Chapter 21

Components and Assemblies

Components and Assemblies

Objectives

After completing this unit you will be able to:

- **Organize large programs using assemblies and components.**
- **Call external COM objects from C#**

Overview of Components and Assemblies

- **Up until now we have been building exclusively monolithic applications, consisting of a single executable file.**
- **Our applications have been logically modular, consisting of several classes, typically distributed among a number of files.**
- **But these files have all been compiled together, forming a single EXE.**
- **Modern large applications are rarely monolithic but instead are made up of a number of executable units.**
- **In the Windows environment an application will normally consist of an EXE and a number of DLLs.**
- **In this chapter we will see how to create class library DLLs, or components, which will expose classes and their methods to external programs.**
- **We begin by using command-line tools, and later in the chapter we will use Visual Studio.NET. We will also examine *assemblies*, which are the unit of deployment in .NET.**
- **An assembly can be a single EXE or DLL, or it can consist of several files, called *modules*.**

OverView (Cont'd)

- **An assembly also contains a *manifest*, which describes how the elements of the assembly relate to each other and to external elements.**
- **An application in .NET can be composed of assemblies built using different languages, and you can even inherit across languages.**
- **As a somewhat larger example, we present a componentized version of our bank case study.**
- **We conclude the chapter by showing how to call a COM component from .NET.**

Building Components Using .Net SDK

- **Prior to .NET there was a big divide between using classes within an application and creating "software components" that implemented classes but could be called from independent executable units.**
- **With Microsoft software, the mechanism for creating such independent components was the Component Object Model, or COM.**
- **Implementing a COM component is nontrivial, as much "plumbing" code must be provided to facilitate proper operation across executable boundaries.**
- **Tools such as the Active Template Library in C++ were developed in individual languages to simplify the process.**

Building Components Using .Net SDK (Cont'd)

- **Visual Basic provided an easy way to create COM components, but this was specific to Visual Basic, and not all features of COM were supported.**
- **.NET changes the picture completely.**
- **It is totally trivial to create a component from a class.**
- **You just need to build a different kind of project, a "class library."**
- **The .NET Framework takes care of the plumbing for you automatically.**
- **In this section we will see how to create a component using the command-line compiler.**

Creating a Component: A Demonstration

- **Let's begin with a simple demonstration.**
- **Do your work in the *Demos* directory for this chapter.**
- **A completed version of the demonstration is available in *HelloSDK*.**

Class Library

- **Using any text editor, create the file *HelloLib.cs*.**

```
// HelloLib.cs

class Hello
{
   private string greeting = "Hello, I'm a DLL";
   public string Greeting
   {
      get
      {
         return greeting;
      }
      set
      {
         greeting = value;
      }
   }
}
```

- **Please type in this program exactly as shown, including the lack of any access specifier in front of class *Hello*.**
  - This class simply exposes the property **Greeting**.
  - Notice that there is no **Main** method.
  - This class is not intended to be used in an EXE file.

Building a Class Library

- **To compile the file as a class library, enter the following at the command line:**

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

- **The command switch */t*, or */target*, specifies the kind of file to create. There are four options, as shown in the table**

Option	Meaning
/target:exe	Console application EXE
/target:winexe	Windows application EXE
/target:library	Class library DLL
/target:module	Module (no manifest)

- **The default is */target:exe*, or console application, which is what we have been building up until now.**
- **We will discuss modules later in this chapter.**
- **We are now building a class library DLL.**
  - If you did not make any typing mistakes, you should get a clean compilation, and if you use the DOS **dir** command, you should see that the file **HelloLib.dll** has been created.

Client Program

- **To exercise our class library we will need to create a client program.**
  - This could be either a console application or a Windows application.
  - We will create a simple console application as a test program.

Client Program (Cont'd)

- **Type in the following program and save in the file *TestHello.cs*.**

```
// TestHello.cs

using System;

public class TestHello
{
   public static void Main()
   {
      Hello obj = new Hello();
      Console.WriteLine(obj.Greeting);
   }
}
```

- **Compile this program using the following command:**

```
csc /r:HelloLib.dll TestHello.cs
```

- **The compiler option */r*, or */reference*, is used to import metadata from the specified class library.**
  - This makes any public classes in the class library available to the current compilation unit.
  - You will get compiler error messages:

```
TestHello.cs(9,3): error CS0122: 'Hello' is
inaccessible due to its protection level
TestHello.cs(10,21): error CS0234: The type or
namespace name 'obj' does not exist in the class or
namespace ''
```

Internal and Public Access

- **The problem comes from the fact that in the file *HelloLib.cs* we did not place any access modifier on the *Hello* class.**
  - The default access is **internal**, which means that the class can be accessed within the current assembly.
  - We mentioned **internal** access in Chapter 13, but were not able to demonstrate its implications, because up until now, all our programs have consisted of only a single assembly.
  - We will discuss assemblies in detail in the next section.
- **The fix is simply to make the *Hello* class public.**

```
// HelloLib.cs

public class Hello
{
   ...
```

- **Make this change, and then recompile both files. Now you should get a clean compilation. You can then run the file *TestHello.exe* and obtain the expected output.**

```
C:\OI\CSharp\Chap21\Demos>testhello
Hello, I'm a DLL
```

Monolithic Program

- **The directory *MonolithicHello* contains a monolithic version of this program.**
  - There is a Visual Studio project consisting of the two files.
  - The **Hello** class is left with no access modifier, so the default **internal** access is in place.
  - This time there is no problem, because we are building only a single assembly.

