

# it courseware™

*TRAINING MATERIALS FOR IT PROFESSIONALS*

.NET Framework  
Using C#

# **.NET Framework Using C#**

*Student Guide*

**Revision 4.8.5**

# **.NET Framework Using C#**

## **Rev. 4.8.5**

### **Student Guide**

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.



™ is a trademark of Object Innovations.

**Author:** Robert J. Oberg

**Special Thanks:** Michael Stiefel, Peter Thorsteinson, Dana Wyatt, Paul Nahay, Ed Strassberger, Robert Seitz, Sharman Staples, Matthew Donaldson

Copyright ©2019 Object Innovations Enterprises, LLC All rights reserved.

Object Innovations  
877-558-7246  
[www.objectinnovations.com](http://www.objectinnovations.com)

Published in the United States of America.

## Table of Contents (Overview)

Chapter 1	.NET Fundamentals
Chapter 2	Class Libraries
Chapter 3	Assemblies, Deployment and Configuration
Chapter 4	Metadata and Reflection
Chapter 5	I/O and Serialization
Chapter 6	.NET Programming Model
Chapter 7	.NET Threading
Chapter 8	.NET Security
Chapter 9	Interoperating with COM and Win32
Chapter 10	ADO.NET and LINQ
Chapter 11	Debugging Fundamentals
Chapter 12	Tracing
Chapter 13	More about Tracing
Appendix A	.NET Remoting
Appendix B	Learning Resources

# Directory Structure

---

- **The course software installs to the root directory *C:\OIC\NetCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for hand-on work during lectures.
- **Data files install to the directory *C:\OIC\Data*.**
- **The directory *C:\OIC\Deploy* is provided to practice deployment.**

# Table of Contents (Detailed)

<b>Chapter 1 .NET Fundamentals.....</b>	<b>1</b>
What Is Microsoft .NET?.....	3
Open Standards and Interoperability .....	4
Windows Development Problems.....	5
Common Language Runtime .....	6
CLR Serialization Example .....	7
Attribute-Based Programming.....	10
Metadata.....	11
Types.....	12
.NET Framework Class Library.....	13
Interface-Based Programming .....	14
Everything is an Object.....	15
Common Type System.....	16
ILDASM .....	17
.NET Framework SDK Tools .....	19
Fixing the PATH Variable .....	20
Language Interoperability.....	21
Managed Code .....	22
Assemblies .....	23
Assembly Deployment.....	24
JIT Compilation .....	25
ASP.NET and Web Services.....	26
The Role of XML.....	27
Performance .....	28
.NET Native .....	29
.NET Core .....	30
.NET Frameworks.....	31
XML Serialization Example .....	32
Summary .....	34
<b>Chapter 2 Class Libraries .....</b>	<b>35</b>
Objects and Components .....	37
Limitation of COM Components .....	38
Components in .NET .....	39
Class Libraries at the Command Line.....	40
Component Example: Customer Management System .....	41
Monolithic versus Component.....	43
Demo: Creating a Class Library .....	44
Demo: A Console Client Program .....	46
Class Libraries Using Visual Studio.....	47
Demo: Class Library in Visual Studio.....	48
Project References in Visual Studio .....	51

References at Compile Time and Run Time .....	54
Project Dependencies .....	55
Specifying Version Numbers .....	56
Lab 2 .....	57
Summary .....	58
<b>Chapter 3 Assemblies, Deployment and Configuration .....</b>	<b>61</b>
Assemblies .....	63
Customer Management System .....	64
ILDASM .....	65
Assembly Manifest .....	66
Assembly Dependency Metadata .....	67
Assembly Metadata .....	68
Versioning an Assembly .....	69
AssemblyVersion Attribute .....	70
Strong Names .....	71
Digital Signatures .....	72
Verification with Digital Signatures .....	73
Hash Codes .....	74
Digitally Signing an Assembly .....	75
Digital Signing Flowchart .....	76
Signing the Customer Assembly .....	77
Signed Assembly Metadata .....	78
Private Assembly Deployment .....	79
Assembly Cache .....	80
Deploying a Shared Assembly .....	81
Signed Assembly Demo .....	82
Versioning Shared Components .....	84
How the CLR Locates Assemblies .....	85
Resolving an Assembly Reference .....	86
Version Policy in a Configuration File .....	87
Finding the Assembly .....	88
Lab 3A .....	89
Application Settings .....	90
Application Settings Using Visual Studio .....	91
Application Settings Demo .....	92
Application Configuration File .....	97
User Configuration File .....	98
Lab 3B .....	99
Summary .....	100
<b>Chapter 4 Metadata and Reflection .....</b>	<b>107</b>
Metadata .....	109
Reflection .....	110
Sample Reflection Program .....	111
System.Reflection.Assembly .....	114

System.Type.....	115
System.Reflection.MethodInfo .....	117
Dynamic Invocation.....	118
Late Binding .....	119
LateBinding Example .....	120
Lab 4 .....	121
Summary .....	122
<b>Chapter 5 I/O and Serialization .....</b>	<b>127</b>
Input and Output in .NET .....	129
Directories.....	130
Directory Example Program .....	131
Files and Streams .....	134
“Read” Command .....	136
Code for “Write” Command .....	137
Serialization .....	138
Attributes .....	139
Serialization Example .....	141
Lab 5 .....	144
Summary .....	145
<b>Chapter 6 .NET Programming Model .....</b>	<b>147</b>
Garbage Collection .....	149
Finalize Method .....	150
C# Destructor Notation .....	151
Dispose.....	152
Finalize/Dispose Example .....	153
Finalize/Dispose Test Program.....	155
Garbage Collection Performance.....	157
Generations .....	158
Processes .....	159
Threads.....	160
Asynchronous Calls .....	161
Asynchronous Delegates.....	162
Using a Callback Method.....	163
BackgroundWorker.....	166
Asynchronous Programs in C# .....	167
Task and Task<TResult> .....	168
Aysnc Methods .....	169
New Async Example.....	170
Synchronous Call .....	171
Async Call.....	172
Threading .....	173
Lab 6A .....	174
Lab 6B.....	175
Application Isolation.....	176

Application Domain.....	177
Application Domains and Assemblies .....	178
AppDomain.....	179
CreateDomain .....	180
App Domain Events .....	181
Lab 6C.....	182
Summary .....	183
<b>Chapter 7 .NET Threading .....</b>	<b>191</b>
Threads.....	193
.NET Threading Model .....	194
Console Log Example.....	195
Race Conditions.....	199
Race Condition Example .....	200
Thread Synchronization.....	204
Monitor .....	205
Monitor Example .....	206
Using C# <i>lock</i> Keyword.....	207
Synchronization of Collections.....	208
ThreadPool Class .....	209
ThreadPool Example.....	210
Starting a ThreadPool Thread .....	211
Foreground and Background Threads.....	212
Synchronizing Threads .....	213
Improved ThreadPool Example .....	214
Task Parallel Library (TPL).....	216
Task Example.....	217
Starting Tasks .....	218
Waiting for Task Completion .....	219
Data Parallelism.....	220
Lab 7 .....	221
Summary .....	222
<b>Chapter 8 .NET Security .....</b>	<b>231</b>
Fundamental Problem of Security .....	233
Authentication.....	234
Authorization .....	235
The Internet and .NET Security .....	236
Code Access Security .....	237
Role-Based Security .....	238
.NET Security Concepts .....	239
Permissions .....	240
IPermission Interface .....	241
IPermission Demand Method .....	242
IPermission Inheritance Hierarchy .....	243
Stack Walking.....	244

Assert .....	245
Demand .....	246
Other CAS Methods.....	247
Security Policy Simplification .....	248
Simple Sandboxing API.....	249
Sandbox Example .....	250
Setting up Permissions.....	251
Creating the Sandbox.....	252
Role-Based Security in .NET.....	253
Identity Objects.....	254
Principal Objects.....	255
Windows Principal Information.....	256
Custom Identity and Principal .....	258
BasicIdentity.cs.....	259
BasicSecurity.cs.....	260
Users.cs.....	263
Roles.cs .....	265
RoleDemo.cs.....	267
Sample Run.....	268
PrincipalPermission .....	269
Lab 8 .....	270
Summary .....	271
<b>Chapter 9 Interoperating with COM and Win32 .....</b>	<b>275</b>
Interoperating Between Managed and Unmanaged Code .....	277
COM Interop and PInvoke.....	278
Calling COM Components from Managed Code .....	279
The TlbImp.exe Utility .....	280
TlbImp Syntax .....	281
Using TlbImp.....	282
Demonstration: Wrapping a Legacy COM Server.....	283
Register the COM Server.....	285
OLE/COM Object Viewer .....	286
64-bit System Considerations .....	287
Run the COM Client .....	288
Implement the .NET Client Program .....	290
The Client Target Platform Is 32-bit.....	292
Import a Type Library Using Visual Studio .....	294
Platform Invocation Services (PInvoke) .....	296
A Simple Example .....	297
Marshalling <i>out</i> Parameters .....	299
Translating Types .....	301
Lab 9 .....	303
Summary .....	304

<b>Chapter 10 ADO.NET and LINQ.....</b>	<b>307</b>
ADO.NET .....	309
ADO.NET Architecture .....	310
.NET Data Providers .....	312
ADO.NET Interfaces .....	313
.NET Namespaces .....	314
Connected Data Access .....	315
AcmePub Database .....	316
Creating a Connection .....	317
SQL Express LocalDB.....	318
SqlLocalDB Utility.....	319
Using Server Explorer .....	320
Performing Queries .....	321
Connecting to a Database .....	322
Database Code .....	323
Connection String .....	325
Using Commands.....	326
Creating a Command Object.....	327
Using a Data Reader .....	328
Data Reader: Code Example.....	329
Generic Collections.....	330
Executing Commands .....	331
Parameterized Queries .....	332
Parameterized Query Example .....	333
Lab 10A .....	334
DataSet.....	335
DataSet Architecture.....	336
Why DataSet? .....	337
DataSet Components.....	338
DataAdapter .....	339
DataSet Example Program.....	340
Data Access Class .....	341
Retrieving the Data .....	342
Filling a DataSet .....	343
Accessing a DataSet.....	344
Using a Standalone DataTable.....	345
DataTable Update Example .....	346
Adding a New Row.....	349
Searching and Updating a Row .....	350
Deleting a Row .....	351
Row Versions.....	352
Row State .....	353
Iterating Through DataRows .....	354
Command Builders .....	355
Updating a Database .....	356

