UC BERKELEY EXTENSION

Database/Application/Programming Courses

Instructor: Michael Kremer, Ph.D.

Course Title: Building Applications using C# and .NET Framework

Course Subtitle: Object-Oriented Programming

Course Number: X428.6

Building Applications using C# and .NET Framework

Instructor: Michael Kremer, Ph.D.

E-mail: asp.net@ucb-access.org

Web Site: http://www.ucb-access.org

4th Edition

ICON KEY	
	Note
	Example
	Warning

TABLE OF CONTENT

Convention used in this Course Reader:

Names of objects, such as forms, buttons, etc. are enclosed in double quotes: “MainMenu”
Keyboard keys are enclosed in square brackets, such as [Enter] key
Identifiers are shown in italics
Keywords are shown in bold
C# syntax is shown in a gray background

CLASS 1

1. INTRODUCTION TO THE .NET PLATFORM

1.1 Overview of the .NET Framework

The .NET platform consists of the .NET framework and the tools to develop applications targeting the .NET framework.

Warning: Many times, the terms framework and platform are used interchangeably.

The .NET framework provides a common set of services that application programs written in a .NET language such a C# can use to run on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as Mac OS X and various Unix/Linux distributions.

The .NET platform overcomes the various problems encountered in previous application development frameworks, such as the Microsoft Component Object Model (COM), also referred to as "DLL Hell" and "Registry Hell". The major advantages of the .NET platform are listed below:

Interoperability with existing code: Existing COM binaries can commingle (i.e., interop) with newer .NET binaries and vice versa.

Support for numerous programming languages: .NET applications can be created using any number of programming languages (C#, Visual Basic, F#, S#, and so on).

A common runtime engine shared by all .NET-aware languages: One aspect of this engine is a well-defined set of types that each .NET-aware language understands.

Complete and total language integration: .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging of code.

A comprehensive base class library: This library provides shelter from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.

A simplified deployment model: Under .NET, there is no need to register a binary unit into the system registry. Furthermore, .NET allows multiple versions of the same *.dll to exist in harmony on a single machine.

The .NET framework is divided into two main components: the .NET Framework Class Library and the Common Language Runtime as shown in Figure 1:

- The .NET Framework Class Library (FCL) consists of segments of pre-written code called classes that provide many of the functions that you need for developing .NET applications.
- The Common Language Runtime (CLR) provides the services that are needed for executing any application that is developed with one of the .NET languages.

The .NET framework interacts with the operating system (OS) and the hardware on one side and with the applications or services on the other side.

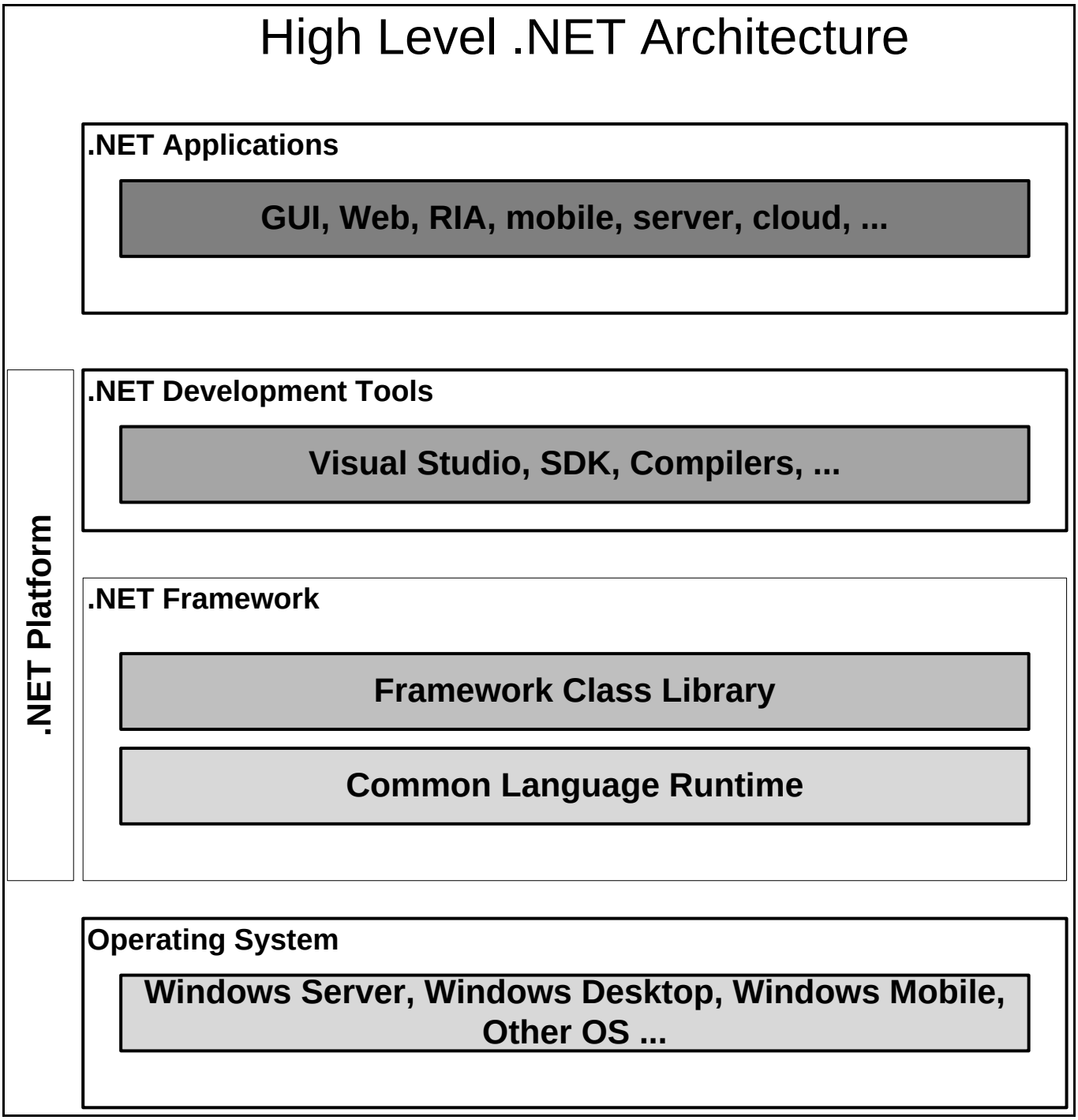


Figure 1: .NET Architecture

1.2 .NET Framework Class Library (FCL)

The .NET platform provides a framework class library that is available to all .NET programming languages. Not only does this library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering systems, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications. The structure of the Framework Class Library is shown in Figure 2:

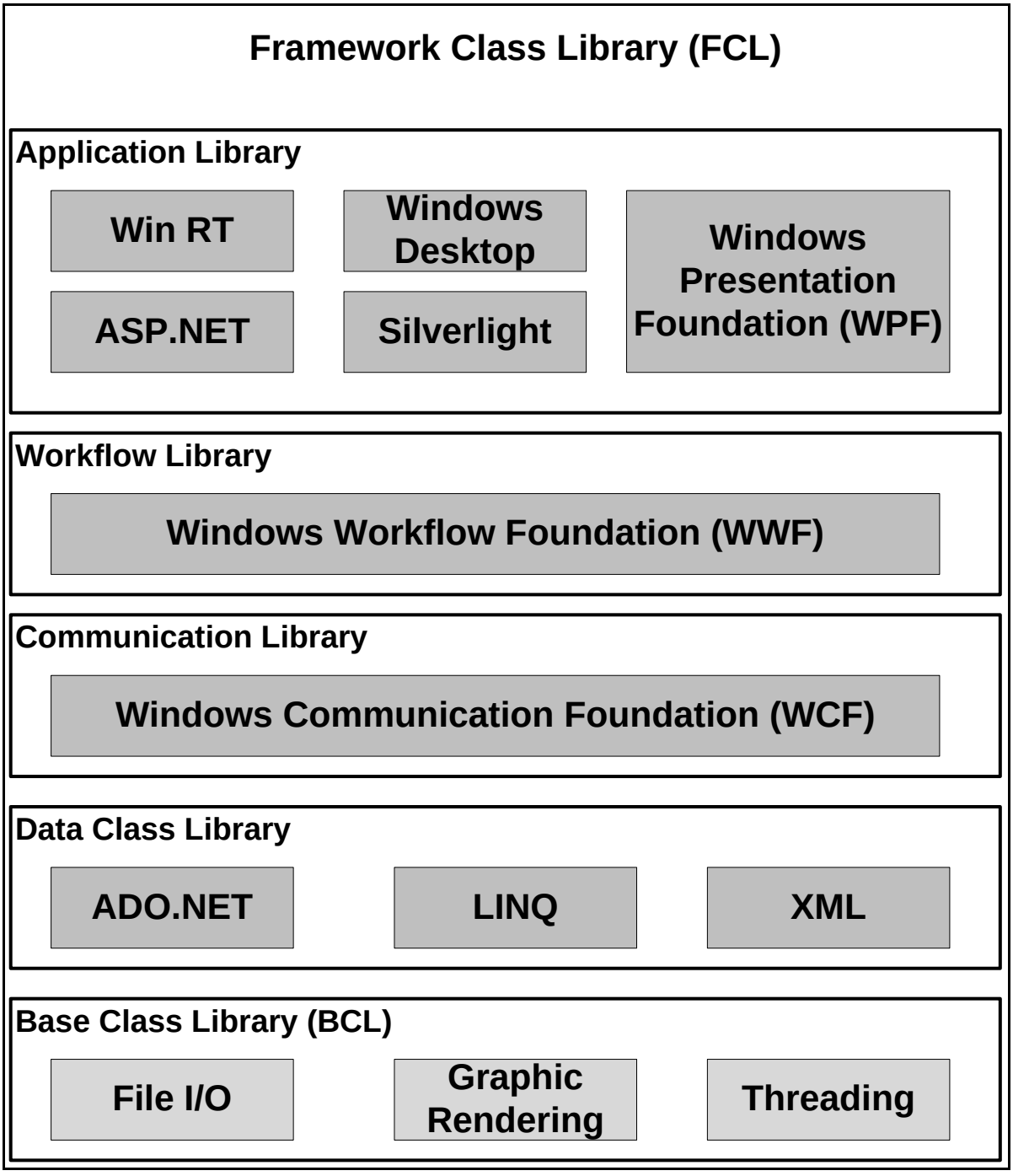


Figure 2: .NET Framework Class Library (FCL)

For example, the Windows Desktop classes are used for developing Windows Forms applications. Other classes let you work with databases, manage security, access files, and perform many other functions.

The class library is organized in a hierarchy of namespaces. Most of the built-in APIs are part of either System.* or Microsoft.* namespaces. These class libraries implement a large number of common functions, such as file reading and writing, graphic rendering, database interaction, and XML document manipulation, among others. The .NET class libraries are available to all Common Language Infrastructure (CLI) (covered in chapter 1.3) compliant languages.

 **Note:** A namespace is a logical grouping mechanism similar to organizing files into folders in the operating system. We will cover this in more detail later on.

The .NET Framework class library is divided into two main parts:

- Base Class Library
- Extended Class Library or Framework Class Library

The Base Class Library (BCL) includes a small subset of the entire class library and is the core set of classes that serve as the basic API of the Common Language Runtime. The classes in mscorlib.dll and some of the classes in System.dll and System.core.dll are considered to be a part of the BCL. The BCL classes are available in both .NET Framework as well as its alternative implementations including .NET Compact Framework, Microsoft Silverlight and Mono.

The Extended Class Library or Framework Class Library (FCL) is a superset of the BCL classes and refers to the entire class library that ships with .NET Framework. It includes an expanded set of libraries, including Windows Forms, ADO.NET, ASP.NET, Language Integrated Query, Windows Presentation Foundation, Windows Communication Foundation among others. The FCL is much larger in scope than standard libraries for languages like C++, and comparable in scope to the standard libraries of Java.

1.3 Common Language Infrastructure (CLI)

The purpose of the Common Language Infrastructure (CLI) is to provide a language-neutral platform for application development and execution, including functions for exception handling, garbage collection, security, and interoperability. By implementing the core aspects of .NET Framework within the scope of the CLI, this functionality will not be tied to a single language but will be available across the many languages supported by the framework. Microsoft's implementation of the CLI is called the Common Language Runtime, or CLR (see next chapter).

The CIL code is housed in CLI assemblies. As mandated by the specification, assemblies are stored in the Portable Executable (PE) format, common on the Windows platform for all DLL and EXE files.

The assembly consists of one or more files, one of which must contain the manifest, which has the metadata for the assembly. The complete name of an assembly (not to be confused with the filename on disk) contains its simple text name, version number, culture, and public key token. Assemblies are considered equivalent if they share the same complete name, excluding the revision of the version number. The core components of the CLI are shown in Figure 3:

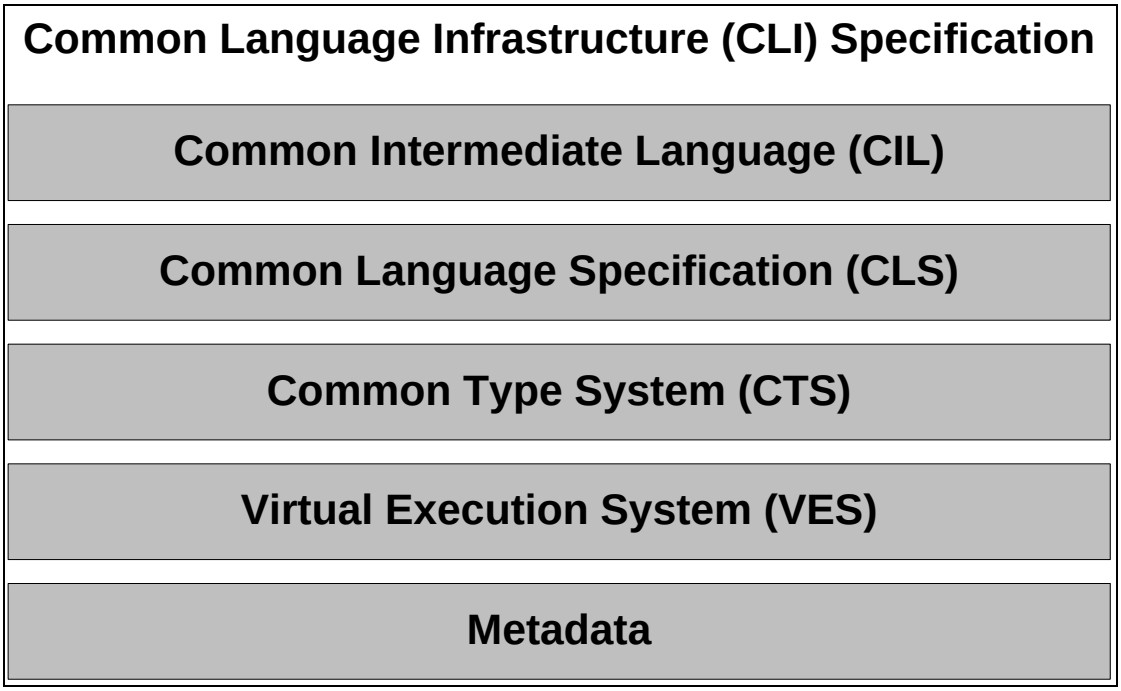


Figure 3: Common Language Infrastructure (CLI)

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

Understand that a given .NET-aware language might not support each and every feature defined by the CTS. The Common Language Specification, or CLS, is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can

agree on. Thus, if you build .NET types that only expose CLS-compliant features, you can rest assured that all .NET-aware languages can consume them.

Conversely, if you make use of a data type or programming construct that is outside of the bounds of the CLS, you cannot guarantee that every .NET programming language can interact with your .NET code library. Thankfully, it is very simple to tell your C# compiler to check all of your code for CLS compliance.

You can view the CIL by exploring an assembly using the tool ildasm.exe (discussed later). A sample method output in CIL is shown in Figure 4:

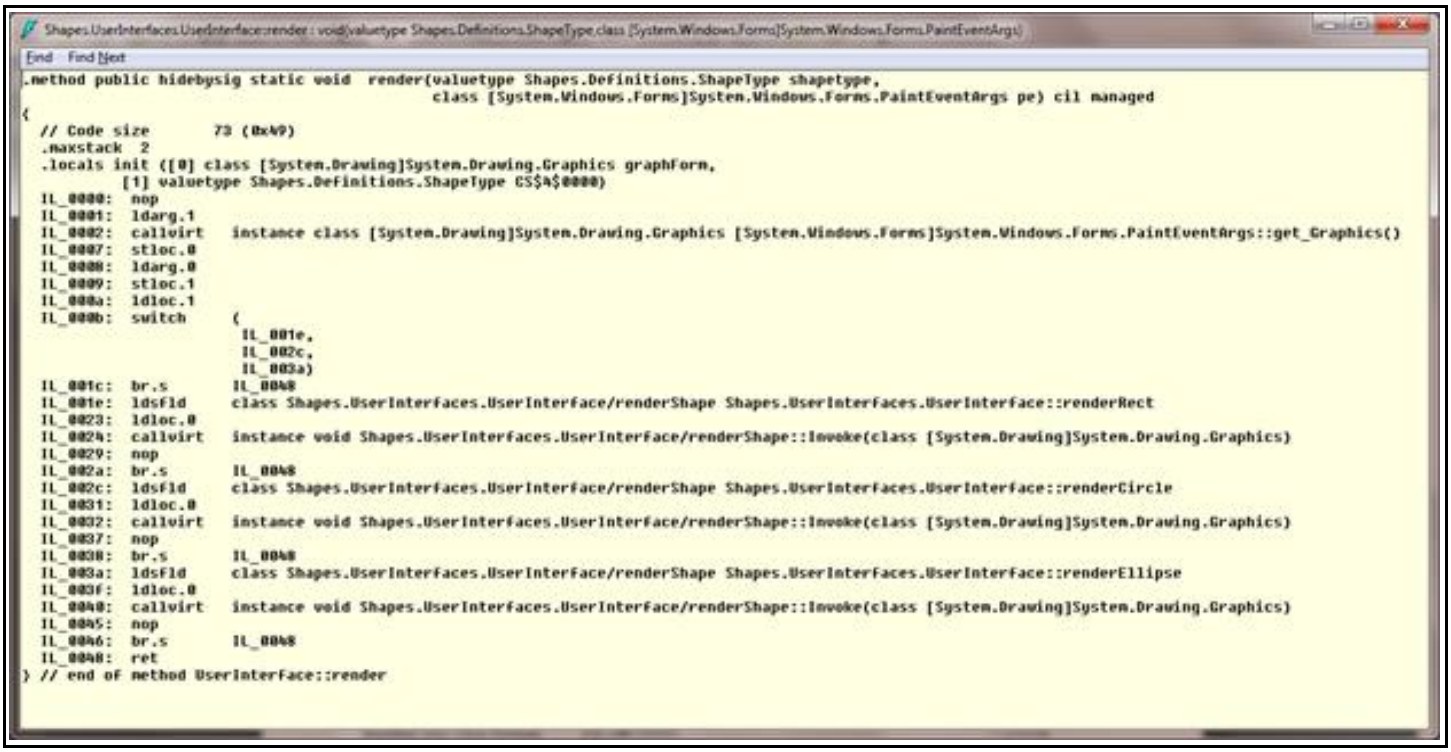


Figure 4: CIL Example

Do not worry if you are unable to even make sense out of this, the point is that the end product of the stage 1 compilation process is the CIL, and not machine or byte code.

Warning: During the development of .NET, the official term for IL was Microsoft Intermediate Language (MSIL). However, with the final release of .NET, the term was changed to Common Intermediate Language (CIL). Thus, as you read the .NET literature, understand that IL, MSIL, and CIL are all describing the same entity. In keeping with the current terminology, I will use the abbreviation CIL throughout this text.

The .NET entire compilation process is shown in Figure 5. The source code can be written in any .NET compliant language, it is then compiled using the appropriate (built into Visual Studio, for

example) into the Common Intermediate Language (CIL). The assembly consists of the actual code and the metadata describing the class references and dependencies. When you run the code (the .exe file), the CLR compiles it into byte code and then executes it in the virtual machine.

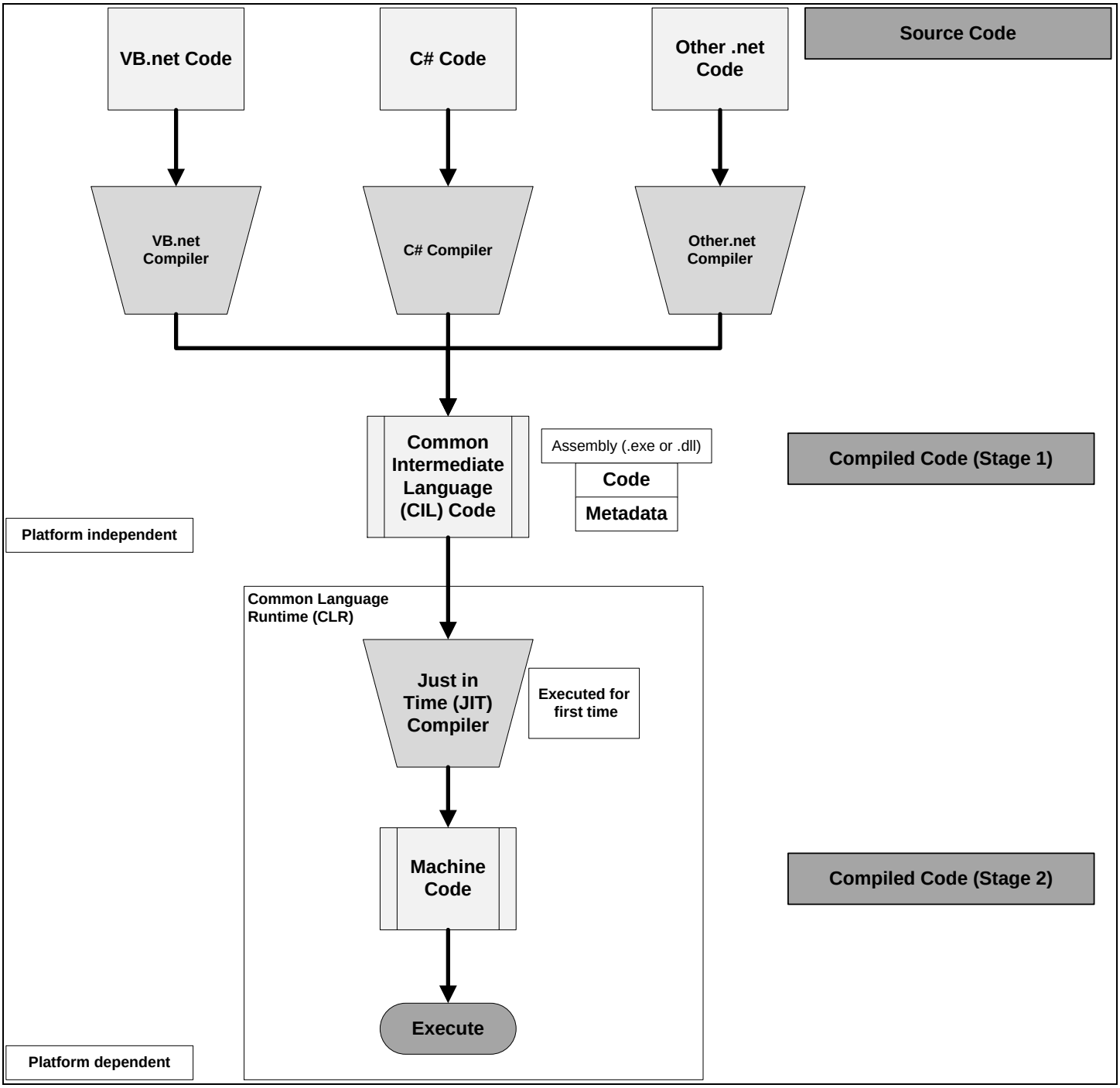


Figure 5: .NET Compilation Process

1.4 .NET Common Language Runtime (CLR)

As mentioned in the previous chapter, the CLR is Microsoft's implementation of the CLI.

The Common Language Runtime (CLR) provides the services that are needed for executing any application that is developed with one of the .NET languages. This is possible because all of the .NET languages compile to a common intermediate language (CIL).

The CLR is also based on the Common Type System that defines the data types that are used by all .NET languages. Because all of the .NET applications are managed by the CLR, they are sometimes referred to as managed applications. The basic structure of the CLR is shown in Figure 6.

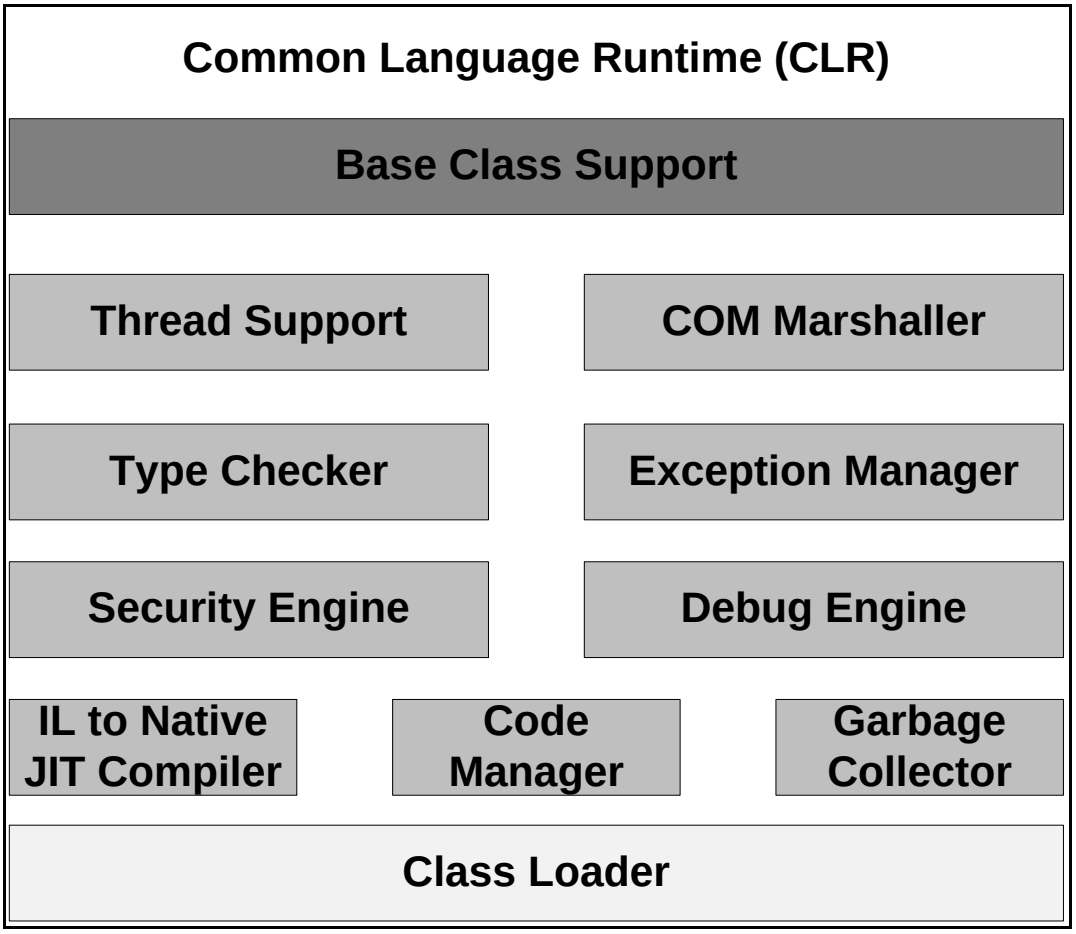


Figure 6: .NET Common Language Runtime (CLR)

Class Loader: The class loader is initiated the first time a type is referenced. It looks in different places (application config file, in the global assembly cache (GAC), and the metadata that is part of the portable execution (PE) format) to locate the class. It then loads the .NET classes into memory and prepares them for execution.

