BUILDING C# APPLICATIONS

The C# Command-Line Compiler (csc.exe)

Configuring the C# Command-Line Compiler

To equip your development machine to compile *.cs files from any directory, follow these steps (which assume a Windows XP installation; Windows NT/2000 steps will differ slightly):

1. Right-click the My Computer icon and select Properties from the pop-up menu.

2. Select the Advanced tab and click the Environment Variables button.

3. Double-click the Path variable from the System Variables list box.

4. Add the following line to the end of the current Path value (note each value in the Path variable is separated by a semicolon):

C:\Windows\Microsoft.NET\Framework\v2.0.50215

Building C# Applications Using csc.exe

To build a simple single file assembly named TestApp.exe using the C# command-line compiler and Notepad. First, you need some source code. Open Notepad and enter the following:

```
// A simple C# application.
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Once you have finished, save the file in a convenient location (e.g., C:\CscExample) as TestApp.cs.

Each possibility is represented by a specific flag passed into csc.exe as a command-line parameter see below table which are the core options of the C# compiler.

Output-centric Options of the C# Compiler

Option	Meaning in Life
/out	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input *.cs file (in the case of a *.dll) or the name of the type containing the program's Main() method (in the case of an *.exe).
/target:exe	This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
/target:library	This option builds a single-file *.dll assembly.
/target:module	This option builds a *module*. Modules are elements of multifile assemblies (fully described in Chapter 11).
/target:winexe	Although you are free to build Windows-based applications using the /target:exe flag, the /target:winexe flag prevents a console window from appearing in the background.

To compile TestApp.cs into a console application named TestApp.exe enter

csc **/target:exe** TestApp.cs

Here explicitly not specify an /out flag, therefore the executable will be named TestApp.exe, given that TestApp is the class defining the program's entry point (the Main() method).

Note that: C# compiler flags support an abbreviated version, such as **/t** rather than **/target**

csc **/t:exe** TestApp.cs

default output used by the C# compiler, so compile TestApp.cs simply by typing

csc TestApp.cs

TestApp.exe can now be run from the command line a shows o/p as;

C:\TestApp

Testing! 1, 2, 3

Referencing External Assemblies

- To compile an application that makes use of types defined in a separate .NET assembly. Reference to the System.Console type mscorlib.dll is *automatically referenced* during the compilation process.
- To illustrate the process of referencing external assemblies the TestApp application to display windows Forms message box.
- At the command line, you must inform csc.exe which assembly contains the "used" namespaces.
- Given that you have made use of the MessageBox class, you must specify the **System.Windows.Forms.dll** assembly using the /reference flag (which can be abbreviated to /r):

csc /**r:System.Windows.Forms.dll** testapp.cs

Then console o/p can be seen as

Compiling Multiple Source Files with csc.exe

Most projects are composed of multiple *.cs files to keep code base a bit more flexible. Assume you have class contained in a new file named HelloMsg.cs:

```
// The HelloMessage class
using System;
using System.Windows.Forms;
class HelloMessage
{
public void Speak(){
MessageBox.Show("Hello...");
}
 }
```

Now, create TestApp.cs file & write below code

```
using System;
class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
HelloMessage h = new HelloMessage();
h.Speak();
}
}
```

You can compile your C# files by listing each input file explicitly:

csc /r:System.Windows.Forms.dll **testapp.cs hellomsg.cs**

As an alternative, the C# compiler allows you to make use of the wildcard character (*) to inform csc.exe to include all *.cs files contained in the project directory as part of the current build:

csc /r:System.Windows.Forms.dll ***.cs**

When you run the program again, the output is identical. The only difference between the two applications is the fact that the current logic has been split among multiple files.

Referencing Multiple External Assemblies

- To reference numerous external assemblies using csc.exe, simply list each assembly using a semicolon-delimited list. You don't need to specify multiple external assemblies for the current example, but some sample usage follows:

  csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs

Generating Bug Reports

- The raw C# compiler provides a helpful flag named /bugreport.

```
csc /bugreport:bugs.txt *.cs
```

- When you specify /bugreport, you will be prompted to enter corrective information for the possible error(s) at hand, which will be saved (along with other details) into the file you specify.
- For example:

```
public static void Main()
{
   ...
   HelloMessage h = new HelloMessage();
   // Note lack of semicolon below!
   h.Speak()   // <= Error!
}
```

- When you recompile this file using the /bugreport flag, you are prompted to enter corrective action for the error at hand in command prompt.
- If you were to open the resulting *.txt file you would find a complete report regarding this compilation cycle And shows message as the **“; expected”.**

Remaining C# Compiler Options

Options of the C# Command Line Compiler

Command Line Flag of csc.exe	Meaning in Life
@	Allows you to specify a response file used during compilation
/? or /help	Prints out the list of all command line flags of csc.exe (which is basically the information in this table)
/addmodule	Used to specify the modules to add to a multifile assembly
/baseaddress	Used to specify the preferred base address at which to load a *.dll
/bugreport	Used to build text-based bug reports for the current compilation
/checked	Used to specify whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time
/codepage	Used to specify the code page to use for all source code files in the compilation

/debug	Forces csc.exe to emit debugging information
/define	Used to define preprocessor symbols
/doc	Used to construct an XML documentation file
/filealign	Specifies the size of sections in the output file
/fullpaths	Specifies the absolute path to the file in compiler output
/incremental	Enables incremental compilation of source code files
/lib	Specifies the location of assemblies referenced via /reference
/linkresource	Used to create a link to a managed resource
/main	Specifies which Main() method to use as the program's entry point, if multiple Main() methods have been defined in the current *.cs file set
/nologo	Suppresses compiler banner information when compiling the file
/nostdlib	Prevents the automatic importing of the core .NET library, mscorlib.dll
/noconfig	Prevents the use of *.rsp files during the current compilation
/nowarn	Suppress the compiler's ability to generate specified warnings
/optimize	Enable/disable optimizations
/out	Specifies the name of the output file
/recurse	Searches subdirectories for source files to compile
/reference	Used to reference an external assembly
/resource	Used to embed .NET resources into the resulting assembly
/target	Specifies the format of the output file
/unsafe	Compiles code that uses the C# "unsafe" keyword
/utf8output	Displays compiler output using UTF-8 encoding
/warn	Used to set warning level for the compilation cycle
/warnaserror	Used to automatically promote warnings to errors
/win32icon	Inserts an .ico file into the output file
/win32res	Inserts a Win32 resource into the output file

Working with csc.exe Response Files

- C# response files contain all the instructions to be used during the compilation of current build or project. By convention, these files end in a *.rsp (response) extension.
- Assume that you have created a response file named TestApp.rsp that contains the following arguments (as you can see, comments are denoted with the # character):