Data Binding .....	357
DataGridView Control.....	358
Language Integrated Query (LINQ) .....	359
LINQ to ADO.NET .....	360
Bridging Objects and Data.....	361
LINQ Demo .....	362
Object Relational Designer .....	363
IntelliSense.....	365
Basic LINQ Query Operators .....	366
Obtaining a Data Source .....	367
LINQ Query Example.....	368
Filtering.....	369
Ordering .....	370
Aggregation .....	371
Obtaining Lists and Arrays .....	372
Deferred Execution .....	373
Modifying a Data Source.....	374
Performing Inserts via LINQ to SQL.....	375
Performing Deletes via LINQ to SQL .....	376
Performing Updates via LINQ to SQL.....	377
Lab 10B.....	378
Summary .....	379
<b>Chapter 11 Debugging Fundamentals .....</b>	<b>389</b>
Compile-Time Errors .....	391
Compile-Time Demo .....	392
Runtime Errors.....	393
Debugging.....	394
Bytes Sample Program.....	395
Project Configurations .....	396
Release Configuration.....	397
Creating a New Configuration .....	398
Build Settings for a Configuration.....	399
Customizing a Toolbar.....	401
Using the Visual Studio Debugger .....	404
Overflow Exception.....	405
Just-in-Time Debugging .....	406
Attach to Running Process.....	409
Standard Debugging – Breakpoints .....	411
Standard Debugging – Watch Variables.....	412
Stepping with the Debugger .....	413
Demo: Stepping with the Debugger.....	414
The Call Stack.....	415
Lab 11 .....	416
Summary .....	417

<b>Chapter 12 Tracing.....</b>	<b>419</b>
Instrumenting an Application .....	421
Order Application .....	422
Debugging Review.....	424
Tracing .....	425
Debug and Trace Classes .....	426
Tracing Example .....	427
Viewing Trace Output .....	428
Debug Statements .....	429
Debug Output.....	430
Assert .....	431
More Debug Output .....	432
WriteLine Syntax .....	433
Lab 12 .....	434
Event Logs .....	435
Viewing Event Logs .....	436
Event Log Entry Types .....	437
.NET EventLog Component .....	438
Quick EventLog Demo .....	439
Full-Blown EventLog Demo.....	442
Viewing Custom Log in Visual Studio .....	443
Retrieving Entries from an Event Log .....	444
DisplayEventLog Sample Program.....	445
Summary .....	446
<b>Chapter 13 More about Tracing.....</b>	<b>451</b>
Trace Switches .....	453
BooleanSwitch .....	454
Sample Program.....	455
Using a Configuration File .....	456
TraceSwitch .....	457
SwitchDemo.....	458
Trace Listeners.....	461
DefaultTraceListener .....	462
Listener Example Program .....	463
A Stream Listener .....	464
A Custom Listener .....	465
Trace Output to a Window.....	466
Summary .....	467
<b>Appendix A .NET Remoting .....</b>	<b>469</b>
Distributed Programming in .NET.....	471
Windows Communication Foundation .....	472
.NET Remoting Architecture .....	473
Remote Objects and Mobile Objects .....	475
Object Activation and Lifetime .....	476

Singleton and SingleCall .....	477
.NET Remoting Example.....	478
.NET Remoting Example: Defs .....	479
.NET Remoting Example: Server .....	480
.NET Remoting Example: Client.....	482
Lab A .....	484
Summary .....	485

<b>Appendix B Learning Resources .....</b>	<b>489</b>
--	------------

EVALUATION COPY  
Unauthorized Reproduction or Distribution Prohibited

# Chapter 1

## **.NET Fundamentals**

EVALUATION COPY  
Unauthorized Reproduction or Distribution Prohibited

# **.NET Fundamentals**

## **Objectives**

---

*After completing this unit you will be able to:*

- **Understand the problems Microsoft .NET is designed to solve.**
- **Understand the basic programming model of Microsoft .NET.**
- **Understand the basic programming tools provided by Microsoft .NET.**
- **Discuss .NET Native, .NET Core and cross-platform development.**

# What Is Microsoft .NET?

---

- **Microsoft .NET was developed to solve three fundamental problems.**
- **First, the Microsoft Windows programming model must be unified to remove the widely varied programming models and approaches that exist among the various Microsoft development technologies.**
- **Second, Microsoft based solutions must be capable of interacting with the modern world of heterogeneous computing environments.**
- **Third, Microsoft needs a development paradigm that is capable of being expanded to encompass future development strategies, technologies, and customer demands.**

# Open Standards and Interoperability

---

- **The modern computing environment contains a vast variety of hardware and software systems.**
  - Computers range from mainframes and high-end servers, to workstations and PCs, and to small mobile devices such as PDAs and cell phones.
  - Operating systems include traditional mainframe systems, many flavors of Unix including Android, Linux, Apple's iOS, several versions of Windows, real-time systems and more.
  - Many different languages, databases, application development tools and middleware products are used.
- **Applications need to be able to work in this heterogeneous environment.**
  - Even shrink-wrapped applications deployed on a single PC may use the Internet for registration and updates.
- **The key to interoperability among applications is the use of *standards*, such as HTML, HTTP, XML, SOAP, and TCP/IP.**

# Windows Development Problems

---

- **In classic Windows development design and language choice often clashed.**
  - Visual Basic vs. C++ approach
  - IDispatch, Dual, or Vtable interfaces
  - VB vs. MFC
  - ODBC or OLEDB or ADO
- **Application deployment was hard.**
  - Critical entries in Registry for COM components
  - No versioning strategy
  - DLL Hell
- **Security was difficult to implement.**
  - No way to control code or give code rights to certain actions and deny it the right to do other actions.
  - Security model is difficult to understand. Did you ever pass anything but NULL to a LPSECURITY\_ATTRIBUTES argument?
- **Too much time is spent in writing plumbing code that the system should provide.**
  - MTS/COM+ was a step in the right direction.

# Common Language Runtime

---

- **The first step in solving the three fundamental problems is for Microsoft .NET to provide a set of underlying services available to all languages.**
- **The runtime environment provided by .NET that provides these services is called the *Common Language Runtime* or CLR.**
  - A runtime provides services to executing programs.
  - Traditionally there are different runtimes for different programming environments. Examples of runtimes include the standard C library, MFC, the Visual Basic 6 runtime and the Java Virtual Machine.
- **These services are available to all languages that follow the rules of the CLR.**
  - C# and Visual Basic are examples of Microsoft languages that are fully compliant with the CLR requirements.
  - Not all languages use all the features of the CLR.
- **As a terminology note, beginning with .NET 2.0, Microsoft has dropped the “.NET” in the Visual Basic language.**
  - The pre-.NET version of the language is now referred to as Visual Basic 6 or VB6.

# CLR Serialization Example

---

- **Let us use serialization to illustrate how the CLR provides a set of services that unifies the Microsoft development paradigm.**
  - Every programmer has to do it.
  - It can get complicated with nested objects, complicated data structures, and a variety of data storages.
  - The programmer should also be able to override the system service if necessary.
- **See the *Serialize* example in this chapter.**

## Serialization Example (Cont'd)

---

- **Ignore the language details covered in a later chapter.**

```
[Serializable]
class Customer
{
    public string name;
    public long id;
}
class Test
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();

        Customer cust = new Customer();
        cust.name = "Charles Darwin";
        cust.id = 10;
        list.Add(cust);

        cust = new Customer();
        cust.name = "Isaac Newton";
        cust.id = 20;
        list.Add(cust);

        foreach (Customer x in list)
            Console.WriteLine(x.name + ":" + x.id);

        Console.WriteLine("Saving Customer List");
        FileStream s = new FileStream("cust.txt",
                                     FileMode.Create);
        SoapFormatter f = new SoapFormatter();
        SaveFile(s, f, list);
    }
}
```

## Serialization Example (Cont'd)

---

```
Console.WriteLine("Restoring to New List");
s = new FileStream("cust.txt",
    FileMode.Open);
f = new SoapFormatter();
ArrayList list2 =
    (ArrayList)RestoreFile(s, f);

foreach (Customer y in list2)
    Console.WriteLine(y.name + ": " + y.id);
}

public static void SaveFile(Stream s,
    IFormatter f, IList list)
{
    f.Serialize(s, list);
    s.Close();
}

public static IList RestoreFile(Stream s,
    IFormatter f)
{
    IList list = (IList)f.Deserialize(s);
    s.Close();
    return list;
}
}
```

# Attribute-Based Programming

---

- **We add two Customer objects to the collection, and print them out. We save the collection to disk and then restore it. The identical list is printed out.**

```
Charles Darwin: 10  
Isaac Newton: 20  
Saving Customer List  
Restoring to New List  
Charles Darwin: 10  
Isaac Newton: 20  
Press enter to continue...
```

- **We wrote no code to save or restore the list!**
  - We just annotated the class we wanted to save with the **Serializable** attribute.
  - We specified the format (SOAP) that the data was to be saved.
  - We specified the medium (disk) where the data was saved.
  - This is typical class partitioning in the .NET Framework.
- **Attribute-based programming is used throughout .NET to describe how code and data should be treated by the framework.**

# Metadata

---

- The compiler adds the *Serializable* attribute to the *metadata* of the Customer class.
- Metadata provides the Common Language Runtime with information it needs to provide services to the application.
  - Version and locale information
  - All the types
  - Details about each type, including name, visibility, etc.
  - Details about the members of each type, such as methods, the signatures of methods, etc.
  - Attributes
- Metadata is stored with the application so that .NET applications are self-describing. The registry is not used.
  - The CLR can query the metadata at runtime. It can see if the **Serializable** attribute is present. It can find out the structure of the Customer object in order to save and restore it.

# Types

---

- ***Types* are at the heart of the programming model for the CLR.**
  - Most of the **metadata** is organized by type.
- **A type is analogous to a class in most object-oriented programming languages, providing an abstraction of data and behavior, grouped together.**
- **A type in the CLR contains:**
  - Fields (data members)
  - Methods
  - Properties
  - Events (which are now full fledged members of the Microsoft programming paradigm).