JIT Compiler: Compiles CIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The just - in -time part of the name reflects the fact that CIL code is only compiled as, and when, it is needed.

Code Manager: The code manager places the objects in memory and controls the execution of the code.

Garbage Collector: One of the most important features of managed code is the concept of garbage collection. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use.

Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of your computer followed by a system crash.

.NET garbage collection works by inspecting the memory of your computer every so often and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but you can be rest assured that it will happen.

Security Engine: .NET has its own security mechanism with two general features: Code Access Security (CAS) and Role-Based Security

Essentially, code access security assigns permissions to assemblies based on assembly evidence. Code access security uses the location from which executable code is obtained and other information about the identity of code as a primary factor in determining what resources the code should have access to. This information about the identity of an assembly is called evidence.

Code Access Security is based on evidence that is associated with a specific assembly. Typically the evidence is the source of the assembly (whether it is installed on the local machine or has been downloaded from the intranet or Internet).

Role-based security utilizes the concepts of users and roles, which is similar to the implementation of security in many current operating systems. Two core abstractions in role-based security are Identity and Principal. Identity represents the user on whose behalf the code is executing. It is important to remember that this could be a logical user as defined by the application or developer and not necessarily the user as seen by the operating system. A Principal represents the abstraction of a user and the roles in which a user belongs.

Debug Engine: To enable debugging an application during runtime.

Type Checker: Type checker will verify types used in the application with CTS or CLS standards supported by CLR, this provides type safety.

Exception Manager: Exception Manager will handle exceptions thrown by application by while executing Try-catch block provided by an exception. In "Try" block used where a part of code expects an error. In "Catch" block throws an exception caught from "try" block, if there is no catch block, it will terminate application.

Thread Support: Threads are managed under the Common Language Runtime. Threading means parallel code execution. Threads are basically light weight processes responsible for multi-tasking within a single application.

COM Marshaler: It allows the communication between the application and COM objects (backward compatibility).

Base Class Support: It provides class libraries support to an application when needed.

2. Overview of Visual Studio

Visual Studio is Microsoft's development tool for building .NET application, whether it is desktop, web, or mobile applications. The Express versions are scaled down versions divided by the application target:

- Visual Studio Express for Desktop
- Visual Studio Express for Windows (Windows Store Apps)
- Visual Studio Express for Web
- Visual Studio Team Foundation Server Express (Version Control)
- Visual Studio Express for Windows Phone

In this course, we are going to use the free Visual Express for Desktop version.

2.1 Visual Express for Desktop IDE

Visual Express for Desktop is a free version as part of Visual Studio. Visual Express for Desktop can be used to develop Console applications (like an old DOS application), Windows Forms or Windows Presentation Foundation (newer Desktop application style) application, or simply just a class library. In this course, we will use Windows Forms as our application platform. For all application types, you can use C# as a programming language to build functionality and enhanced user interfaces.

The Visual Express for Desktop Integrated Development Environment (IDE) looks very similar to the one in Visual Studio, except that it does not contain all the features. However, the IDE interface is structured exactly the same with a few exceptions. Again, this is a free version, so do not expect it to be a full-fledged tool. However, this free version is very powerful and allows you to build powerful applications and business logic. Especially for this course it is a great tool to introduce you to the object-oriented nature of C# and object-oriented programming.

So let us first explore the basic IDE interface. When you run Visual Express for Desktop for the first time you are presented with the main interface as shown in Figure 7. This interface contains a Start Page in the middle of the IDE that, if connected to the Internet, will show you the latest messages and information about Visual Studio.

The Start section allows you to start a new project or open an existing project. The Team Foundation Server is Microsoft's Versioning Solution that can also be accessed through the Start section.

The Recent section allows you to open any recently used projects.

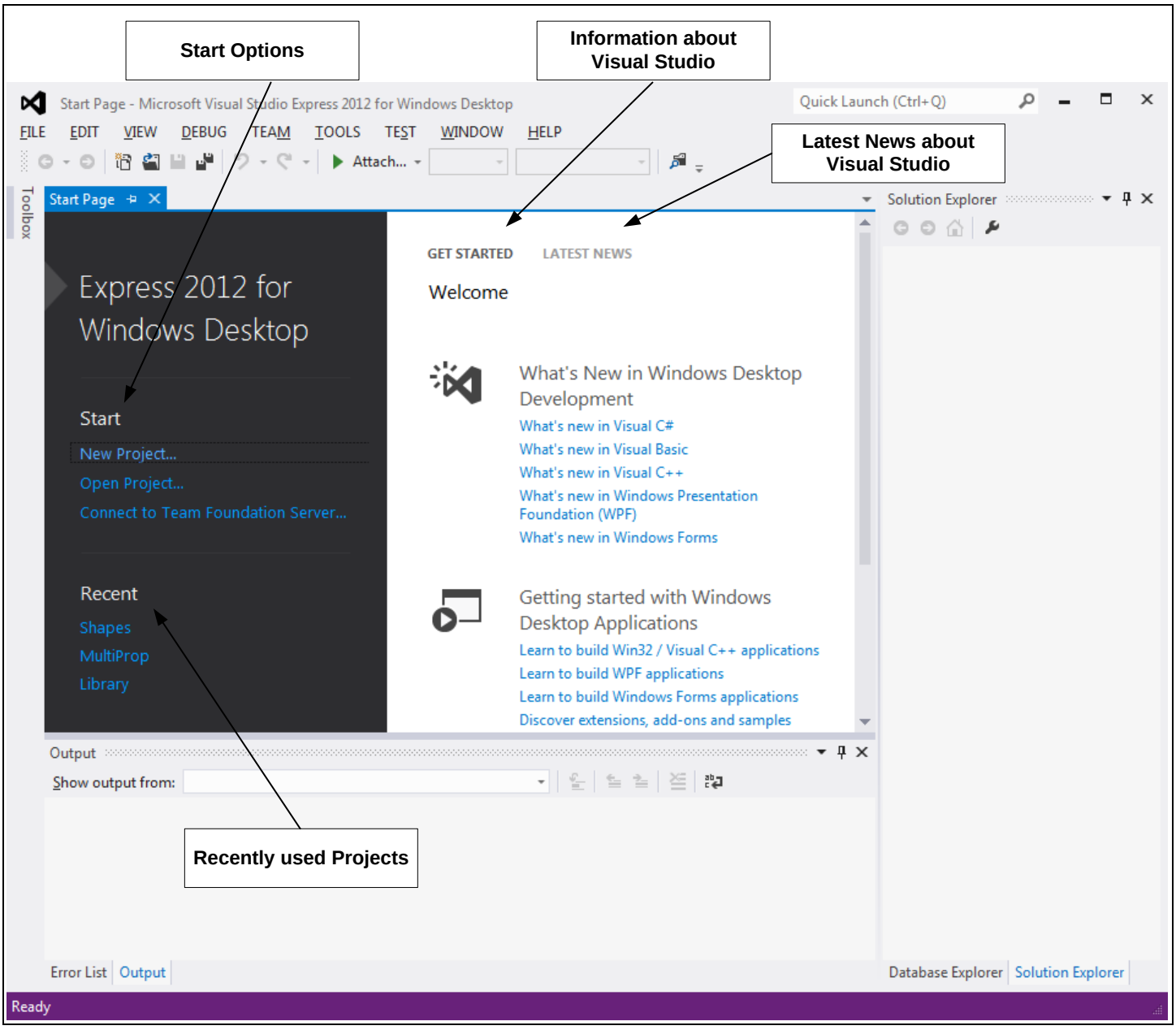


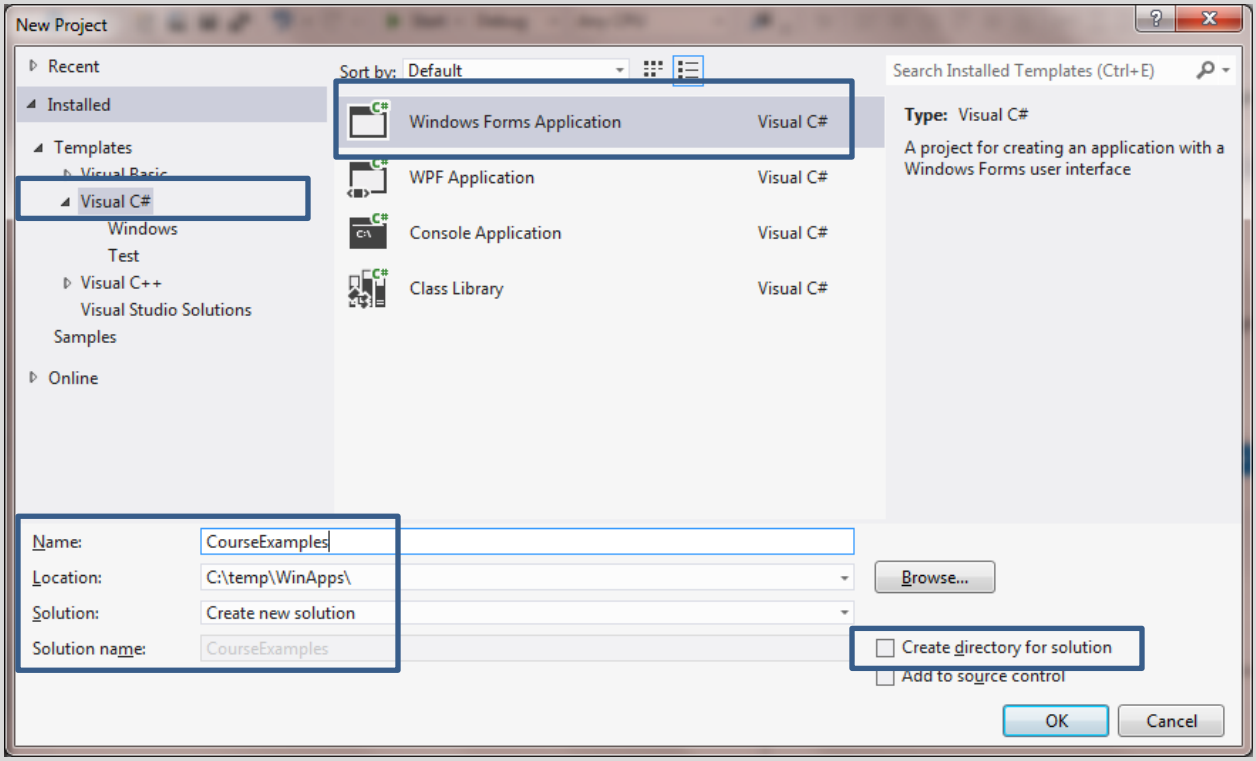
Figure 7: Visual Express for Desktop Start Page

Visual C# Express Edition can also be used to edit XML and HTML files as well as other common data formats. Its color coding of different elements makes it particular convenient to edit those files. To open any file within the IDE, select the pull-down menu FILE, then Open File… To bring back the Start Page, select pull-down menu VIEW, then Start Page.

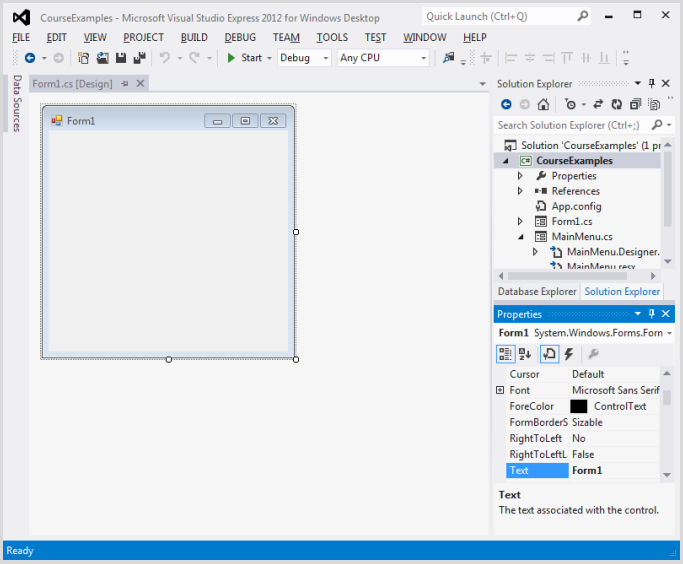
To get more familiar with the IDE, let us build a project and explore the interface in more detail.

Example 1-1: Exploring Visual Express for Desktop

1. Click on the FILE menu, then select New Project.
2. In the New Project dialog box, select Visual C# under Templates, then Windows Forms Application. Click on Browse to navigate to C:\Temp (if it does not exist, create a Temp folder). Create a new Folder under c:\Temp and name it WinApps. Name the Project CourseExamples. Uncheck "Create directory for solution".



3. Click on the OK button located in the lower, right hand corner.
4. Now you should see a default form displayed named form1.cs in the center of the screen.



Now let us explore the IDE interface. First of all, on the left side of the window you will see certain tabs. When you click on them, those tabs expand to a menu. For example, click on the toolbox tab and you see a list of controls and other tools that you can use for your form development tasks.

Note: You can turn on hovering rather than clicking on those menus by navigating to TOOLS, then Options. Under Environment → Tabs and Windows check "Show auto-hidden windows on mouse over"

All of these menus have three buttons in the upper, right hand side. The down arrow allows for setting Windows Position options. The middle button, a push pin, allows for pinning the menu (meaning it does not slide in and out anymore, it is fixed). The last button, the X, is simply for closing this menu window. To bring any closed window back to line, navigate to pull-down menu VIEW. There you find many different other windows that you can display in the IDE.

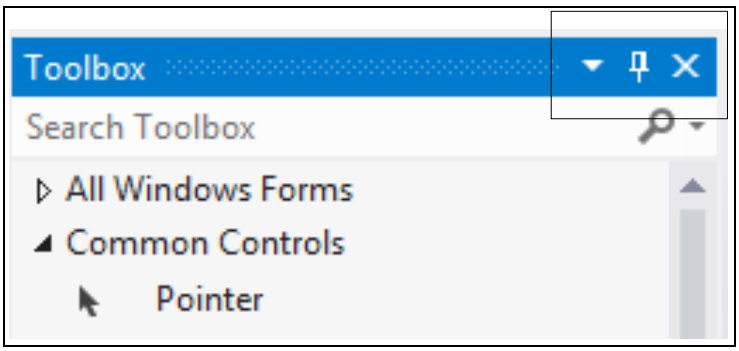


Figure 8: Menu Window in Visual Studio

On the right side, you see two important windows: The Solution Explorer and the Properties Window.

The Solution Explorer (see Figure 9) allows you to manage all the different files in a project. Visual Studio uses the concept of a Solution. A solution can contain one or more projects. In general, you have only one project contained in one solution. But for larger development projects, there could be multiple projects within a given solution.

The Solution Explorer is organized in a tree view. At the very top is the solution, underneath are one or more projects. For each Windows Forms project there is at least one properties folder, one references folder, one form file, one App.config file and one Program.cs file.

The properties folder lists all the properties of the project, they are automatically generated and in general there is no need to edit the files contained in this folder. To access the project properties window, either select the pull-down menu Project or right-click on the project name (not the solution name) and then select Properties. Any options that are changed here are saved into the files under the properties folder.

The References folder contains all references to the namespaces that are included in this project. When you create a Windows Form project, certain references are automatically added needed in particular for a Windows Form project, for example System.Windows.Forms. If later on, you need other references, right-click on the Reference folder to add other references.

The App.config file is an XML based file to set specific and custom properties for this project.

The program.cs file contains a main method which is executed when the application is started.

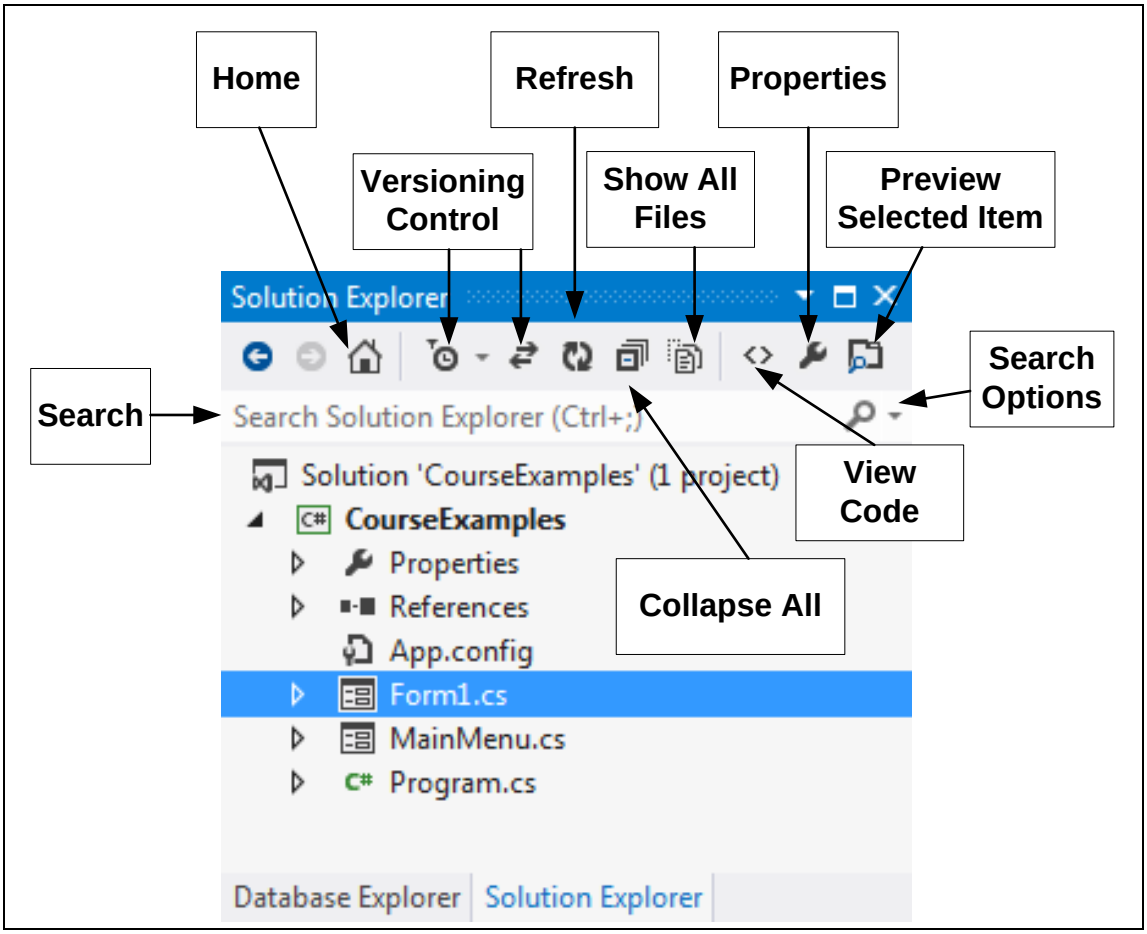


Figure 9: Solution Explorer

The other important window is the Properties window as shown in Figure 10. To view the Properties window, select pull-down menu VIEW, then Properties Window or alternatively, keyboard shortcut [Alt] + [Enter].

This window displays for a selected object the appropriate properties. For example, for a form you can change the Text property (which is the Windows Title), the background color of the form, etc.

This window also allows you to create events for the selected item, if applicable.

At the top of this dialog box you see a drop-down list of items that you can select. In the case shown in Figure 10 you see that the form Form1 is currently selected. There are five buttons below that you allow to further select options in a different view or to access different options.

As shown in figure, the first two buttons simply enable you to either display the properties in a categorized or alphabetical view.

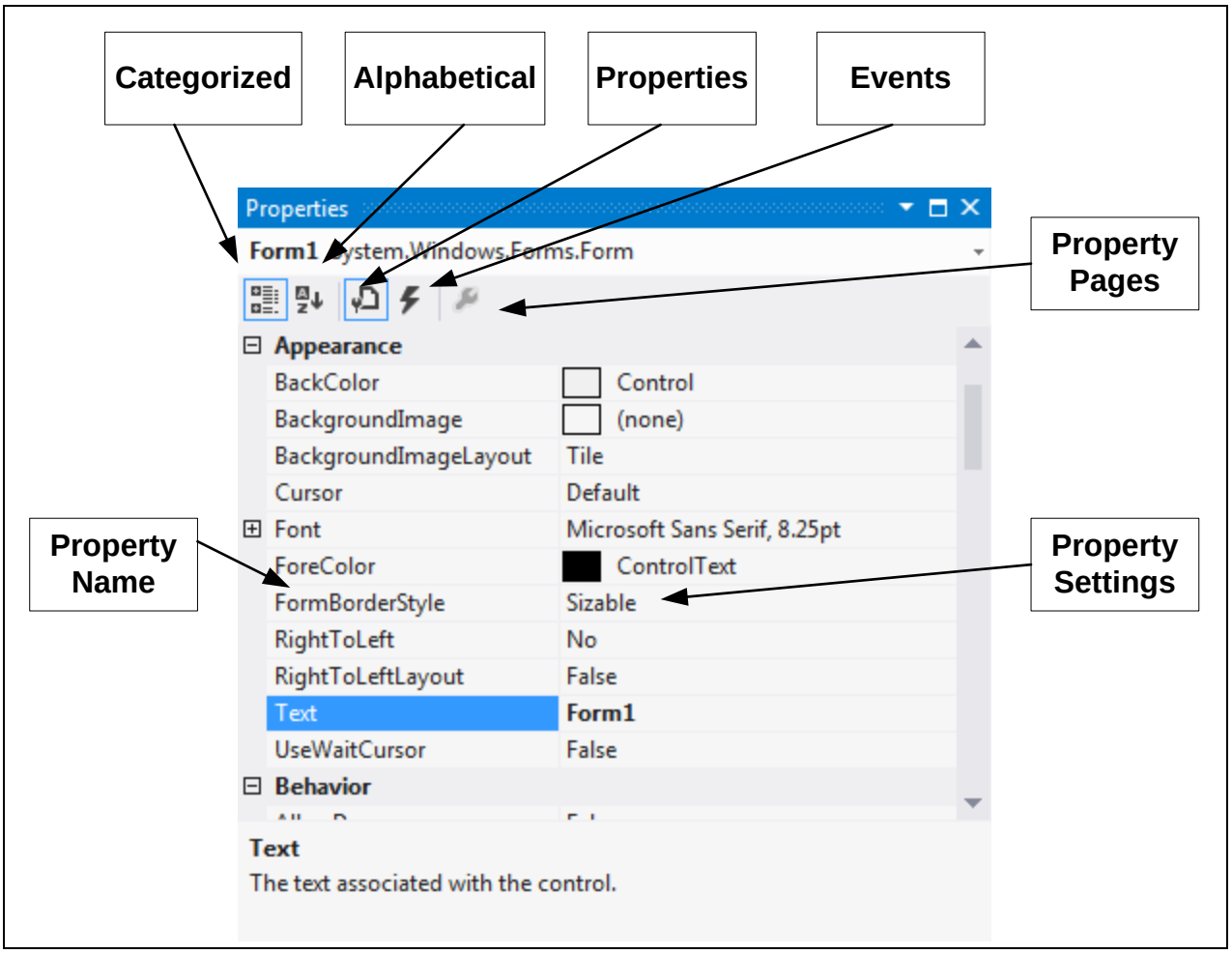


Figure 10: Properties Window

The next two buttons enable you to view the properties or the events of a selected item. The last button is for special items that have property pages, which is either a subset or a superset of the properties shown in this window. We will explore this tool in much more detail as we go along in this course.

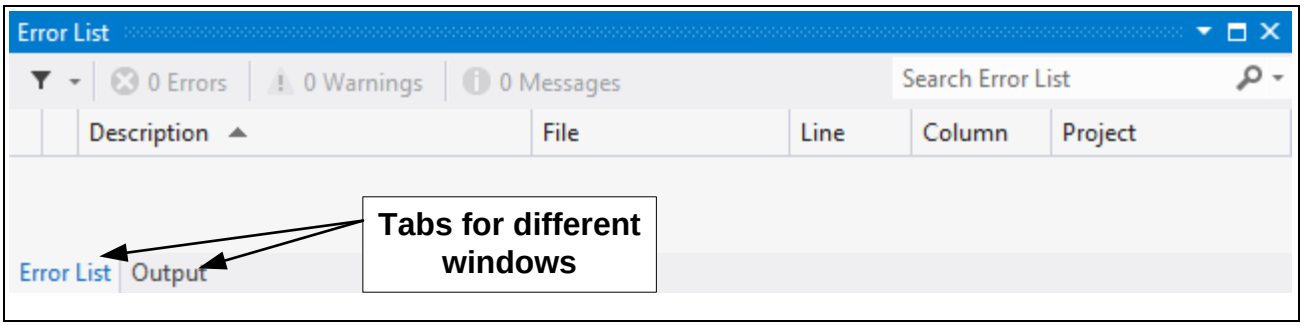


Figure 11: Output Windows

At the bottom of the IDE you may see various output windows, such as Error List and Output window as shown in Figure 11. These windows are arranged in tabbed way, the tabs at the bottom allow for easy navigation.

All these various windows can be moved by dragging them using the title bar. Once they are out of place, they become floating windows.