Assemblies

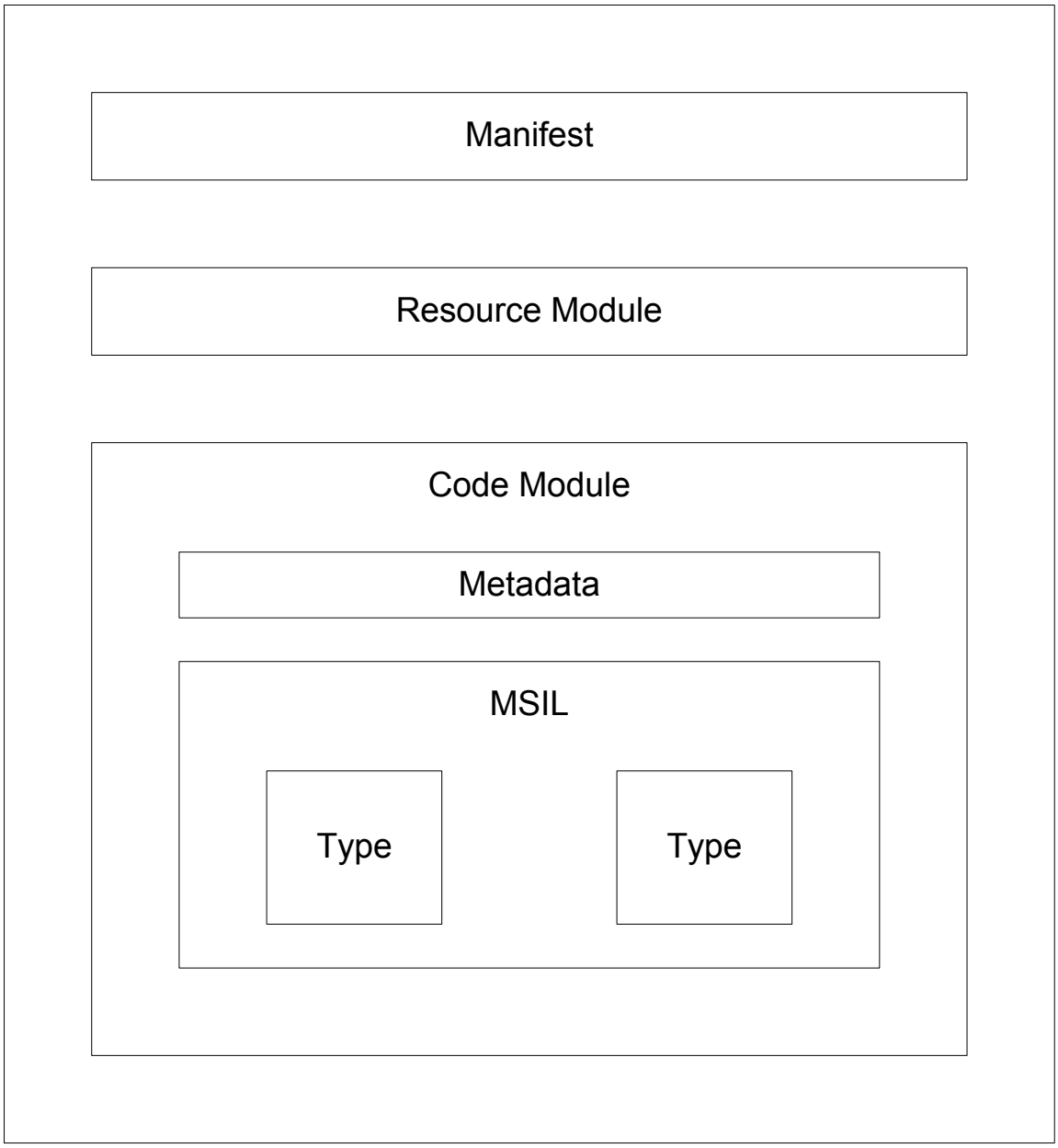
- **In this section we will take a closer look at assemblies.**
- **We begin by examining the structure of assemblies in some detail, and then we will work through an example that illustrates creating and using different types of assemblies.**

Assembly Structure

- **An assembly is a grouping of types and resources that work together as a logical unit.**
  - An assembly consists of one or more physical files, called **modules**, which may be code files or resources (such as bitmaps).
  - An assembly forms the boundary for security, deployment, type resolution, and versioning.
- **Logically, an assembly holds three kinds of information:**
  - MSIL (Microsoft Intermediate Language) implementing one or more types
  - Metadata
  - A *manifest* describing how the elements in the assembly relate to each other and to external elements

Assembly Structure (Cont'd)

- **The general structure of an assembly is shown in the figure.**



Assembly Structure (Cont'd)

- **All of the information in an assembly could be stored in a single file, or it could be distributed among a number of files, called modules.**
  - All of the assemblies we have built so far, including the DLL we built in the preceding section, have been single-file assemblies.
  - In a multiple-file assembly, the manifest could be a standalone file or it could be contained in one of the modules.

Manifest

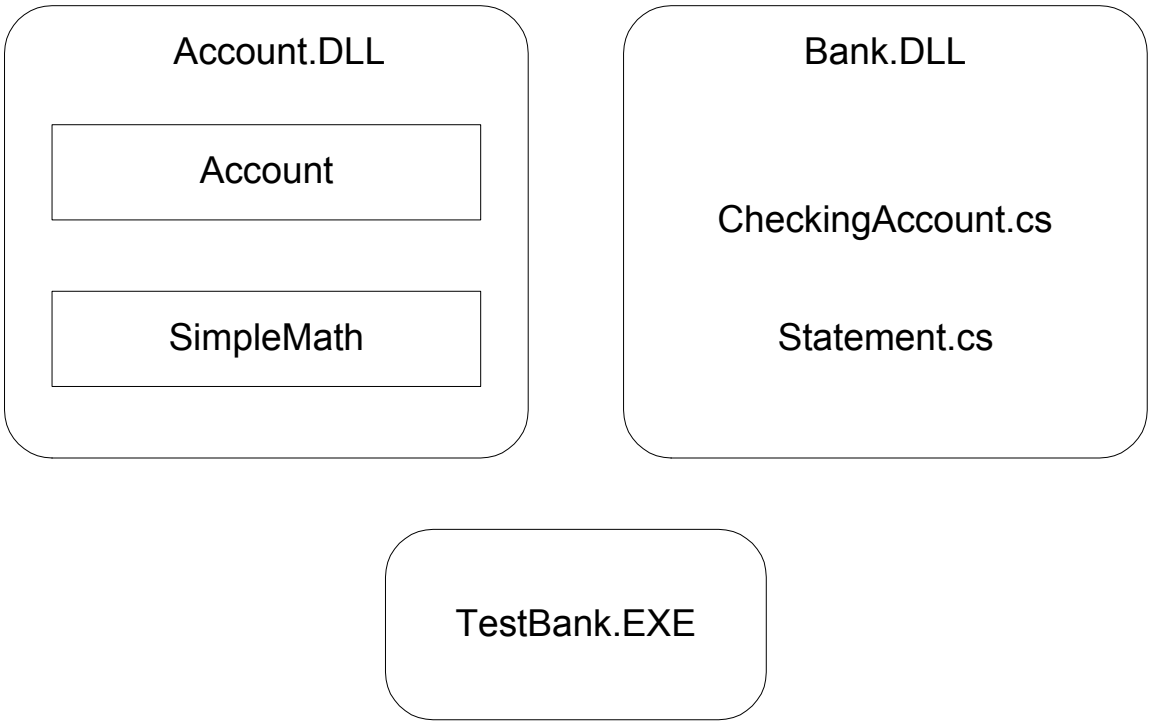
- **The manifest contains comprehensive information about the contents of an assembly and facilitates other assemblies using the assembly. The manifest contains a number of elements:**
  - **Assembly identity**. The name of the assembly, version information (consisting of four parts for very fine-grained versioning), and culture (containing locale in-formation suitable for globalization).
  - **Files**. A list of files in the assembly.
  - **Referenced assemblies**. A list of external assemblies that are referenced.
  - **Types**. A list of all types in the assembly, a mapping to the modules containing the types, and visibility information about the types.
  - **Security permissions**. Details needed by client programs that will determine whether or not they have rights to run the assembly.