# NET Framework Class Library

---

- **The *SoapFormatter* and *FileStream* classes are two of the thousands of classes in the .NET Framework that provide system services.**
- **The functionality provided includes:**
  - Base Class Library (basic functionality such as strings, arrays and formatting).
  - Networking
  - Security
  - Remoting
  - Diagnostics
  - I/O
  - Database
  - XML
  - Web Services
  - Web programming
  - Windows User Interface
- **This framework is usable by all CLR compliant languages.**

# Interface-Based Programming

---

- **Interfaces allow you to work with abstract types in a way that allows for extensible programming.**
- **The *SaveFile* and *RestoreFile* routines are written using the *IList* and *IFormatter* interfaces.**
  - **These routines will work with all the collection classes that support the *IList* interface, and the formatters that support the *IFormatter* interface.**
- **Implementation inheritance permits code reuse.**
- **You can implement the *ISerializable* interface to override the framework's implementation.**
  - The metadata for the type tells the framework that the class has implemented the interface.
- **Interface-based programming allows classes to provide implementations of standard functionality that can be used by the framework.**

# Everything is an Object

---

- **Every type in .NET derives from *System.Object*.<sup>1</sup>**
- **Every type, system or user defined, has metadata.**
  - In the sample the framework can walk through the ArrayList of Customer objects and save each one as well as the array itself.
- **All access to objects in .NET is through object references.**

---

<sup>1</sup> An exception is the pointer type, which is rarely used in C#.

# Common Type System

---

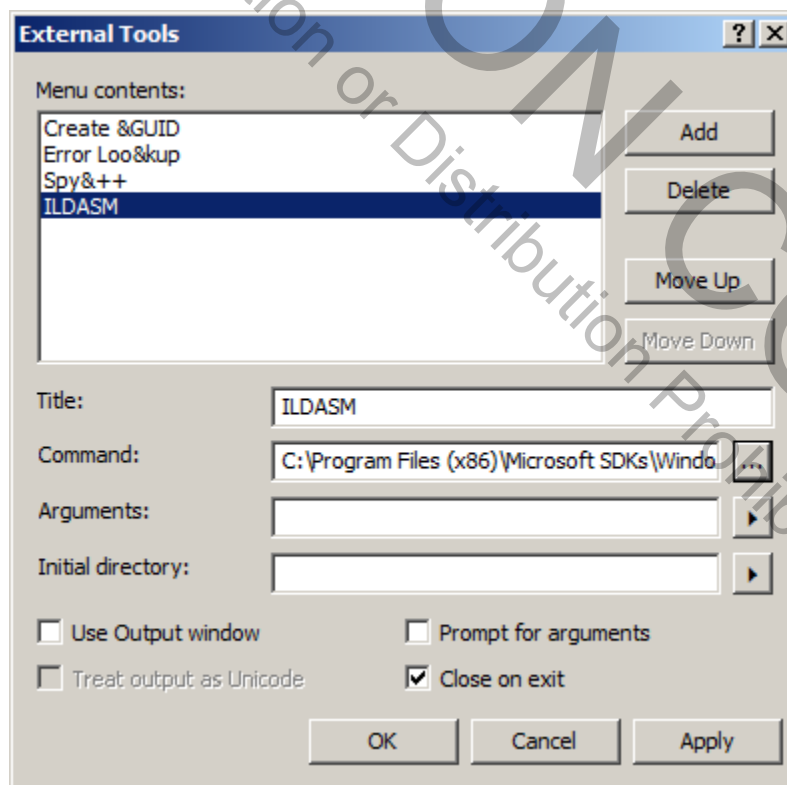
- **The Common Type System (CTS) defines the rules for the types and operations that the CLR will support.**
  - The CTS limits .NET classes to single implementation inheritance.
  - The CTS is designed for a wide range of languages, not all languages will support all features of the CTS.
- **The CTS makes it *possible* to guarantee type safety.**
  - Access to objects can be restricted to object references (no pointers), each reference refers to a defined memory. Access to that layout is only through public methods and fields.
  - By performing a local analysis of the class, you can verify to make sure that the code does not perform any inappropriate memory access. You do not have to analyze the users of the class.
- **.NET compilers emit Common Intermediate Language<sup>2</sup> (CIL) not native code.**
  - CIL is platform independent.
  - Type-safe code can be restricted to a subset of verifiable CIL expressions.
  - Once code is verified, it is verified for all platforms.

---

<sup>2</sup> Formerly called Microsoft Intermediate Language or MSIL.

# ILDASM

- **The Microsoft Intermediate Language Disassembler (ILDASM) can display the metadata and CIL instructions associated with .NET code.**
  - It is a very useful tool both for debugging and increasing your understanding of the .NET infrastructure.
- **You may wish to add ILDASM to your Tools menu in Visual Studio 2019.**
  - Use the command Tools | External Tools. Click the Add button, enter ILDASM for the Title, and click the ... button to navigate to the folder \Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7.2 Tools.



## ILDASM (Cont'd)

- You can use ILDASM to examine the .NET framework code.
  - Here is a fragment of the CIL from the **Serialize** example.

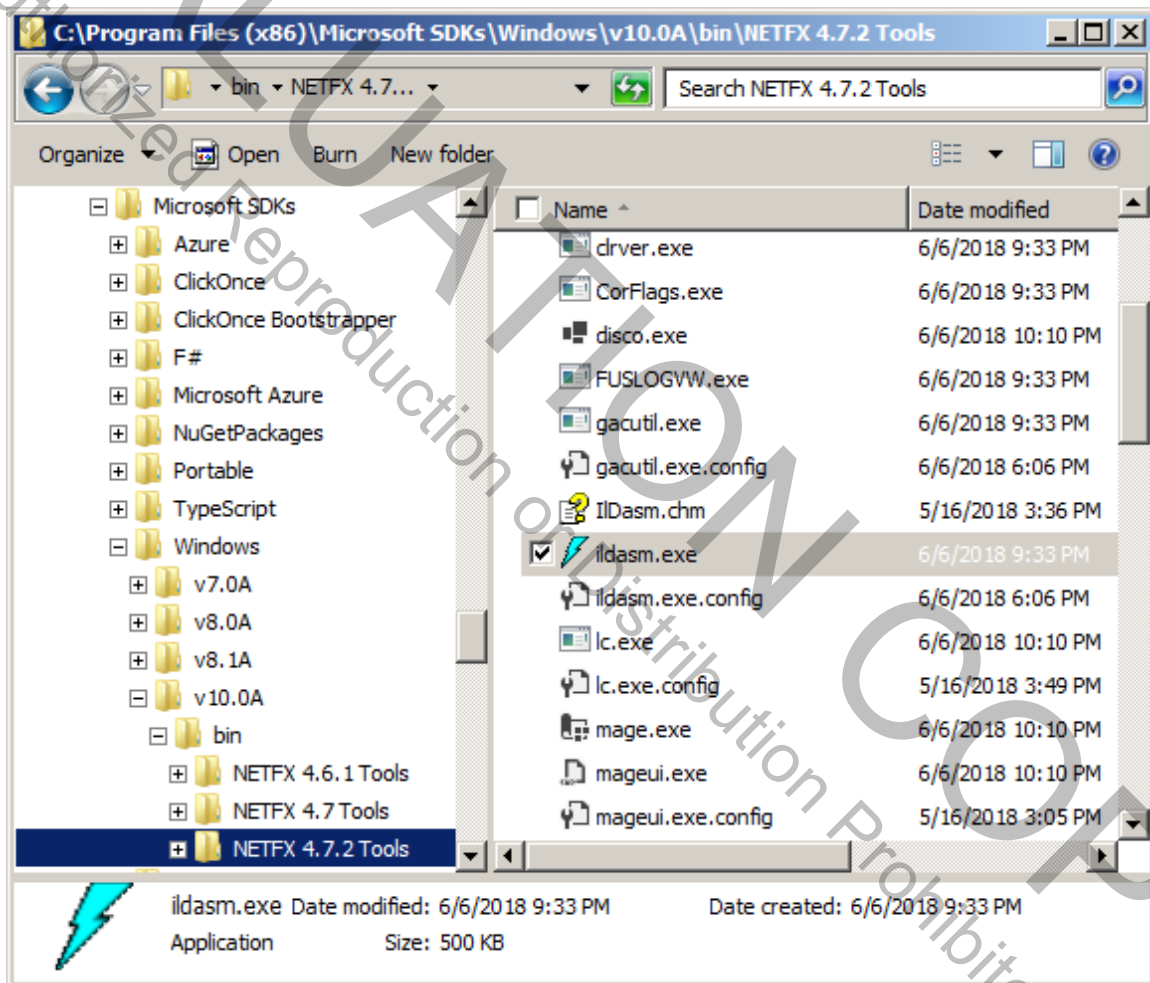
```

Test::Main : void(string[])
Find Find Next
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      344 (0x158)
    .maxstack 3
    .locals init ([0] class [mscorlib]System.Collections.ArrayList list,
        [1] class Customer cust,
        [2] class [mscorlib]System.IO.FileStream s,
        [3] class [System.Runtime.Serialization.Formatters.Soap]System.Runt
        [4] class [mscorlib]System.Collections.ArrayList list2,
        [5] class [mscorlib]System.Collections.IEnumerator U_5,
        [6] class Customer x,
        [7] class [mscorlib]System.IDisposable U_7,
        [8] class [mscorlib]System.Collections.IEnumerator U_8,
        [9] class Customer y)
    IL_0000: nop
    IL_0001: newobj      instance void [mscorlib]System.Collections.ArrayList::..
    IL_0006: stloc.0
    IL_0007: newobj      instance void Customer::.ctor()
    IL_000c: stloc.1
    IL_000d: ldloc.1
    IL_000e: ldstr       "Charles Darwin"
    IL_0013: stfld      string Customer::name
    IL_0018: ldloc.1
    IL_0019: ldc.i4.s    10
    IL_001b: conv.i8
    IL_001c: stfld      int64 Customer::id
    IL_0021: ldloc.0
    IL_0022: ldloc.1
    IL_0023: callvirt   instance int32 [mscorlib]System.Collections.ArrayList::..
    IL_0028: pop
    IL_0029: newobj      instance void Customer::.ctor()
    IL_002e: stloc.1
    IL_002f: ldloc.1
    IL_0030: ldstr       "Isaac Newton"
    IL_0035: stfld      string Customer::name
    IL_003a: ldloc.1
    IL_003b: ldc.i4.s    20
    IL_003d: conv.i8
    IL_003e: stfld      int64 Customer::id

```

## .NET Framework SDK Tools

- **Installing Visual Studio 2019 will also install the .NET Framework 4.7.2 SDK, version 10.0A.**
  - These tools are located in folder \Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7.2 Tools.