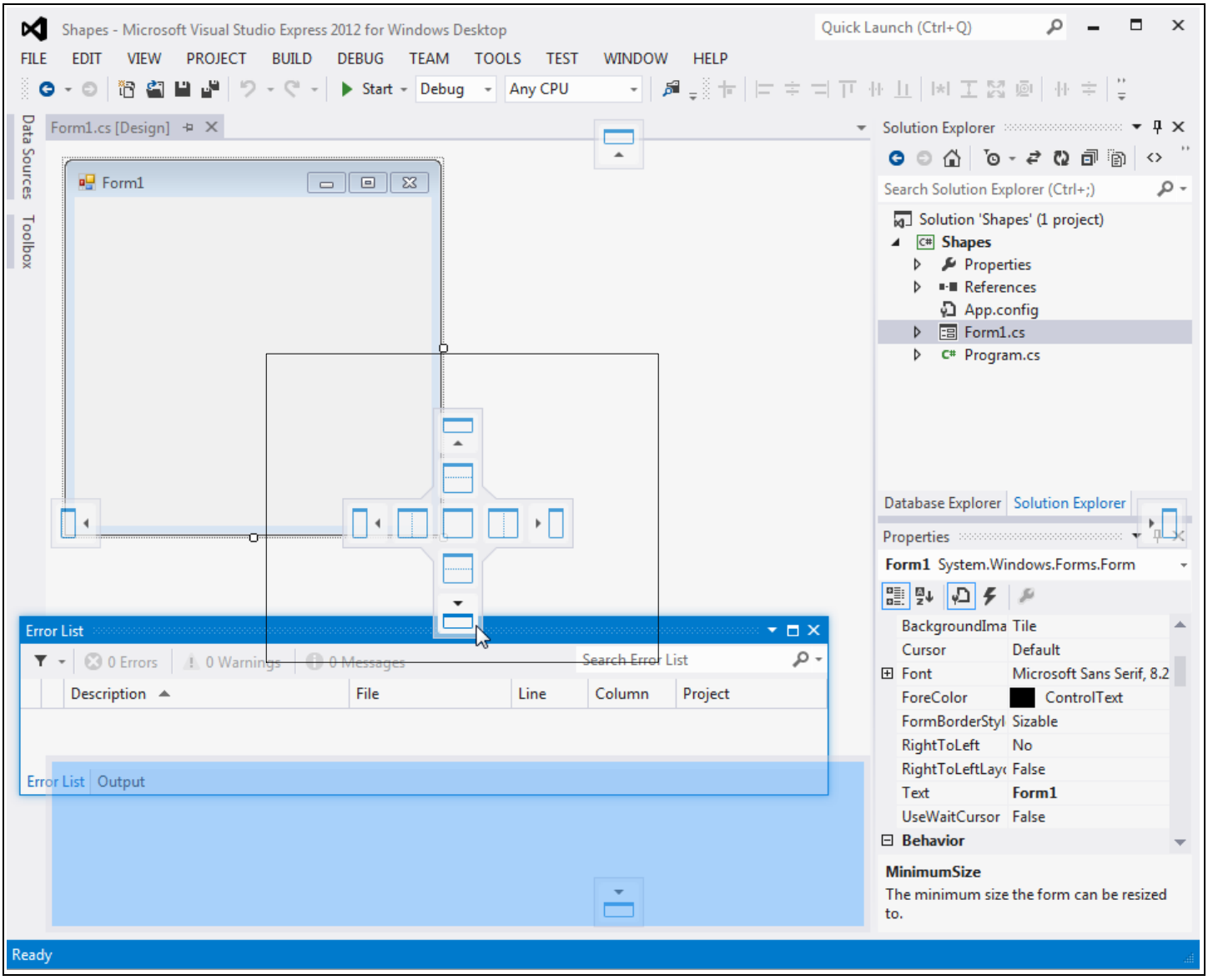


Figure 12: Visual Studio Window Docking Options

To dock them back in place, click inside the title bar and start dragging them. Notice the various positioning arrows within the IDE as shown in Figure 12 in the center. Once you place the window onto one of these arrows, the window becomes integrated into the IDE interface and it is docked again.

2.2 Developing a Desktop Application

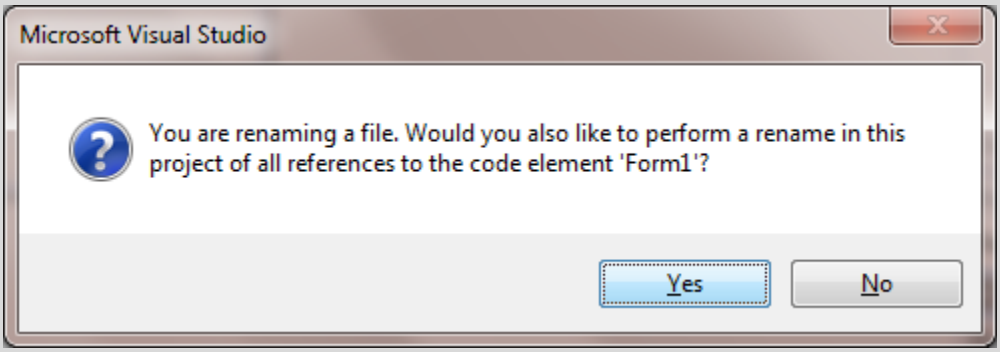
We are now going to develop a very simple application in order to learn how to use Visual Express for Desktop. This chapter is comprised mainly of hands-on examples.

We will built on Example 1-1 and expand the Windows Forms application. The project developed in Example 1-1 will serve as the project containing all the examples we are going to demonstrate during the lecture.

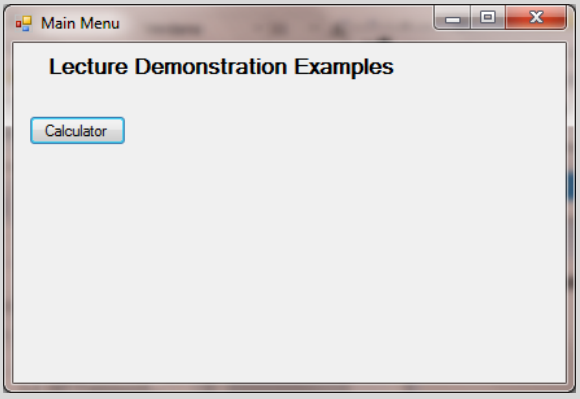
First we will use the form that is already contained in the project to build a main menu form. We will place buttons on this form navigating to other forms which then in turn will contain the lecture examples.

Example 1-2: Building a main menu form

1. Use the existing project created in Example 1-1. First of all, let us rename the Form1. Right-click on the form in Solution Explorer and select rename.
2. Name the form MainMenu.cs and hit the [Enter] key. The following dialog box is displayed:



3. Selecting "Yes" will rename all existing references from "Form1" to "MainMenu".
4. Using the Properties Window set the Text property to "Main Menu". Make sure that form is selected in the design Window or double-click on the "Form1.cs" file in Solution Explorer.
5. From the Toolbox Window, click on a Label control (under category Common Controls) and place it on the form at the top. With the label being selected, set its Text property to "Lecture Demonstration Examples".
6. Locate the Font property, click on plus (+) sign, and increase the Font to 12 and set the Bold property to True.
7. From the Toolbox Window, click on a Button control (under category Common Controls) and place it on the form below the label control. Set its Text property to "Calculator" and the Name property to btnCalculator.
8. To run this project, click on the green arrow button in the toolbar at the top.



2.3 Writing Basic C# Code

We are going to write a few lines of code, do not worry, we will cover the basic of the C# programming language in the next chapter. The code example presented here is simple enough for you to understand, but it also shows you where to place the code and how to use the code editor.

First we will add another form and write two lines of C# code in the OnClick event for the Calculator button to open this new form.

Then on the new form we will add two text box controls, three labels and a button to build a simple calculator.

Example 1-3: Button OnClick event

1. Right-click on the project name CourseExamples and select Add Windows Form.
2. Type frmCalculator as the form name and click on "OK".
3. Set its Text property to "Calculator".
4. Double-click on the Calculator button on the MainMenu form which will create the default OnClick event method.
5. Write the following two lines of code:

```
private void btnCalculator_Click(object sender, EventArgs e)
{
    frmCalculator frm = new frmCalculator();
    frm.Show();
}
```

6. Save all files and click on the green Arrow Start button to run the project.

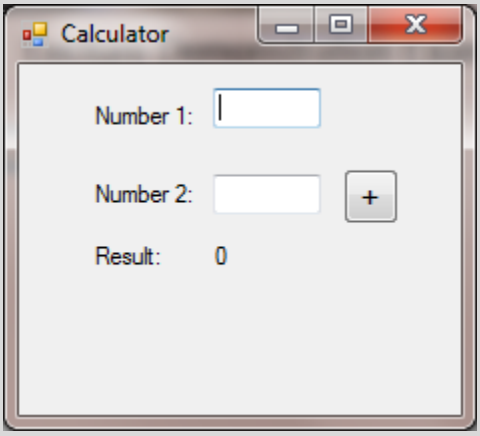
The first line creates an instance of the form class frmCalculator in memory identified by the variable frm. You could have named this variable anything (well, almost anything) you want.

Then we used the Show method on this form instance to actually show this form.

In the next example we will now create a little calculator form. On the existing calculator form, we will add two text box controls where the user can enter some values to be added using a button control.

Example 1-4: Creating the Calculator Form

1. Make sure to display the form frmCalculator in design view.
2. From the toolbox, add two text box controls, four label controls and a button as shown below:



3. Set the first label's Text property to "Number 1:", the second one to "Number 2:", the third one to "Result:" and the last one simply to "0".
4. Set the Name property of the last label (the one that contains "0") to lblResult.
5. Name the first text box control "txtNumber1" and the second one "txtNumber2".
6. Set the button's Text property simply to a plus sign ("+") and the Name property to "btnAdd". Increase the font size to 12.
7. Set the button's width and height property to 25 pixels.
8. Size the form by hovering the mouse over the border of the form, then click and drag to size the form.
9. Now run the project by clicking on the Start button (green arrow in the toolbar)

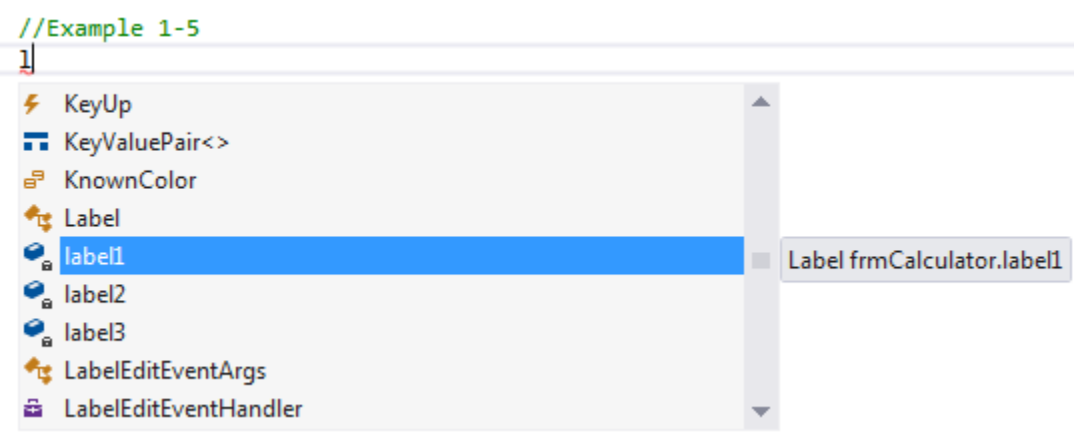
Designing a form is a fairly straightforward process. Many other application design environments use the same concepts and tools. At this point, the form is not functioning at all, except that it is running as a standalone window. We will now add some simple logic to the Add button to perform an arithmetic addition.

Basically, we need to reference the values that are entered in both textbox controls and add them up. The result should be then displayed in the label lblResult.

This logic should be executed when the user clicks on the Add button. So we need to create a small program that is executed when the click event of the button occurs.

Example 1-5: Creating the Button OnClick event

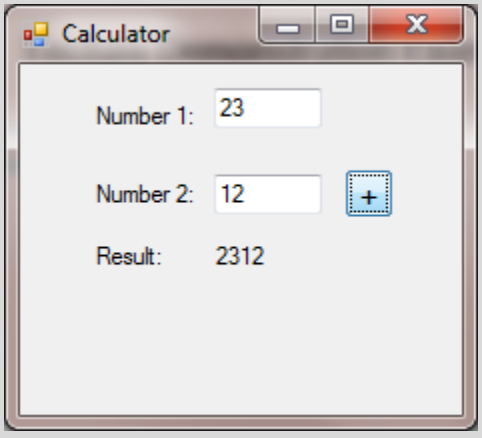
1. Double-click on the button "btnAdd" to create the default OnClick event.
2. The cursor is placed exactly in the event handler that you created by double-clicking on the button.
3. Now type the following line. Notice a little window that is displayed when you start typing. This is the IntelliSense window assisting you and saving you lots of typing.



4. Start typing lblResult, which is our result label that will display the result. Once you see the item you need simply hit the [Enter] or [tab] key.
5. Now type a period or so-called dot operator and start typing the Text property. Again, once you see or find a match in the IntelliSense window, hit the [Enter] key.
6. Now type the equal (=) sign and type txtNumber1.Text + txtNumber2.Text;

```
private void btnAdd_Click(object sender, EventArgs e)
{
    //Example 1-5
    lblResult.Text = txtNumber1.Text + txtNumber2.Text;
}
```

7. Save and run the project using the green arrow Start button. Enter some numbers and click on the "+" button. Notice what happened. Surprised??
8. As you can see, our code added the values in a textual context, simply concatenating. The text property of any control is of data type string, so it used the string data type to perform the operation.



This one line of code shows that C# is pretty picky with data types and does not implicitly convert data types, meaning assuming a certain data type based on the operation. You have to explicitly convert data types in C# in order to achieve the desired result. This makes C# a strongly typed language.

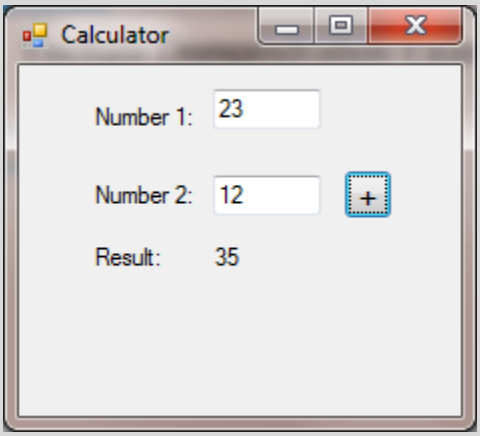
We will use a conversion function to convert the numbers that are entered into the textbox controls into a number data type. We need to convert the result back into a string data type, because that is what the Text property of the label expects. This may seem at first a bit overwhelming, but you get used to it and will see the benefits throughout this course.