```
# This is the response file for the TestApp.exe app
# External assembly references.
/r:System.Windows.Forms.dll
# output and files to compile (using wildcard syntax).
/target:exe /out:TestApp.exe *.cs
```

- Now, assuming this file is saved in the same directory as the C# source code files to be compiled, you are able to build your entire application as follows (note the use of the @ symbol):

```
csc @TestApp.rsp
```

- To specify multiple *.rsp files as input (e.g., csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp).If you take this approach, do be aware that the compiler processes the command options as they are encountered! Therefore, command-line arguments in a later *.rsp file can override options in a previous response file.
- Also note that flags listed explicitly on the command line before a response file will be overridden by the specified *.rsp file. Thus, if you were to enter

```
csc /out:MyCoolApp.exe @TestApp.rsp
```

The name of the assembly would still be TestApp.exe (rather than MyCoolApp.exe), given the /out:TestApp.exe flag listed in the TestApp.rsp response file. However, if you list flags after a response file, the flag will override settings in the response file.

Note The /reference flag is cumulative. Regardless of where you specify external assemblies (before, after, or within a response file) the end result is a summation of each reference assembly.

The Default Response File (csc.rsp)

- C# compiler has an associated default response file (csc.rsp), which is located in the same directory as csc.exe itself
  (e.g., C:\Windows\Microsoft.NET\Framework\v2.0.50215). If you were to open this file using Notepad, you will find that numerous .NET assemblies have already been specified using the /r: flag.
- When you are building your C# programs using csc.exe, this file will be automatically referenced, even when you supply a custom *.rsp file.
- Given the presence of the default response file, the current TestApp.exe application could be successfully compiled using the following command set (as System.Windows.Forms.dll is referenced within csc.rsp):

```
csc /out:TestApp.exe *.cs
```

- In the event that you wish to disable the automatic reading of csc.rsp, you can specify the /noconfig option:

```
csc @TestApp.rsp /noconfig
```

The Command-Line Debugger (cordbg.exe)

- This tool provides dozens of options that allow you to debug your assembly. You may view them by specifying the /? flag:

```
cordbg /?
```

- The flags recognized by cordbg.exe (with the alternative shorthand notation) once you have entered a debugging session.

A Handful of Useful `cordbg.exe` *Command-Line Flags*

Flag	Meaning in Life
`b[reak]`	Set or display current breakpoints.
`del[ete]`	Remove one or more breakpoints.
`ex[it]`	Exit the debugger.
`g[o]`	Continue debugging the current process until hitting next breakpoint.
`o[ut]`	Step out of the current function.
`p[rint]`	Print all loaded variables (local, arguments, etc.).
`si`	Step into the next line.
`so`	Step over the next line.

Debugging at the Command Line

- Before you can debug your application using cordbg.exe, the first step is to generate debugging symbols for your current application by specifying the /debug flag of csc.exe.
- For example, to generate debugging data for TestApp.exe, enter the following command set:

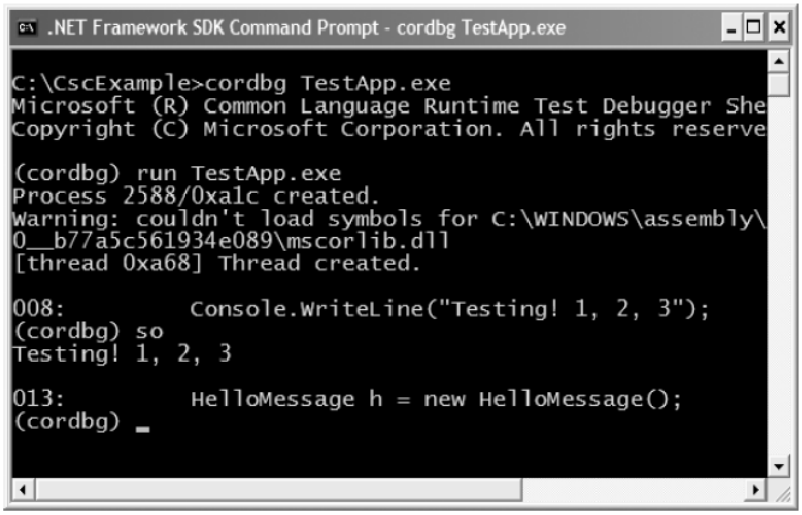
```
csc @testapp.rsp /debug
```

- This generates a new file named (in this case) testapp.pdb. If you do not have an associated *.pdb file, it is still possible to make use of cordbg.exe; however, you will not be able to view your C# source code during the process.

- Once you have generated a *.pdb file, open a session with cordbg.exe by specifying your .NET assembly as a command-line argument (the *.pdb file will be loaded automatically):