- They can be run at the command line from the Visual Studio 2019 Command Prompt<sup>3</sup>, which can be started from All Programs | Visual Studio 2019 | Visual Studio Tools | Developer Command Prompt for VS2019.

<sup>3</sup> You may need to run the Command Prompt as Administrator in some cases.

## Fixing the PATH Variable

---

- In case you are unable to run the .NET4.7.2 Tools from the Developer Command Prompt, you can explicitly add the proper folder to the Windows PATH environment variable:

```
PATH=%PATH%;C:\Program Files (x86)\Microsoft  
SDKs\Windows\v10.0A\bin\NETFX 4.7.2 Tools
```

- We provide the batch file **FixPath.bat** to do this for you.
- It is located in **C:\OIC\NetCs**.

# Language Interoperability

---

- **Having all language compilers use a common intermediate language and common base class makes it *possible* for languages to interoperate.**
  - All languages need not implement all parts of the CTS.
  - One language can have a feature that another does not.
- **The *Common Language Specification* (CLS) defines a subset of the CTS that represents the basic functionality that all .NET languages should implement if they are to interoperate with each other.**
  - For example, a class written in Visual Basic can inherit from a class written in C#.
  - Interlanguage debugging is possible.
  - CLS rule: Method calls need not support a variable number of arguments even though such a construct can be expressed in CIL.
  - CLS prohibits the use of pointers.
- **CLS compliance only applies to public features.**
  - C# code should not define public and protected class names that differ only by case sensitivity since languages as Visual Basic are not case sensitive. Private C# fields could have such names.

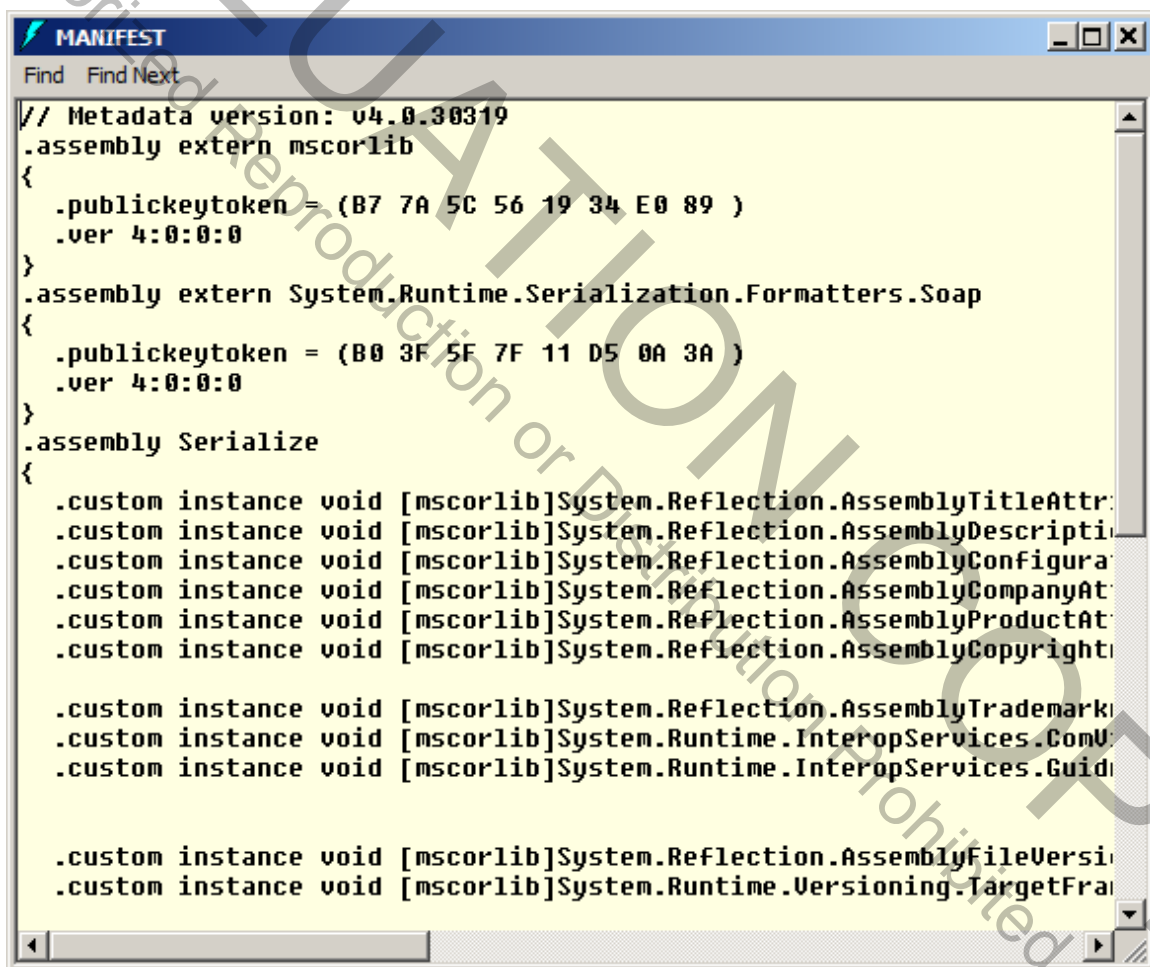
# Managed Code

---

- **In the serialization example we never freed any allocated memory.**
  - Memory that is no longer referenced can be reclaimed by the CLR's garbage collector.
  - Automatic memory management eliminates the common programming error of memory leaks.
  - Garbage collection is one of the services provided to .NET applications by the Common Language Runtime.
- **Managed code uses the services of the CLR.**
  - CIL can express access to unmanaged data in legacy code.
- **Type-safe code cannot be subverted.**
  - For example, a buffer overwrite is not able to corrupt other data structures or programs. Security policy can be applied to type-safe code.
- **Type-safe code can be secured.**
  - Access to files or user interface features can be controlled.
  - You can prevent the execution of code from unknown sources.
  - You can prevent access to unmanaged code to prevent subversion of .NET security.
  - Paths of execution of .NET code to be isolated from one another.

# Assemblies

- .NET programs are deployed as an *assembly*.
  - The metadata about the entire assembly is stored in the assembly's manifest.
  - An assembly has one or more EXEs or DLLs with associated metadata information.



```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly extern System.Runtime.Serialization.Formatters.Soap
{
    .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
    .ver 4:0:0:0
}
.assembly Serialize
{
    .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttr:
    .custom instance void [mscorlib]System.Reflection.AssemblyDescripti
    .custom instance void [mscorlib]System.Reflection.AssemblyConfigura
    .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAt
    .custom instance void [mscorlib]System.Reflection.AssemblyProductAt
    .custom instance void [mscorlib]System.Reflection.AssemblyCopyright

    .custom instance void [mscorlib]System.Reflection.AssemblyTrademark
    .custom instance void [mscorlib]System.Runtime.InteropServices.ComV
    .custom instance void [mscorlib]System.Runtime.InteropServices.Guid

    .custom instance void [mscorlib]System.Reflection.AssemblyFileVersion
    .custom instance void [mscorlib]System.Runtime.Versioning.TargetFra
```

# Assembly Deployment

---

- **The assemblies can be uniquely named.**
  - Assemblies can be versioned and the version is part of the assembly's name.
  - Unique (strong) names use a public/private encryption scheme.
  - The culture used can also be made part of the assembly name.
- **Assemblies are self-describing. Information is in the metadata associated with the assembly, not in the System Registry.**
- **Private, or xcopy deployment requires only that all the assemblies an application needs are in the same directory.**
  - This makes deployment of components much simpler.
- **Public assemblies require a strong name and an entry in the Global Assembly Cache (GAC).**
- **Either approach means the end of DLL Hell!**
  - Components with different versions can be deployed side by side and need not interfere with each other.

# JIT Compilation

---

- **Before executing on the target machine, CIL is translated by a just-in-time (JIT) compiler to native code.**
- **Some code typically will never be executed during a program run.**
  - Hence it may be more efficient to translate CIL as needed during execution, storing the native code for reuse.
- **When a type is loaded, the loader attaches a stub to each method of the type.**
  - On the first call the stub passes control to the JIT, which translates to native code and modifies the stub to save the address of the translated native code.
  - On subsequent calls to the method transfer is then made directly to the native code.
- **As part of JIT compilation code goes through a verification process.**
  - Type safety is verified, using both the CIL and metadata.
  - Security restrictions are checked.