Example 1-5 (continued): Creating the Button OnClick event

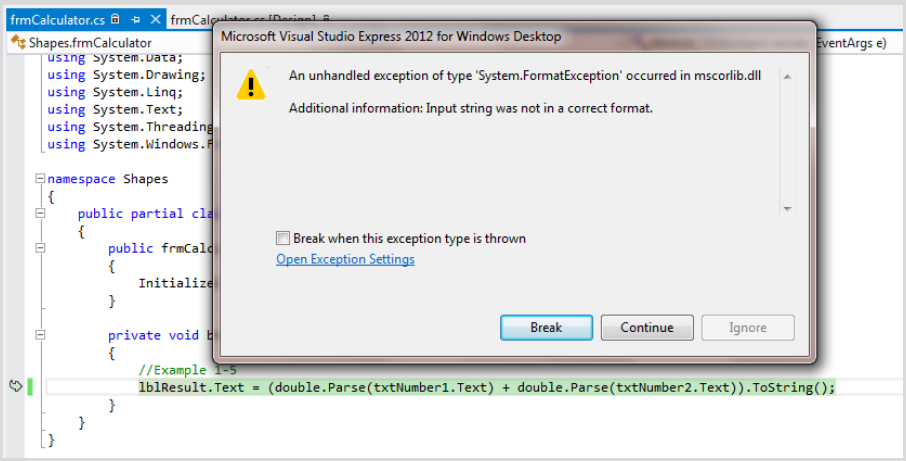
1. Modify the line of code as shown below:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    //Example 1-5
    lblResult.Text = (double.Parse(txtNumber1.Text) + double.Parse(txtNumber2.Text)).ToString();
}
```

2. The double.Parse method is trying to parse the text into a number. Once you have added the numbers up, the end result is then converted back into a string using the ToString() method.
3. Now run this project and repeat your test. It should now add up the numbers arithmetically.



4. Now the result is correct. But what happens when we do not enter a valid number. Well, an exception is thrown which we have not handled yet. The program enters suspension mode at the offending line.



2.4 Deploying an .NET Desktop Application

Deploying a .NET is not much different from conventional applications. At a minimum there is at least one executable file. But there may be other supporting files for a particular application needed, such as database files or other data input files (Text, Excel). Furthermore, you may also have an application configuration file (XML format) that stores application specific information, such as an application title, application version and database connection information.

Overview of Deployment Methods

When deploying an application, we basically need to distribute the assembly or assemblies to the target location. The entire process from building an application using Visual Studio and compiling the project into an assembly and then executing the assembly to run the application is shown in Figure 13.

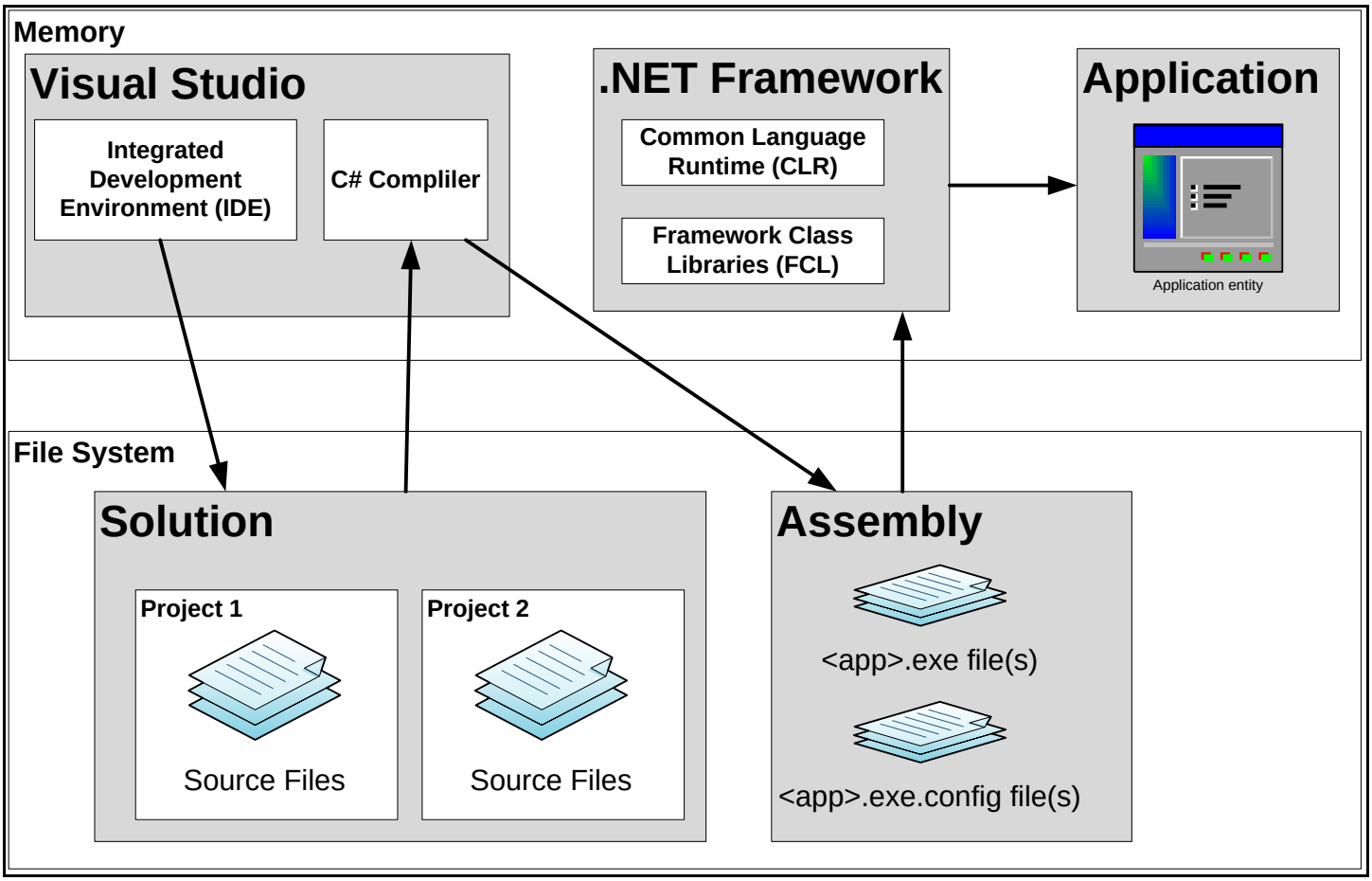


Figure 13: Visual Studio Compilation

An assembly in the Common Language Infrastructure (CLI) is a compiled code library used for deployment, versioning, and security. There are two types: process assemblies (EXE) and library assemblies (DLL). A process assembly represents a process that will use classes defined in library assemblies. CLI assemblies contain code in CIL, which is usually generated from a CLI language, and then compiled into machine language at run time by the just-in-time compiler. In the .NET framework implementation, this compiler is part of the Common Language Runtime (CLR).

An assembly can consist of one or more files. Code files are called modules. An assembly can contain more than one code module. And since it is possible to use different languages to create code modules, it is technically possible to use several different languages to create an assembly.

Physically, assemblies are files located somewhere on disk and are per definition so-called Portable Executable (PE). Assemblies are loaded on demand. They are not loaded if not needed.

Assemblies have Metadata stored to provide version information along with a complete description of methods and types. Part of this metadata is a Manifest. The manifest includes identification information, public types and a list of other used assemblies.

To view an assembly, Microsoft provides a tool called ildasm.exe. Its shortcut name basically stands for Intermediate Language Disassembler. To launch this tool, you have to either use the shortcut installed in the program menu as shown in Figure 14 (left side) or create your own shortcut on the desktop as shown in Figure 14 (right side).

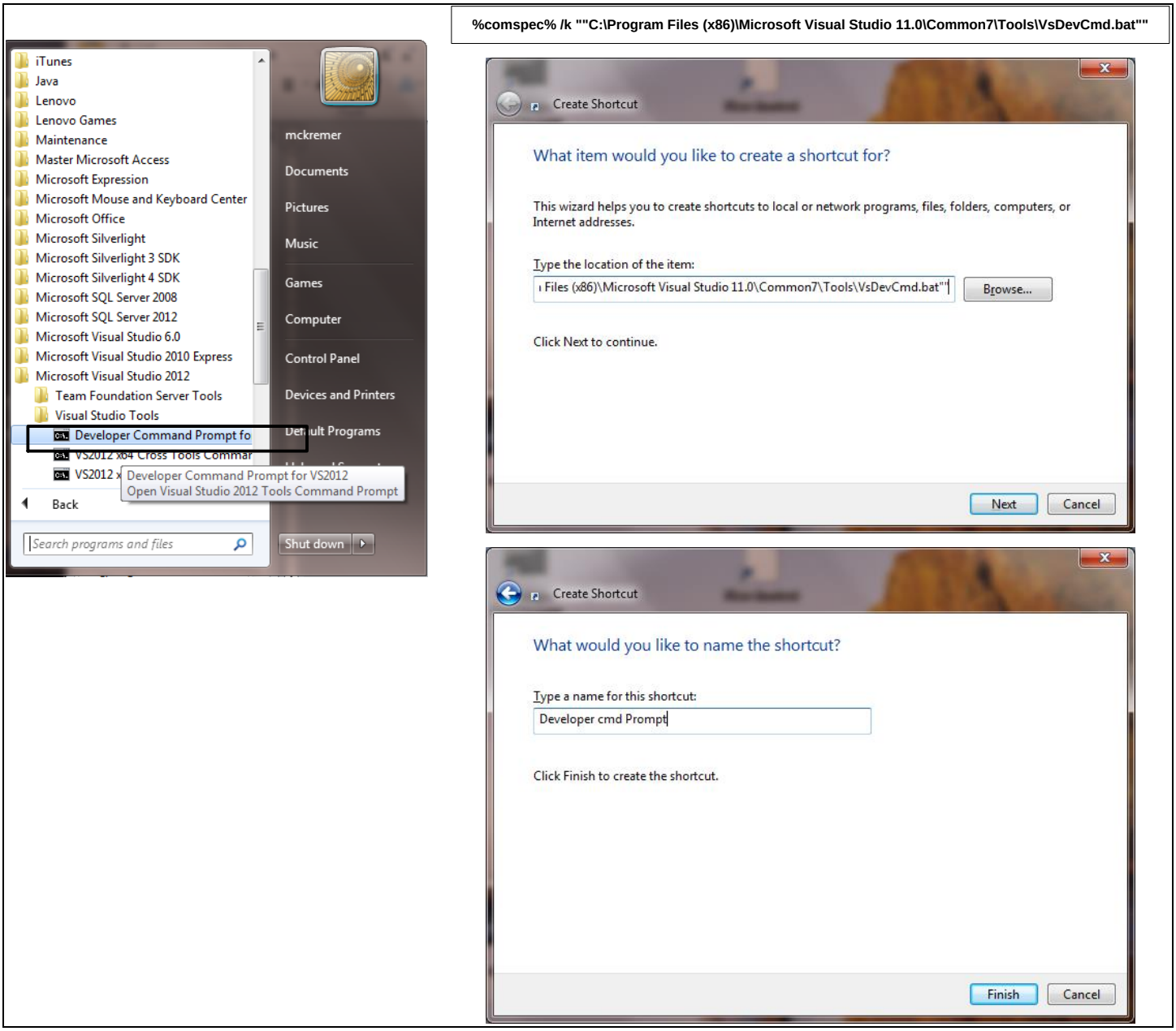


Figure 14: Developer Command Prompt for Visual Studio

In the command prompt window that is displayed when running the developer command prompt, type ildasm.exe and hit the [Enter] key. The ildasm application is displayed as shown in Figure 15.

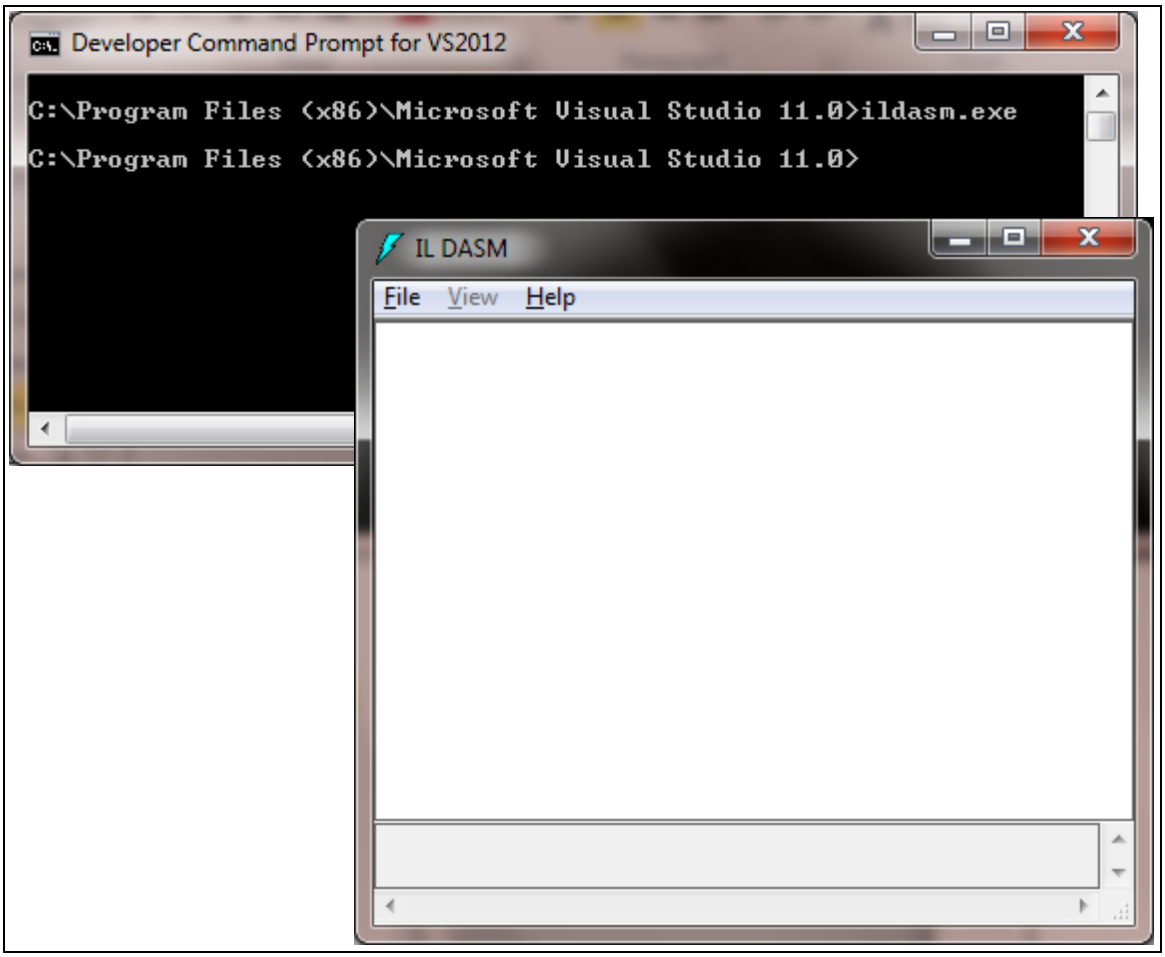


Figure 15:ILDASM.exe Application

Using this application you can now open any .net executable file (=assembly) and view the metadata and the IL code modules as shown in Figure 16.

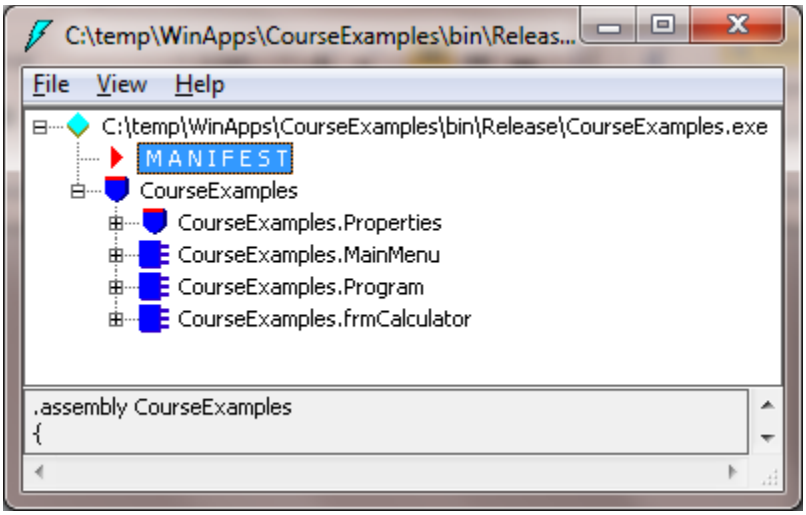


Figure 16: Viewing an Assembly

You can now double click on the Manifest to view the manifest information. This is basically metadata that tracks references libraries and dependencies as shown in Figure 17.

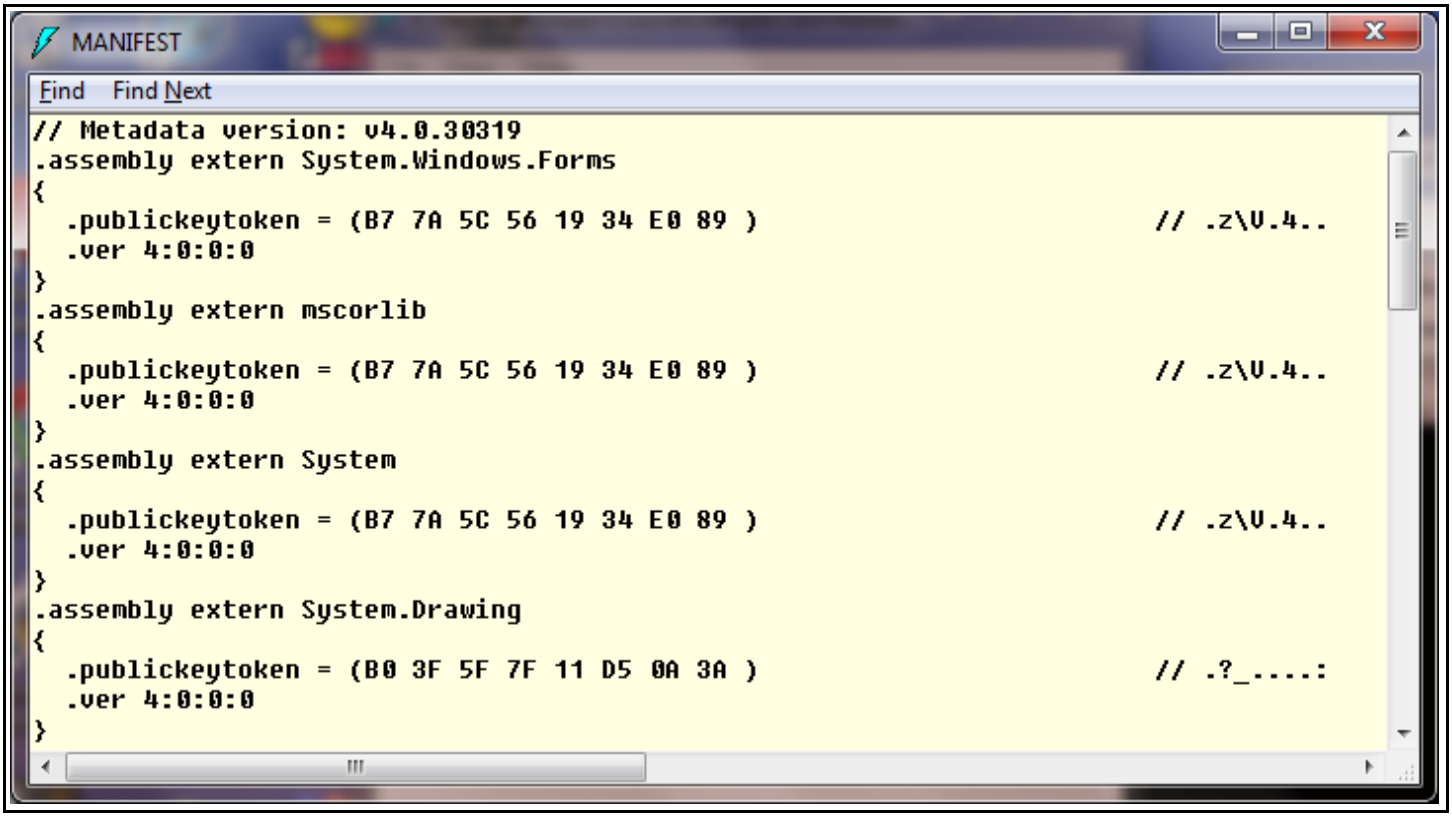


Figure 17: Manifest Information

You can also expand on the individual nodes to view the CIL as shown in Figure 18.

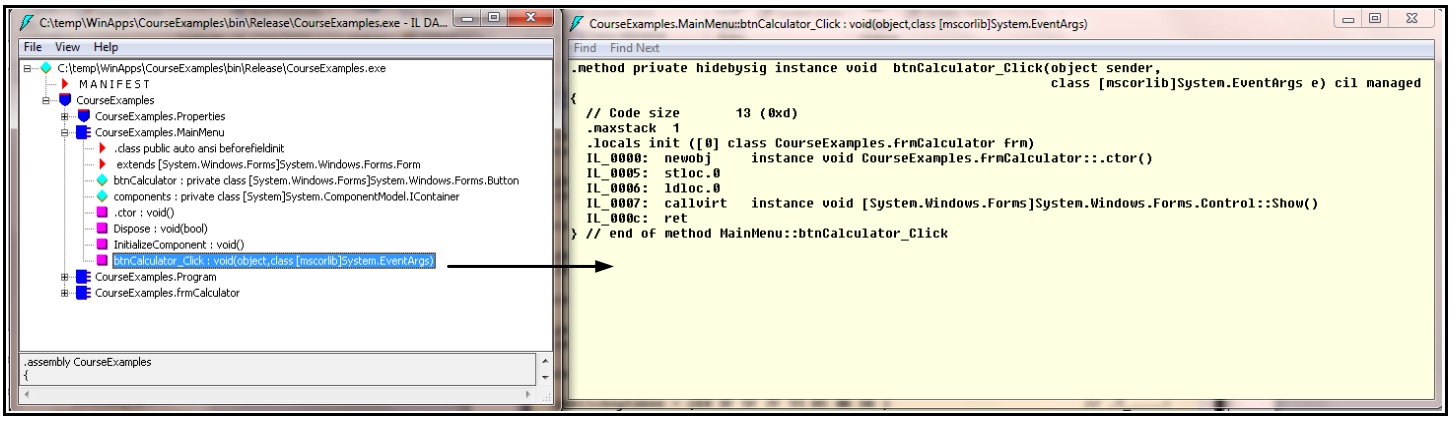


Figure 18: CIL in Assembly

There are basically three different methods in Visual Studio to deploy an application

- XCopy (File deployment)
- ClickOnce (Web deployment)
- Setup Program (not available in Express versions)

We will discuss each method in more detail in the following sections.

XCopy

The XCopy method is the simplest and the oldest way to deploy an application. You simply copy the executable files and possibly other dependent files to your target environment (other user's computers, for example).

The files for deployment are stored under the bin folder of your project, in either the debug or release folder. The kind of files that you may find in this folder are shown in Figure 1

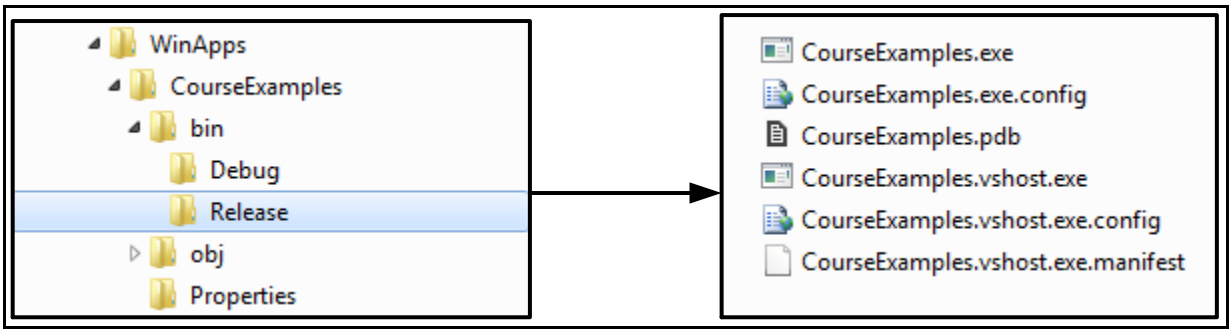


Figure 19: Release Folder

When you look at the project folder structure as shown in Figure 19, you notice a bin folder. The word bin stands for binary, this is a common terminology to store compiled, binary files in this folder. This folder contains the file CourseExamples.exe, which is the compiled (CIL), executable file that can be deployed into a target environment.

The other important file is the app.config file that stores application related settings, such as database connection string. In the debug or release folder this file is renamed to <project_name>.exe.config. This file is in an XML format and can be edited on the target's computer in case the connection string has to be adjusted, for example.

There are other files in either the debug or the release folder, all files containing "vshost" in their file names are simply for supporting debugging an application. These files do not need to be deployed. The file <project_name>.pdb is also supporting the debug process and does not need to be deployed either.

You could write a batch file to copy all the files to the target computer. Then copy the batch file and the deployment files to a network server so that you can deploy these files easily to many other target computers.

Click Once

As you may know, web applications have a big advantage over client applications that is there is no installation required on the client side. You simply need a browser. The ClickOnce method in Visual Studio is trying to bridge that gap by using a mechanism where the user can deploy and update the application through a web page.

To setup the ClickOnce method, you must run the Publish Wizard to set up this process. You select from the pull-down menu BUILD → Publish project_name.

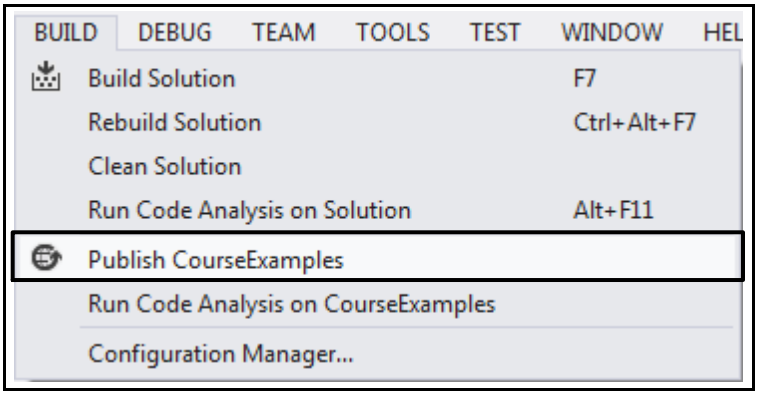


Figure 20: BILD--> Publish Menu

Note: The Publish menu name contains the name of your project, in the example above CourseExamples.

The first step of the publish wizard requires you to specify the location from which you want to install the application is shown in Figure 21.

There are four options:

- Disk path
- File share
- FTP Server
- Web Site

The most commonly used option is the Web Site option. However, other options are a local or network disk path or an FTP server.

The next step is to decide whether the application will be available offline. That means, it will be installed on the user's hard drive, a shortcut will be added to the Start menu, and the application can be uninstalled via the Uninstall or Change Programs dialog box in the Control Panel.

However, if you choose the online option only, then the application is loaded only into memory on the user's computer from the source location (network, web server). As a result, there is no shortcut in the Start menu, you always must have access to the source files (network, web) and you must download the application every time before you can run it.

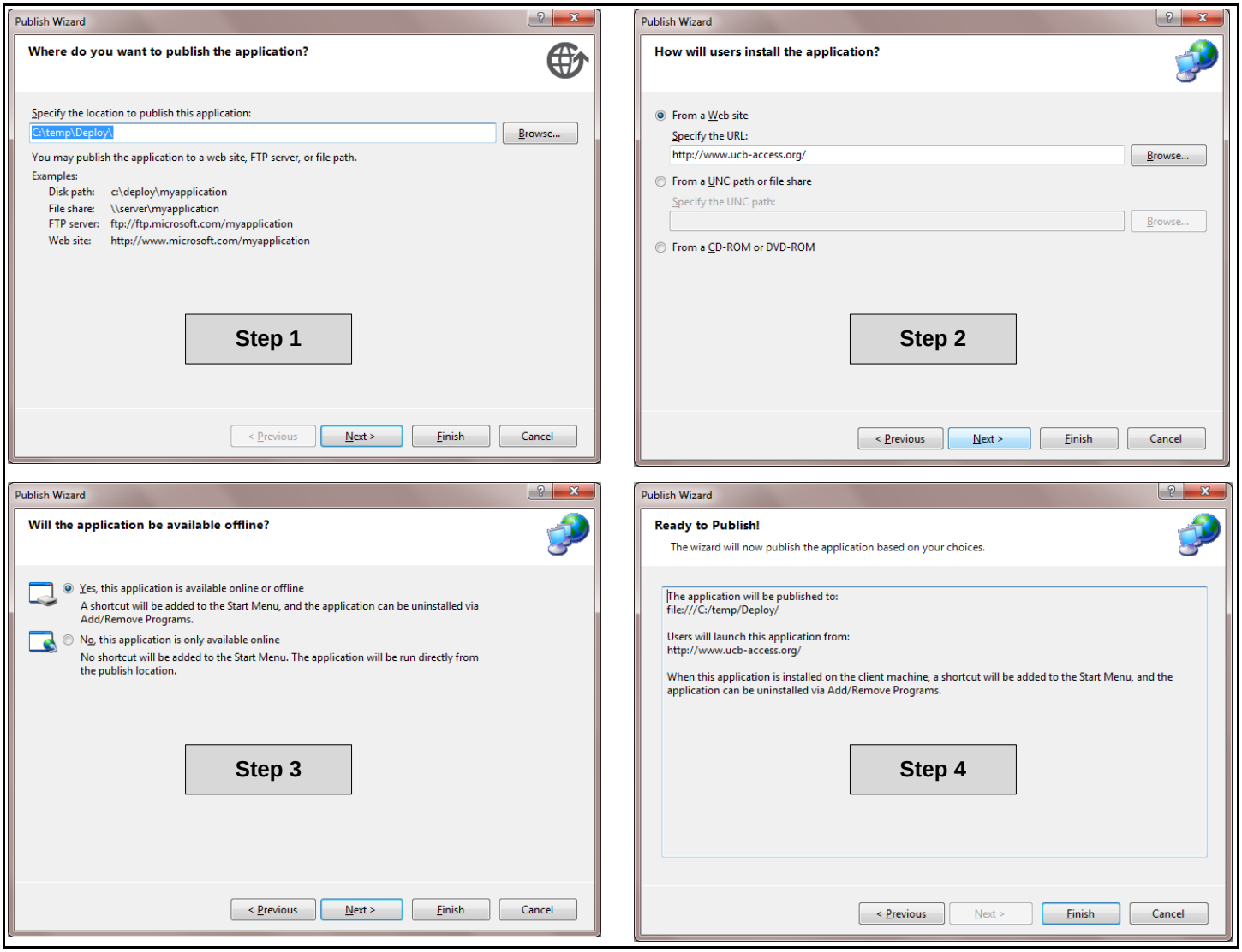


Figure 21: Publish Wizard, Step 1 – Step 4

If the source files for the application are stored on a web server, you can then bring up the following page as shown in Figure 22.

If the application is already installed, then you can click on the Launch link to run the application. Otherwise, click on the Run button to install and then run the application.

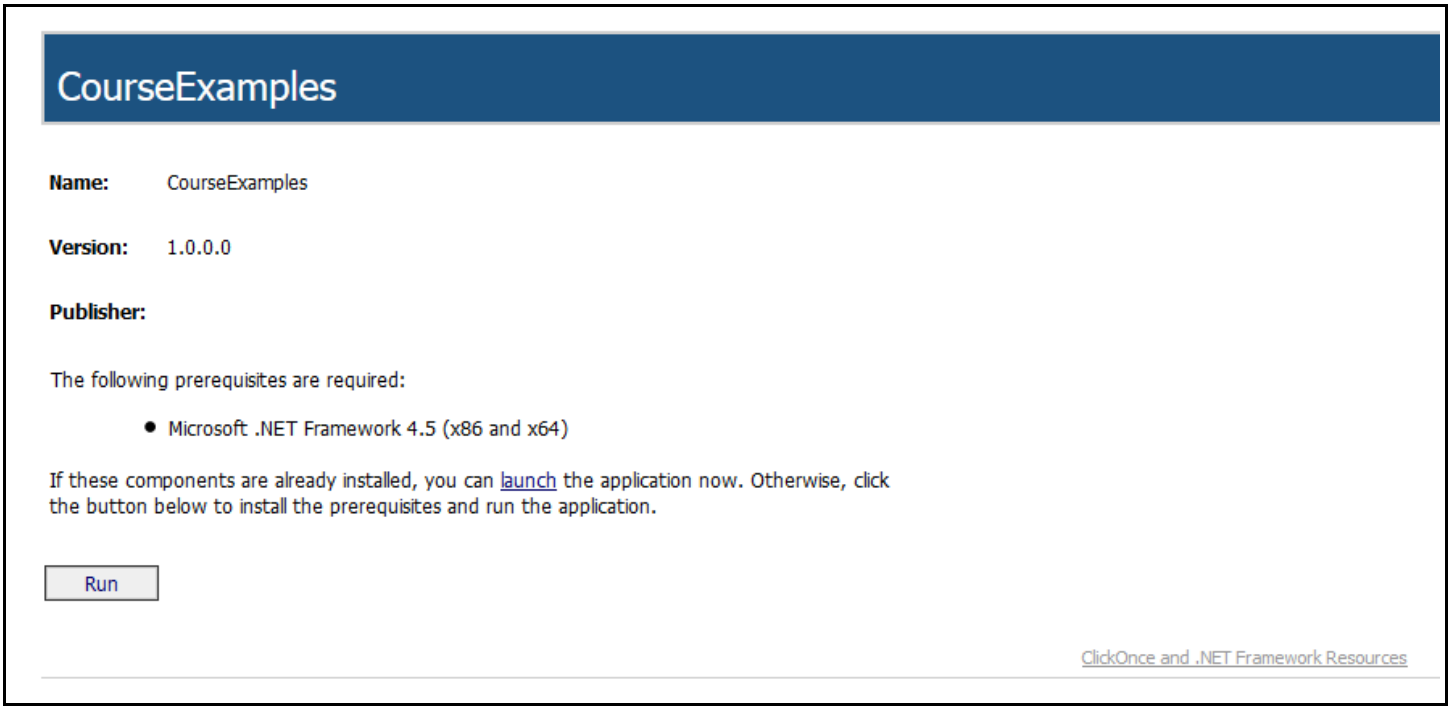


Figure 22: Web Page for Installing Application

When you use ClickOnce deployment, an application can be updated automatically if the user is connected to a web server that manages the installation.

An Update Available dialog box is displayed when the user runs the application and the local version number is different from the one on the web server.

This makes is a very convenient and powerful process for users to update their application to the latest version. It almost functions like a web application (meaning the updates happen automatically), except the program is installed locally.

Setup Program

If the first two deployment options are not adequate, you can create a Setup Program and package it up for your users to install the application.

Warning: As mentioned earlier, this option is not available in the Express versions of Visual Studio. However, there are many free setup program programs available, such as Inno Setup.

To create a setup program in Visual Studio, you would add a setup project to your solution which is provided in Visual Studio. This project is a specific type of Windows application that is used to install other Windows application.

Warning: Starting with version 2012, the setup program project has been removed. A free InstallShield limited edition program can be used with Visual Studio.

Then you customize the setup project based on your installation needs. Then you select Build from the pull-down menu to build the setup project. This will create the executable setup file that you would then distribute to your users.

CLASS 2

3. Introduction to C# Programming Language

C# bears a strong resemblance to the C++ and Java programming languages, having borrowed (or improved) features provided by these languages. The origins of both Java and C++ can be traced back to a language called C, which is a highly dangerous and entertaining language which was invented in the early 1970s. C is famous as the language the UNIX operating system was written in, and was specially designed for this.

3.1 Overview of C#

C# (pronounced C-Sharp) is no doubt the language of choice in the .Net environment. It is a whole new language free of the backward compatibility curse with a whole bunch of new, exciting and promising features. It is an Object Oriented Programming language and has at its core, many similarities to Java, C++ and VB. In fact, C# combines the power and efficiency of C++, the simple and clean OO design of Java and the language simplification of Visual Basic.

C is a dangerous language. So what does that mean? Consider the chain saw, for an unexperienced chain saw user I would expect it to come with lots of built in safety features such as guards and automatic cut outs.

In programming terms what this means is that C lacks some safety features provided by other programming languages. This makes the language much more flexible, but on the other hand there is a greater chance of crashing the computer with a C program than with a safer language.

The C# language attempts to get the best of both worlds in this respect. A C# program can contain managed or unmanaged parts. The managed code is fussed over by the system which runs it. This makes sure that it is hard (but probably not impossible) to crash your computer running managed code. However, all this fussing comes at a price, causing your programs to run more slowly.

The C# language is object oriented. Objects are an organizational mechanism which let you break your program down into sensible chunks, each of which is in charge of part of the overall system. Object Oriented Design makes large projects much easier to design, test and extend. It also lets you create programs which can have a high degree of reliability and stability.

C# is a compiled programming language. The computer cannot understand the language directly, so a program called a compiler converts the C# text into the low level instructions which are much simpler (CIL). These low level instructions are in turn converted into the actual commands to drive the hardware which runs your program (Binary Code).

3.2 What is a C# Program?

A program can be regarded as a cooking recipe, but written for a computer to follow, not a cook. The ingredients will be values (called variables) that you want your program to work with. The program itself will be a sequence of actions (called statements) that are to be followed by the computer. Rather than writing the program down on a piece of paper you instead put it into a file on the computer, often called a source file.

A windows forms application contains two main places where programs are placed. One place is the code file behind each form object. You normally put code into the form object that is directly related to the form and cannot be shared (or should not be shared) with other programs. Then there are code files that are standalone files and not related to a form object. Here you develop code that is used in multiple places in your application, hence shared. In C#, the code files are always called classes.

A class contains three things:

- Instructions to the compiler
- Information about the structures which will hold the data to be stored and manipulated
- Instructions which manipulate the data

Programs work by processing data. The data has to be stored within the computer while the program processes it. All computer languages support variables of one form or another. A variable is simply a named location in which a value is held while the program runs.

C# also lets you build up structures which can hold more than one item (array, object), for example a single structure could hold all the information about a particular bank customer. As part of the program design process you will need to decide what items of data need to be stored. You must also decide on sensible names that you will use to identify these items.

The actual instructions which describe your solution to the problem must also be part of your program. A single, simple, instruction to do something is in a C# program is called a statement. A statement is an instruction to perform one particular operation, for example add two numbers together and store the result.

The really gripping thing about programs is that some statements can change which statement is performed next, so that your program can look at things and decide what to do. In the case of C# you can group statements together to form a program which does one particular task. Such a program is called a method.

A method can be very small, or very large. It can return a value which may or may not be of interest. It can have any name you like, and your program can contain as many methods as you see fit. One method may refer to others. The C# language also has a huge number of libraries available which you can use. These save you from "re-inventing the wheel" each time you write a program.

3.3 Identifiers and Keywords

You give a name to each method that you create, and you try to make the name of the function fit what it does, for example ShowMenu or SaveToFile. The C# language actually runs your program by looking for a method with a special name, Main. This method is called when your program starts running, and when Main finishes, your program ends. In the CourseExamples project you notice a standalone program named Program.cs in Solution Explorer. When you double-click on you will see the source code as shown in Figure 23. This is the Main program of a Windows Forms application.

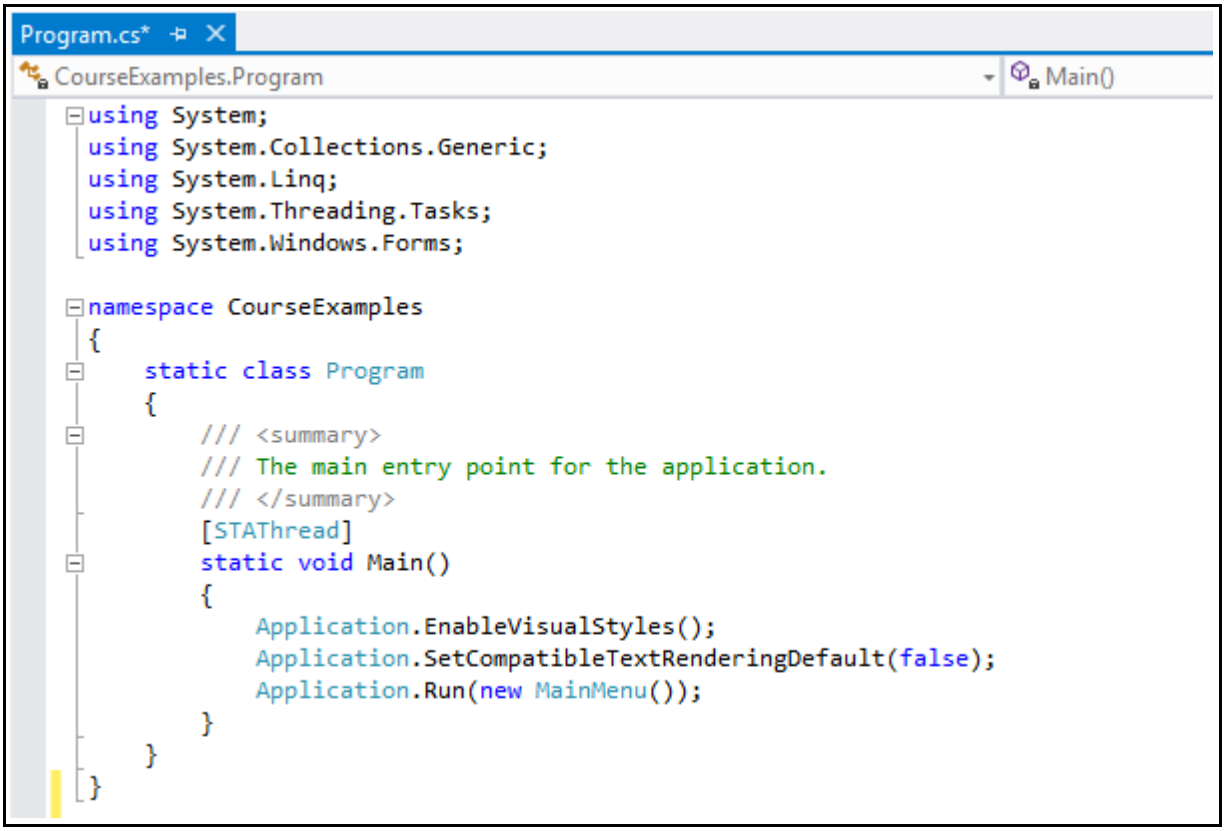


Figure 23: Main Program in Windows Forms Application

The names that you "invent" to identify things are called identifiers. You also create identifiers when you make things to hold values; *amountSugar* might a good choice when we want to hold the amount of sugar required. Later on we will look at the rules and conventions which you must observe when you create identifiers.

The words which are part of the C# language itself are called keywords. In a recipe a keyword would be something like "mix" or "heat" or "until". They would let you say things like "heat sugar until molten" or "mix until smooth". In fact, you will find that programs look a lot like recipes. Keywords will appear in dark blue in Visual Studio.

3.4 Classes and Objects

We will introduce in this chapter the concept of a class and an object, mainly to understand and learn about the C# programming language, since this language is object-oriented. Later on this course, we will extend on this concept to actually build our own classes to model business problems.

Class Concept

A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on data (methods) and accessors to data (properties).

For example, there is a class String in the System namespace of .Net Framework Class Library (FCL). This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(), Trim(), Substring(), etc. It also has some properties like Length (used to find the length of the string).

Another example (outside the programming world) is an architectural blue print of a house. This blue print tells the builder how to exactly build a house. The blue print is not the house, but it is the abstract model to build a house.

Object Concept

As mentioned above, a class is an abstract model. An object is the concrete realization or instance built on the model specified by the class. An object is created in memory using the keyword 'new' and is referenced by an identifier called a "reference".

An object models some part of reality and is therefore something that exists in time and space. An object has state, behavior, and identity, the structure and behavior of similar objects are defined in their common class.

 **Note:** The terms instance and object are interchangeable.

To instantiate an object from a class in C#, you would use the following syntax:

class_name *myObject* = **new class_name()**;

class_name	This is the type definition for the object *myObject*. That means the object *myObject* is of type class_name. This is similar to a data type, when you declare a variable to be of a specific data type.
myObject	This is the name of the identifier pointing to an instance of the class_name in memory. This is like a variable, except it does not hold only one simple value, but lots of data.
new	This is a C# keyword, instructing the compiler to create an instance of the class class_name in memory.
class_name()	Whenever you see parentheses after a name then you have a method. This is the constructor method to initialize the object based on the instructions in the class. This is a special method because you use the class name as the method name.

If you remember from the previous example where we coded an event for the button to open a new form frmCalculator. This piece of code is shown in Figure 24.

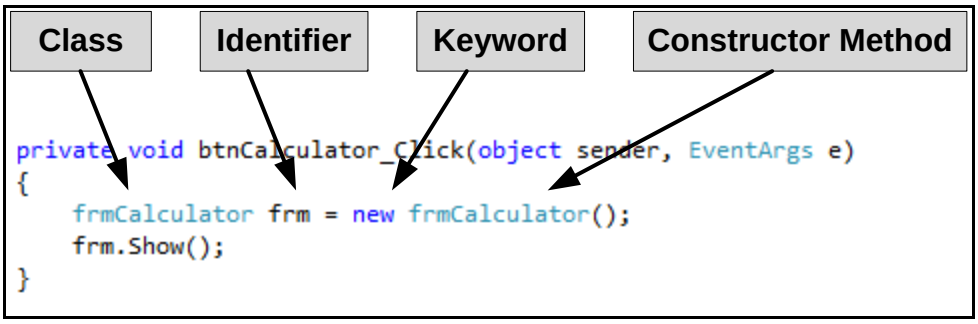


Figure 24: C# Syntax Example for Creating an Object

As you can see here, the form frmCalcuator is the class, the name frm is the identifier pointing to a specific instance of this form in memory, the keyword new to instantiate an object from a class, and the constructor method frmCalculator().

As you can imagine, one wants to open a form more than once displaying different data (from different records in the database, for example). Rather than creating multiple form classes to achieve this goal of having multiple, equal forms being displayed, you create one class and derive or instantiate multiple instances.

Back to our architectural blue print example, using this blue print you can now build one or many houses (a new housing development, for example). The individual houses are the objects, or instances, of the same blue print.

Static Class

One more important concept of a class we need to discuss here, we will also expand on this throughout this course. Sometime you need to write code that does not need to instantiated, meaning creating multiple versions in memory. In C#, all code is placed into classes. There is no other place for writing code, therefore, the concept of a static class exists. A static class acts like a module, you cannot instantiate an object from this class, you use the class directly to call its methods. In the C# programming language, there are many static classes, such as System.Math. You would call mathematical methods by using this class directly rather than first instantiating it.

Back to our blueprint example, sometimes an architect creates an architectural plan just for one unique house. There will be only one house built from this plan, so all the details about this specific house are included in the plans.

 **Note:** A singleton class is a class that only allows one instance. A singleton class is useful if you want to take advantage of the instance class related features but only allow one instance.

4. C# Language Fundamentals

As mentioned earlier, any C# code must be placed into a Class. In this chapter we will take a look at the basic structure of an entire class file and introduce the basic syntax of C#

4.1 Basic Structure of C#

Since C# is object-oriented, we have to first understand how to set or refer to objects and their characteristics.

Objects exhibit three distinct concepts:

- Properties
- Methods
- Events

Properties define the object's characteristics and data. For example, the Name property assigns a name to an object so that you can refer to the object later. The Text property holds a value that is displayed within the control (depending on the type of the control).

Methods of an object determine the operations that can be performed by the object. For example, a Form object has a Show method that will display the form on the screen after this method is issued.

Events of an object are signals sent by the object to your application that something has happened that can be responded to. For example, a Command Button has a Click event. If the user clicks this button, this event is generated in that moment. Your application can respond to this event by executing some C# code to handle this event.

As you write code, you often need to refer to the properties, methods, and events of an object. In the Code Editor window, you simply type the name the object followed by a period (also known as dot operator) and the name of the property, method, or event as shown below:

txtSum.Text = "15"	Assigns a string to the Text property of the text box control txtSum.
txtSum.Focus()	The Focus() method moves the focus to the text box control txtSum.
txtSum.Leave	Refers to the Leave event of the text box control txtSum

As mentioned above you refer to properties, methods, or events by typing the dot operator after the object or class. This process is facilitated by the use of the IntelliSense list. This list shows you all available options based on the kind of object that you are referring to as shown in Figure 25. It also displays a tool-tip like box showing some basic information about the selected property, method, or event.

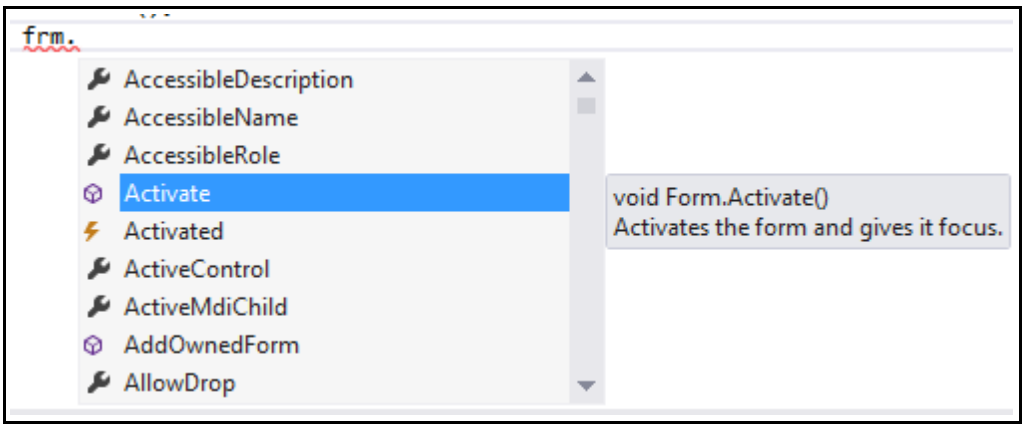


Figure 25: Intelli-Sense Window in Visual Studio

Properties

Properties are the characteristics of an object. In C# code, you simply refer to the property on the left side, use the equal sign(=) and type a value or an expression on the right side.

Object.property = value;

Most properties are read and write properties, but some properties are read-only properties, they cannot be set in C# code. You can use those property values in decision structures, such as an If statement. When you try to assign a value to such properties, you will receive a syntax error indicated by the red wavy line below the statement as show in Figure 26.

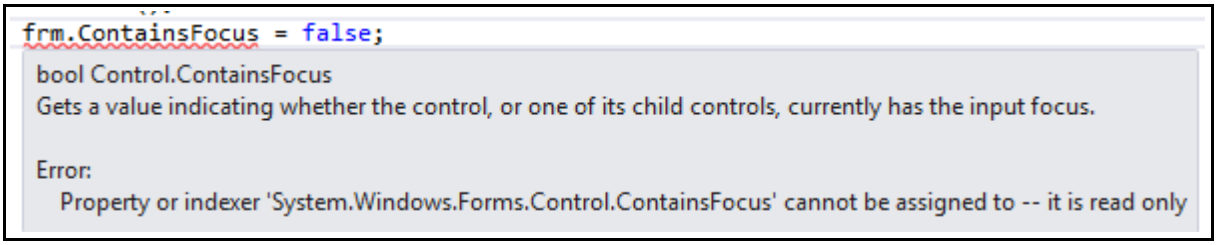


Figure 26: Syntax Error (Assign value to read-only property)

Methods

Methods are like procedures and functions in other programming languages. In C#, they are simply called methods. The way you code the method determines whether it is simply a procedure (executing some statements and **not** returning a value) or a function (executing some statements and returning a value). Methods fulfill the same functionality as procedures or functions, centralizing and reusing code. If you find yourself copying and pasting code or writing very similar code, then this is exactly the situation where you want to place this code somewhere central and call it from the places where this functionality is needed.

You can either place a method in the current form's class or create a separate class and make it a public method. A method may take some input values through its parameters and may return a value of a particular data type.

The basic syntax of a method is shown below:

