style function pointer.
- The key difference is that a .NET delegate is a *class* that derives from System.MulticastDelegate, rather than a simple pointer to a raw memory address.
- In C#, delegates are declared using the **delegate keyword**:

```
// This C# delegate type can 'point to' any method
// returning an integer and taking two integers as input.
public delegate int BinaryOp(int x, int y);
```

- Delegates are useful when you wish to provide a way for one entity to forward a call to another entity, and provide the foundation for the .NET event architecture
- Delegates have intrinsic support for multicasting (i.e., forwarding a request to multiple recipients) and asynchronous method invocations.

CTS Type Members

- A ***type member*** is constrained by the set {constructor, finalizer, static constructor, nested type, operator, method, property, indexer, field, read only field, constant, event}.
- For example,
  - Each member has a given visibility trait (e.g., public, private, protected, and so forth).
- Some members may be declared as abstract to enforce a polymorphic behavior on derived types as well as virtual to define a canned (but overridable) implementation.
- Also, most members may be configured as static (bound at the class level) or instance (bound at the object level).

Intrinsic CTS Data Types

Unique keyword used to declare an intrinsic CTS data type, all language keywords ultimately resolve to the same type defined in an assembly named **mscorlib.dll.**

The Intrinsic CTS Data Types

CTS Data Type	VB .NET Keyword	C# Keyword	Managed Extensions for C++ Keyword
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

Understanding the Common Language Specification

- Different languages express the same programming constructs in unique, language specific terms.
- For example, in C# you denote string concatenation using the plus operator (+), while in VB .NET you typically make use of the ampersand (&).

' VB .NET method returning nothing.

Public Sub MyMethod()

' Some interesting code...

End Sub

// C# method returning nothing.

public void MyMethod()

{ // **Some interesting code.. .** }

As .NET runtime, compilers (vbc.exe or csc.exe) emit a similar set of CIL instructions.

- ***The Common Language Specification (CLS)*** is a set of rules that describe in vivid detail the minimal and complete set of features a given .NET-aware compiler must support to produce code that can be hosted by the CLR, while at the same time be accessed in a uniform manner by all languages that target the .NET platform.
- The CLS is ultimately a set of rules that compiler builders must conform too thier function seamlessly within the .NET universe. Each rule is assigned a simple name (e.g.,"CLS Rule 6") and describes how this rule affects those who build the compilers as well as those who (in some way) interact with them.

Rule 1: CLS rules apply only to those parts of a type that are exposed outside the defining assembly.

The only aspects of a type that must conform to the CLS are the member definitions themselves (i.e., naming conventions, parameters, and return types).

To illustrate, the following Add() method is not CLS-compliant, as the parameters and return values make use of unsigned data (which is not a requirement of the CLS):

```
public class Calc
{
// Exposed unsigned data is not CLS compliant!
public ulong Add(ulong x, ulong y)
{ return x + y;}
}
```

However, if you were to simply make use of unsigned data internally as follows:

```
public class Calc
{
public int Add(int x, int y)
{
// As this ulong variable is only used internally,we are still CLS compliant.
ulong temp;
...
return x + y;
}
}
```

Ensuring CLS Compliance

Instruct c# compiler to check the code for CLS compliance using a single .NET attribute:

// Tell the C# compiler to check for CLS compliance.

[assembly: System.CLSCompliant(true)]

[CLSCompliant] attribute will instruct the C# compiler to check each and every line of code against the rules of the CLS. If any CLS violations are discovered, you receive a compiler error and a description of the offending code.