# ASP.NET and Web Services

---

- **.NET includes a totally redone version of the popular Active Server Pages technology, known as *ASP.NET*.**
- **Whereas ASP relied on interpreted script code interspersed with page formatting commands, ASP.NET relies on *compiled* code.**
  - The code can be written in any .NET language, including C#, Visual Basic, JScript.NET and C++/CLI.
- **ASP.NET provides *Web Forms* which vastly simplifies creating Web user interfaces.**
  - Drag and drop in Visual Studio 2019 makes it very easy to lay out forms.
  - Also supported are ASP.NET MVC and Web API.
- **For application integration across the internet, *Web services* use the SOAP protocol.**
  - The beautiful thing about a Web service is that from the perspective of a programmer, a Web service is no different from any other kind of service implemented by a class in a .NET language.
- **However, Web API and REST have become more important than SOAP.**
- **Web services and C# (or Visual Basic) as a scripting language allows Web programming to follow an object-oriented programming model.**

# The Role of XML

---

- **XML is ubiquitous in .NET and is highly important in Microsoft's overall vision.**
- **Some uses of XML in .NET include:**
  - XML is used for encoding requests and responses in the SOAP protocol.
  - XML is the serialization format for disconnected datasets in ADO.NET.
  - XML is used extensively in configuration files.
  - XML documentation can be automatically generated by .NET languages.
  - .NET classes provide a very convenient API for XML programming as an alternative to DOM or SAX.
- **Also, as we shall see shortly, CLR Serialization is not available in .NET Core, while XML Serialization may be used to achieve serialization in a cross-platform manner.**

# Performance

---

- **Concerns about performance of managed code are similar to the concerns assembly language programmers had with high level languages.**
- **Garbage collection usually produces faster allocation than C++ unmanaged heap allocation. Deallocation is done on a separate thread by the garbage collector.**
- **JIT compilation takes a hit the first time when verification and translation take place, but subsequent executions pay no penalty.**
- **There is a penalty when security checks have to be made that require a stack walk.**
- **Compiled ASP.NET code is going to be a lot faster than interpreted ASP pages.**
- **Bottom line: for most of the code that is written, any small loss in performance is far outweighed by the gains in reliability and ease of development.**
  - High performance servers might still have to use technologies such as ATL Server and C++.
- **Apps targeting the Windows 10 platform may use .NET Native to achieve higher performance.**

# **.NET Native**

---

- **.NET Native is a precompilation technology for building and deploying apps to Windows 10.**
  - Rather than generating CIL, Windows apps are compiled directly to native code for faster startup and execution.
- **.NET Native changes the way in which .NET Framework applications are built and executed.**
  - During precompilation required portions of the .NET Framework are statically linked to your code, enabling the compiler to perform global code optimization.
  - The .NET Native runtime is optimized for static precompilation.
  - .NET Native uses the same backend as the C++ compiler for superior performance.
- **.NET Native brings the performance of C++ to managed code.**
- **See MSDN for more information:**

<https://docs.microsoft.com/en-us/dotnet/framework/net-native/index>

# **.NET Core**

---

- **.NET Core is a modular subset version of the .NET Framework that is portable across multiple platforms.**
- **Rather than one large assembly, .NET Core is released through NuGet in smaller feature-specific assembly packages.**
- **.NET Core provides key functionality used in applications regardless of platform.**
  - This common functionality provides for shared code that can be used across platforms.
  - Your application then links in additional platform-specific code.
- **Microsoft platforms you can target include traditional desktop Windows and Windows phones.**
- **Through third-party tools such as Xamian you can target Android and iOS.**
- **Visual Studio 2019 provides support for cross-platform development.**

# .NET Frameworks

---

- With the advent of .NET Core the term “framework” becomes more complex.
- Traditionally, there was just one .NET Framework (in various versions).
- But now there are multiple .NET frameworks.
- A framework defines an API you can rely on when you target that framework.
  - A framework is versioned as new APIs are added.
- But there are differences between the .NETCoreApp framework API and the classic .NET API.
  - Thus apps built for classic .NET may not run with a simple recompile on .NET Core.
- For example, the *Serialization* example does not run on .NET Core.
  - The CLR Serializer does not work with .NET Core, but you can achieve serialization with the XML Serializer.
- For more information about .NET Core see [this Object Innovations course](#):

4012 .NET Core Frameworks

# XML Serialization Example

---

- **Although you cannot use CLR Serialization as illustrated earlier in the chapter with .NET Core, you can use XML Serialization.**

– See the **XmlSerialize** example in this chapter<sup>4</sup>.

```
class Customer
{
    public string name;
    public long id;
}
class Program
{
    static void Main(string[] args)
    {
        List<Customer> list = new List<Customer>();

        Customer cust = new Customer();
        cust.name = "Charles Darwin";
        cust.id = 10;
        list.Add(cust);

        cust = new Customer();
        cust.name = "Isaac Newton";
        cust.id = 20;
        list.Add(cust);

        foreach (Customer x in list)
            Console.WriteLine(x.name + ": " + x.id);

        Console.WriteLine("Saving Customer List");

        XmlSerializer ser = new
            XmlSerializer(typeof(List<Customer>));
    }
}
```

---

<sup>4</sup> This example targets .NET Framework, but it can be built to target .NET Core. However, .NET Core is beyond the scope of this course.

## XML Serialization (Cont'd)

```
FileStream s = new FileStream("cust.xml",  
                             FileMode.Create);  
ser.Serialize(s, list);  
s.Flush();  
s.Dispose();  
  
Console.WriteLine("Restoring to New List");  
  
FileStream s2 =  
    new FileStream("cust.xml", FileMode.Open);  
list = (List<Customer>)ser.Deserialize(s2);  
  
foreach (Customer y in list2)  
    Console.WriteLine(y.name + ": " + y.id);  
}
```

- The data is serialized to the file *cust.xml*.

```
<?xml version="1.0" encoding="utf-8"?>  
<ArrayOfCustomer xmlns:xsi="http://www.w3.org/2001/  
XMLSchema-instance"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
  <Customer>  
    <name>Charles Darwin</name>  
    <id>10</id>  
  </Customer>  
  <Customer>  
    <name>Isaac Newton</name>  
    <id>20</id>  
  </Customer>  
</ArrayOfCustomer>
```

# Summary

---

- **.NET solves problems of past Windows development.**
- **It provides one development paradigm for multiple languages.**
- **Design and programming language no longer conflict.**
- **.NET uses managed code with services provided by the Common Language Runtime that uses the Common Type System.**
- **Plumbing code for fundamental system services is there, yet you can extend it or replace it if necessary.**
- **The .NET Framework is a very large class library available consistently across many languages.**
- **.NET Core is a modular subset version of the .NET Framework that is portable across multiple platforms.**
- **CLR Serialization is not available in .NET Core, but you can use XML Serialization.**

## **Chapter 2**

### **Class Libraries**

EVALUATION COPY  
Unauthorized Reproduction or Distribution Prohibited

# Class Libraries

## Objectives

---

*After completing this unit you will be able to:*

- **Describe the role of components in software development.**
- **Compare components in COM and .NET.**
- **Build .NET class libraries at the command line.**
- **Create class libraries using Visual Studio 2019.**
- **Use components in client programs by obtaining references to class libraries.**

# Objects and Components

---

- **An *object* encapsulates data and behavior, and it facilitates reuse—at the source code level.**
  - A class library, such as Microsoft Foundation Classes (MFC) provides reusable code in the form of a hierarchy of classes.
  - But you cannot use MFC classes in a Visual Basic 6 program.
- **By contrast, a *component* is a binary piece of software that can be reused in many different programming languages.**
  - For example, you can use COM components and ActiveX controls (a particularly rich type of COM component, typically with a graphical user interface) in Visual Basic 6 programs.
  - The component could be implemented in some other language, such as C++. The Visual Basic 6 programmer does not care.
- **The basic concept of component is a black box piece of software that can be reused.**
  - By this loose definition, a DLL would be a component.
- **Usually, somewhat more is meant, such as some kind of “object orientation.”**
  - Examples of such object-oriented components are COM objects, JavaBeans, and CORBA objects.

## Limitation of COM Components

---

- **COM lacks support for implementation inheritance.**
  - You cannot start with a base component and inherit its methods. (You can achieve similar reuse by other techniques, such as containment and aggregation, but such reuse is not as easy or convenient as inheritance.)
- **Another drawback of COM components is the requirement that the component implement “plumbing” code that allows it be called in a black box fashion from another piece of software.**
  - Visual Basic 6 hid the plumbing code, but it was there, and VB6 programs could only use a subset of the capabilities of COM.
  - In C++ you could fully utilize COM, but there was a lot of work to be done in implementing the plumbing code. Specialized libraries like the Active Template Library (ATL) could do a lot of the work for you, but that required you to learn yet another piece of technology, and it applied only to C++.

# Components in .NET

---

- **The .NET Framework provides an exceptionally attractive environment for creating and using software components.**
- **By simply setting an appropriate compiler switch or choosing a specific project type in Visual Studio, you can build a *class library*.**
  - A class library is the .NET version of a component and is a DLL that packages the code for a set of classes.
- **There is no special plumbing code that must be provided.**
- **These class library DLLs can easily be used by other .NET programs, and you can mix .NET languages freely.**
- **Also, you can inherit from a class implemented in a class library.**
  - This inheritance mechanism extends across languages, since a class library is a binary component.

# Class Libraries at the Command Line

---

- **To create a .NET class library from the command line, compile using the switch */target:library*.**

- You can abbreviate the **/target** switch as simply **/t**.

```
csc /t:library Customer.cs
```

- This command creates the DLL **Customer.dll**.

- **To use the class library from another program, you must obtain a *reference* to the class library.**

- Compiling at the command line, you can use the **/reference** or **/r** switch.

```
csc /r:Customer.dll TestCustomer.cs
```

- This command creates the executable **TestCustomer.exe**.

- **Run commands from the Visual Studio 2019 command-line prompt.**

- Start it via All Programs | Visual Studio 2019 | Visual Studio Tools | Developer Command Prompt for VS2019.

# Component Example: Customer Management System

---

- We will illustrate how to work with components in .NET by componentizing a monolithic application.
- The application manages a list of customers.
  - The **Customers** class manages a collection of customers, as defined by the **CustomerListItem** structure.

```
public struct CustomerListItem
{
    public int CustomerId;
    public string FirstName;
    public string LastName;
    public string EmailAddress;
}
```

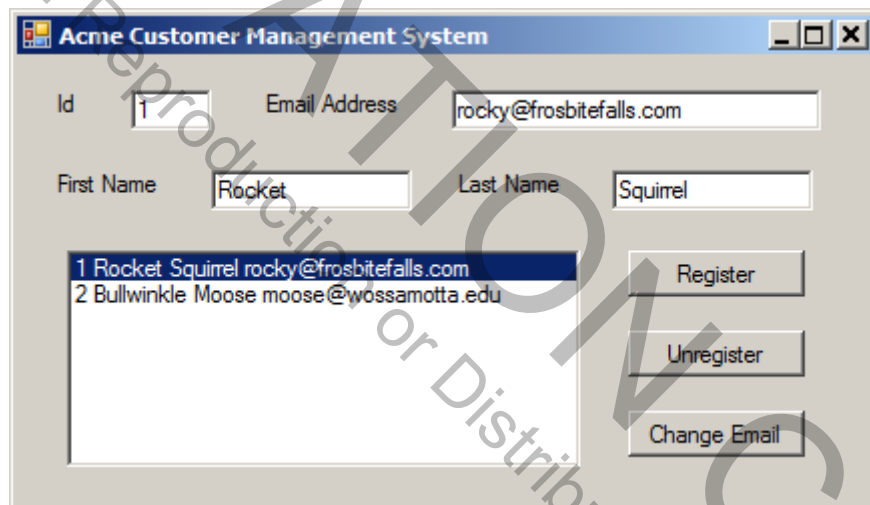
- The *Customers* class implements the *ICustomer* interface.

```
public interface ICustomer
{
    int RegisterCustomer(string firstName,
        string lastName, string emailAddress);
    void UnregisterCustomer(int id);
    ArrayList GetCustomer(int id);
    void ChangeEmailAddress(int id,
        string emailAddress);
}
```

- The **GetCustomer** method returns a list of customers (all customers if input ID is -1).