Manifest (Cont'd)

- **Product information**. Information such as company, trademark, and copyright.
- **Custom attributes**. Special attribute information specific to this assembly. We touched on attributes in Chapter 20.
- **An (optional) shared name and hash**. This information facilitates running the assembly from a common location where multiple programs may access it. The hash protects client programs from running a corrupted version of the assembly.

Assembly Example

- **We will illustrate assemblies, including an assembly with multiple modules, with a version of our bank example.**
  - The general logic of this program should be very familiar by this point. Our example is intended to focus on different ways of packaging the units of the program.
- **There are three assemblies in the example, as illustrated in the figure.**



  - **Account.dll** is an assembly consisting of two modules.
  - **Bank.dll** is an assembly built from two source files.
  - **TestBank.exe** is a test program.

Monolithic Version

- **We begin with a monolithic version, consisting of a single assembly.**
- **A Visual Studio solution is provided in the directory *BankAssembly\Step1*.**
  - Since all the classes are in the same assembly, we can utilize **internal** accessibility.
  - In the class **Account**, the field **numXact** has **internal** access modifier.
  - The **namespace** directive will be discussed shortly.

```
// Account.cs
namespace OI.NetCs.Examples
{
   using OI.NetCs.Examples;
   public class Account
   {
      protected int balance;
      private string owner;
      static private string bankName = "Fiduciary
Bank";
      internal int numXact = 0;  // number of
transactions
      ...
      public int Transactions
      {
         get
         {
            return numXact;
         }
      }
   }
}
```

Monolithic Version (Cont'd)

- This means that the **Statement** class can access this field directly, without going through the public **Transactions** property.

```
// Statement.cs

using OI.NetCs.Examples;

public class Statement
{
   public static string
GetStatement(CheckingAccount acc)
   {
      string s = "Statement for " + acc.Owner +
"\n" +
         acc.numXact + " transactions, balance = "
         + acc.Balance + ", fee = " + acc.Fee;
      return s;
   }
}
```

Namespace

- **This example also illustrates the *namespace* directive.**
  - Look again at the **Account** class, which is enclosed in the namespace **OI.NetCs.Examples**.

```
// Account.cs

namespace OI.NetCs.Examples
{
...
```

- **The name of the class *Account* is rather generic, and if this code were part of a large system, with components acquired from many third-party vendors, we might well run into a name collision.**
- **By enclosing the class in a namespace, we can remove ambiguity.**
  - To refer to the **Account** class in another program, we would have to use the long, fully qualified name **OI.NetCs.Examples.Account**.
  - Such usage does indeed get around the possible name collision, but is also quite cumbersome.

Namespace Illustration

- **Hence, client programs can make use of the *using* directive, as illustrated in *Statement.cs*.**

```
// Statement.cs

using OI.NetCs.Examples;

public class Statement
{
   public static string
GetStatement(CheckingAccount acc)
   ...
```

  - Here we employ the short name **CheckingAccount** (the **CheckingAccount** class was also created in the namespace **OI.NetCs.Examples**).

- **This example of a namespace also illustrates a common sort of hierarchy:**
  - **OI** is a company/brand (Object Innovations).
  - **NetCs** is an acronym for this book (*Introduction to C# Using .NET*).
  - **Examples** is for the example programs.

Multiple Assemblies

- ***BankAssembly\Step2* illustrates building multiple assemblies.**
  - You may compile at the command line.
- **First, build *Account.dll*:**

```
csc /t:module /out:SimpleMath.mod SimpleMath.cs
csc /t:library /addmodule:SimpleMath.mod Account.cs
```

  - The first line illustrates the **/t:module** option for compiling **SimpleMath.cs**.
  - This will create a module without a manifest. We then could not obtain metadata in another compilation using the /reference option.
  - To make clear that the output file is not an ordinary DLL, we use the **/out** option to explicitly allow us to name the output file.
  - We make up a **.mod** extension (this is not standard, we are just using it to remind ourselves that we have created a module).

Multiple Assemblies (Cont'd)

- **Since we cannot use the /reference switch to bring in metadata from SimpleMath.mod, we need another mechanism.**
- **The C# compiler provides the /addmodule option for this purpose.**
  - As the name suggests, this option will add a module to the current assembly that is being built.
  - The target is a library, so the output file will be a DLL.
  - We don't specify a name for the output file, so it will be **Account.dll**.
- **Next we will build *bank.dll* using the following command:**

```
csc /t:library /r:Account.dll /out:Bank.dll
CheckingAccount.cs Statement.cs
```

  - Again we are building a class library, so we use the **/t:library** option.
  - We need a reference to **Account.dll**.
  - The source files are **CheckingAccount.cs** and **Statement.cs**.
  - Finally, we specify the name of the output file **bank.dll**.

Multiple Assemblies (Cont'd)

- **The last thing we build is *TestBank.exe*:**

```
csc /t:exe /r:Bank.dll /r:Account.dll
/out:TestBank.exe
TestBank.cs InputWrapper.cs
```

  - This time we need references to two DLLs, **Bank.dll** and **Account.dll**.

  - The build is automated by the batch file **build.bat**. You can run this batch file at the command line simply by typing build. (In Windows Explorer you could run the batch file by double-clicking on **build.bat**).

Multiple Language Applications

- ***BankAssembly\Step3*** **illustrates inheriting from a VB.NET class.**
- **The class *CheckingAccount*, implemented in C#, now inherits from *Account* implemented in VB.NET.**
  - We are continuing to work at the command line, and the batch file **buildvb.bat** builds the assemblies.
  - There is no equivalent of **/addmodule** in the VB.NET compiler, so **SimpleMath.dll** is built as a separate assembly.
  - There is no case sensitivity in VB.NET, so some variables are changed in **Account.vb**, which has a ripple effect into **CheckingAccount.cs**. (If you are going to do this sort of thing, you should pay attention to the Common Language Specification!)
- **In case you are curious about VB.NET, look at the code for *Account.vb* (next slide).**
  - Semantically, the class is identical to the C# class **Account.cs**. (Actually, the Step 3 example is somewhat simplified. There is now no **Owner** property.)
  - The syntax is somewhat different.
  - There are many new Visual Basic keywords in VB.NET to denote inheritance, name-spaces, and so on.