```
cordbg.exe testapp.exe
```

Now debugging mode and may apply any number of cordbg.exe flags at the (cordbg) command prompt:



Debugging with cordbg.exe

When you wish to quit debugging with cordbg.exe, simply type exit (or shorthand **ex**).

Using the Visual Studio .NET IDE

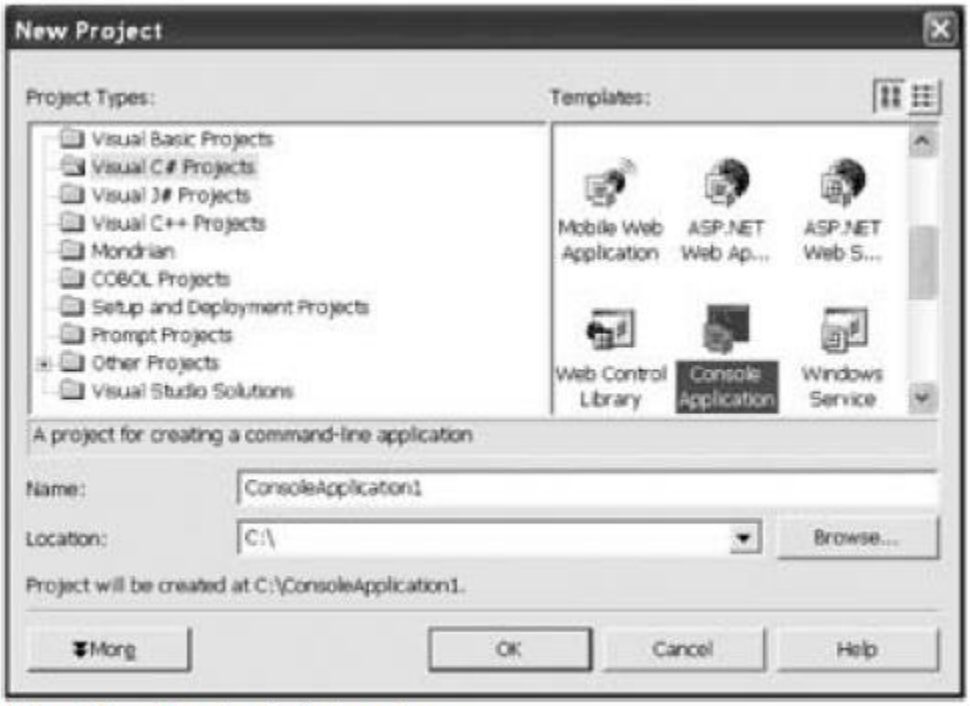
- This product allows you to build applications using any number of .NET-aware (and unaware) languages. Thus, you are able to use VS .NET when building C#, J#, VB .NET, MFC, ATL (4.0) or traditional C-based Win32 applications.
- The one thing you *cannot* do is build a traditional Visual Basic 6.0 application using VS .NET.
- If you want to create classic VB 6.0 COM servers (or any additional VB 6.0 project types) you need to make use of the Visual Basic 6.0.

The VS .NET Start Page

- By default, the first thing you see when you launch Visual Studio .NET is the Start Page. For the present, you need only be concerned with the Projects and My Profile tabs (the remaining options allow you to connect to the Internet to view online resources, obtain product updates, and whatnot).
- The Projects view allows you to open existing projects as well as create a brand-new project workspace. Be aware that these options are also available via the File menu.
- When you click the My Profile tab, Here, you are able to control how the VS .NET IDE should be configured each time you launch the tool.
- For example, the Keyboard Scheme drop-down list allows you to control which keyboard mapping should be used. If you want, you can opt to have your shortcut keys configured to mimic VB 6.0, Visual C++ 6.0, or the default VS .NET settings.
- Other options allow you to configure how online Help should be filtered (and displayed), as well as how the core IDE windows (i.e., Properties, Toolbox, etc.) should dock themselves. To check things out first-hand, take a moment to select the various options found under the Window Layout drop-down list and find a look and feel you are comfortable with.
- Finally, be aware that if you close the Start Page window (and want to get it back), access the Help | Show Start Page menu option.

Creating VS.NET Project Solution

- Open the New Project dialog box by clicking the New Project button from the Start Page, or by choosing the File | New | Project menu selection. In figure, project types are grouped (more or less) by language.



The New Project dialog box

Core Project Workspace Types

Project Type	Meaning in Life
Windows Application	This project type represents a Windows Forms application.
Class Library	This option allows you to build a single file assembly (*.dll).
Windows Control Library	This type of project allows you to build a single file assembly (*.dll) that contains custom Windows Forms Controls (as you can guess, this is the .NET version of a COM-based ActiveX control).
ASP.NET Web Application	Select this option when you want to build an ASP.NET Web application.
ASP.NET Web Service	This option allows you to build a .NET Web Service. As shown later in this text, a Web Service is a block of code, reachable using HTTP requests.
Web Control Library	VS .NET also allows you to build customized Web controls. These GUI widgets are responsible for emitting HTML to a requesting browser.
Console Application	The good old command window. As mentioned, you spend the first number of chapters working with this type of project type, just to keep focused on the syntax and semantics of C#.
Windows Services	.NET allows you to build NT/2000 services. As you may know, these are background worker applications that are launched during the OS boot process.

Building a VS .NET Test Application

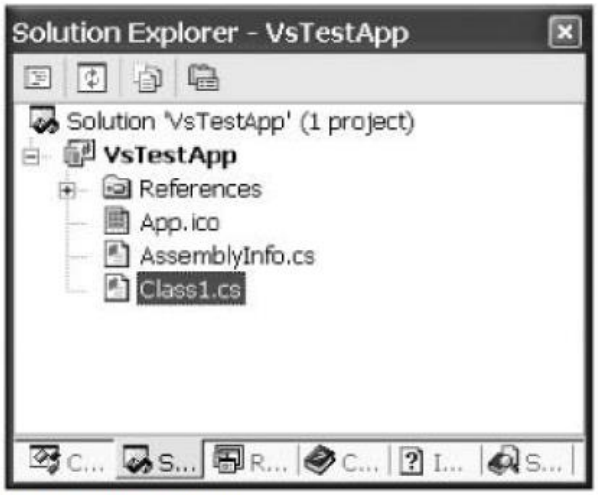
Create a brand-new C# Console Application project named Vs TestApp. Once you hit OK, you will find a new folder has been created that contains a number of starter files and project subdirectories. Below table describes the key items:

The Structure of a VS .NET Console Application

Generated Item	Meaning in Life