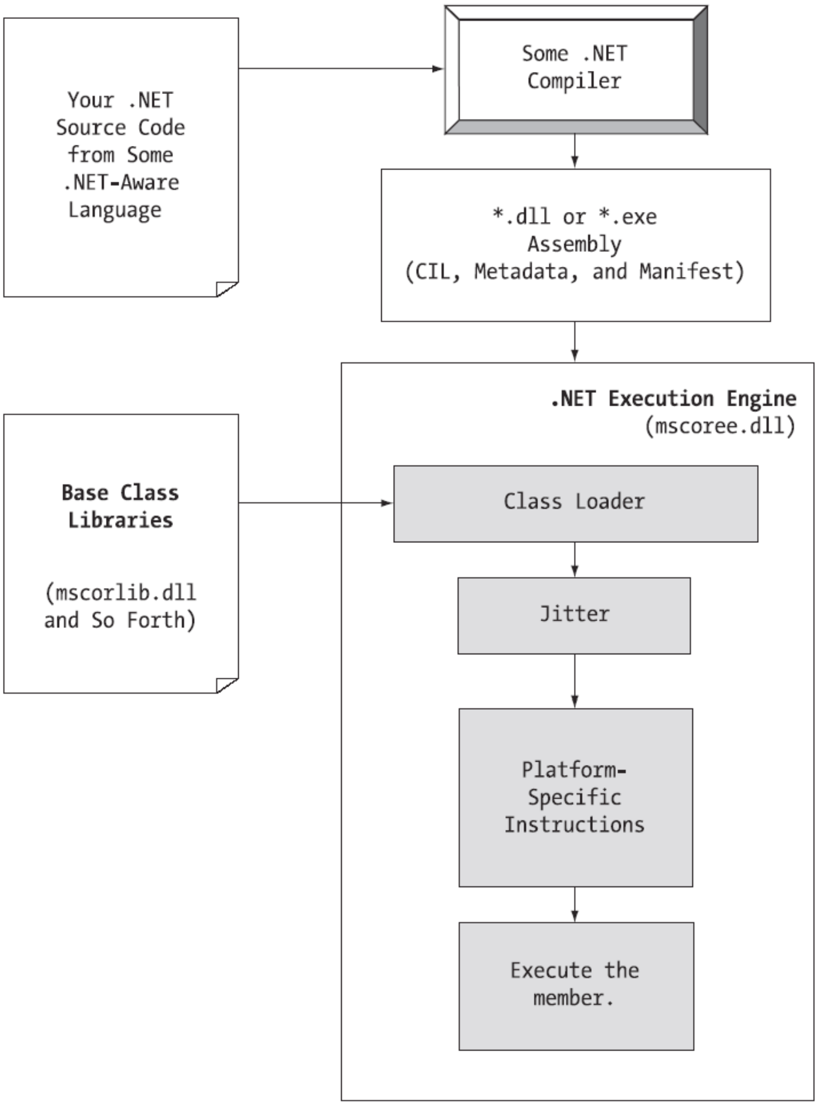
<u>Understanding the Common Language Runtime</u>

- .NET runtime provides a single well-defined runtime layer that is shared by *all* languages and platforms that are .NET-aware.
- The crux of the CLR is physically represented by a library named mscoree.dll (aka the Common Object Runtime Execution Engine). When an assembly is referenced for use, mscoree.dll is loaded automatically, which in turn loads the required assembly into memory. The runtime engine is responsible for a number of tasks.
- They are:

1. It is the entity in charge of resolving the location of an assembly and finding the requested type within the binary by reading the contained metadata.
2. The CLR then lays out the type in memory, compiles the associated CIL into platform-specific instructions, performs any necessary security checks, and then executes the code in question.
3. The CLR will also interact with the types contained within the .NET base class libraries when required. Although the entire base class library has been broken into a number of discrete assemblies, the key assembly is mscorlib.dll.

mscorlib.dll contains a large number of core types that encapsulate a wide variety of common programming tasks as well as the core data types used by all .NET languages. When you build .NET solutions, you automatically have access to this particular assembly.