VB.NET Code Example

```
' Account.vb -- Step 3
Imports OI.NetCs.Examples
Namespace OI.NetCs.Examples
  Public Class Account
    Protected bal As Integer
    Private Shared bank As String = "Fiduciary Bank"
    Protected numXact As Integer = 0  ' number of
transactions
      Public Sub New(balance As Integer)
          Me.bal = balance
      End Sub
    Public Sub Deposit(amount As Integer)
      numXact = numXact + 1
      bal = SimpleMath.Add(balance, amount)
      End Sub
    Public Sub Withdraw(amount As Integer)
      numXact = numXact + 1
      bal = SimpleMath.Subtract(balance, amount)
      End Sub
    Public ReadOnly Property Balance() As Integer
      Get
        Return bal
      End Get
    End Property
    Public Shared Property BankName() As String
      Get
        Return bank
      End Get
      Set
        bank = Value
      End Set
    End Property
    Public ReadOnly Property Transactions() As Integer
      Get
        Return numXact
      End Get
    End Property
  End Class
End Namespace
```

Building Components Using Visual Studio. NET

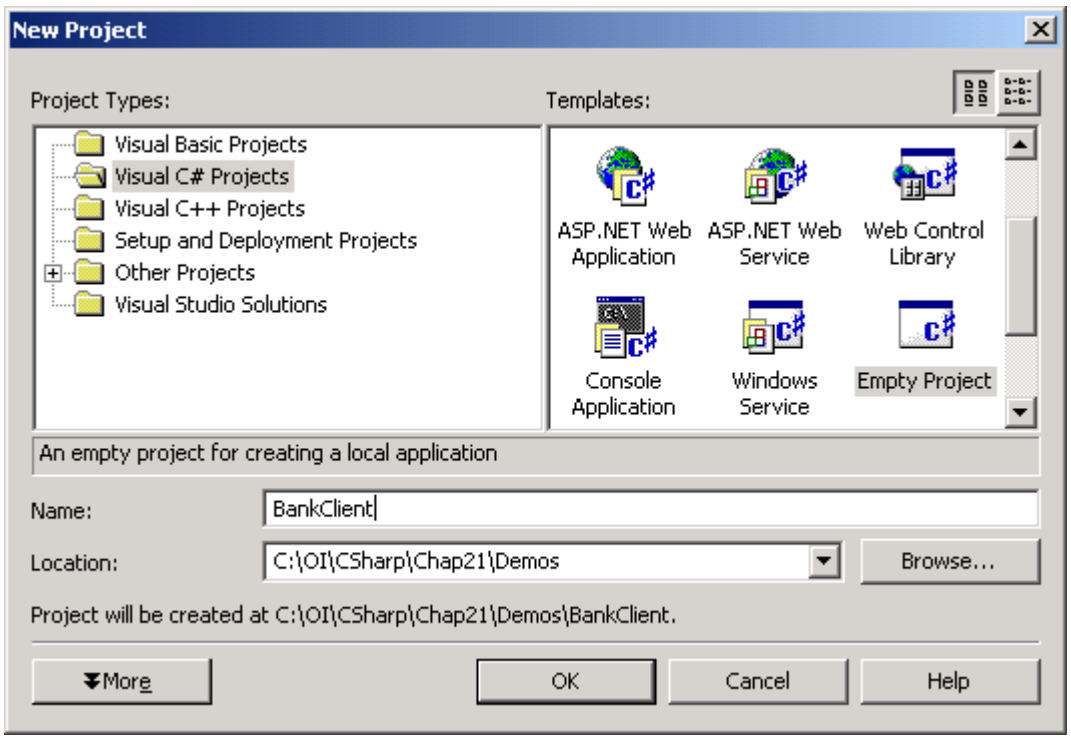
- **So far in this chapter we have done all our component building at the command line.**
  - By using the various command-line compiler options, you should by now be fairly familiar with the build process, including especially the use of references via the **/reference** option.
- **In this section we will see how to use Visual Studio for building class libraries and client programs.**
  - The class library and the client program will be separate projects within one solution.
  - If you would like to look at the complete solution, see **BankClientAnswer**.

Demonstration: Solution with Multiple Projects

- **To learn how to create a multiple-project solution, including a class library, go through the following demonstration.**
- **Do your work in the *Demos* directory for this chapter.**
  - To save typing, you may copy source files from **BankMonolithic**.

Creating the First Project

- **From Visual Studio main menu choose File | New | Project.... This will bring up the New Project dialog.**
- **For Project Types choose "Visual C# Projects" and for Templates choose "Empty C# Project."**
- **Click the Browse button and navigate to Demos and click Open.**
- **In the *Name* field, type *BankClient*. See the figure. Click OK.**



Adding the Source Files

- **Using Windows Explorer, copy the files *BankClient.cs* and *Account.cs* from *BankMonolithic* into the current directory, *Demos\BankClient*.**
- **In the Solution Explorer right-click over *BankClient*, and from the context menu choose Add | Add Existing Item....**
  - In the dialog that comes up, select the two files *BankClient.cs* and *Account.cs* (you may use multiple selection by having the Control key pressed). Click Open.

Adding the Source Files (Cont'd)

- **Examine the code for the test program *BankClient.cs*.**
  - A new **Account** object is created, initialized with a starting balance of 100.
  - A deposit is made, followed by a withdrawal.
  - The balance is displayed at the beginning and after each transaction. (The prompt to read a string at the end will prevent a quick close of the application if you run it in the debugger.)

```
// BankClient.cs

using System;

public class BankClient
{
   static int Main(string[] args)
   {
      Account acc = new Account(100);
      Console.WriteLine("balance = {0}",
acc.Balance);
      acc.Deposit(25);
      Console.WriteLine("balance = {0}",
acc.Balance);
      acc.Withdraw(50);
      Console.WriteLine("balance = {0}",
acc.Balance);
      Console.WriteLine("press Enter to exit");
      string s = Console.ReadLine();
      return 0;
   }
}
```

Adding the Source Files (Cont'd)

- **Examine the code for the *Account* class *Account.cs*.**
  - There is a constructor which initializes the starting balance and the methods **Deposit** and **Withdraw**.
  - There is also a property **Balance**.

```
// Account.cs

public class Account
{
   private int balance;
   public Account(int balance)
   {
      this.balance = balance;
   }
   public void Deposit(int amount)
   {
      balance += amount;
   }
   public void Withdraw(int amount)
   {
      balance -= amount;
   }
   public int Balance
   {
      get
      {
         return balance;
      }
   }
}
```