\bin\Debug	The \bin\Debug folder contains the debug version of your compiled .NET assembly. If you configure a release build, a new folder (\bin\Release) will be generated that contains a copy of your assembly, stripped of any debugging information. You can switch between debug and release builds using the "Build | Configuration Manager" menu selection.
\obj*	Under the \obj folder there are numerous subfolders used by VS .NET during the compilation process. You are always safe to ignore this folder set.
App.ico	An *.ico file used to specify the icon for the current program.
AssemblyInfo.cs	This file allows you to establish assembly-level attributes for your current project.
Class1.cs	This file is your initial class file.
*.csproj	This file represents a C# project that is loaded into a given solution.
*.sln	This file represents the current VS .NET solution (which by definition is a collection of individual projects).

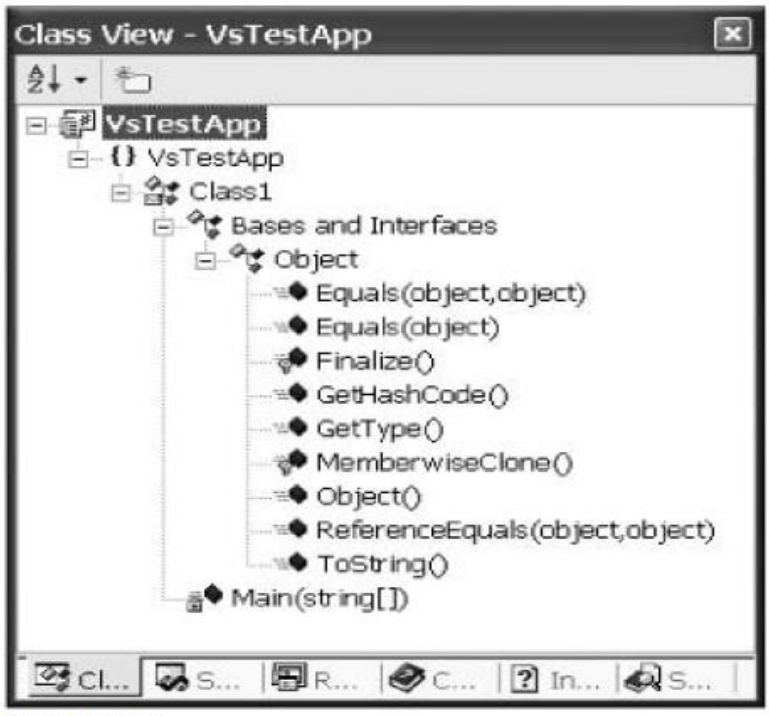
Examining the Solution Explorer Window

- VS.NET logically arranges a given project using a solution metaphor. Simply put, a "solution" is a collection of one or more "projects."
- Each project contains any number of source code files, external references, and resources that constitute the application as a whole.
- The *.sln file can be opened using VS .NET to load each project in the workspace. Using the Solution Explorer window, you are able to view and open any such item. Notice the default name of your initial class is "Class1.cs."



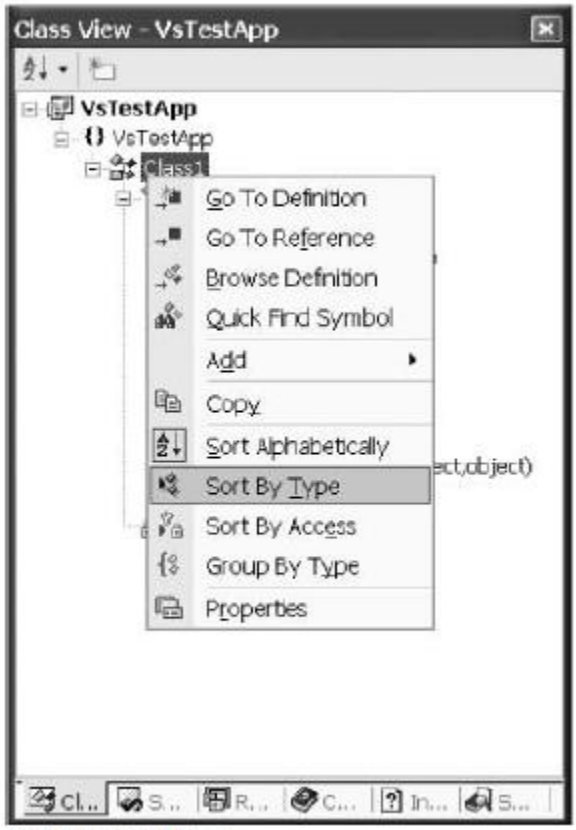
The Solution Explorer

- Solution explorer window provides a Class View tab, which shows the object oriented view of your project. One nice feature of this perspective is that if you double click on a given icon, you are launched to the definition of the given type member.



Class View

When you right-click a given item, you activate a context-sensitive pop-up menu. The menu lets you access a number of tools that allow you to configure the current project settings and sort items in a variety of ways. For example, right-click the Solution node from Class View and check out below figure



Type sorting options

Configuring a C# Project

- In the Solution Explorer tab. Right-click the VS TestApp project node, and select Properties from the context menu. This launches the all-important Project Property Page.
- That dialog box provides a number of intriguing settings, which map to various flags of the command line compiler. To begin, when you select the General node, you are able to configure the type of output file that should be produced by csc.exe.
- You are also able to configure which item in your application should be marked as the "Startup object" (meaning, the class in the application that contains the Main() method).

- By default, this value is set to Not Set, which will be just fine unless your application happens to contain multiple class types that define a method named Main() (which will seldom, if ever, be the case).
- Finally, notice that the General node also allows you to configure the 'default' namespace for this particular project (when you create a new project with VS .NET, the default namespace is the same as your project name).

The Properties Window

- This window allows you to interact with a number of characteristics for the item that has the current focus. This item may be an open source code file, a GUI widget, or the project itself.
- For example, to change the name of your initial *.cs file, select it from the Solution Explorer and configure the FileName property

File names may be changed using the Properties window.

- You can change the name of your initial class, select the Class1 icon from ClassView and edit the Properties window accordingly.

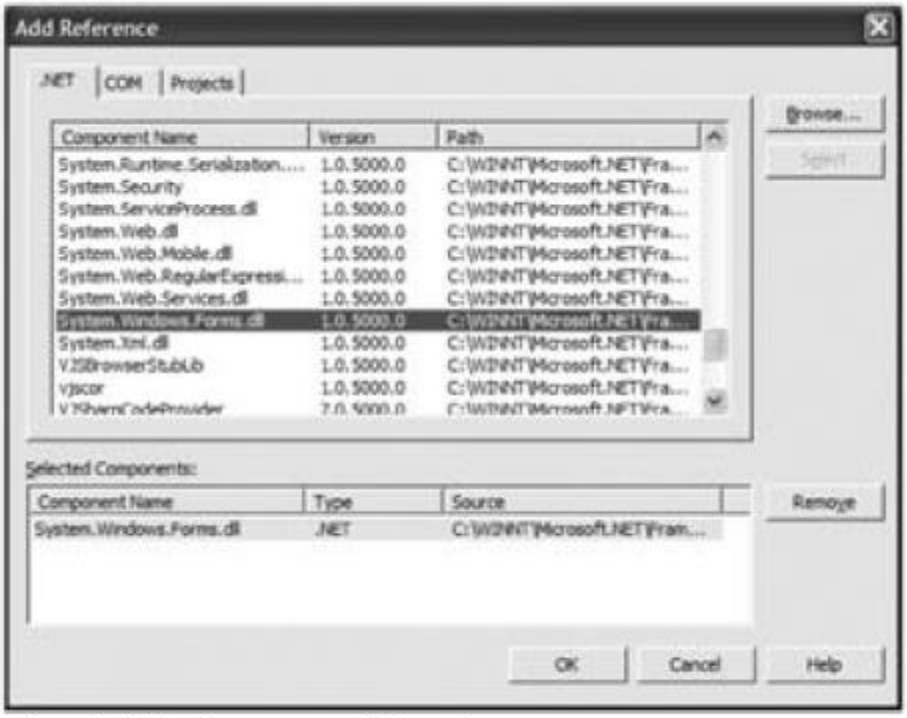
Adding Some Code

Now that you have configured your new C# project workspace, you can add source code. Within the Main() method, print a line to the command window and display a Windows Forms message box (don't forget to specify the correct C# using statements!):