```
<access modifier> <return type> <method name> (<type> <identifier1>, <type> <identifier 2>,…)
{
        //Body of method
        return <value or object>
}
```

access modifier	Whether the method is public or private. There are other modifiers which we will introduce later.
return type	The type to return, or if the method does not return anything, then use the keyword void.
method name	A name for the method Choose a meaningful name, plus use a verb at the beginning, such as get, set, calculate, etc.
type	The type of a parameter used inside the method. The type can be a data type, but it can also be an object.
identifier	The name of the reference to a parameter.
return	A C# keyword that instructs the method to return some kind of value or object.
<value or object>	The value or object to be returned by the method. The type has to match the methods return type

In the next example, we will create a simple method to perform the addition operation. Later on, we will expand on this so that we can include other operations as well.

Example 2-1: Creating a method

1. Navigate to the frmCalculator.cs file.
2. Below the event method (after the curly braces of btnAdd_click) type the following:
   private string performCalc(string strNum1, string strNum2)
   {

   }
3. You may notice the red, wavy line under the method name. This is due to the fact that we have not yet specified the return argument. Inside the method type the following:

```
private string performCalc(string strNumber1, string strNumber2)
{
    return (double.Parse(strNumber1) + double.Parse(strNumber2)).ToString();
}
```

4. Now use this method in the btnAdd_Click event handler. Comment out the previous statement by placing two forward slashes in front of the statement.
5. Modify the event as follows:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    //Example 1-5
    //lblResult.Text = (double.Parse(txtNumber1.Text) + double.Parse(txtNumber2.Text)).ToString();
    //Example2-1
    lblResult.Text = performCalc(txtNumber1.Text, txtNumber2.Text);
}
```

6. Save the project and run it. Test it to make sure it still works!
7. There are three important things to mention about the method we just coded earlier:

- The method returns a string data type
- The method is private, meaning it can only be used within this class
- The method has two parameters, both are of string data type (strNum1, strNum2)

Events

Windows Forms applications are event-driven. This is now common among the modern programming environments. Events are caused by user actions, and you write code to respond to those events, if necessary.

When you create an event handler, the name of the handler is comprised of the object name, an underscore, and the event name. For example, in the earlier example we created an event for the Add button. The name of the event handler was btnAdd_Click.

This is the default name for an event. You can choose any name based on your own naming convention. What you have to know is how the event handler is wired behind the scenes to respond to a certain event. This event wiring is done by the concept of a delegate.

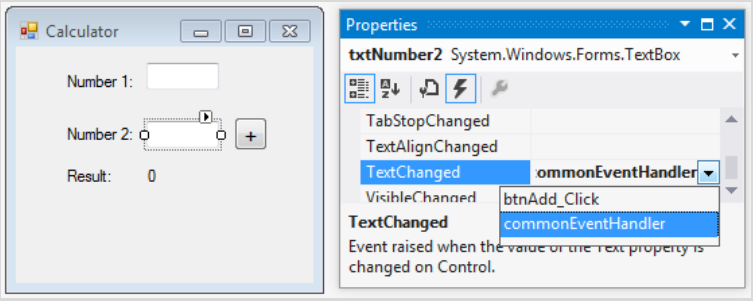
This concept allows you to code one event handler that can be used (or better wired) for many events. Take a look at the following example.

Example 2-2: Creating a custom event handler

1. Navigate to the frmCalculator.cs file.
2. Write the following event handler:

```
//Example 2-2
private void commonEventHandler(object sender, EventArgs e)
{
    lblResult.Text = performCalc(txtNumber1.Text, txtNumber2.Text);
}
```

3. Navigate to the form in design view.
4. Select the textbox control txtNumber2, in the property sheet click on the lightning bolt, then click inside the TextChanged event property. In the drop-down list, select commonEventHandler.



5. Now select the button and link the click event to the same event handler.
6. Save the project, run it and test it. When you enter a number in the second textbox control, the event is fired and the event handler is executed.
7. Open the file frmCalculator.Designer.cs file, and view the Windows Form Designer generated code. In this section you find the wiring for both controls.

```
//
// txtNumber2
//
this.txtNumber2.Location = new System.Drawing.Point(97, 55);
this.txtNumber2.Name = "txtNumber2";
this.txtNumber2.Size = new System.Drawing.Size(54, 20);
this.txtNumber2.TabIndex = 1;
this.txtNumber2.TextChanged += new System.EventHandler(this.commonEventHandler);
```

Class File Structure

The very basic structure of a C# class file is shown in Figure 27.

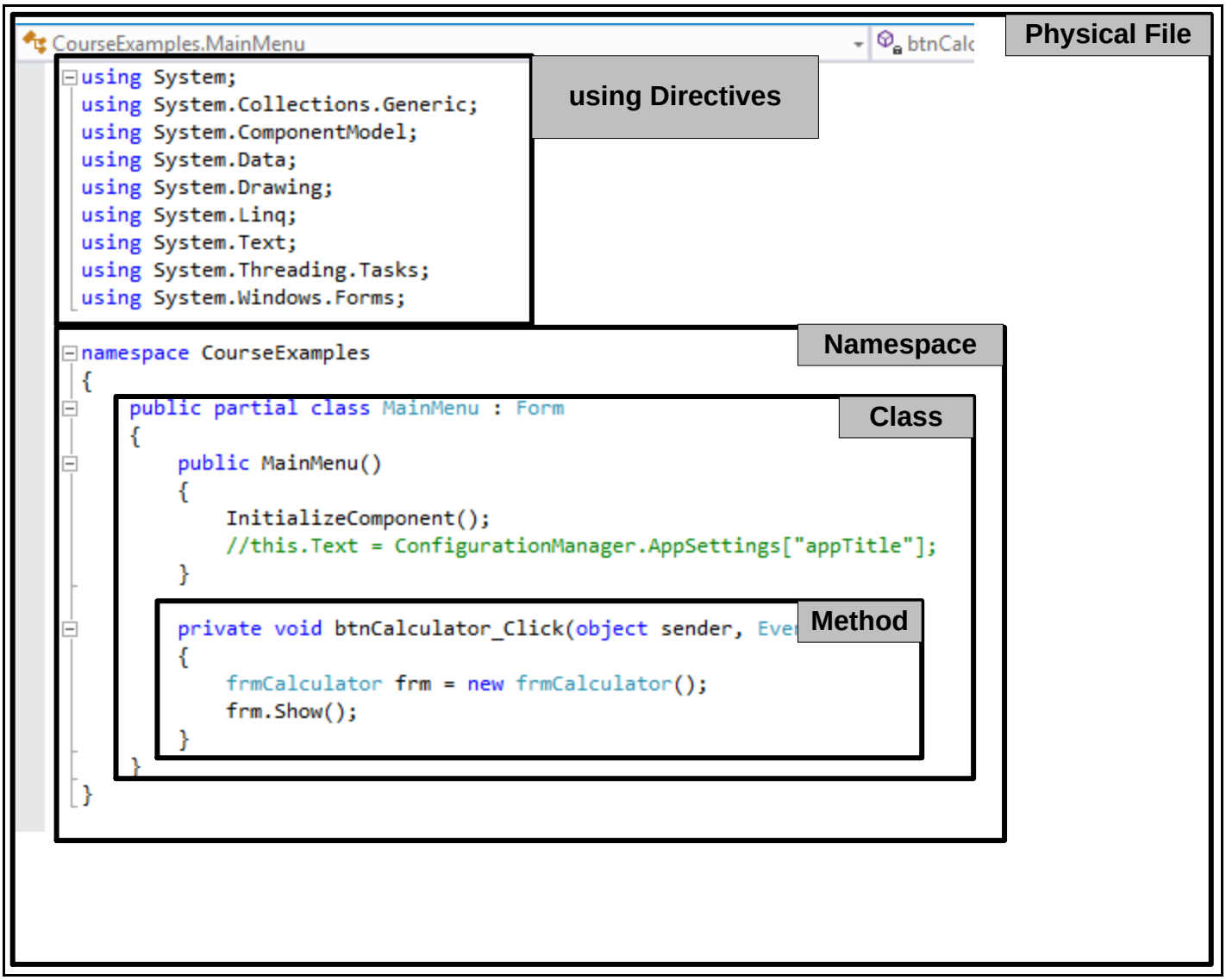


Figure 27: C# Class File Structure

First of all, you have the physical file that is displayed in the Solution Explorer. Once you open the file, you see the following core components:

- using Directives
- Namespace
- Class definition
- Methods

using Directives:
At the very top you see many "using" statements. These are called directives and they are explained in detail in the next section. Basically, they instruct the compiler to use the namespace listed after the using keyword. You do not have to use a using directive to include a namespace, it is just used for simplifying references to classes contained in these namespaces.

Namespace:
A namespace is a logical object to simply organize your code. As you know, the .NET Framework class library comes with over 10,000 classes. To categorize those classes, namespaces are used. There is no relation between a namespace and a physical file or folder, it is purely logical.

For example, there are two main namespaces:

- System
- Microsoft

The Microsoft namespace contains classes that relates specifically to Microsoft Windows, meaning it contains classes that allow you to interface with Microsoft Windows.

The System namespace is a more generic namespace that contains pretty much everything else, such as File I/O, data types, drawing etc.

There are a few other namespaces, but the two mentioned above are the most important ones.

Class:
The class keyword makes this code page a class, remember, all C# code is container in classes. It is required to use the class keyword before writing any line of code. To refer to methods within the class, you use the class name, the dot operator, and then the name of the method.

Method:
Within the class you can have various different constructs, such as methods. We have already used some basic methods in the previous examples, this is where you actually place individual code components. Methods are the smallest container for your code.

Namespaces, Using Directives, and References

A directive construct exists in many programming environments. They are normally placed at the top of the source code file and they apply to the entire file. In scripting languages you find an #include directive that basically reads other files and include those with the current source file.

In C# you mostly notice the using directive. The using directive lists various .NET namespaces which are to be included in the current source file. A namespace is nothing else but a way of organizing code. The .NET framework is comprised of thousands of namespaces that include pre-built classes that you can use. Sometimes you may have the same name for a method or class, in order to differentiate these objects, you use a higher-level category feature called namespace. You can compare a namespace to a folder of an operating system, or a module object in VBA.

When you create your own project, every code file that you create will include by default a namespace using the project name.

You do not need to use the directives at all at the top of your code page. You could always include the fully-qualified reference in your code, as shown in Figure 28.

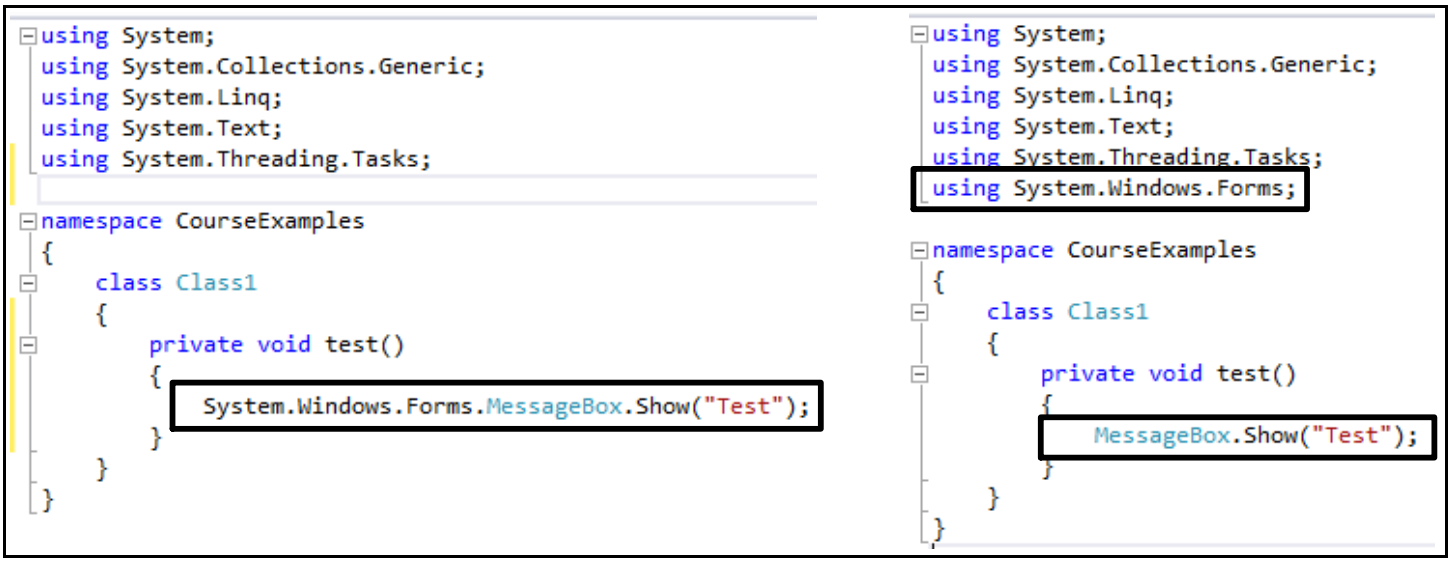


Figure 28: Using Directive

Warning: The using directive allows you only to access the classes in the referenced namespace only and not in its internal/child namespaces. For example, using System does not include using System.IO, you need to add this directive separately.

Adding a using directive does not mean that a certain class is available. Remember, a using directive simply makes referencing classes easier and shorter. To be able a use a class, you must reference the corresponding .NET assembly that includes the class.

The .NET Framework libraries are all stored in the global assembly cache (GAC), located in c:\windows\assembly (.NET 3.5 and earlier) or c:\Windows\Microsoft.NET\assembly (version 4.0 and higher). This is the main location where most of the FCL assemblies are located as shown in Figure 29.