Changing the Output Path

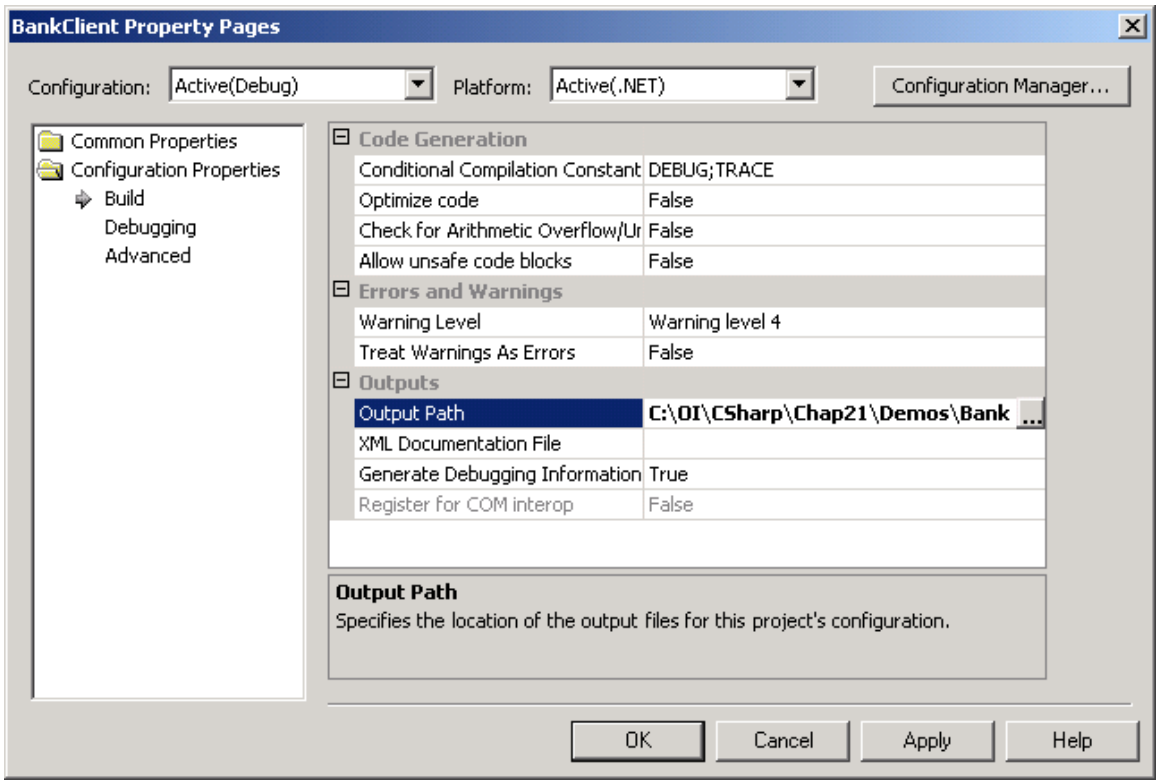
- **Up until now when we have built a project in Visual Studio, we have always excepted the default location for the executable.**
  - If we are building a **Debug** version of our project, the executable will be located in the **bin\Debug** directory.
- **When we are working with an application that consists of several assemblies, it will be convenient for the our executable and component files to all reside in the same directory.**
  - At runtime our application can then easily find its components.
- **We can achieve this goal by changing the output path of our executable to be the source directory.**
  - Later we will build our component to reside in the same directory.

Changing the Output Path (Cont'd)

1. 1. In Solution Explorer, right-click on the *BankClient* project, and choose Properties.

2. 2. In the Property Pages window that comes up, click on Configuration Properties and then on Build.

3. 3. Click in the right side of the Output Path area, and then click on the three dots. Navigate to the *BankClient* directory. See the figure. Click OK.

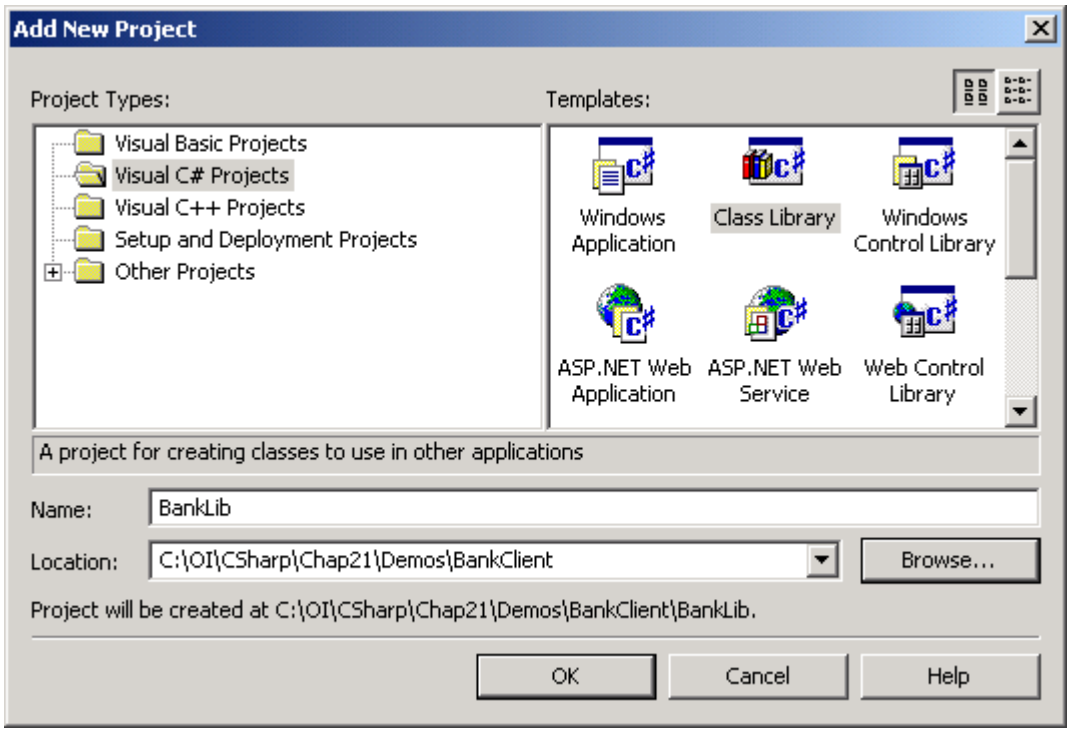


4. 4. Build and run. You should see the following output:

```
balance = 100
balance = 125
balance = 75
press Enter to exit
```

Adding a Second Project

- **In Solution Explorer, right-click over the solution and choose Add | New Project.**
  - The Add New Project dialog comes up.
- **Choose "Visual C# Projects" as the Project Type and "Class Library" as the template.**
- **Click the Browse button and navigate to the Demos\BankClient directory.**
- **Click Open. Type BankLib as the name of the project. See the figure. Click OK.**
- **Observe that a subdirectory BankLib will be created underneath BankClient.**