```
using System;
using System.Windows.Forms;
namespace VsTestApp
{
    class MainClass
    {
        // See note below...
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("Hello again!");
            MessageBox.Show("Yo!");
        }
    }
}
```

Referencing External Assemblies via VS .NET

When you need to add external references (such as System.Windows.Forms.dll) into your current project, access the Project | Add Reference menu selection (or right-click the References node from the Solution Explorer window). Whichever way you go, you end up with the dialog box shown in figure



The Add References dialog box

Inserting New C# Type Definitions

When you want to add new custom types (such as a class, interface, or enumeration) to your current project you are always free to insert a blank *.cs file and manually flesh out the details of the new item.

As an alternative, you are also free to use the Project | Add Class menu selection and modify the initial code accordingly. Pick your poison, and insert a new class named HelloClass. Add a simple methodnamed SayHi():

```
using System;
using System.Windows.Forms;

namespace VsTestApp
{
    public class HelloClass
    {
       public HelloClass() {}
       public void SayHi()
       {
           MessageBox.Show("Hello from HelloClass...");
       }
    }
}
```

Now, update your existing main() method to create a new instance of this type, and call the sayhi() member:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello again!");
    // Make a HelloClass type.
    HelloClass h = new HelloClass();
    h.SayHi();
}
```

Once you have done so, you are able to run your new VS .NET application using the Debug | Start Without Debugging menu option.