## Customer Management (Cont'd)

- The folder *CustomerMonolithic* contains a GUI version of a customer management system, with all code in one project.
  - Examine the code, and then build and exercise the program.
  - The list of customers is shown in the list box, and buttons are provided to register a new customer, unregister a customer, and change the email address of a customer.



# Monolithic versus Component

---

- In this monolithic version of the program, the business logic (as specified by the *ICustomer* interface) and the user interface code are mixed together.
- A better structure is to isolate the business logic in a separate component.
  - Create a DLL **Customer.dll** that contains the **Customers** class implementing the **ICustomer** interface.
- The .NET Framework then provides various mechanisms by which a client program can access this functionality, and there can be different kinds of clients:
  - Windows GUI client
  - Console application client
  - Web client
  - and so forth

## Demo: Creating a Class Library

---

- We will create a class library *Customer.dll* from the C# source file *Customer.cs*.

– See **CustomerLib** in this chapter's directory.

1. Examine the code file **Customer.cs**. Note that the code is within the namespace **OI.Acme**.

```
using System;
using System.Collections;

namespace OI.Acme
{
    public struct CustomerListItem
    {
        public int CustomerId;
        public string FirstName;
        public string LastName;
        public string EmailAddress;
    }

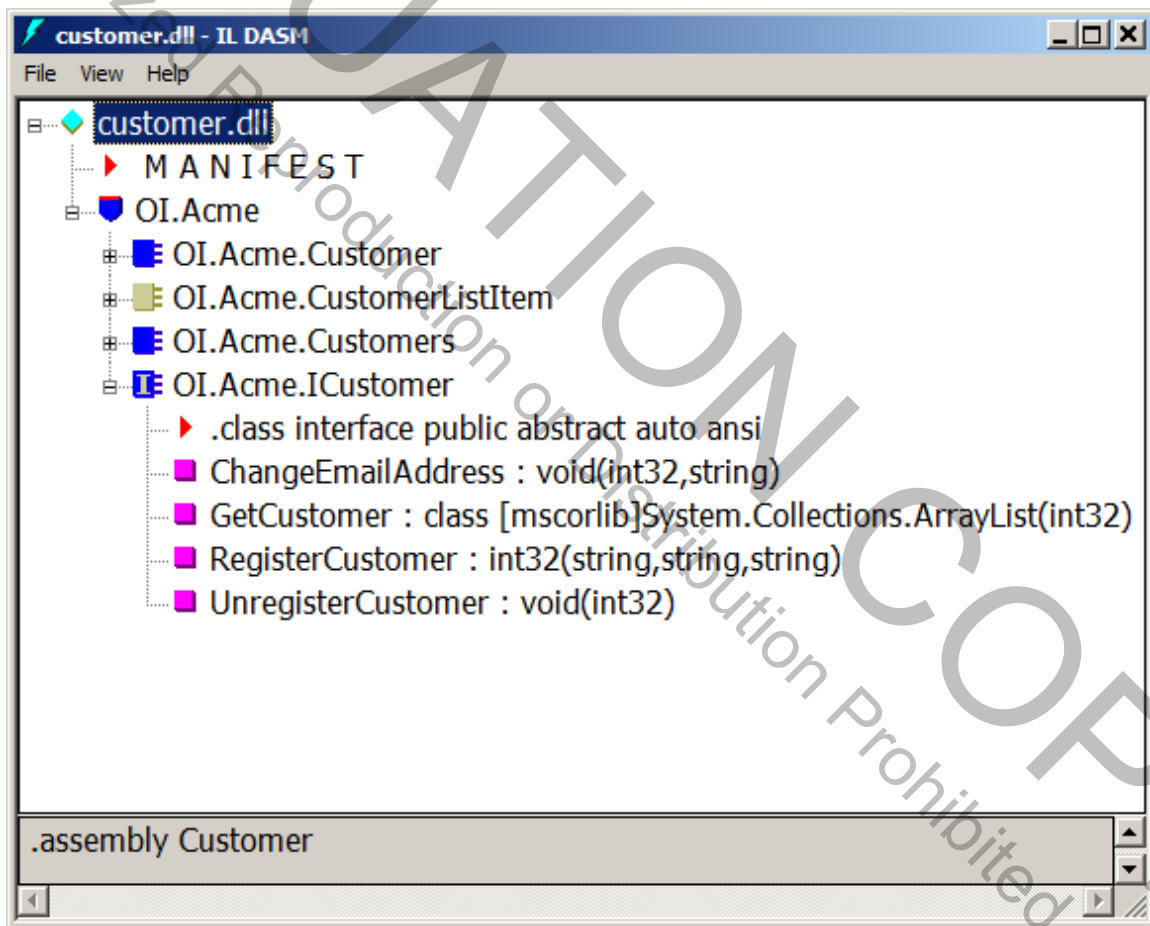
    public interface ICustomer
    {
        int RegisterCustomer(string firstName,
            string lastName, string emailAddress);
        void UnregisterCustomer(int id);
        ArrayList GetCustomer(int id);
        void ChangeEmailAddress(int id,
            string emailAddress);
    }
    ...
}
```

## Creating a Class Library (Cont'd)

2. Create the class library **Customer.dll** via the command:

```
csc /t:library Customer.cs
```

3. Examine the assembly **Customer.dll** that was created by using the **ILDASM** tool<sup>1</sup>. Notice that this assembly looks a little different from the ones we've created before (for example, there is no **Main** method).



<sup>1</sup> If you can't run the .NET 4.7.2 Tools, such as ILDASM, from the Developer Command Prompt, run the file `\OIC\NetCs\FixPath.bat`, as discussed in Chapter 1.

## Demo: A Console Client Program

---

- Create a console client program *TestCustomer.exe* from the C# source file *TestCustomer.cs*.

- See **TestCustomer**.

1. Examine the code file **TestCustomer.cs**. Notice the **using** statement to bring in the **OI.Acme** namespace.

```
using System;
using System.Collections;
using OI.Acme;
```

```
class TestCustomer
{
    ...
}
```

2. Copy **Customer.dll** from **CustomerLib** to **TestCustomer**<sup>2</sup>.

3. Create the assembly **TestCustomer.exe** via the command:

```
csc /r:Customer.dll TestCustomer.cs
```

4. You can now run the test program, which exercises the **Customers** class by adding a new customer, changing the email address of a customer, and unregistering a customer.

### Output:

```
1 Rocket      Squirrel  rocky@frosbitefalls.com
2 Bullwinkle  Moose     moose@wossamotta.edu
New customer Christopher Robin id = 3
1 Rocket      Squirrel  stallone.com
3 Christopher Robin  chris@pooh.com
```

<sup>2</sup> The batch file **build.bat** will both build the DLL and the EXE without having to copy the DLL. The batch file must be run from the Developer Command Prompt.

# Class Libraries Using Visual Studio

---

- **Visual Studio makes it very easy to work with .NET class libraries.**

- You can create a class library by using the Class Library project type.
- You can use Solution Explorer to add references.
- It is all quite painless.

- **There is one nuance involved when using Visual Studio that you need to be aware of.**

- Visual Studio by default creates a root namespace based on the name of the project.

```
using System;  
using System.Collections.Generic;  
using System.Text;
```

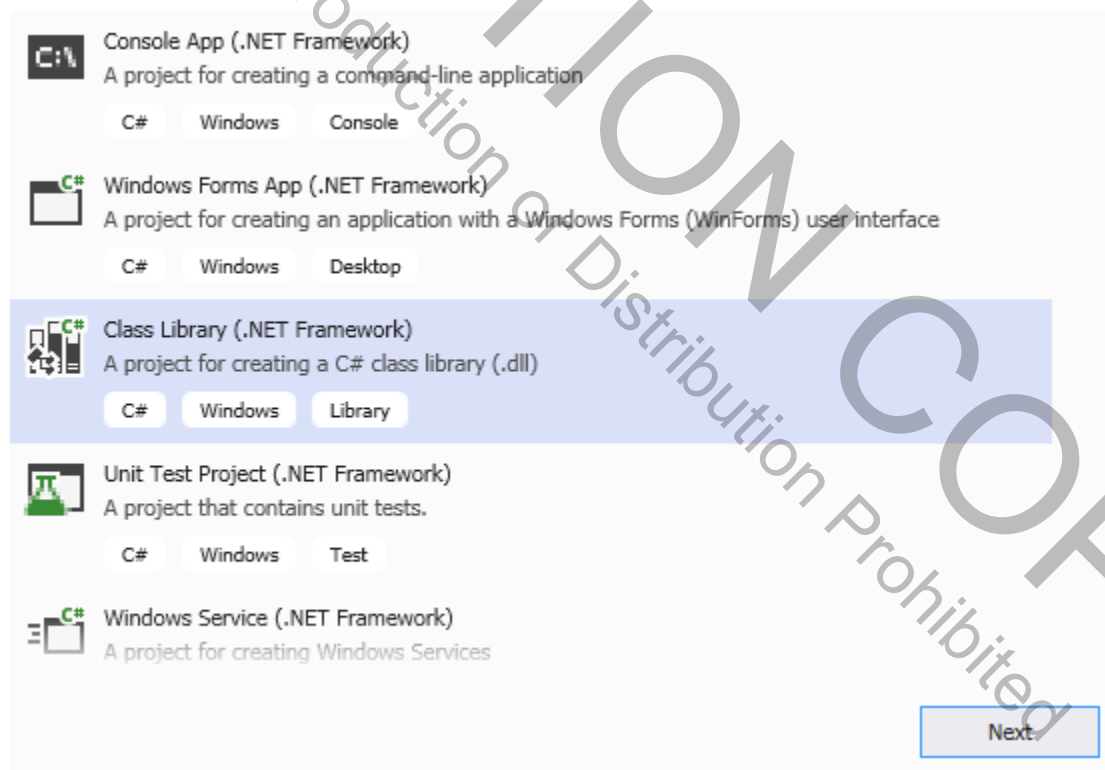
```
namespace ClassLibrary1  
{  
    public class Class1  
    {  
    }  
}
```

- This applies to both class library and application projects.