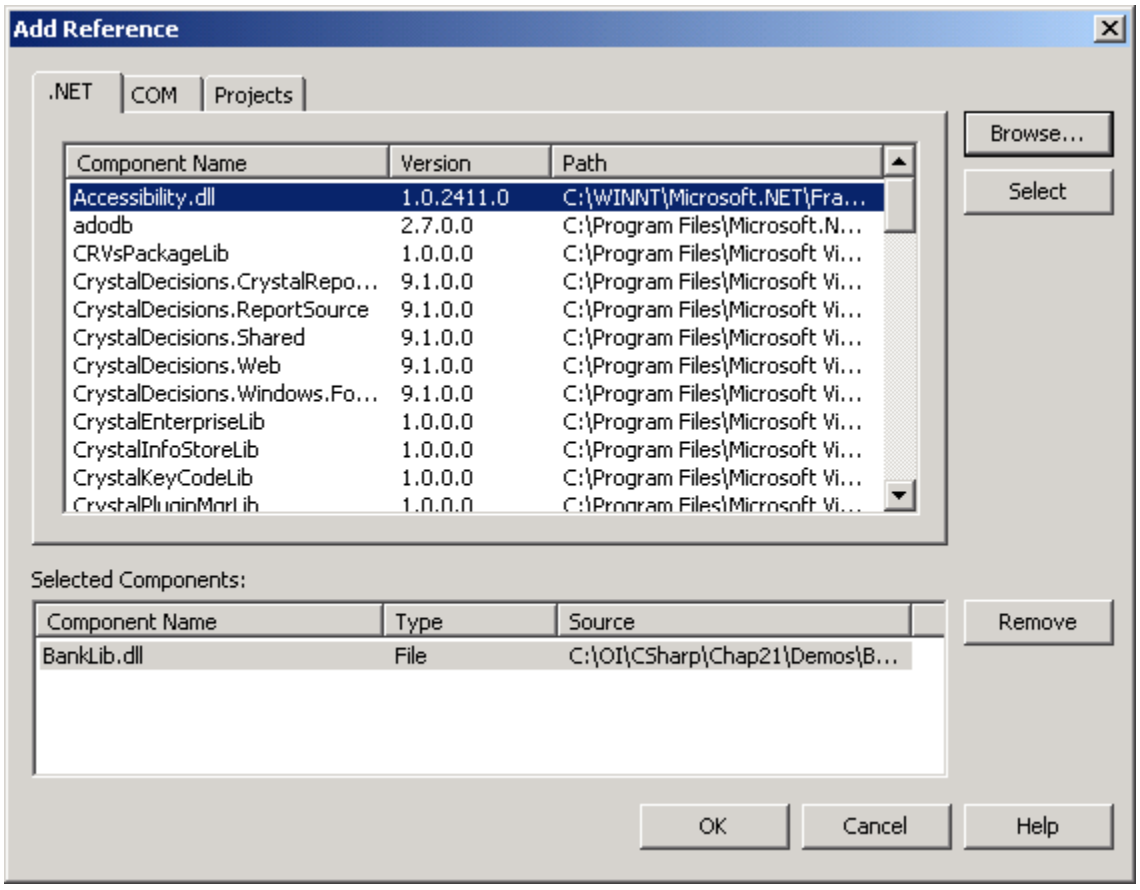


Move Code to the Second Project

- **The file *AssemblyInfo.cs* contains code that can be used to customize the manifest in the assembly.**
  - For simplicity, remove the two files **AssemblyInfo.cs** and **Class1.cs** from the new project (use Delete key or right-click on the project and choose Delete).
- **Move the file *Account.cs* from *BankClient* to *BankLib* (you can drag inside Solution Explorer).**
- **Following the same procedure we used previously for the *BankClient* project, change the output path for *BankLib* to be the *BankClient* source directory (so that *BankClient.exe* and *BankLib.dll* will wind up in the same directory).**
- **Build the class library: Right-click on *BankLib* and choose Build.**
- **Try to build the client test program: Right-click on *BankClient* and choose Build.**
  - You will get a number of error messages pertaining to **Account** not existing in the current namespace.

Adding a Reference

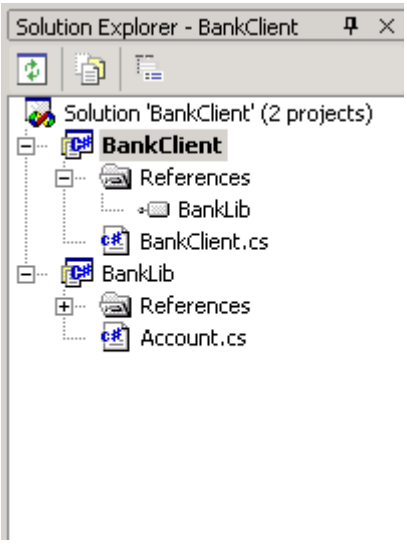
- **In Solution Explorer right-click on *BankClient* and choose Add Reference.**
- **Click Browse and navigate to *BankLib.dll*. (It is in *BankClient*.) Click Open. See the figure.**



- **Click OK. You will now see a Reference in Solution Explorer.**
  - Note also the two projects in our solution.
- **Now build the client again.**
  - This time it should work! Run the client.

Solutions and Projects

- **We have now constructed a solution with two projects.**
- **The figure shows Solution Explorer with this solution and its two projects.**



Building a Solution

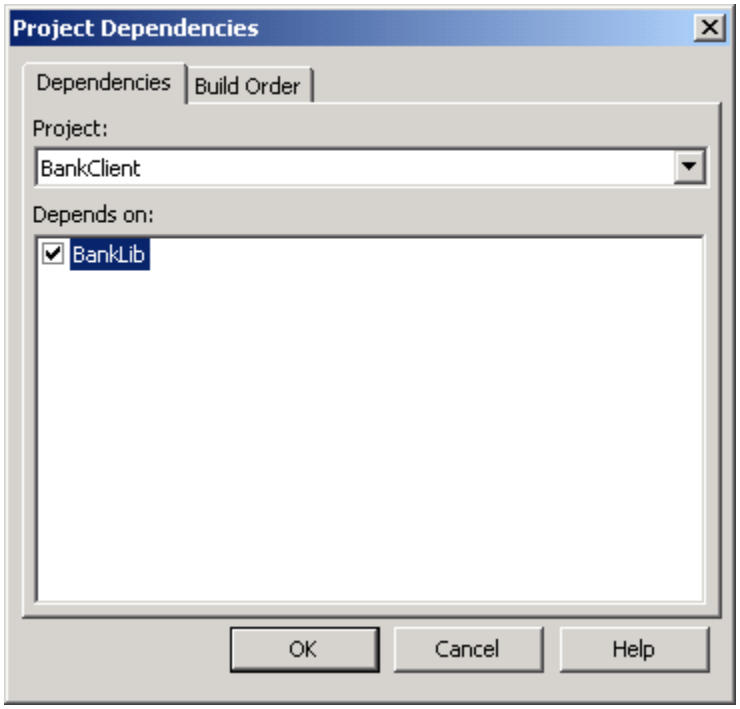
- **There are two build toolbar buttons:** 
- **The first button builds the currently selected project.**
  - You can select a project by clicking on it in Solution Explorer.
- **The second button builds the entire solution.**

Project Dependencies

- **Continuing our demonstration, close the solution (menu File | Close Solution) and delete the *bin* and *obj* directories for both projects. (If you have trouble deleting the directories, close down Visual Studio.)**
- **Open up the solution again and build the entire solution.**
  - You will get build errors, because the client program is built first and the class library does not yet exist.
  - To fix this problem, open up the Project Dependencies dialog from the menu Project | Project Dependencies.