Debugging with the Visual Studio .NET IDE

- Visual Studio .NET contains an integrated debugger, which provides the same functionality of cordbg.exe using a friendly user interface. To illustrate the basics, begin by clicking in the far left gray column of an active code window to insert a breakpoint.
- When you initiate a debug session (via the Debug | Start menu selection), the flow of execution halts at each breakpoint.
- Using the Debug toolbar (or the corresponding keystroke commands), you can step over, step into, and step out of a given function.
- The integrated debugger hosts a number of debug-centric windows (e.g., Call Stack, Autos, Locals, Breakpoints, Modules, Exceptions, and so forth).To show or hide a particular window, simply access the Debug | Windows menu selection while in a debugging session.

"Running" Versus "Debugging"

- When you run an application, you are instructing VS .NET to ignore all breakpoints, and to automatically prompt for a keystroke before terminating the current console window.
- If you debug a project that does not have any breakpoints set, the console application terminates so quickly that you will be unable to view the output.
- To ensure that the command window is alive regardless of the presence of a given breakpoint, one surefire technique is to simply add the following code at the end of the Main() method:

```
static void Main(string[] args)
{
...
// Keep console window up until user hits return.
Console.ReadLine();
}
```

Other Key Aspects of the VS .NET IDE

Examining the Server Explorer Window

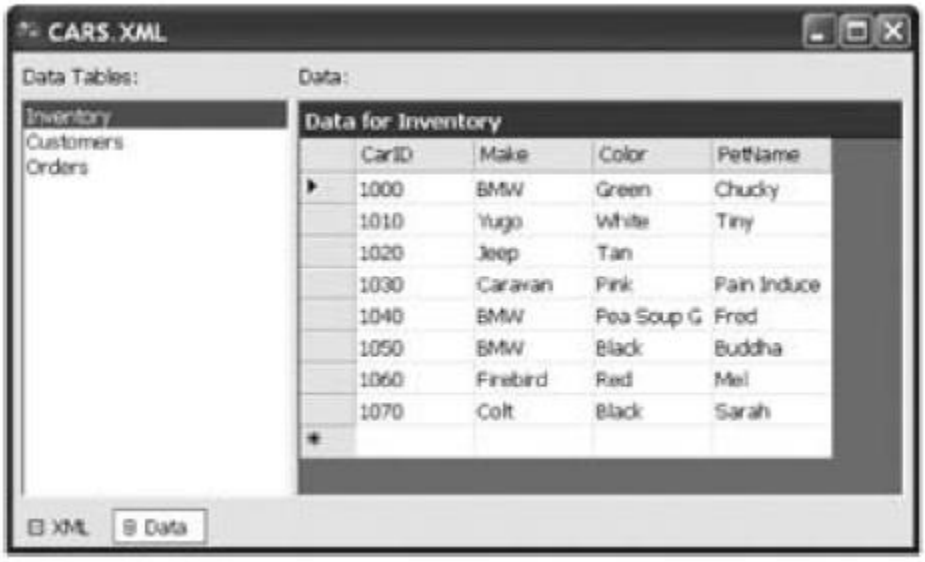
- One extremely useful aspect of Visual Studio .NET is the Server Explorer window, which can be accessed using the View menu.

The Server Explorer window

- This window can be thought of as the command center of a distributed application you may be building.
- Using the Server Explorer, you can attach to and manipulate local and remote databases (and view any of the given database objects), examine the contents of a given message queue, and obtain general machine-wide information (such as seeing what services are running and viewing information in the Windows event log).

XML-Related Editing Tools

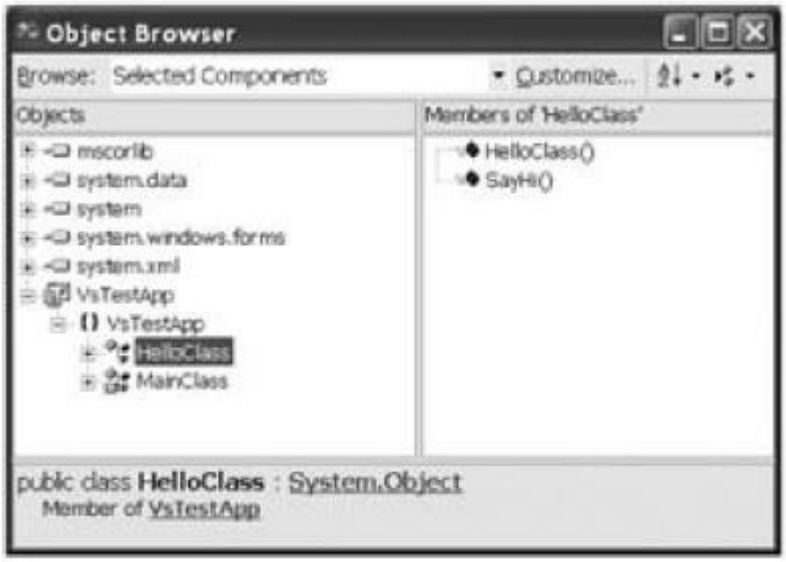
- Visual Studio .NET also provides numerous XML-related editors. As a simple example, if you insert a new XML file into your application (via the Project | Add New Item... menu selection), you are able to manipulate the underlying XML using GUI design-time tools (and related toolbars).



The integrated XML editor

The Object Browser Utility

- The object browser allows you to view the namespaces, types, and type members of each assembly referenced by the current solution.