The workflow that takes place between your source code (which is making use of base class library types), a given .NET compiler, and the .NET execution engine.



mscoree.dll in action

The Assembly/Namespace/Type Distinction

- A namespace is a grouping of related types contained in an assembly.
- Keep all the types within the base class libraries well organized, the .NET platform makes possible by of the *namespace* concept.
- For example,
  - The System.IO namespace contains file I/O related types;
  - The System.Data namespace defines basic database types
- Very important single assembly (such as mscorlib.dll) can contain any number of namespaces, each of which can contain any number of types.

- The key difference between this approach and a language-specific library such as MFC is that any language targeting the .NET runtime makes use of the *same* namespaces and *same* types.
- For example, the following three programs all illustrate the "Hello World" application, written in C#, VB .NET, and Managed Extensions for C++:

```
// Hello world in C#
using System;
public class MyApp
{
static void Main()
{
Console.WriteLine("Hi from C#");
}
}
```

```
' Hello world in VB .NET
Imports System
Public Module MyApp
Sub Main()
Console.WriteLine("Hi from VB .NET")
End Sub
End Module
```

```
// Hello world in Managed Extensions for C++
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
Console::WriteLine(L"Hi from managed C++");
return 0;
}
```

- Notice that each language is making use of the **Console** class defined in the **System namespace**.

- Beyond minor syntactic variations, these three applications look and feel very much alike, both physically and logically.
- The most fundamental namespace to get your hands around is named System. This namespace provides a core body of types that you will need to leverage time and again as a .NET developer.
- In below table we shown many namespaces which are used commonly.

A Sampling of .NET Namespaces

.NET Namespace	Meaning in Life
`System`	Within `System` you find numerous useful types dealing with intrinsic data, mathematical computations, random number generation, environment variables, and garbage collection, as well as a number of commonly used exceptions and attributes.
`System.Collections` `System.Collections.Generic`	These namespaces define a number of stock container objects (`ArrayList`, `Queue`, and so forth), as well as base types and interfaces that allow you to build customized collections. As of .NET 2.0, the collection types have been extended with generic capabilities.

`System.Data`
`System.Data.Odbc`
`System.Data.OracleClient`
`System.Data.OleDb`
`System.Data.SqlClient`
	These namespaces are used for interacting with databases using ADO.NET.

`System.Diagnostics`	Here, you find numerous types that can be used to programmatically debug and trace your source code.

`System.Drawing`
`System.Drawing.Drawing2D`
`System.Drawing.Printing`
	Here, you find numerous types wrapping graphical primitives such as bitmaps, fonts, and icons, as well as printing capabilities.

`System.IO`
`System.IO.Compression`
`System.IO.Ports`
	These namespaces include file I/O, buffering, and so forth. As of .NET 2.0, the `IO` namespaces now include support compression and port manipulation.

`System.Net`	This namespace (as well as other related namespaces) contains types related to network programming (requests/responses, sockets, end points, and so on).
`System.Reflection` `System.Reflection.Emit`	These namespaces define types that support runtime type discovery as well as dynamic creation of types.
`System.Runtime.` `InteropServices`	This namespace provides facilities to allow .NET types to interact with "unmanaged code" (e.g., C-based DLLs and COM servers) and vice versa.
`System.Runtime.Remoting`	This namespace (among others) defines types used to build solutions that incorporate the .NET remoting layer.
`System.Security`	Security is an integrated aspect of the .NET universe. In the security-centric namespaces you find numerous types dealing with permissions, cryptography, and so on.
`System.Threading`	This namespace defines types used to build multithreaded applications.
`System.Web`	A number of namespaces are specifically geared toward the development of .NET web applications, including ASP.NET and XML web services.
`System.Windows.Forms`	This namespace contains types that facilitate the construction of traditional desktop GUI applications.
`System.Xml`	The XML-centric namespaces contain numerous types used to interact with XML data.