Project Dependencies (Cont'd)

– With the **BankClient** project selected from the Project dropdown, check **BankLib** in the Depends On list. See the figure. Click OK.



- **Now try building the solution again. This time the build should succeed, because the library will be built first.**

Bank Case Study: Componentized Version

- **We have built some small components.**
  - Now let's try something a little more elaborate.
- **We will build a componentized version of our bank case study.**
  - The starting point is the Step 7 version from Chapter 18.
- **We have a copy in the current chapter in *CaseStudy\Monolithic*.**
  - If you want to refresh yourself on the case study, you may build and exercise this monolithic version.

Bank Case Study: Componentized Version (Cont'd)

- **Our componentized version consists of three pieces:**
  - **Account.dll**. This class library is built from **AccountDefinitions.cs**, **Account.cs**, **CheckingAccount.cs**, and **SavingsAccount.cs**.
  - **BankLib.dll**. This class library is built from **Bank.cs**, **Atm.cs**, and **InputWrapper.cs**. It contains a reference to **Account.dll**.
  - **BankClient.exe**. This console application is built from **TestBank.cs** and **InputWrapper.cs**. It contains references to **Account.dll** and **BankLib.dll**.
- **Projects for these pieces are in the directories *CaseStudy\AccountLib*, *CaseStudy\BankLib*, and *CaseStudy\BankClient*.**
  - You may wish to examine these projects, build them, and exercise the client program.
  - The BankLib directory contains a copy of Account.dll. The BankClient directory contains copies of BankLib.dll and Account.dll.

Interoperating With COM

- **We have seen that .NET components can be built in different languages, such as C# and VB.NET, and interoperate with each. It is also possible for .NET components to interoperate with COM components.**
  - You can call a COM component from .NET.
  - You can call a .NET component from COM.
- **The .NET Framework supports both of these kinds of interoperability, and tools are provided.**
- **We will illustrate the first scenario, which is the common one, of a .NET application calling a legacy COM component.**
  - We will use the **Type Library Importer** tool, which imports a type library for a COM component and generates a .NET proxy for calling a COM component from .NET. This tool is transparently invoked by Visual Studio when you add a reference to a COM component.
- **The subject of Microsoft's Component Object Model, or COM, is a large one, far be-yond the scope of this book. For a discussion of COM and COM+ you can refer to the book Understanding and Programming COM+, by Robert J. Oberg. In Chapter 12 of that book you will find a discussion of the *Logger* component, which we use as an illustration in this section.**

Calling a COM Component from .NET

- **We will demonstrate COM interoperability with a .NET application that calls a COM Logger component, which happens to be written in C++ using ATL, the Active Template Library.**
- **The sample program is in the directory *LoggerVc*.**
  - If you are curious about the code used to create a COM component, where the infrastructure code is in the component itself, you can examine the C++ code in **LoggerVc\Source**.
  - It is much simpler to create components using .NET.
- **First we will run the sample program, and then we will switch to the *Demos* directory and create the test program ourselves.**

Running the Test Program

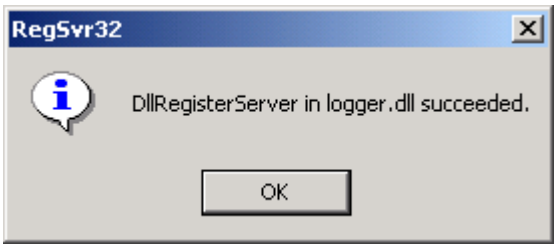
- **Open up the test program in *LoggerVc\TestLogger*. Try to build it. You will get an error message:**

```
error CS0234: The type or namespace name
'LOGGERLib' does
not exist in the class or namespace ''
```

- **The reason for this error is that the COM component must be registered before it can be used.**

Running the Test Program (Cont'd)

- **You can register the component by running the batch file *reg_logger.bat*. (In Windows Explorer you can double-click on this file to run it.)**
  - You will see a message box announcing that registration is successful. See the figure. (If by chance this component was already registered on your machine, you can try unregistering it by running the batch file **unreg_logger.bat**.)



- **Now you should succeed in building** TestLogger**.**
  - Running the program does not produce any output, because what the program does is write to a logfile, which for convenience we have located at the root of the **c:** drive.
  - In any text editor (for example, Notepad), examine the file **c:\logfile.txt**. You should see the following test messages displayed:

```
first line
second line
```

Creating the Test Program

- **Now let's create the test program. Do your work in the *Demos* directory.**
- **Open up Visual Studio and create a new Empty C# project *TestLogger* in *Demos*.**
- **Add a new empty C# file called *TestLogger.cs* to the project.**
- **Type the following code into this file.**