The integrated Object Browser

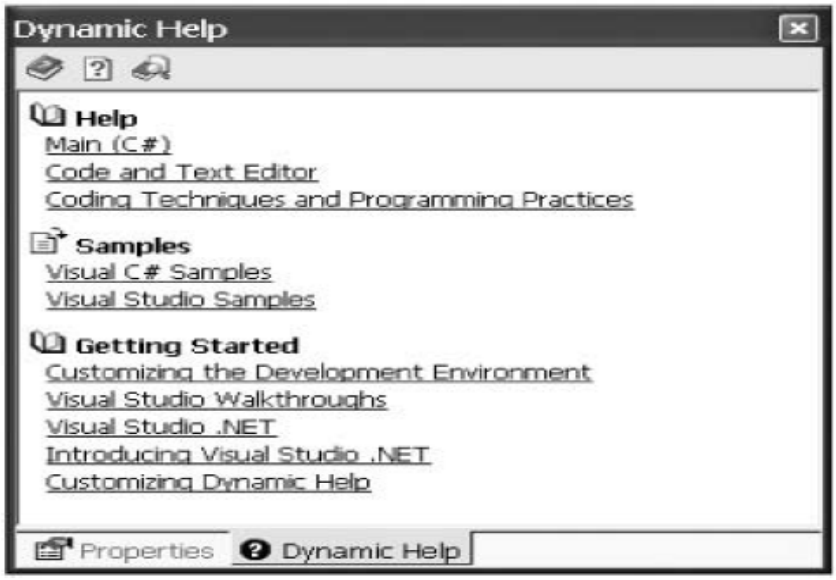
Database Manipulation Tools

- Integrated database support is also part of the VS .NET IDE. Using the Server Explorer window, you can open and examine any database object from within the IDE. For example, below figure shows a view of the Inventory table of the Cars database you build during our examination of ADO.NET.

Integrated database editors

Integrated Help

- The final aspect of the IDE is integrated Help system.
- The .NET documentation is extremely good, very readable, and full of useful information. Given the huge number of predefined .NET types (which number well into the thousands) you must be willing to roll up your sleeves and dig into the provided documentation.
- If you resist, you are doomed to a long, frustrating, and painful existence. To prevent this from happening, VS .NET provides the Dynamic Help window, which changes its contents (dynamically!) based on what item (window, menu, source code keyword, etc.) is currently selected.
- For example, if you place the cursor on Main() method, the Dynamic Help window displays what's shown in figure below



Visual Studio .NET Integrated Help

C# "Preprocessor" Directives

- C# supports the use of various symbols that allow you to interact with the compilation process.
- Preprocessing directives are processed as part of the lexical analysis phase of the compiler. Nevertheless, the syntax of the "C# preprocessor" is identical to that of the other members of the C family. Specifically, C# supports the tokens you see in below table

C# Preprocessor Directives

C# Preprocessor Symbol	Meaning in Life
#define, #undef	Used to define and un-define conditional compilation symbols.
#if, #elif, #else, #endif	Used to conditionally skip sections of source code (based on specified compilation symbols).
#line	Used to control the line numbers emitted for errors and warnings.
#error, #warning	Used to issue errors and warnings for the current build.
#region, #endregion	Used to explicitly mark sections of source code. Under VS .NET, regions may be expanded and collapsed within the code window, other IDEs (including simple text editors) will ignore these symbols.

Specifying Code Regions

- The most interesting of all preprocessor directives are **#region** and **#endregion**. Using these tags, you are able to specify a block of code that may be hidden from view and identified by a friendly textual marker.
- The use of regions can help keep lengthy *.cs files more manageable. For example, you could create one region for a type's constructors, another for type properties, and yet another for internal helper classes.
- The following class has nested two internal helper types, which have been wrapped in a region block:

```
class ProcessMe
{
...
    // Nested types will be examined later.
    #region Stuff I Don't Care About
    public class MyHelperClass
    { // stuff...
    }
    public interface MyHelperInterface
    {// stuff...
    }
    #endregion
}
```

When you place your mouse cursor over a collapsed region, you are provided with a snapshot of the code lurking behind.

Regions at work

Conditional Code Compilation

- The preprocessor directives (#if, #elif, #else, #endif) allows you to conditionally compile a block of code, based on predefined symbols.
- The classic use of these directives is to include additional (typically diagnostic-centric) code only compiled under debug builds.
- For example, that when your current application is configured under a debug build, you wish to dump out a number of statistics to the console:

```
using System;
class ProcessMe
{
...
    static void Main(string[] args)
    {
        // Are we in debug mode?
        #if (DEBUG)
        Console.WriteLine("App directory: {0}",
            Environment.CurrentDirectory);
        Console.WriteLine("Box: {0}",
            Environment.MachineName);
        Console.WriteLine("OS: {0}",
            Environment.OSVersion);
        Console.WriteLine(".NET Version: {0}",
            Environment.Version);
        #endif
    }
}
```

- Here, we are checking for a symbol named DEBUG. If it is present, we dump out a number of interesting statistics using some (quite helpful) static members of the System.Environment class.
- If the DEBUG symbol is not defined, the code placed between #if and #endif will not be compiled into the resulting assembly, and effectively ignored.

Where exactly is the DEBUG symbol defined?