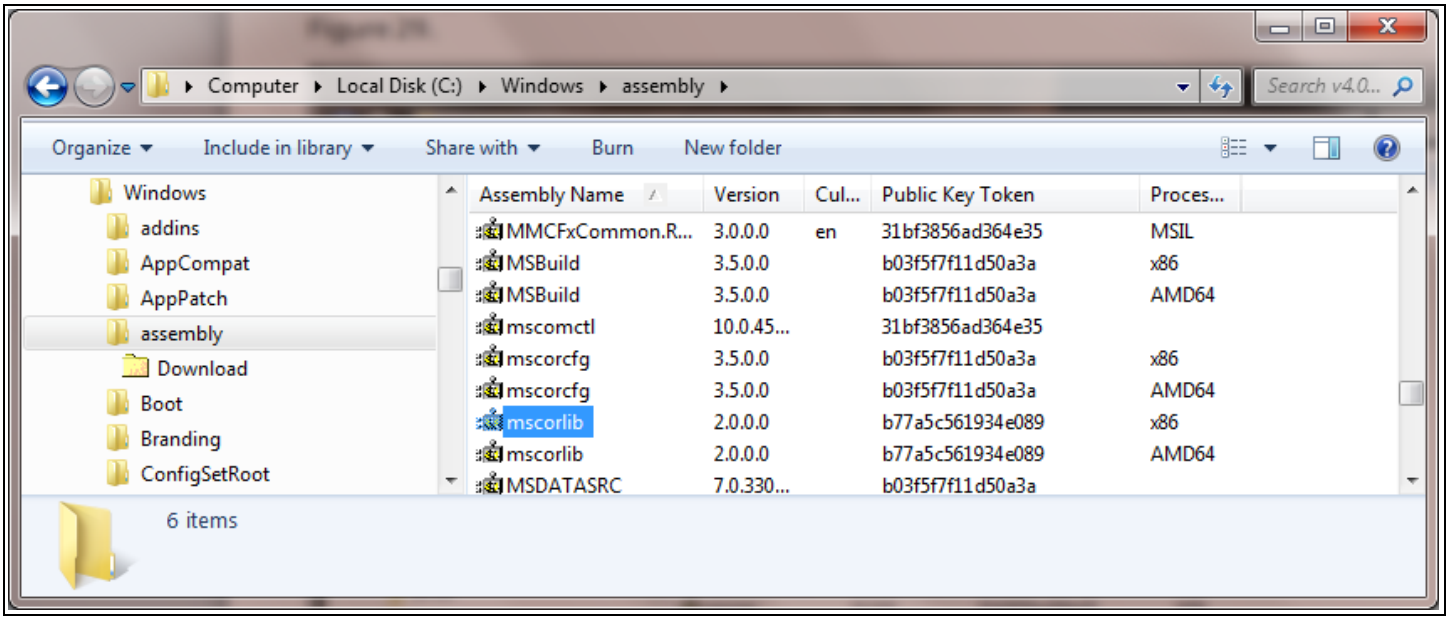


Figure 29: C:\Windows\assembly (GAC) (Version 3.5 or earlier)

These are the physical files, such as the highlighted file in Figure 29, mscorlib.dll. From .NET 4.0 on the physical files are stored in c:\Windows\Microsoft.NET\assembly as shown in Figure 30.

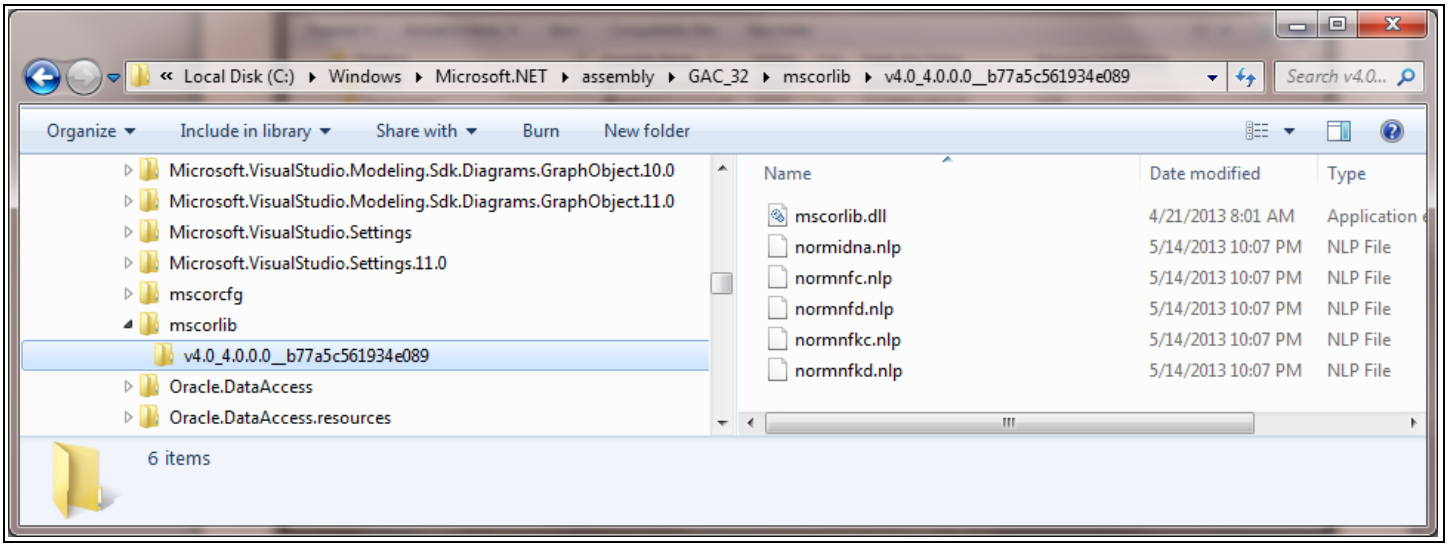


Figure 30: c:\Windows\Microsoft.NET\assembly (GAC) (version 4.0 and higher)

You can view the namespaces and the classes contained in this file in the Object Browser in Visual Studio as shown Figure 31.

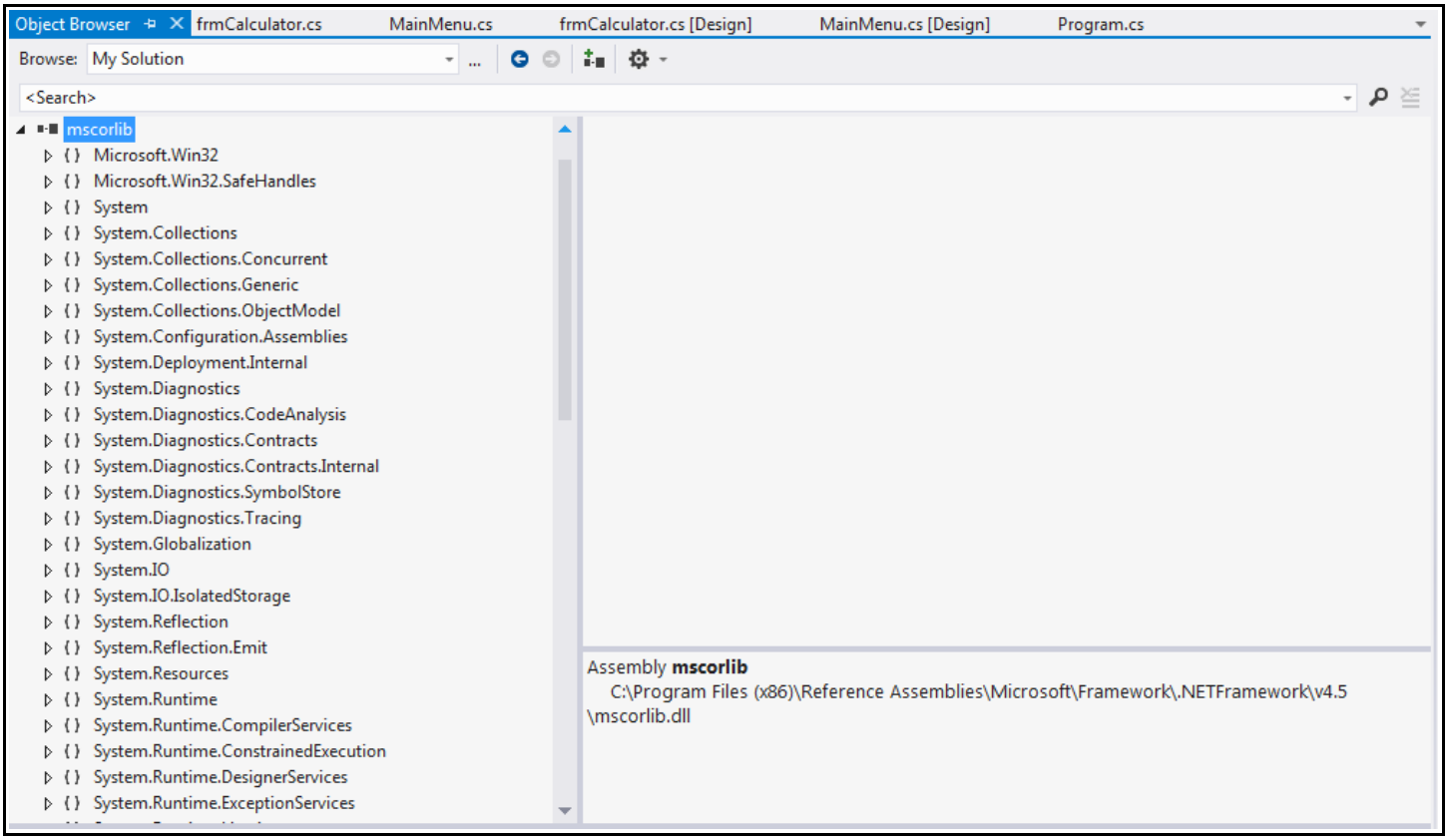


Figure 31: Object Browser in Visual Studio

Note: The file location of .NET assemblies for the Object Browser is not the GAC. The location is shown below:

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5

The files in folder Reference Assemblies are empty stubs that are only used to filter IntelliSense in Visual Studio for different target frameworks.

4.2 Basic Syntax of C#

Now that we have already written quite a few lines of code, let us examine the syntax of C# in more detail. Let us summarize what we have already seen in the few code examples so far:

- First and foremost, every executable line must end in a semicolon.
- Blocks of code are surrounded by curly braces ({ })
- C# is case-sensitive
- C# is strongly typed

Comments

Like in many programming environments, you can comment your code, in fact, you want to. To comment a single line, simply precede the comment with a double forward slash. The comment can also be placed after an executable line of code as shown in Figure 32.

```
private string performCalc(string strNumber1, string strNumber2)
{
   //This is a line comment
   return (double.Parse(txtNumber1.Text) + double.Parse(txtNumber2.Text)).ToString();//End of line comment
}
```

Figure 32: Comments in C#

To comment out lines of code you have two options:

- Select all the lines and click on Comment button in the toolbar. To uncomment those lines, click on the Uncomment button in the toolbar.
- Use the begin comment (/*) and end comment (*/) syntax. Anything between these two symbols is comment

Comments are used for two main purposes. First and foremost, it should be used to document your code. Secondly, it may be used to temporarily disable executable lines of code for testing or debugging.

Example 2-3: Creating method/class XML documentation

1. Navigate to the frmCalculator.cs file.
2. Before the performCalc method create a blank line.
3. Now type three forward slashes. Notice the comments that were created.

```
/// <summary>
///
/// </summary>
/// <param name="strNumber1"></param>
/// <param name="strNumber2"></param>
/// <returns></returns>
private string performCalc(string strNumber1, string strNumber2)
{
    return (double.Parse(txtNumber1.Text) + double.Parse(txtNumber2.Text)).ToString();
}
```

To document methods and classes, you can use a special feature that greatly enhances the productivity of documenting your code. Create a blank line before the method or class, then simply type three forward slashes. You will notice a transformation in front of your eyes, a block of comments is created for you in XML format. Now you just have to fill in the blanks.

Collapsible Code and Regions

You may have noticed the plus and minus sign in the Code Editor on the left side. They allow you to collapse and expand code structures within your code page.

The Code Editor automatically treats namespaces, classes, and methods as regions that you can collapse in order to make other parts of the source code file easier to find and read. You can also create your own collapsible regions by surrounding code with the #region and #endregion directives to as shown in Figure 33.

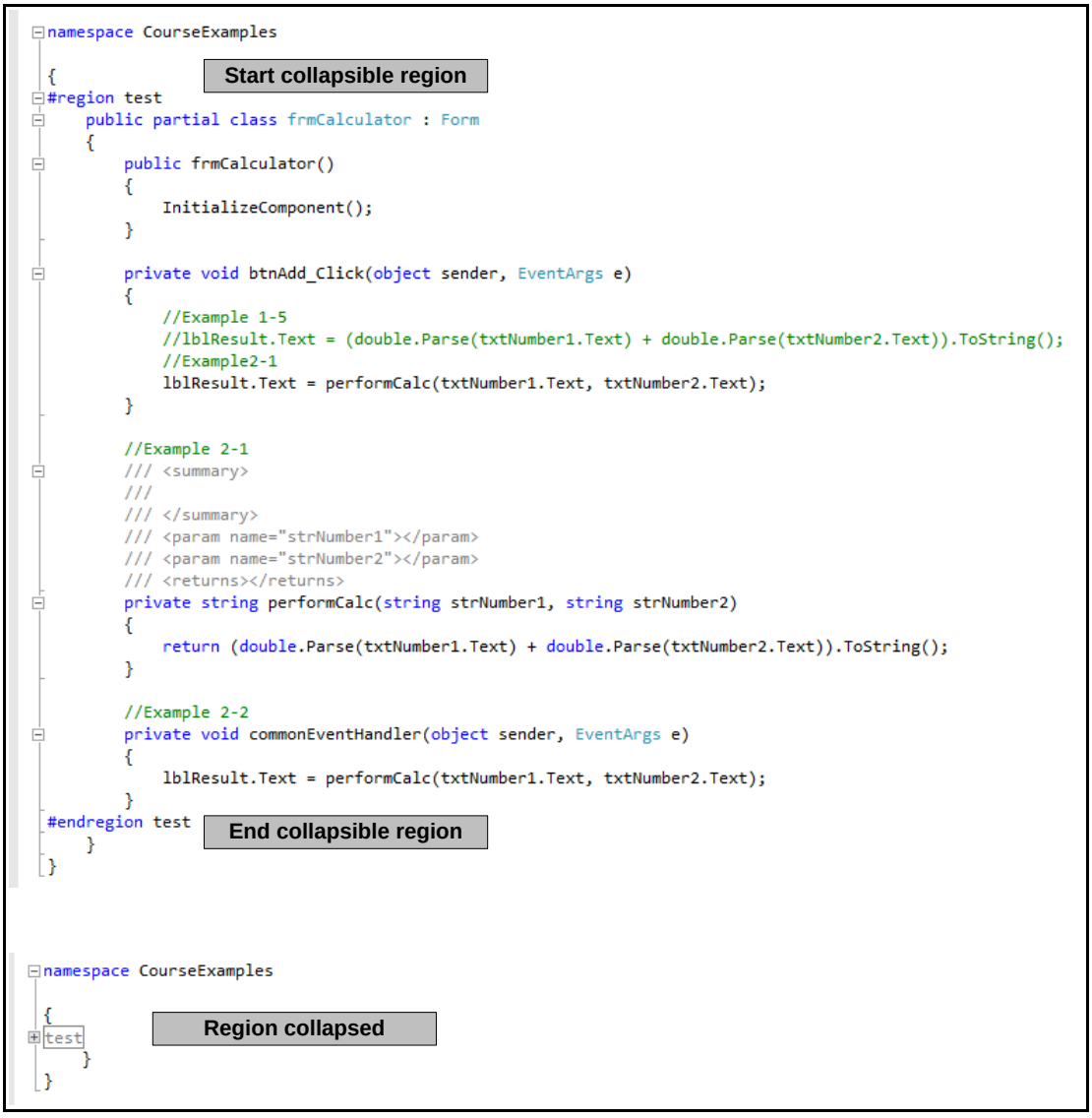


Figure 33: Collapsible Regions

You can name your regions by appending your own name after the directives (top and bottom) so that it is easier to identify them when you collapse them.

Naming Rules/Conventions

Naming identifiers in C# must follow some rules (compiler-based rules). These rules are shown below:

- Identifier name must start with an alphabetic character
- Identifier names can contain alphanumeric characters and the underscore after the first character
- The maximum length is 512 characters

Besides those hard rules, for all of your identifier names you should follow a certain naming convention. There are some recommended rules, all of them differ somewhat. You should select one and if necessary, adjust it. Once you have finalized your convention, stick to it. Consistency is important, especially for maintenance work down the road.

Since C# is case sensitive, I do not recommend using the same spelling but different casing for two different identifiers. Especially in light of VB.NET which is not case sensitive.

Pascal**C**asing is a style where every component within a name starts with a capital letter.

Example: CalculateStatistics, ClientActivityResults

camelCasing on the other hand uses lower case for the first component and then initial capital letter for the following components.

Example: calculateStatistics, clientActivityResults

There are two main rules where many developers agree:

- Use PascalCasing for class names and method names.
- Use camelCasing for method arguments and local variables.

 **Note:** The use of the Hungarian notation is not recommended anymore in the .NET environment. Furthermore, the use of the underscore is generally discouraged as well.

4.3 Variables in C#

Like many other programming languages C# has many different types of variables. One important distinction is the concept of a value vs. a reference variable. We will start out with the simple type variable, the value variable.

Value Variable

A value-type variable is also referred to as a simple variable. A value variable simply holds one value, such as an integer variable x may hold the number 5. A value-type variable is always stored in the stack memory. The stack memory is the faster type of the memory of a computer. It uses the Last In, First Out (LIFO) method. Therefore it is called stack, it only has one opening at the top. Whatever you put in last resides at the top, and things that were added earlier are more at the bottom.

When a variable is declared, it goes into the stack memory at the top. It can now be accessed very quickly since it resides at the top of the stack. The stack memory can only be accessed from the top. When the variable x goes out of scope because the method in which it was defined has finished executing, the value is discarded from the stack.

Using the stack is efficient, but the limited lifetime of value types makes them less suited for sharing data between different classes.

A value variable actually holds the value rather than a reference or pointer to a location in memory (see reference variables).

We will demonstrate this concept with an example as this is very important to understand, especially when it comes to reference variables.

Step 1: Create two string variables named val1 and val2

Step 2: Assign an initial value to val1 from a textbox control on a form

Step 3: Assign val1 into val2

Step 4: Display both value variables in a message box

Step 5: Assign another value to val1 from a second textbox on a form

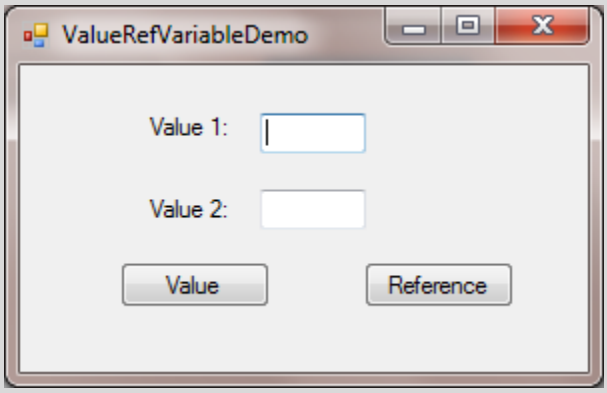
Step 6: Display both value variables again in a message box

You will notice that in the last step variable val2 still holds the initial value, whereas variable val1 shows the new value. This demonstrates that value variables holds the actual value and not any references in memory where a value is stored.

You will also see in the next section that the behavior for reference variables is very different.

Example 2-4: Value Variables

1. In the project CourseExamples add another form by right-clicking on the project name in Solution Explorer, select Add → Windows Form….
2. Name this form frmValueRefVariableDemo and click on the "Add" button.
3. Add the following controls (two labels, two text boxes and two buttons as shown on the right.
4. Set the text properties of the labels and the buttons as shown on the right.
5. Name the text boxes txtVal1 and txtVal2.
6. Name the buttons btnValue and btnReference.



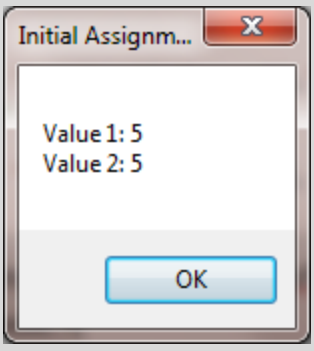
7. Double click on the Value button to create the default click event and add the following code:

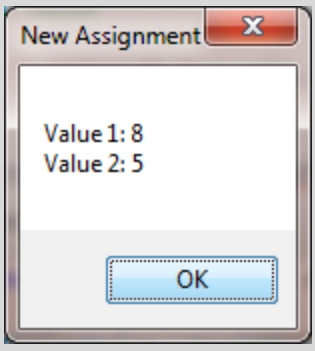
```
private void btnValue_Click(object sender, EventArgs e)
{
    //Initial Assignment
    string val1 = txtVal1.Text;
    string val2 = "";
    //Copy variable val1 into val2
    val2 = val1;
    MessageBox.Show("Value 1: " + val1 + "\nValue 2: " + val2,"Initial Assignment");
    //Assigning variable val1 a different value
    val1 = txtVal2.Text;
    MessageBox.Show("Value 1: " + val1 + "\nValue 2: " + val2, "New Assignment");
}
```

8. Add a button on the Main Menu form, name it btnValRefVariable, double-click on it to create the event and write the following code:

```
private void btnValRefVariable_Click(object sender, EventArgs e)
{
    frmValueRefVariableDemo frm = new frmValueRefVariableDemo();
    frm.Show();
}
```

9. Save the project and run it. Enter two different numbers and click on the Value button.
10. Notice the two message boxes and their values as shown on the right:
11. In the initial assignment step, they both show the same value as we copied val1 into val2.
12. In the new assignment we just assigned a new value into val1. Val2 remains unaffected by this assignment.





A value-type variable is a small and simple variable. C# defines the following value-type variables as shown in Table 1.

C# Keyword	CLS Compliant?	System Type	Range	Description
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit number
ulong	No	System.UInt64	0 to 18,446,744,073,709,551, 615	Unsigned 64-bit number
float	Yes	System.Single	±1.5 ×10 –45 to ±3.4 ×10 38	32-bit floating-point number
double	Yes	System.Double	±5.0 ×10 –324 to ±1.7 ×10 308	64-bit floating-point number
decimal	Yes	System.Decimal	±1.0 ×10 –28 to ±7.9 ×10 28	96-bit signed number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character

Table 1: Value Type Data Types in C#

 **Note:** You should always use the C# keyword for defining data types as this is a shortcut to the underlying defined data type in the Common Type System (CTS) in the .NET Framework.

In general, when dealing with numbers, you really need only to remember three data types:

- long for integers,
- double for floating point decimals
- decimal for fixed-point decimals

 **Warning:** Recall that CLS-compliant .NET code can be used by any managed programming language. If you expose non–CLS-compliant data from your programs, other languages may not be able to make use of it.

Each of the numerical types, such as short or int, map to a corresponding structure in the System namespace. Simply put, structures are value types allocated on the stack. On the other hand, string and object are reference types (see next section), meaning the data stored in the variable is allocated on the managed heap. You will examine full details of value and reference types in the next section.

For the time being, however, simply understand that value types can be allocated into memory very quickly and have a fixed and predictable lifetime.

Reference Variable

In contrast to a value type, a reference type, such as an instance of a class (object) or an array, is allocated in a different area of memory called the heap. In the example below, the space required for the ten integers that make up the array is allocated on the heap.

```
int[] Numbers = new int[10];
```

This memory is not returned to the heap when a method finishes; it is only reclaimed when C#'s garbage collection system determines it is no longer needed. There is a greater overhead in declaring reference types, but they have the advantage of being accessible from other classes.

In the above example, the variable Numbers is a reference-type variable. It does not hold the actual value, it stores a pointer to the object or array in the heap memory. The pointer of the reference variable is actually stored in the stack area of the memory pointing to an object in the heap area of the memory as shown in Figure 34.

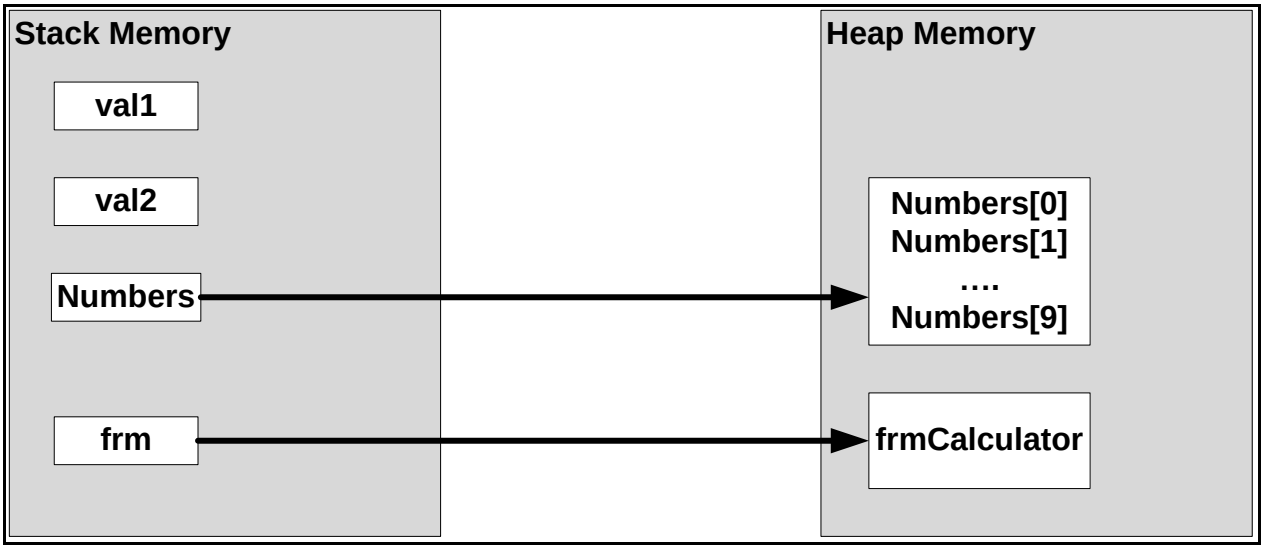


Figure 34: Stack and Heap Memory Allocation

Unlike arrays, strings, or enumerations, C# structures do not have an identically named representation in the .NET library (that is, there is no System.Structure class), but are implicitly derived from System.ValueType. Simply put, the role of System.ValueType is to ensure that the derived type (e.g., any structure) is allocated on the stack rather than the garbage collected heap. Simply put, data allocated on the stack can be created and destroyed very quickly, as its lifetime is determined by the defining scope.

Heap allocated data, on the other hand, is monitored by the .NET garbage collector, and has a lifetime that is determined by a large number of factors. Functionally, the only purpose of System.ValueType is to override the virtual methods defined by System.Object to use value-based, versus reference-based, semantics.

Now we are using the same example 2-4, except we are going to use reference variables to demonstrate the different behavior.

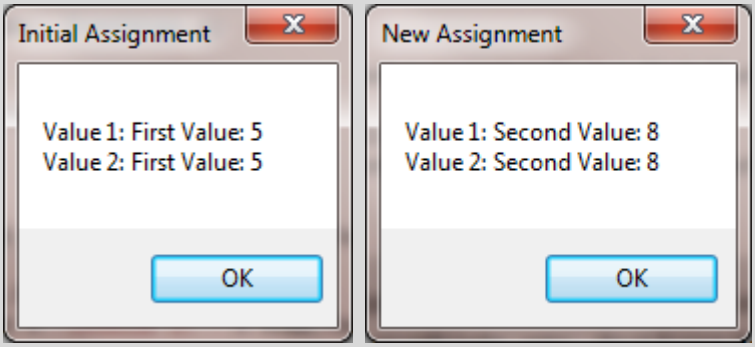
You will notice that when you assign an object into another object, you are not copying the value (which really does not exist in an object anyway) but only the reference pointing to the same object on the heap. Therefore the second dialog box will display the same values.