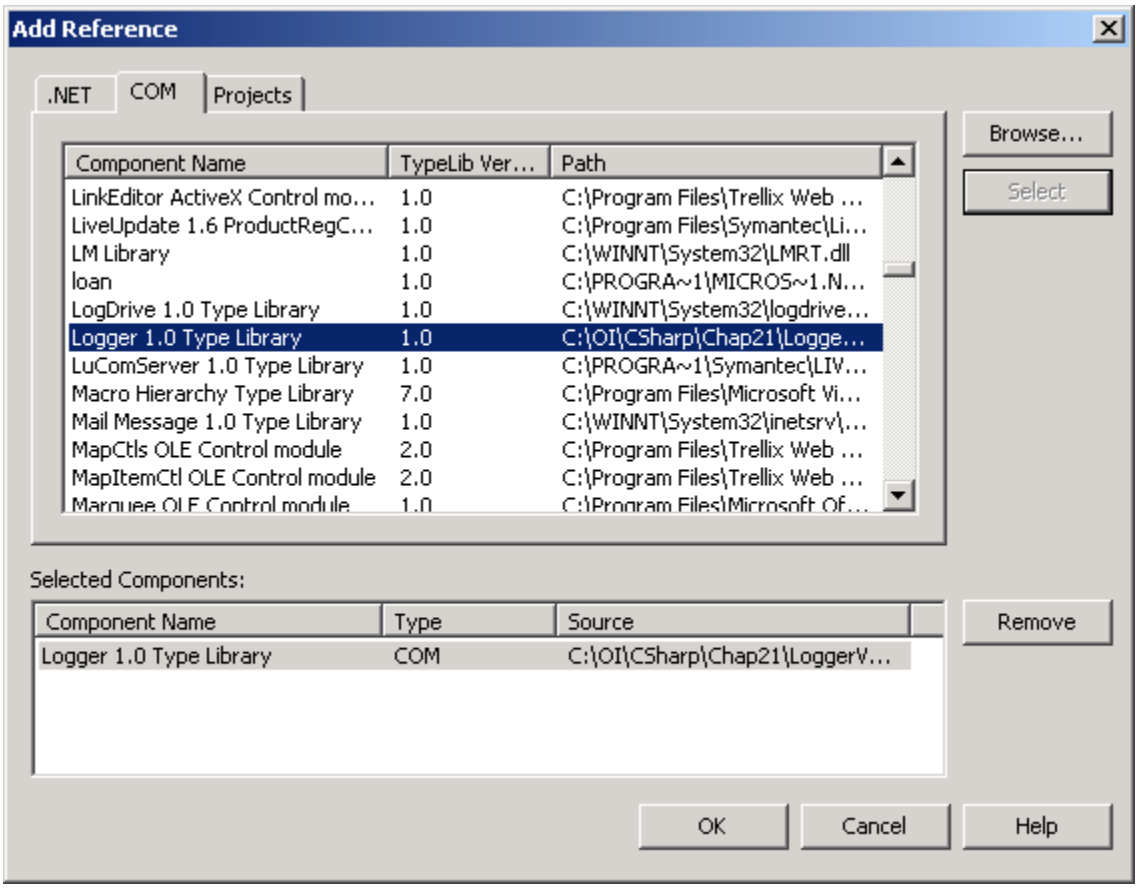
```
// TestLogger.cs

using LOGGERLib;

public class TestLogger
{
   public static void Main()
   {
      Log log = new Log();
      log.Write("first line");
      log.Write("second line");
   }
}
```

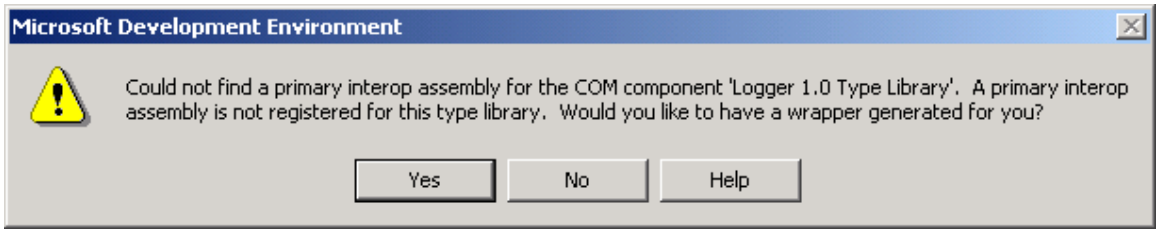
Creating the Test Program (Cont'd)

- **In Solution Explorer right-click over References and choose Add References from the context menu.**
  - Click on the COM tab and click on "Logger 1.0 Type Library." Click Select. See the figure. Click OK.



Creating the Test Program (Cont'd)

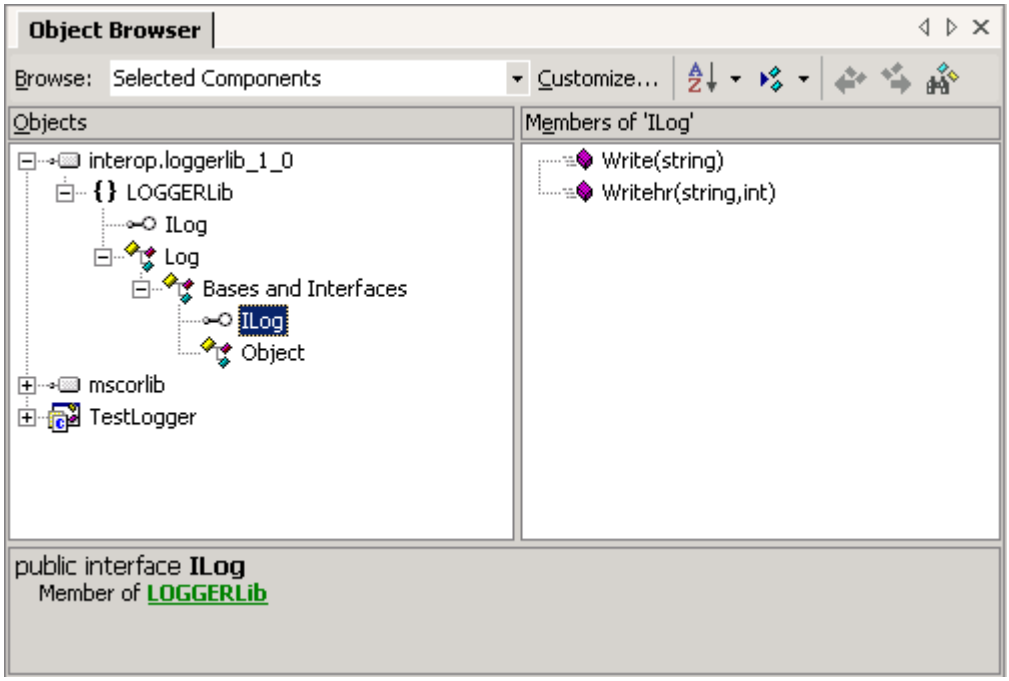
- **A dialog will come up talking about a "primary interop assembly" and asking you if you would like to have a wrapper generated for you. See the figure. Click Yes. (The primary interop assembly is the DLL *Interop.LOGGERLib_1_0.dll*, which was in the *bin\Debug* directory of the *TestLogger* program that was provided for you.)**



- **You should now be able to build and run the project.**
  - You should see two lines added to the log file **c:\logfile.txt**.

Creating the Test Program (Cont'd)

- **You can view the LOGGERLib library in the Object Browser, which you can bring up from the menu View | Other Windows | Object Browser. See the figure.**



Summary

- **Modern large applications are rarely monolithic but instead are made up of a number of executable units.**
- **In the Windows environment an application will normally consist of an EXE and a number of DLLs.**
- **In this chapter we saw how to create class library DLLs, or *components*, which expose classes and their methods to external programs.**
- ***Assemblies* are the unit of deployment in .NET.**
- **An assembly can be a single EXE or DLL, or it can consist of several files, called *modules*. An assembly also contains a *manifest*, which describes how the elements of the assembly relate to each other and to external elements.**
- **An application in .NET can be composed of assemblies built using different languages, and you can even inherit across languages.**
- **You can call a COM component from .NET.**