- If you wish to define this symbol on a file by file basis, you are able to make use of the #define preprocessor symbol as follows (do note that the #define directive *must* be listed before anything else in the *.cs source code file):

```
#define DEBUG
using System;
namespace Preprocessor
{
    class ProcessMe
    {
        static void Main(string[] args)
        {
            // Are we in debug mode?
            #if (DEBUG)
            // Same code as before...
            #endif
        }
    ...
    }
}
```

- To the DEBUG symbol to apply for each *.cs file in your application, you can simply select a Debug build of the current project using the IDE's Standard toolbar or using the Build | Configuration Manager Menu selection.

Issuing Warnings and Errors

These directives allow you to instruct the C# compile to generate a warning or error on the fly. For example, assume that you wish to ensure that when a given symbol (such as DEBUG) is defined in the current project, a compiler warning is issued.

```
static void Main(string[] args)
{
    // Are we in debug mode?
    #if (DEBUG)
    #warning Beware!  Debug is defined...configure release build.
    // Same code as before...
    #endif
}
```

Altering Line Numbers

The final directive, #line. Basically, this directive allows you to alter the compiler's recognition of #line numbers during its recording of compilation warnings and errors. The #line directive allows you

1. To specify the name of a file to dump said compiler anomalies.
2. To reset the default line numbering, you may specify the default tag.

By way of a simple example, ponder the following code:

```
// The warning below will be issued on line
// Set line number to 3000.
#line 3000
#warning "Custom warning on line 3000 (for no good reason)...
// Resume default line numbering.
#line default
```

On a related note, understand that you are able to view the "real" line numbering for a given source code file by selecting the "Tools | Options" menu command and enabling the Line Numbers check box

The System.Environment Class

- The System.Environment class allows you to obtain a number of details regarding the context of the operating system hosting your .NET application using various static members.
- The System.Environment class can be used to retrieve information like below :
    - Command-line arguments.
    - The exit code.
    - Environment variable settings.
    - Contents of the call stack.
    - Time since last system boot.
    - The version of the common language runtime.

```
// Here are some (but not all) of the interesting
// static members of the Environment class.
using System;
class PlatformSpy
{
public static int Main(string[] args)
{
// Which OS version are we running on?
Console.WriteLine("Current OS: {0} ",
Environment.OSVersion);
// Directory?
Console.WriteLine("Current Directory: {0} ",
Environment.CurrentDirectory);
// Here are the drives on this box.
string[] drives = Environment.GetLogicalDrives();
for(int i = 0; i < drives.Length; i++)
```

```
Console.WriteLine("Drive {0} : {1} ", i, drives[i]);
// Which version of the .NET platform?
Console.WriteLine("Current version of .NET: {0} ",
Environment.Version);
return 0;
}
}
```

Environment Member	Description
CommandLine	Returns a string that includes the command line arguments.
CurrentDirectory	Gets or sets the full path of the current directory in which this process starts with the format of drivename + backslash character + subdirectory (e.g., C:\ or C:\dir1).
ExitCode	Returns a success or failure status from a process.
HasShutdownStarted	Represents a Boolean value that reads *true* if the CLR is shutting down.
MachineName	Returns the NetBIOS name of the computer. There are two types of naming standards for a computer: NETBIOS and FQDN.
NewLine	Returns the defined NewLine string. This is mostly \r\n on Windows-related operating systems but on other operating systems this value may differ.
OSVersion	Returns an operating system object with the current version information.
StackTrace	Returns a string that shows the call stack.
SystemDirectory	Returns the fully qualified path where the underlying operating system is installed.
TickCount	Returns the number of milliseconds passed since the operating system was last started.
UserDomainName	Returns the domain name of the current user.
UserInteractive	Returns a Boolean that indicates whether the process is running in user-interactive mode or not. For service and Web applications, this value will return *false*.
UserName	Returns the name of the user who started the current thread of this process.
Version	Returns a version object from which we can learn the major, minor, build, and revision numbers of the CLR.

WorkingSet	Returns a 64-bit number representing the amount of physical memory used by the process.
Exit()	Quits this process with a specified exit code to pass to the operating system.
ExpandEnvironmentVariables()	Sets the environment variables by parsing the specified string containing environment variables quoted with the percent sign. Returns the new environment variable settings in a string.
GetCommandLineArgs()	Returns an array of strings where each element contains a command line argument passed to the process.
GetEnvironmentVariable()	Gets the value of the passed environment variable.
GetEnvironmentVariables()	Gets all environment variables and their values and returns an IDictionary interface for accessing them
GetFolderPath()	Returns the fully qualified path to the system special folder (set by the system and on demand explicitly by the user) of the passed-in-type SpecialFolder enumeration. This enumeration has values such as Favorites, History, Cookies, Startup, Startup, and InternetCache, all indicating a special folder of the operating system.
GetLogicalDrives()	Gets the names of the logical drives that exist on the computer as an array of strings.

IMPORTANT QUESTIONS

1. Explain how CSC.exe computer is used to build c# application. Explain any five flags with appropriate examples. 6m
2. What is cordbg.exe? List and explain any five command line flags recognized by crdbg.exe while running .Net assemblies under debug mode. 7m
3. What is csc.rsp file? Where is it located? 3m
4. Explain the following, w.r.t compilation of c# program in command prompt.
   I. Referencing external assemblies
   II. Compiling multiple source files
   III. Response files
   IV. Generating bug reports. 10m
5. Explain c# preprocessor directives:
   I. #region,#endregion
   II. Conditional code compilation 5m

   Or

   Explain any three preprocessor directives. 8m

6. Write a c# program to display the following information using system environment class.
   I. Current directory of application
   II. OS version
   III. Logical drivers
   IV. host name
   V. .NET version 8m
7. Explain the building of a c# application using command line compiler CSE.exe. 4m