Accessing a Namespace Programmatically

- Consider System namespace assume that System.Console represents a class named *Console* that is contained within a namespace called *System*.
- In C#, the using keyword simplifies the process of referencing types defined in a particular namespace.
- The main window renders a bar chart based on some information obtained from a back-end database and displays your company logo, for this we need namespaces.

```
// Here are all the namespaces used to build this application.
using System; // General base class library types.
using System.Drawing;            // Graphical rendering types.
using System.Windows.Forms;      // GUI widget types.
using System.Data;               // General data-centric types.
using System.Data.SqlClient;     // MS SQL Server data access types.
```

- Once you have specified some number of namespaces (and set a reference to the assemblies that define them), you are free to create instances of the types they contain.
- For example, an instance of the Bitmap class created as (defined in the System.Drawing namespace), you can write:

```
// Explicitly list the namespaces used by this file.
using System;
using System.Drawing;
class MyApp
{
public void DisplayLogo()
{ // Create a 20_20 pixel bitmap.
Bitmap companyLogo = new Bitmap(20, 20); …
}
}
```

- Because your application is referencing System.Drawing, the compiler is able to resolve the Bitmap class as a member of this namespace. If you did not specify the System.Drawing namespace, you would be issued a compiler error.

You can declare variables using a *fully qualified name* as well:

```
// Not listing System.Drawing namespace!
using System;
```

```
class MyApp
{
public void DisplayLogo()
{
// Using fully qualified name.
System.Drawing.Bitmap companyLogo =
new System.Drawing.Bitmap(20, 20);
...
}
}
```

- While defining a type using the fully qualified name provides greater readability, I think you'd agree that the C# using keyword reduces keystrokes.
- However, always remember that this technique is simply a shorthand notation for specifying a type's fully qualified name, and each approach results in the *exact* same underlying CIL (given the fact that CIL code always makes use of fully qualified names) and has no effect on performance or the size of the assembly.

Referencing External Assemblies

In addition to specifying a namespace via the C# using keyword, you also need to tell the C# compiler the name of the assembly containing the actual CIL definition for the referenced type.

A vast majority of the .NET Framework assemblies are located under a specific directory termed the ***global assembly cache*** (***GAC***).

On a Windows machine, this can be located under **%windir%\Assembly.**

Deploying the .NET Runtime

However, if you deploy an assembly to a computer that does not have .NET installed, it will fail to run. For this reason, Microsoft provides a setup package named dotnetfx.exe that can be freely shipped and installed along with your custom software. This installation program is included with the .NET Framework 2.0 SDK, and it is also freely downloadable from Microsoft.

Once dotnetfx.exe is installed, the target machine will now contain the .NET base class libraries, .NET runtime (mscoree.dll), and additional .NET infrastructure (such as the GAC).

Note Do be aware that if you are building a .NET web application, the end user's machine does not need to be configured with the .NET Framework, as the browser will simply receive generic HTML and possibly client-side JavaScript.

IMPORTANT QUESTIONS

1. Briefly explain the history of .NET. Explain the building components of .NET and their responsibilities. 6M
2. What are the building blocks of .NET frame work? Show their relationship, with a neat block diagram. Explain CTS in detail. 10M
3. Explain features and building blocks of .NET framework. 10M
4. What are the building blocks of .NET platform? Give the relationship between .NET runtime layer and the base class Library. 8M
5. Explain Jitter, along with its benefits. Explain how CLR host an application on .NET platform .Give the block diagram. 6M
6. What is an assembly? Explain each component of an assembly. Differentiate b/n singlefile assembly and multifile assembly. 8M
7. What is .NET assembly? What does it contain? Explain each of them. 10M
8. Write a note on .NET Namespace . 4M
9. Explain the role of the common intermediate Language. 6M
10. Explain the limitations and complexities found within the technologies prior to .NET. Briefly explain how .NET attempts to simplify the same. 10M
11. Explain the formal definitions of all possible CTS types. 10M
12. What is the role of .NET type Metadata? Give an example. 4M
13. Explain the CLR. Illustrate the workflow that takes place b/n the source code, given .NET compiler and the .NET execution engine. 8M