Example 2-5: Value Variables

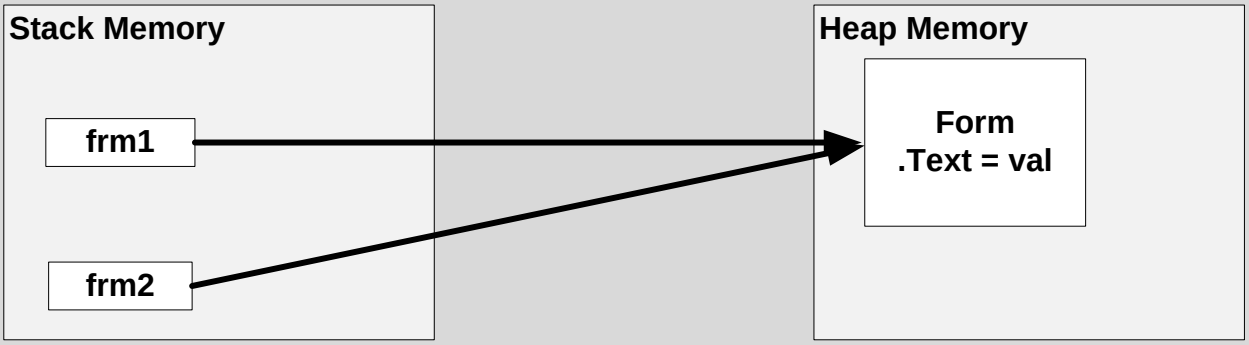
1. Use the form frmValueRefVariableDemo and double-click on the button btnReference to create the click event. Write the following code:

```
private void btnReference_Click(object sender, EventArgs e)
{
    Form frm1 = new Form();
    frm1.Text = "First Value: " + txtVal1.Text;
    Form frm2;

    frm2 = frm1;
    MessageBox.Show("Value 1: " + frm1.Text + "\nValue 2: " + frm2.Text, "Initial Assignment");

    frm1.Text = "Second Value: " + txtVal2.Text;
    MessageBox.Show("Value 1: " + frm1.Text + "\nValue 2: " + frm2.Text, "New Assignment");

}
```

2. As you can see, we are using a form object and storing it into a reference variable frm. An object contains many properties, methods, etc., but here for simplicity we are just changing the text property of the form, which is the Windows title bar caption.
3. Save the project and run it.
4. Enter two different values in the text box controls.
5. Then click on the Reference button. The two message boxes are shown below:



6. As you can see, the output is exactly the same. The second reference variable frm2 has the same pointer to the form object in memory.



A reference-type variable points to a more complex object that does not hold just one value, but a variety of data, properties and methods. C# defines the following built-in reference types as shown in Table 2.

C# Keyword	CLS Compliant?	System Type	Range	Description
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
object	Yes	System.Object	Can store any data type in an object variable	The base class of all

Table 2: Reference Type Variables in C#

Besides the built-in reference types there are many other reference type variables that you will create. When you create your own classes, for example, you will want to create a variable that points to an instance of your class so that you can use the functionality of your class in your code.

 **Note:** There is no date/time data type defined in the CTS. In C#, a DateTime struc (=structure→ simplified class) is created to allow for date and time values.

All built-in data types are in the System namespace of the .NET framework. A structure (struct) defines a value type whereas a class defines a reference type. A struct is very similar to a class except it lacks certain features, for example inheritance. A struct is a simplified class.

In Figure 35 you see again the object class from which the reference types are derived (=inherited). These types are created on the heap memory which is managed by the garbage collection. On the other hand, you see the Value Type class from which the value types are derived. Those types are created on the stack memory, which is not garbage collected. Once a variable goes out of scope, the variable is removed from the stack.

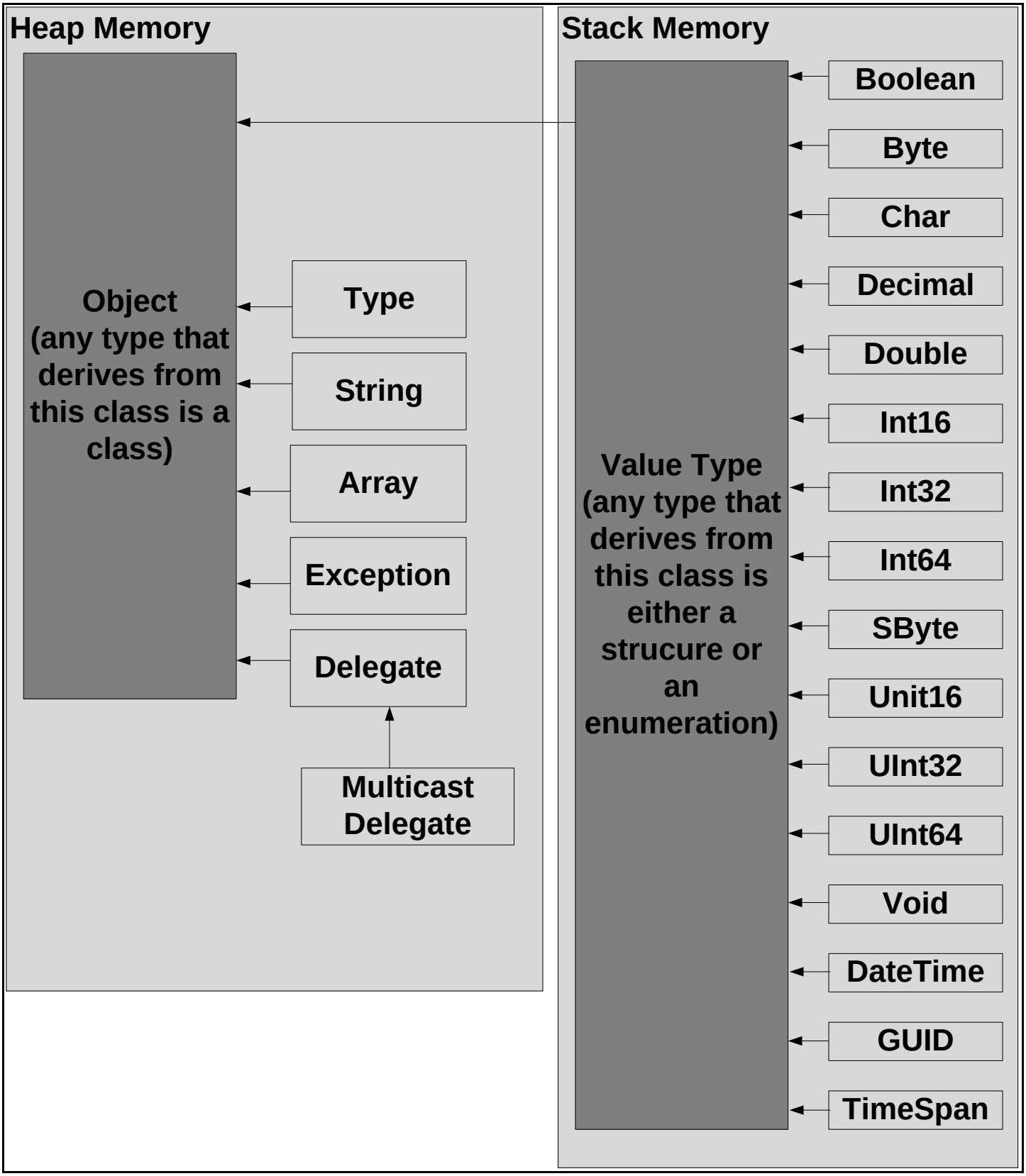


Figure 35: Reference vs. Value Type

Since all types are derived from the Object class, they inherited the following public methods from the object class:

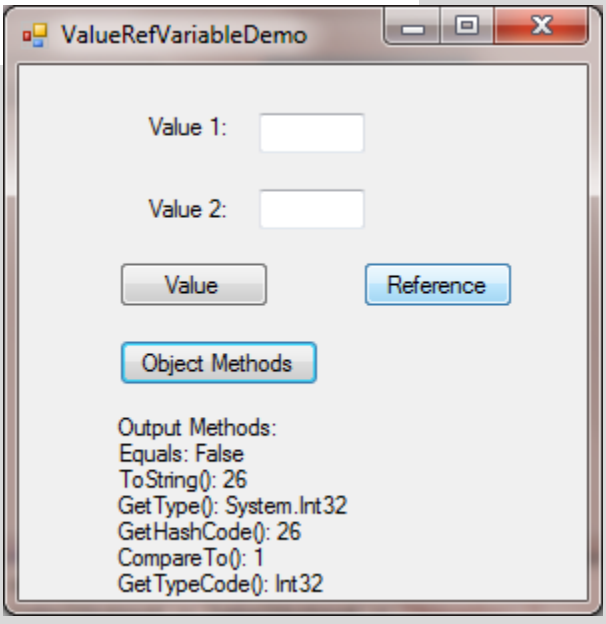
- GetHashCode()
- Equals()
- CompareTo()
- ToString()
- GetType()
- GetTypeCode()

Example 2-6: Public Methods of Object Class

1. Use the form frmValueRefVariableDemo and add another label and button at the bottom.
2. Name the label lblObjectMethods and set its Text property to "Object Methods: ".
3. Name the button btnObjectMethods and set its Text property Object Methods.
4. You may need to widen the button to fit the text and increase the height of the form.
5. Double-click on the button btnObjectMethods to create the default click event. Type the following code:

```
private void btnObjectMethods_Click(object sender, EventArgs e)
{
    int wholeNumber = new int();
    string labelOutput = "Output Methods: \n";
    wholeNumber = 26;
    labelOutput = labelOutput + "Equals: "        + wholeNumber.Equals(30).ToString() + "\n";
    labelOutput = labelOutput + "ToString(): "    + wholeNumber.ToString() + "\n";
    labelOutput = labelOutput + "GetType(): "     + wholeNumber.GetType().ToString() + "\n";
    labelOutput = labelOutput + "GetHashCode(): " + wholeNumber.GetHashCode().ToString() + "\n";
    labelOutput = labelOutput + "CompareTo(): "   + wholeNumber.CompareTo(20).ToString() + "\n";
    labelOutput = labelOutput + "GetTypeCode(): " + wholeNumber.GetTypeCode().ToString() + "\n";
    lblObjectMethods.Text = labelOutput;

}
```

6. Save the project and run it. Click on the button Value Ref Variable in the main menu form, then on the button Object Methods. You should see the output in the label control as shown on the right.



Declaring and Initializing Variables

To declare a variable, use the following basic syntax:

type *variable_name* [= value] [,variable_name [= value]][,…];

You may declare multiple variable of the same type in the same line by comma separating them.

 **Warning:** You must initialize a variable before you can use it, otherwise you will receive a compiler error.

Due to the above warning, it is a good idea to initialize your variables at the time of declaration, even if it is a zero or zero-length string value.

Example: **int** numberOfCopies = 0, totalGrade = 1;

All built-in data types support what is known as a default constructor. This feature allows you to create a variable using the new keyword, which automatically sets the variable to its default value:

- bool variables are set to false.
- Numeric data is set to 0(or 0.0in the case of floating-point data types).
- char variables are set to a single empty character.
- BigInteger variables are set to 0.
- DateTime variables are set to 1/1/0001 12:00:00 AM.
- Object references (including strings) are set to null.

The syntax is shown below:

type variable_name = **new type();**

Example: int numberOfCopies = new int();

 **Warning:** Variables in C# are by default not nullable, meaning you cannot assign a null value to a variable.

Value types can never be assigned the value of null as opposed to reference types where it is used to establish an empty object reference.

Since the release of .NET 2.0, it has been possible to create nullable datatypes. Simply put, a nullable type can represent all the values of its underlying type, plus the value null. Thus, if we declare a nullable bool, it could be assigned a value from the set {true, false, null}.

This can be extremely helpful when working with relational databases, given that it is quite common to encounter undefined columns in database tables. Without the concept of a nullable data type, there is no convenient manner in C# to represent a numerical data point with no value.

To define a nullable variable type, the question mark symbol (?) is suffixed to the underlying data type. Do note that this syntax is only legal when applied to value types. If you attempt to create a nullable reference type (including strings), you are issued a compile-time error.

Example: **int?** numberOfCopies = **null**;

To declare constants, use the const keyword in front of the type declaration. Constants cannot be changed in your program, they are as the name suggests, well constant. In general, constant variable names should start with a capital letter. You must assign a value in the same line as declaration of the constant.

const type *Constant_name* =value;

Scope of Variables

We have seen that when we want to store a quantity in our program we can create a variable to hold this information. The C# compiler makes sure that the correctly sized chunk of memory is used to hold the value and it also makes sure that we only ever use that value correctly. The C# compiler also looks after the part of a program within which a variable has an existence. This is called the scope of a variable.

A block is a number of statements which are enclosed in curly brackets. Any block can contain any number of local variables, i.e. variables which are local to that block. The scope of a local variable is the block within which the variable is declared. In C#, you can declare a variable at any point in the block, but you must declare it before you use it.

When the execution of the program moves outside a block any local variables which are declared in the block are automatically discarded. We have seen that in C# the programmer can create blocks inside blocks. Each of these nested blocks can have its own set of local variables:

```
{
      int i ;
      {
            int j ;
      }
}
```

The variable j has the scope of the inner block. This means that only statements in the inner block can use this variable. In other words the following code example would cause an error:

```
{
      int i ;
      {
            int j ;
      }
      j = 99 ;
}
```

The variable j does not exist at this point in the program, it is out of scope.

In order to keep you from confusing yourself by creating two versions of a variable with the same name, C# has an additional rule about the variables in the inner blocks:

```
{
      int i ;
      {
            int i ;
      }
}
```

This is not a valid program because C# does not let a variable in an inner block have the same name as one in an outer block. This is because inside the inner block there is the possibility that you may use the "inner" version of i when you intend to use the outer one. In order to remove this possibility the compiler refuses to allow this.

 **Note:** This is in contrast to the situation in other languages, for example C++, where this behavior is allowed.

It is however perfectly acceptable to reuse a variable name in successive blocks because in this situation there is no way that one variable can be confused with another.

```
{
      int i ;
}
{
      int i ;
      {
            int j ;
      }
}
```

The first incarnation of i has been destroyed before the second, so this code is OK.

A special kind of variable can be used when you create a for loop construction. This allows you to declare a control variable which exists for the duration of the loop itself:

```
for ( int i = 0 ; i < 10 ; i = i + 1 )
{
      Some code
}
```

The variable i is declared and initialized at the start of the for loop and only exists for the duration of the block itself.

5. Number Data Types

We have already introduced the number data types as part of the built-in value data types. This section covers how to work with variables of the number data types and use the built-in methods and properties to perform operations. The most important number data types are:

Whole Numbers: byte, short, int, long

Floating-point decimals: float, double

Fixed-point decimals: decimal

In general, you only need long for whole numbers, double for floating-point decimals, and decimal for fixed-point-decimals.

5.1 Methods and Properties of Number Data Types

The .NET data types are all implemented as objects, in case of numbers as value types derived from the generic object class. These data types come already with a number of prebuilt, useful methods and properties as shown in Table 3.

Data Type	Methods	Properties
byte, short, int, long	Equals, Parse, TryParse	Min, Max
float, double	Equals, IsInfinity, IsNaN, IsNegativeInfinity, IsPositiveInfinity, Parse, TryParse	Epsilon, Min, Last, NaN, PositiveInfinity, NegativeInfinity,
decimal	Add, Ceiling, Compare, CompareTo, Divide, Equals, Floor, GetBits, Multiply, Negate, Parse, Remainder,Round, Subtract, ToByte … ,TryParse, Truncate	MaxValue, MinusOne, MinValue, One, Zero

Table 3: Property and Methods of Number Data Types

While most of these properties and methods are self-explanatory, others represent more advanced number data type features.

For example, the Min and Max properties return the lower and upper range values of the respective data type.

We will cover a few of these methods in the upcoming sections, for example the Parse and TryParse methods, which help you convert string values into a data type.

5.2 Arithmetic Expressions

When building arithmetic expression using numbers, you have a large selection of arithmetic operators as shown in Table 4.

Oper-ator	**Name**	**Description**	**int x = 11;** **int y =5;**	**Operator Type**
+	Addition	Adds to operands together	int result = x + y (result = 16)	Binary
-	Subtraction	Subtracts the right operand from the left operand.	int result = x – y (result = 6)	Binary
*	Multiplication	Multiples the right operand with the left operand	int result = x * y (result = 55)	Binary
/	Division	Divides the right operand into the left operand. If both operands are integers, then the result is an integer.	int result = x / y (result = 2)	Binary
%	Modulus	Returns the remainder of a division operation (right operand divided by the left operand)	int result = x % y (result = 1)	Binary
+	Positive sign	Positive sign before an operand	int result = +x – y (result= 6)	Unary
-	Negative sign	Negative sign before an operand, changes a positive operand into a negative and a negative one into a positive one.	int result = x – (-y) (result =16)	Unary
++	Increment	Adds 1 to the operand	int result = ++x (result = 12)	Unary
--	Decrement	Subtracts 1 from the operand	int result = -- y (result = 4)	Unary

Table 4: C# Arithmetic Operators

The first five operators are binary operators as they work on two operands. The last four operators are unary since they only work on one operand.

Warning: Increment, Decrement operators: You can prefix or postfix the increment and decrement operators. Postfixing means that the right operand is first assigned to the left operand and then incremented or decremented as shown below:

int x = 0

int y = x++ (y = 0, x = 1)

Prefixing, on the other hand, performs the opposite operation. It first increments or decrements the right side operand and then assigns it to the left side operand as shown below:

int z = ++x (x = 2, z = 2)

5.3 Assignment Statements

We have already discussed the initial variable assignment, and in general you use the equal (=) sign to assign a value to a number data type variable. However, you can also combine an assignment with an arithmetic operation. You can use the additional assignment operators shown in Table 5.

Operator	Name	Description
=	Assignment	Assigns the value on the right side into the left side.
+=	Additive assignment	Adds the right operand to the value stored in the variable and assigns the result to the variable.
-=	Subtractive assignment	Subtracts the right operand from the value stored in the variable and assigns the result to the variable.
*=	Multiplicative assignment	Multiplies the variable by the right operand and assigns the result to the variable.
/=	Division assignment	Divides the variable by the right operand and assigns the result to the variable. If the variable and the operand are both integers, then the result is an integer.
%=	Modulo assignment	Divides the variable by the right operand and assigns the remainder to the variable.

Table 5: C# Number Assignment Operators

The syntax for a simple assignment statement is as follows:

variable_name = expression;

Using the above assignment operators, you can now simplify certain assignment statements. For example, in loops you commonly need to keep track of the number of loops by using a counter variable. For every loop, you want to increase (or decrease) the counter variable.

A common assignment statement looks like the one shown below:

intI = intI + 1;

Now we can simplify it as follows:

intI += 1;

You can further simplify it by using the Increment operator:

intI++;

Like in any programming language, there is an order of precedence for arithmetic operators. Unless parentheses are used to prescribe an order of arithmetic operations, the following order of precedence applies in C#:

1. Increment, Decrement
2. Positive, Negative
3. Multiplication, Division, Modulus
4. Addition, Subtraction

When assigning literal number values into variables, you have to understand that any literal whole number in C# is by default the first of these types in which its value can be represented: int, uint, long, ulong, see Figure 36.

For example, the number 2,147,483,647 is the upper bound of the data type int, therefore, when typing this number in C#, it is automatically of type int, see Figure 36 when you hover the mouse over the number.

Once you increment this number by one (2,147,483,648), it is automatically of type uint, since it is larger than the upper bound of type int.

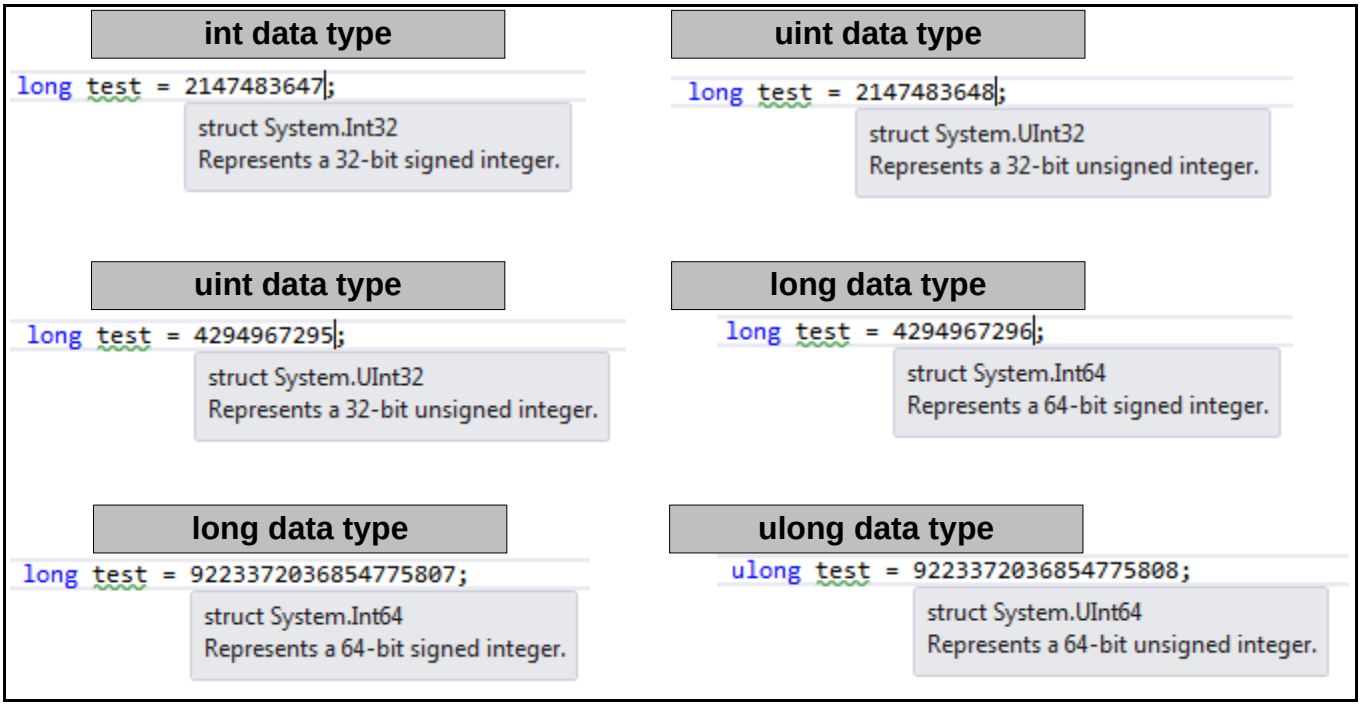


Figure 36: C# Default Literal Whole Number Data Types

When assigning a literal whole number into a byte data type, for example, the literal which is of type int is implicitly(automatically) converted into type byte, provided, of course, it is less than 256.

Example:

byte test = 231; (231 is converted into type byte)

byte test = 543; (compiler throws error, number is too large for type byte)

To explicitly convert a whole number to a long data type when assigning it, suffix the number with an l or L.

Example:

long test = 231**L**; (231 is of type int by default, the L at the end of the number makes it automatically a number of type long)

When dealing with literal decimal numbers (as opposed to whole numbers), a real numeric literal on the right side of the assignment operator is treated as double by default. When assigning a literal whole number into a double data type, for example, the literal which is of type int is implicitly (automatically) converted into type byte.

Example:

double test = 5;

To explicitly convert a whole number to a double data type when assigning it, suffix the number with a d or D.

Example:

double test = 5**D**;

The implicit conversions take place automatically, if a conversion is possible. If that is not the case, then the compiler throws an error. The following assignments will all throw an error as shown Figure 37.

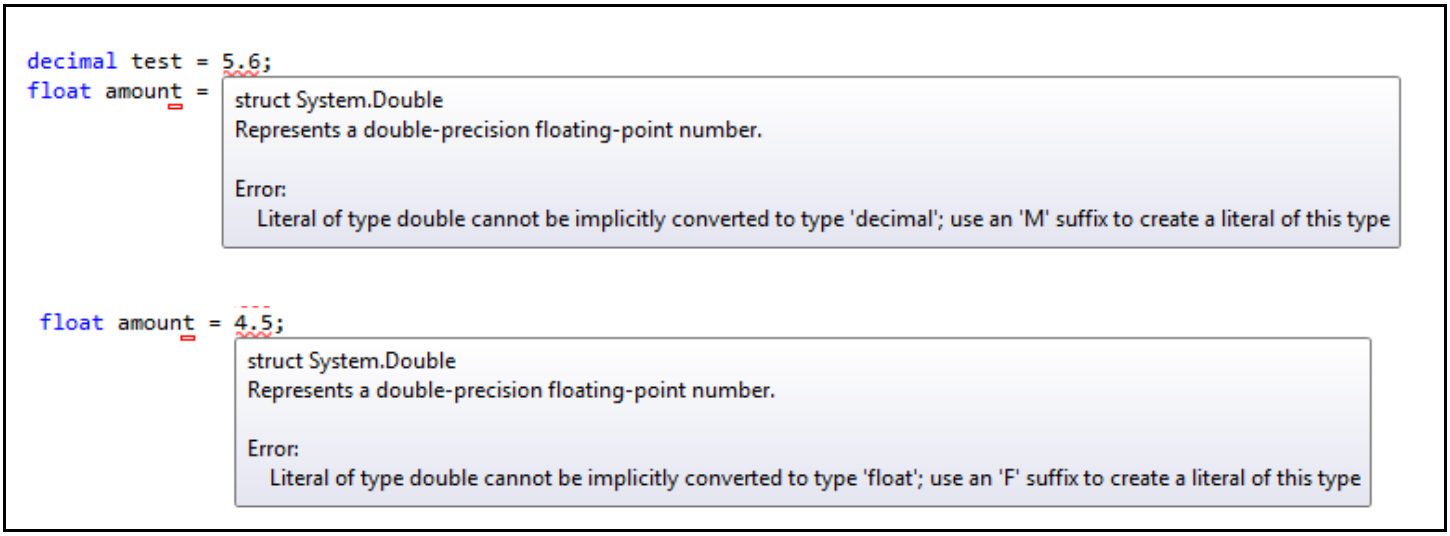


Figure 37: Implicit Assignment Conversion Errors

To explicitly convert a literal number into a decimal, use the suffix m or M.

To explicitly convert a literal number into a float, use the suffix f or F.

Example:

float test = 5.3**F**;

decimal amount 3.5**M**;

5.4 Casting/Converting between Number Data Types

When writing any code, you almost always need to convert data types for a variety of reasons. One good example is that when you read data from a textbox control, the data is always of type string. Even when you enter only numeric characters into the textbox, the type is string. If you want to perform some arithmetic operation on that number, you need to convert the string into a number type.

Casting is one method of converting. There are other methods in C# to convert data type (and object types, for that matter), and there are some minor differences between them.

Casting of data is performed within a base of data types, such as number data types. Converting of data means converting across base data types, such as converting a string representation of a number into a number data type.

C# will perform implicit casting if no information is lost during the conversion process. The most common situation when you convert numbers with a less precise type to a more precise type. This is called widening conversion.