## Demo: Class Library in Visual Studio

---

- We will use Visual Studio to create a class library containing the *Customers* class.
    - Do your work in the **Demos** directory. The completed example is available in **CustomerVs** in the chapter folder.
1. Open up the solution in **Demos\CustomerVs**, which is a copy of **Chap02\CustomerMonolithic**. Build and run.
  2. Add a new project by right clicking over the solution. Choose Add | New Project from the context menu. Filter by C#. Choose the project template Class Library (.NET Framework).



3. Click Next.

## Demo: Class Library in VS (Cont'd)

4. Type **CustomerLib** for the Project name.

### Configure your new project

Class Library (.NET Framework) C# Windows Library

Project name

CustomerLib

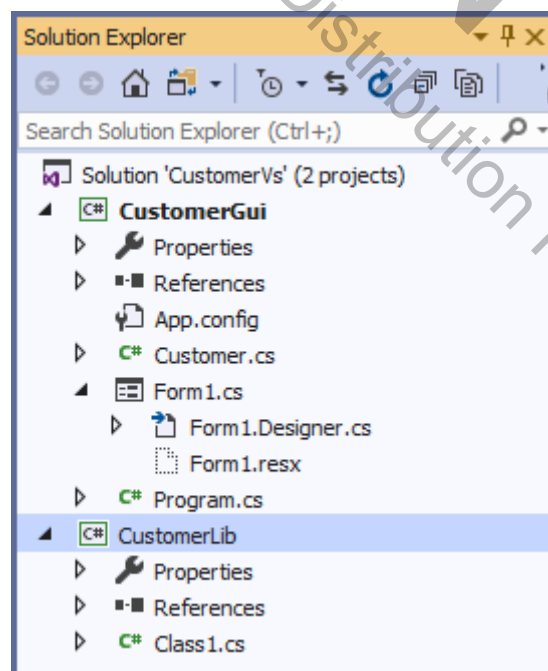
Location

C:\OIC\NetCs\Demos\CustomerVs

Framework

.NET Framework 4.7.2

5. Click Create. In Solution Explorer you will now see two projects in the solution.



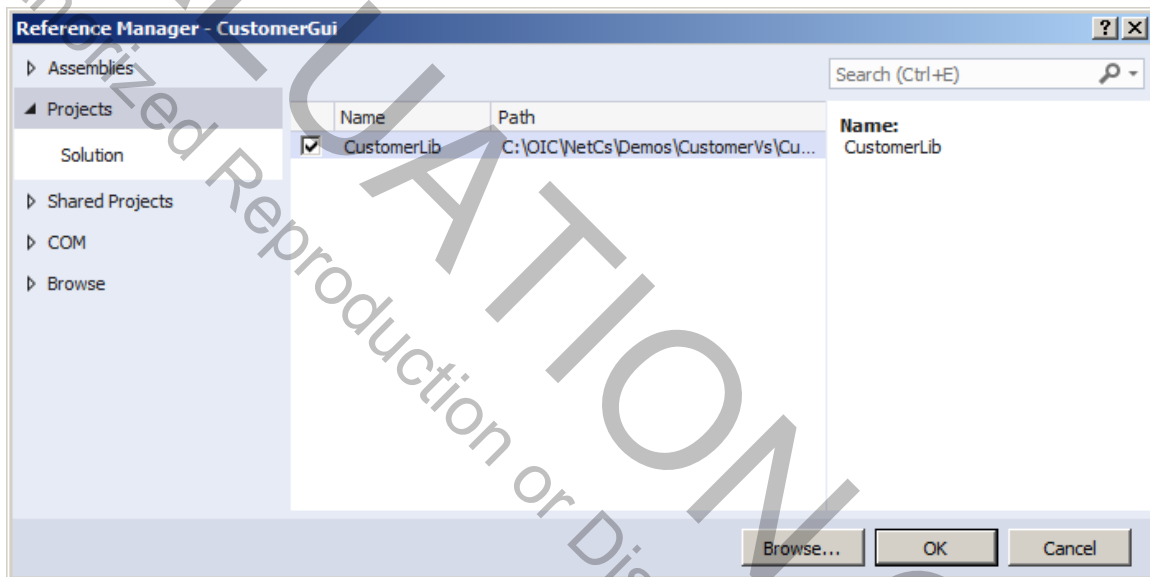
## Demo: Class Library in VS (Cont'd)

---

6. Delete **Class1.cs** from the new project.
7. Copy the file **Customer.cs** from the **CustomerGui** project to the **CustomerLib** project. You can do this right within Solution Explorer, and the file will be automatically added to the new project.
8. You should now be able to build the class library. The new DLL **CustomerLib.dll** will be created in **Demos\CustomerVs\CustomerLib\bin\Debug**.
9. Delete the **Customer.cs** file from the **CustomerGui** project. Do this in Solution Explorer, and the file will be removed both from the project and from the folder.
10. Try building the solution.
11. You will get errors. You will need to add a reference to the **CustomerLib** project in the **CustomerGui** project.

## Project References in Visual Studio

12. In Solution Explorer, right-click over the **CustomerGui** project and choose Add | Reference from the context menu, bringing up the Reference Manager dialog.
13. Click Solution, and you will see Projects highlighted. Check **CustomerLib**.

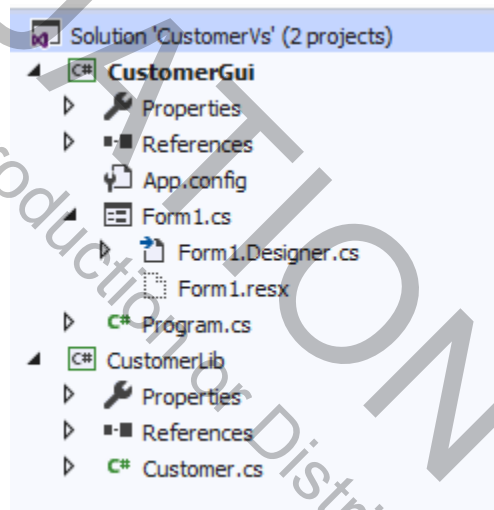


14. Click OK.
15. Now you should be able to build and run the solution. The behavior should be identical to that of **CustomerMonolithic**.

# Startup Project

- **In a multiple-project solution, you should specify a startup project.**

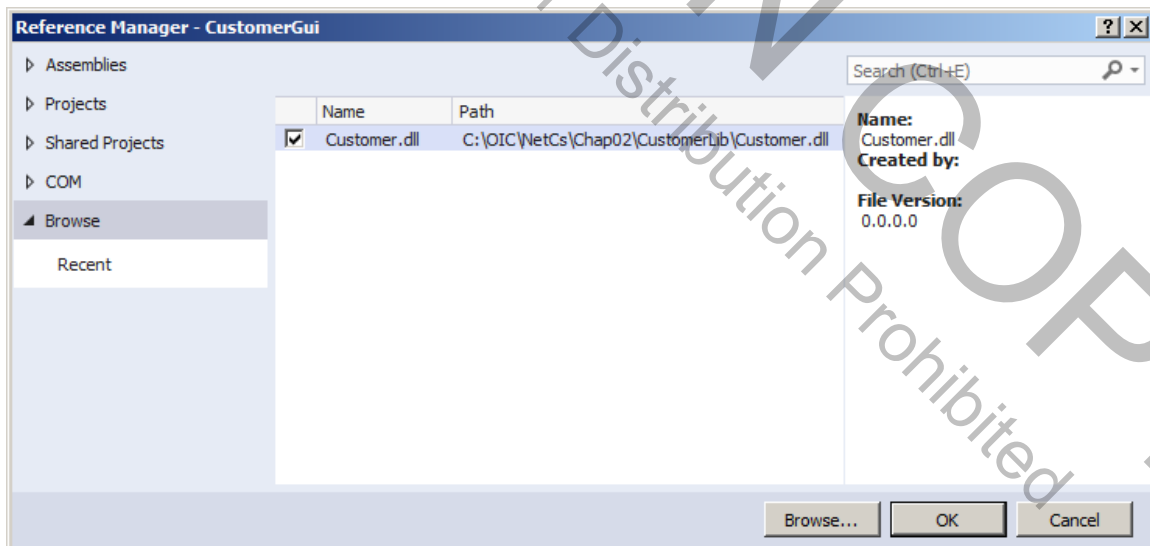
- By default, the first project that was in the solution is the startup project. (In our case it was CustomerGui, which is what we want.)
- The startup project is shown bolded in Solution Explorer.



15. To assign the startup project, right-click over the desired project and choose Set As Startup Project from the context menu.

# Assembly References

- The class library does not need to be a project in the same solution.
- You can add a reference to any assembly (DLL file).
  - For example, **CustomerLib.dll** in **Chap02\CustomerLib**.
  - If the DLL is not there, build it by running the **build.bat** batch file from a Visual Studio Command Prompt.
- As a quick demo, delete the project reference in the *CustomerGui* project.
  - Then right-click over References in Solution Explorer and choose Add Reference from the context menu.
  - Browse for **CustomerLib**, check it, and click OK.



- Then you should be able to build and run as before.

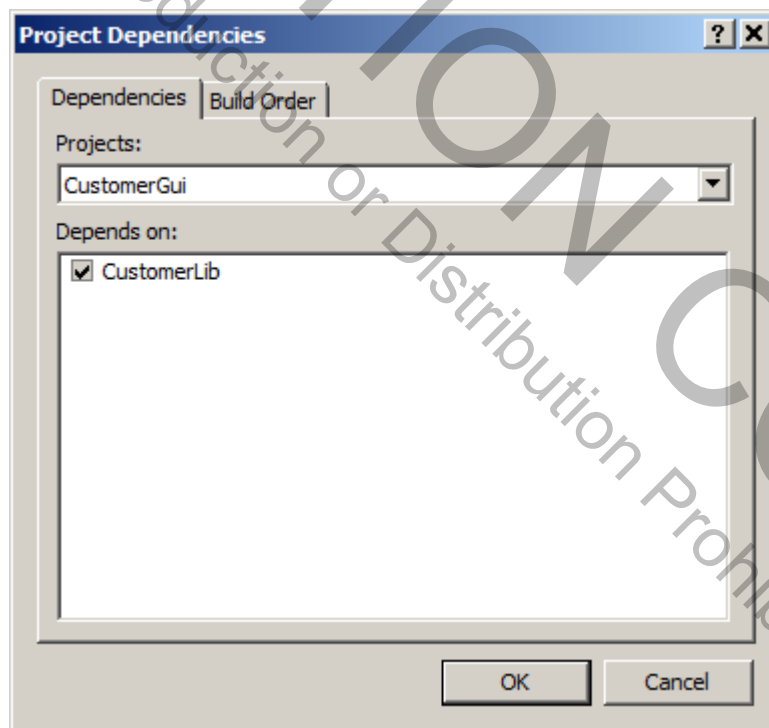
# References at Compile Time and Run Time

---

- **The assembly *CustomerLib.dll* is used both at compile time and at runtime.**
  - At compile time the metadata in the assembly is used, and at runtime the code is called.
  - When you build the application, Visual Studio will copy the DLL into the **bin\Debug** or the **bin\Release** folder, so it will reside in the proper place at runtime.
  - Examine the folder **CustomerGui\bin\Debug**. You should now see both **CustomerGui.exe** and **Customer.dll**.
- **In this case we are using *private* deployment in which all assemblies reside in the same directory.**
- **With .NET it is also possible to have *shared* deployment, using the Global Assembly Cache.**
  - We will discuss these issues in detail in the next chapter.

# Project Dependencies

- **When you have multiple projects in a solution, it becomes important to specify project dependencies.**
  - In our example, you need to build the class library before the Windows application.
- **To set project dependencies, right-click over the solution and choose Project Dependencies from the context menu.**
  - When we specified a Project Reference, the dependency was set automatically for us.



# Specifying Version Numbers

---

- **It is easy to specify version numbers and other assembly information using Visual Studio.**
  - A generated project contains a file **AssemblyInfo.cs** in the **Properties** folder.
  - Edit the **[assembly]** attributes.

```
...
[assembly: AssemblyTitle("CustomerGui")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CustomerGui")]
[assembly: AssemblyCopyright("Copyright © 2015")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

...
// Version information for an assembly consists of
// the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

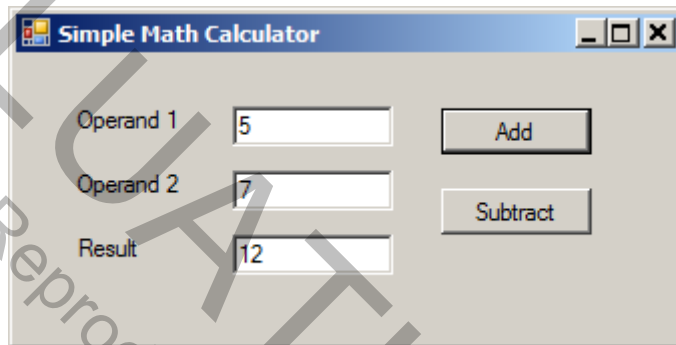
- You can then specify the Assembly Version and the File Version, as well as other information about the assembly.
- **You can view the Assembly Version of an assembly using ILDASM, and you can view the File Version of a file using Windows Explorer.**

## Lab 2

---

### Implementing a Class Library

In this lab you will implement a class library **SimpleMath.dll** that performs elementary arithmetic operations. You will also create a GUI test program to exercise your class library.



Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 30 minutes

## Summary

---

- A *component* is a binary piece of software that can be reused in many different programming languages.
- The .NET Framework makes it easy to create and use components, known in .NET as *class libraries*.
- You can build class libraries at the command line by using the */target* compiler switch.
- You can build class libraries in Visual Studio by creating a class library project.
- To use a class library in a client program you must specify a *reference* to it.

## Lab 2

### Implementing a Class Library

#### Introduction

In this lab you will implement a class library **SimpleMath.dll** that performs elementary arithmetic operations. You will also create a GUI test program to exercise your class library.

Rather than create two projects in the same solution, in this exercise you will create two separate solutions. This structure is carried over into the next chapter on assemblies.

**Suggested Time:** 30 minutes

**Root Directory:** OIC\NetCs

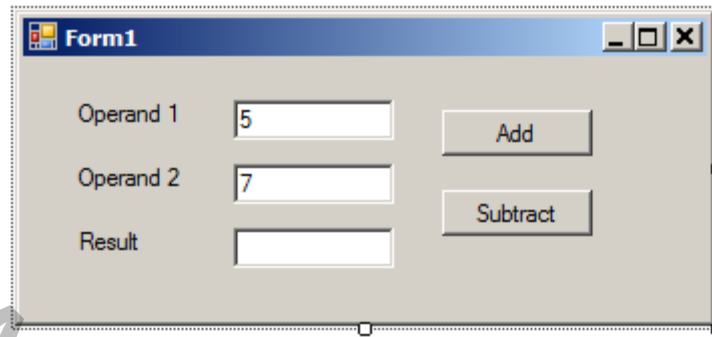
<b>Directories:</b>	<b>Labs\Lab2\SimpleMath</b>	(create your class library here)
	<b>Labs\Lab2\TestMath</b>	(create your test program here)
	<b>Chap02\SimpleMath\Step1</b>	(class library answer)
	<b>Chap02\TestMath</b>	(test program answer)

#### Instructions

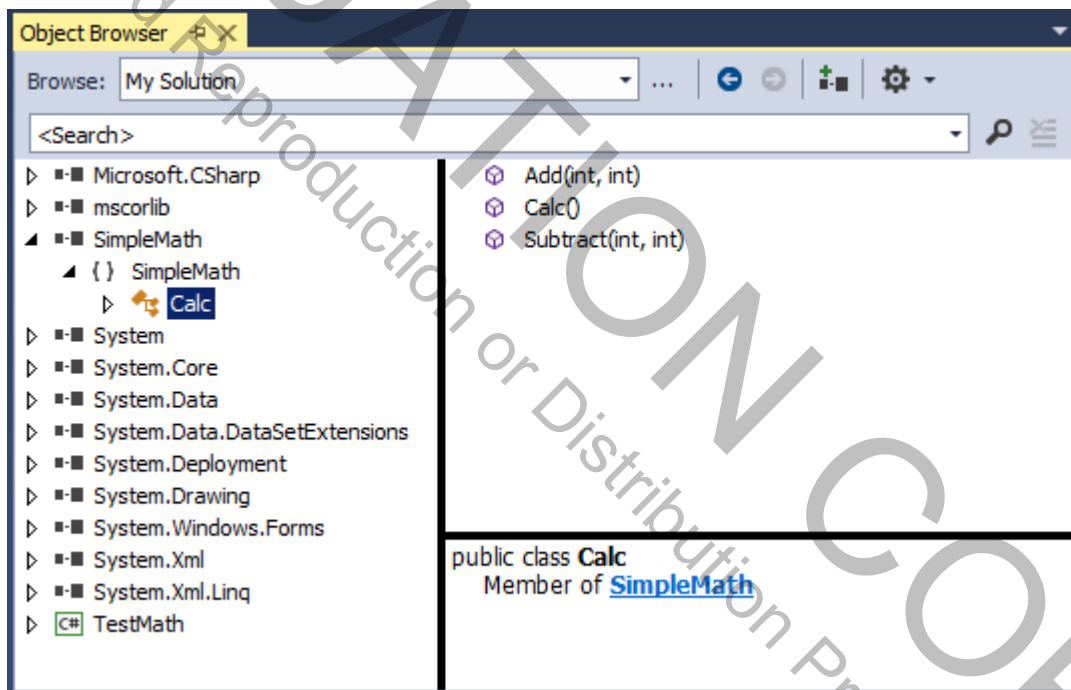
1. Use Visual Studio 2019 to create a class library project **SimpleMath** in the folder **Labs\Lab2**.
2. Rename the file **Class1.cs** to **SimpleMath.cs**. Rename the class to **Calc**. Note that the wizard generated code places your class inside the namespace **SimpleMath**.
3. Implement static methods to add and subtract two integers.
4. Edit the **AssemblyInfo.cs** file in the **Properties** folder to set both the Assembly Version number and the File Version number to 1.1.0.0. The former will be shown in ILDASM, and the latter using Windows Explorer.

```
[assembly: AssemblyVersion("1.1.0.0")]
[assembly: AssemblyFileVersion("1.1.0.0")]
```

5. Build your class library, and examine it in ILDASM.
6. Use Visual Studio to create a Windows Application project **TestMath** in the folder **Labs\Lab2**.
7. Create a GUI consisting of three text boxes and two buttons that can test your **SimpleMath** class library.



8. Copy the **SimpleMath.dll** to the **TestMath** directory. Add a reference to this assembly. Instead of using the Solution | Projects tab, use the Browse tab.
9. Examine the **SimpleMath** DLL in the Object Browser, which you can bring up from the View menu.



10. Notice that the class **Calc** is in the namespace **SimpleMath**. Provide an appropriate **using** statement in the file **Form1.cs**.
11. Implement handlers for the two buttons that will call the corresponding methods of the **SimpleMath** class library.
12. Build and test.



7400 E. Orchard Road, Suite 1450 N  
Greenwood Village, Colorado 80111  
Ph: 303-302-5280  
[www.ITCourseware.com](http://www.ITCourseware.com)