Therefore, you can implicitly convert the following:

byte → short →int →long → float → double → decimal

char → int

If you have to convert from a double to an integer, for example, you must explicitly cast the double type value into an integer. This is done using the following syntax:

(type) expression

Example:

```
double dblA = 5.6;

int intI;

intI = (int) dblA; (intI = 5)
```

As you can see in this example, you will lose information, in this case the fractional part (0.6). An implicit conversion would not work in this case, an error message is generated by the compiler. You can also use an explicit cast in an arithmetic expression. Then the casting is performed before the arithmetic operation. Take a look at the following example:

Example:

```
int intA = 5;
int intB = 2;

decimal decResult;
decResult = intA/intB;                          (decResult = 2)
decResult = (decimal) intA/(decimal) intB; (decResult = 2.5)
```

You can see from this example that casting the individual operands into a decimal type makes a difference since the division is now performed on decimal operands rather than integer operands.

Casting only works on "similar" types (or in the object world on objects that are related to each other). You cannot use the following cast, even though it may make sense:

decimal decA = **(decimal)** "34.56"; → compiler will generate an error

For this kind of conversion, other conversion must be used and you have to take care of the error handling in case the conversion fails.

To convert data across base data types, use the Convert class. The two most common methods for data conversion are the ToString([format]) and the Parse(string) methods. The ToString() method lets you convert any value into a string. The Parse method performs the reverse operation, it converts a string into another data type. Then there is also the TryParse method allowing you to handle the situation when the conversion fails. TryParse returns a Boolean false when the conversion fails, otherwise it returns true.

Examples:

decimal subtotal = **Convert.ToDecimal(**txtTotal.Text**)**;
int years = **Convert.ToInt32(**txtYears.Text**)**;

double length = **double.Parse(**txtLength.Text**)**;

double dblNumber;
bool convert = **double.TryParse(**"test", **out** dblNumber**)** → convert = false, dblNumber = 0
bool convert = **double.TryParse(**"3.5E-6", **out** dblNumber**)** → convert =true, dblNumber = 3.5E-6

5.5 The Math Class

The Math class is a comprehensive library of advanced mathematical functions. The Math class is a static class, meaning you do not have to instantiate it. We learn much more about this later in this course, but I thought it is worth mentioning here.

You simply type the word math followed by the dot operator and then select one of the methods from the IntelliSense list. Many of the methods are self-explanatory. Some of the more common and important ones are listed below:

Math.Round(number[, number_of_digits[,midpointrounding]]**)**

The round method by default is using banker's rounding, meaning the digit 5 is rounded up or down depending on whether the last digit to be rounded is even or odd. If it is even, than it is not changed, if it is odd, it is round up to the next even number.

Example:

Math.Round(4.245,2**)** → 4.24 (last digit to be rounded is 4, it is even so it remains 4)

Math.Round(4.255,2**)** → 4.26 (last digit to be rounded is 5, it is changed to 6)

You can specify the rounding method by using the third argument. MidPointRounding.ToEven represents the default banker's rounding, MidPointRounding.AwayFromZero represents the common rounding rule (meaning the digit 5 is always rounded up).

The Intelli-Sense windows containing all the methods and properties of the math class are shown in Figure 38.

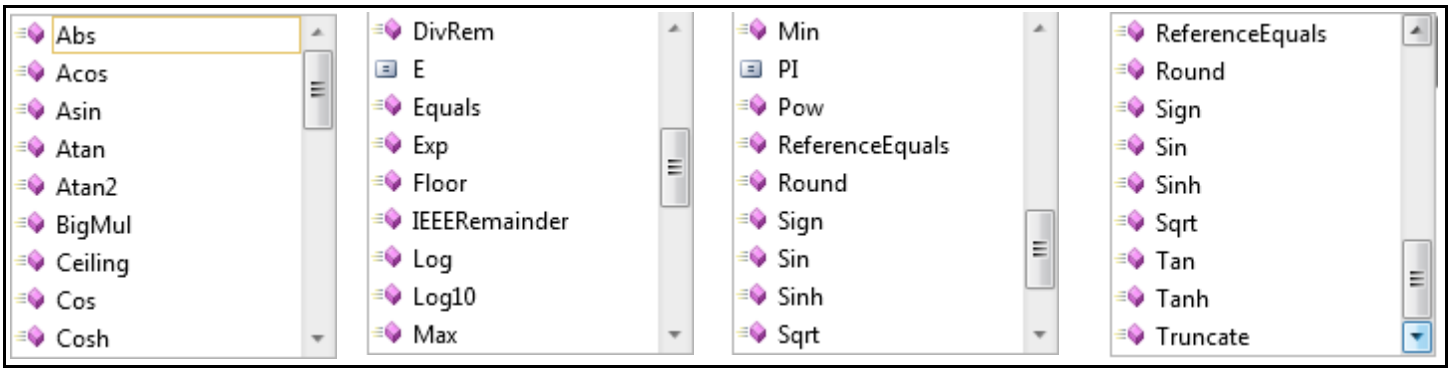
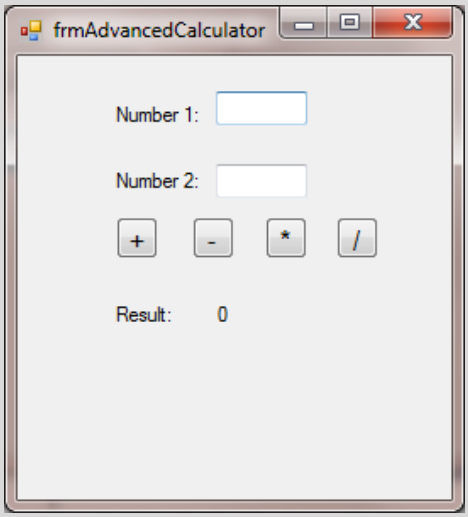


Figure 38: Math Class Properties and Methods

Example 2-7: Building an Advanced Calculator

1. Add a new form to the CourseExamples project and name it frmAdvancedCalculator.cs .
2. Copy all controls (labels, textboxes, and the button) on the new form.
3. Add three more buttons for subtracting, multiplying, and dividing operations. Name the buttons btnSubtract, btnMultiply, and btnDivide. Set the text property to the appropriate operator symbol.The form layout is shown on the right.



4. Select the first textbox control, navigate to the properties window, click on the lightning bolt, and double-click in the event Validating.
5. Enter the following code to perform some basic validation:

```
private void txtNumber1_Validating(object sender, CancelEventArgs e)
{
    double Number1 = new double();
        bool Parse = double.TryParse(txtNumber1.Text, out Number1);
        if (!Parse)
        {
            MessageBox.Show("Invalid number");
            e.Cancel = true;
        }
}
```

6. We are using the TryParse method to convert the entry of the textbox control into a valid number of type double. If it fails, the method returns a false. The If statement checks whether the boolean Parse is false (!Parse), and the issues a message box informing the user that the value is entered is an "Invalid Number". Then we also cancel the event so that the user cannot move forward.
7. Add a new button on the Main Menu form, name it btnAdvancedCalculator, and set its Text property to "Advanced Calculator".
8. Double-click on the button to create the default click event and add the following code:

```
private void btnAdvancedCalculator_Click(object sender, EventArgs e)
{
    frmAdvancedCalculator frm = new frmAdvancedCalculator();
    frm.Show();
}
```

9. Save the application and test it. Notice the validation for the first textbox.
10. But also notice, when the textbox is empty and you are trying to close the form, the validation takes place, the TryParse method returns a false, and the form does not close. We will fix this later. Enter a valid number and then close the form.

Example 2-7(continued): Building an Advanced Calculator

11.Create the click event for all calculator buttons and type the following code:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = (Convert.ToDouble(txtNumber1.Text) + Convert.ToDouble(txtNumber2.Text)).ToString();
}

private void btnSubtract_Click(object sender, EventArgs e)
{
    lblResult.Text = (Convert.ToDouble(txtNumber1.Text) - Convert.ToDouble(txtNumber2.Text)).ToString();
}

private void btnMultiply_Click(object sender, EventArgs e)
{
    lblResult.Text = (Convert.ToDouble(txtNumber1.Text) * Convert.ToDouble(txtNumber2.Text)).ToString();
}

private void btnDivide_Click(object sender, EventArgs e)
{
    lblResult.Text = (Convert.ToDouble(txtNumber1.Text) / Convert.ToDouble(txtNumber2.Text)).ToString();
}
```

12.Save the project and test it. Perform all operations.

13.The last step is to use a round function to round the result of the division operation. Modify the btnDivide_click event procedure as shown below:

```
private void btnDivide_Click(object sender, EventArgs e)
{
    double Result = new double();
    Result = Convert.ToDouble(txtNumber1.Text) / Convert.ToDouble(txtNumber2.Text);
    lblResult.Text = Math.Round(Result, 4, MidpointRounding.AwayFromZero).ToString();
    lblResult.Text += " (" + Result.ToString() + ")";
}
```

14.Save the project and test the division operation.

6. Date/Time Data Type

As you may recall, there is no Date/Time data type in the Common Type System in the .NET framework. C# though defines a Date/Time data type through a struct (=structure), which is a simplified class. As you can imagine, you must instantiate a Date/Time variable using the new keyword.

6.1 Declaring Date/Time Variables

To create date and time variables in C# you use the DateTime structure. A DateTime variable is a value-type variable like many others. This structure provides a variety of properties and methods for retrieving information about dates and times, formatting of DateTime values, and performing operations on dates and times.

A DateTime variable can be created in several ways as shown below:

DateTime *variable_name* = **new DateTime**(year,month,day[,hour,minute,second [,millisecond]])

DateTime *variable_name* = **DateTime.Parse(**string**)**

DateTime *variable_name* = **DateTime.TryParse(**string, **out** *DateTime_variable*)

DateTime *variable_name* = **DateTime.TryParseExact(**string, **string[]** formats, **out** *DateTime_variable*)

 **Note:** Creating DateTime variables is different from the way we create other built-in value types. The main reason is that C# does not provide a keyword for using this structure as it does for the other value types.

When you create a DateTime variable using the new keyword, you must at least specify the year, month, and day. The time component is then set to midnight (12:00 AM). You can optionally set the time value by specifying the hour, minute, and second part.

When you use the DateTime.Parse method, you must specify the date and/or time component in a string using a valid date/time format, such as 5/10/2010.

 **Note:** C# stores a date/time value as a number of ticks (1 tick = 100 nanaseconds) that have elapsed since January 1, 0001 midnight. Therefore, it is actually a long data type.

6.2 Date/Time Methods and Properties

As you can imagine, there are many properties and methods associated with the DateTime structure. As with any data type, there are static properties and methods that work on the DateTime struct such as get the current date and/or time. Then there are instance methods and properties that work on a specific instance of a DateTime struct, such as get the month of a given date. Table 6 shows some static properties.

Static Property	Description
MinValue	The minimum value of the DateTime structure
MaxValue	The maximum value of the DateTime structure
Now	The current date and time
Today	The current date
UtcNow	Gets a DateTime object that is set to the current date and time on this computer, expressed as the Universal Coordinated Time (UTC)

Table 6: Static Properties of DateTime Structure

Table 7 shows some static methods of the DateTime structure.

Static Method	Description
Compare(DateTime Date1,DateTime Date2)	Compares two instances of DateTime structures and returns an integer, indicating whether date1 is earlier than, the same as, or later as date2.
DaysInMonth(int Year, int Month)	Returns the number of days in the month and year specified.
IsLeapYear(int Year)	Returns an indication whether the specified year is a leap year.
Parse(string Datestring, Try Parse(string Datestring, out DateTime)	Returns a DateTime object based on date/time in a string format.
ParseExact, TryParseExact	Same as above, but in addition must match one more speficied date formats

Table 7: Static Methods of DateTime Structure

The instance properties and methods are shown in Figure 39. You first create a DateTime variable and then access the properties and methods to operate on this particular instance.

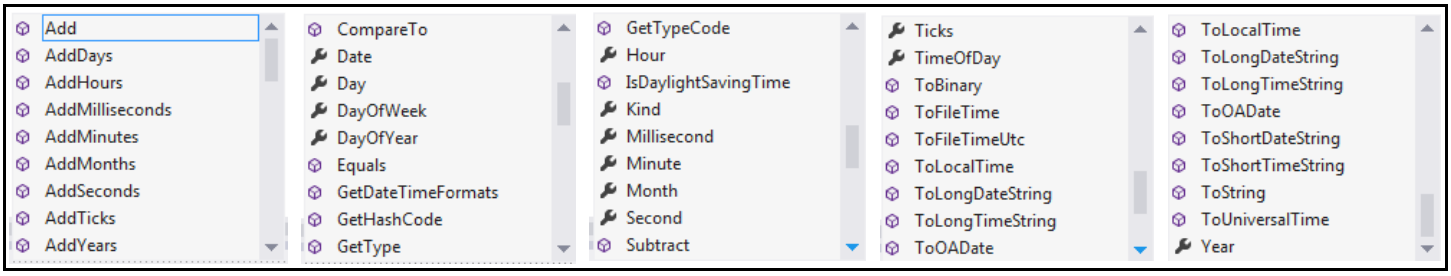


Figure 39: Instance Properties and Methods of DateTime Structure

The majority of these properties and methods can be categorized as shown in Figure 40.

Category	Instance Properties & Methods
Date/Time Arithmetic	Add, AddDays, AddHours, AddMilliseconds, AddMinutes, AddMonths, AddSeconds, AddTicks, AddYears, Subtract
Date/Time Information	Date, Day, DayOfWeek, DayOfYear, Hour, Millisecond, Minute, Month, Second, Ticks, TimeOfDay, Year, IsDaylightSavingTime, GetDateTimeFormats
Date/Time Formatting	ToLocalTime, ToLongDateString, ToLongTimeString, ToShortDateString, ToShortTimeString, ToString, ToUniversalTime

Figure 40: Categories of Instance Properties & Methods

Below are some examples to demonstrate the usage of the properties and methods of the DateTime structure.

Examples:

DateTime dtCurrentDateTime = **DateTime.Now**;

DateTime dtCurrentDate = **DateTime.Today**;

To get the current time, you have to use the TimeOfDay property on the dtCurrentDateTime variable. This property returns a TimeSpan value:

TimeSpan tsCurrentTime = dtCurrentDateTime.**TimeOfDay**;

Or to format this into a string, use the ToString() method:

dtCurrentDateTime.**TimeOfDate.ToString("t");** t formats the time into short time format

To retrieve the month, use the Month property:

intMonth = dtCurrentDate.**Month**;

The examples below show some date/time arithmetic.

Examples:

```
int daysMonth = DateTime.DaysInMonth(2010, 7);

DateTime dueDate = dtCurrentDate.AddDays(30);
```

For weekday functionality, use the enumeration DayOfWeek. An example is shown below:

Example:

The example shows an if logic where when the weekday is either Saturday or Sunday then perform some statements, otherwise if the weekday is a day where we have to work (Mon – Fri) then perform some other logic.

```
DayOfWeek dw = dtCurrentDate.DayOfWeek

if (dw == DayOfWeek.Saturday || DayOfWeek.Sunday)
{
        //do something
}
else
{
        //do something else
}
```

7. Character/String Data Type

A string can consist of any letters, numbers, and any special characters based on the character set used. The string data type is a special type since it is based on a class named String and therefore a reference type. We will explore this in more detail in the next sections.

Also, remember there is a Char value type that can store a single character.

7.1 String Basics

Variables declared as string are not a value-type but a reference-type variable. The main reason for that is that strings are of variable length, they are not predictable in terms of memory resources and therefore are placed on the heap memory. Therefore, a string variable is merely a pointer to the string object on the heap. Kind of weird, isn't it? On the other hand, you can use and work with strings as if they were value-type variables.

The other important fact is that strings are immutable, meaning once they are assigned a value, you cannot change the value anymore. This does not really have a practical impact as shown in Figure 41.

The above example will work and no error is generated. So for all intense purposes, we can simply change the value of a string variable. However, what goes on behind the scenes is important to know and understand, mainly for performance reasons.

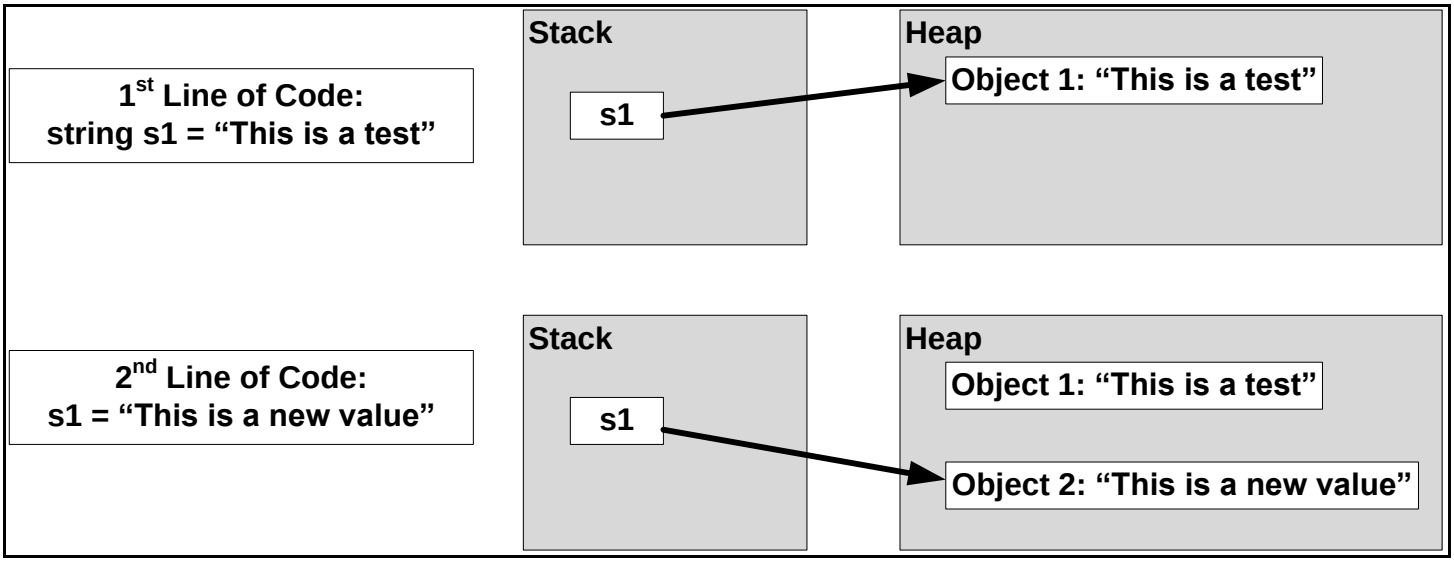


Figure 41: String Objects and Memory Resources

When you assign a new value to an existing string a new string object is actually created in memory. The old string object will also remain in memory, and since strings are objects, they are placed on the heap memory which you cannot really clean up using code.

So the first line contains a value-type variable s1 pointing to object on the heap containing "This is a test". In the second line, the value of s1 is changed and now points to a new object on the heap having a value of "This is a new value". The old object is still out on the heap until it is garbage collected since no more reference exists.

As a consequence, if you have to manipulate large strings, you should use the StringBuilder class for manipulating string for performance reasons as shown in a later section.

Since the string data type is based on the String class, you also get many methods. This is the good news. All these various string manipulation functions that are floating around in other programming languages are combined into the String class. This is a very powerful concept.

First of all, let us explore the string data type. To declare and initialize a string, use the same syntax as for all other value-type variables:

string *variable_name* = "value";

Warning: String values are always enclosed in double quotes, whereas Char values are enclosed in single quotes.

To join or concatenate strings together, you use the plus (+) sign as shown below:

Example:

string fname = "Bob";

string lname = "Smith";

string name = lname + ", " + fname;

7.2 Char Data Type

The other data type that is related to strings is the char data type. This is an actual value type since can contain only a single character. When assigning values into a char variable, you must single quotes as shown below:

char chrTest = ‘g’;

The char type is special for various reasons. First of all, the char type is considered a counting type. It is used to control loops for strings and extract one character at a time. We will discuss some examples later. And secondly, the char type holds besides the individual character also the ASCII character code. You can simply convert a char variable into an integer type without receiving an error as shown below:

int intI = (**int**)chrTest;

Some characters within a given character set are not printable. The tab key, the new line, carriage return are just a few examples. To use those special characters, you use the so-called escape sequence. C# uses the backslash for this purpose. Table 8 shows some of the non-printing characters and the C# equivalent:

C# special character	Description	C# special character	Description
'\n'	New line	'\r'	Carriage return
'\t'	Tab	'\\'	Backslash
'\0'	Null character	'\"'	Quotation

Table 8: Non-Printing Characters

The last two special characters are really not non-printing characters. But because of their usage as a delimiter for strings (") or escape sequence (\), they are needed if you want to include a double-quote of a backslash in your string without this special meaning.

The other option you have is to use a verbatim string literal. This is especially useful for strings with many backslashes, for example. You use the @ (at) sign at the beginning of the string, then the usual double-quote to begin your string. Within the string, you can now use a single backslash to show a backslash as shown below:

Example:

```
string strPath = "c:\\Temp\\WinApps\\";
string strPath = @"c:\Temp\WinApps\";
```

Note: The @ sign basically tells C# to ignore the backslash as an escape character.

7.3 String Class Properties & Methods

As mentioned earlier, the string data type is based on a class. Hence, there are many methods and properties built into this class that you can use the work with string objects. Many times you need to work with the individual characters within a string, to change the case, to concatenate strings, and so forth.

Table 9 shows some of the most commonly used indexers, properties, and methods of the string class.

Indexer	**Description**
[index]	Gets the character at the specified position.
Property	
Length	Gets the number of characters in a string.
Method	
StartsWith(string)	Returns a boolean indicating whether or not the string starts with the specified string.
EndsWith(string)	Returns a boolean indicating whether or not the string ends with the specified string
IndexOf(string[, startIndex])	Returns an integer representing the position of the first occurrence of the specified string starting at the specified position. If the starting position is not specified (optional), the search starts at the beginning of the string. If the string is not found, -1 is returned.
LastIndexOf(string [, startIndex])	Returns an integer representing the position of the last occurrence of the specified string starting at the specified position. If the starting position is not specified (optional), the search starts at the end of the string. If the string is not found, -1 is returned.
Insert(startIndex, string)	Returns a string with the specified string inserted beginning at the specified position.
PadLeft(totalWidth)	Returns a string that is right-aligned and padded on the left with spaces so that total width of the string equals the specified width.
PadRight(totalWidth)	Returns a string that is left-aligned and padded on the right with spaces so that total width of the string equals the specified width.
Remove(startIndex, count)	Returns a string with the specified number of characters removed starting at the specified position.
Replace(oldstring,newString)	Returns a string with all occurrences of the old string replaced with the new string.
Substring(startIndex [,length])	Returns the string that starts at the specified position and has the specified length. If length is not specified, all characters from specified position to the end are returned.
ToLower(), ToUpper()	Returns a string in lower/upper case.
Trim()	Returns a string with leading and trailing spaces removed.

Table 9: String Methods & Properties

The following examples simply demonstrate some of the methods and properties:

Examples:

Using indexer

string strTest = "abcd";
char chr1 = strTest[0]; (holding character a, the first character in strTest)
char chr2 = strTest[3]; (holding character d, the fourth character in strTest)

For Next loop showing each individual character in string strTest

```
for (int i=0; I < strTest.Length; i++)
        {
                MessageBox.Show(strTest[i]);
        }
```

StartsWith, EndWith methods

bool blnStart_ab = strTest.StartsWith("ab"); (returns true)
bool blnEnd_ab = strTest.EndsWith("ab"); (returns false)

Extracting last name from name:

string strName = "Robert Killian";
string strLastName = strName.Substring(strName.IndexOf(" ") + 1); (returns "Killian")

Code that copies one string to another string

string s1 = "test";
string s2 = s1; (copies the value from s1 into s2)
s2 = "New Test"; (this does not change the value stored in s1, s1 still equals "test")

7.4 StringBuilder Class

As mentioned earlier, strings are immutable, meaning they cannot be changed. If you change a value of an existing string, a new string object is created. For manipulating large strings, use the StringBuilder class. The StringBuilder class is part of the System.Text namespace. Either include this reference at the top of your class files (using System.Text) or use the fully qualified reference in your code.

Table 10 show some of the properties and methods of the StringBuilder class.

Indexer	**Description**
[index]	Gets the character at the specified position.
Property	
Length	Gets the number of characters in the string
Capacity	Gets or sets the number of characters the string can hold
Method	
Append(string)	Adds the specified string to the end of the string.
Insert(index, string)	Inserts the specified string at the specified index in the string.
Remove(startIndex, count)	Removes the specified number of characters from the string
Replace(oldString,newString)	Replaces all occurrences of the old string with the new string.
ToString()	Converts the StringBuilder object to a string.

Table 10: StringBuilder Class Properties & Methods

To use the StringBuilder class, you have to create an instance of this class. You can immediately assign it a value and/or capacity or do this at a later stage in your program. Once you are done manipulating the large string, then "convert" the final output of the StringBuilder using the ToString() method to assign back into a string variable.

Example:

```
StringBuilder sbSQL = new StringBuilder("SELECT LastName, FirstName ",500);
sbSQL.Append("FROM employees ");
sbSQL.Append("ORDER BY LastName;");

string strSQL;
strSQL = sbSQL.ToString();
```

7.5 Formatting of Number and Date/Time Values

When you format numbers or date/time values, the resulting output is represented in a string format (that is the reason why you find this discussion under the string section).

Many built-in formats exist, but you can also create your own custom formats. Table 11 shows the standard numeric formatting codes.

Code	Format	Description
C or c	Currency	Formats the number as currency with the specified number of decimal places.
P or p	Percent	Formats the number as percent with the specified number of decimal places.
N or n	Number	Formats the number with thousands separators and the specified number of decimal places.
F or f	Float	Formats the number as a decimal with the specified number of decimal places.
D or d	Digits	Formats an integer with the specified number of digits.
E or e	Exponential	Formats the number in scientific notation the specified number of decimal places.
G or g	General	Formats the number as a decimal or scientific notation depending on which is more compact.

Table 11: Standard Numeric Formatting Codes

You can either use the ToString() or the String.Format() method to format numbers as shown below:

Examples:

string strBonus = dblBonus.**ToString("c");** (dblBonus = 1525.75, strBonus = $1,525.75)

string strInterest = dblInterest.**ToString(**"P2"**)**; (dblInterest = 0.0725, strInterest = 7.25%)

string strInterest = **String.Format(**"{0:p2}", 0.0725**)**;

The String.Format() method allows you to create a string using multiple numbers embedded in the string. The number is the index of the numbers supplied in the format method as shown below:

Example:

string strAccount = **String.Format(**"Your interest rate is {0:p2} and the current balance is {1:c}",0.0725,3500**)**

The format displays as: "Your interest rate is 7.25% and the current balance is $3,500.00"

Date/Time formats are created in the same fashion. The standard Date/Time formatting codes are shown in Table 12.

Code	Format	Code	Format
d	Short date	f	Long date, short time
D	Long date	F	Long date, long time
t	Short time	g	Short date, short time
T	Long Time	G	Short date, long time

Table 12: Date/Time Formatting Codes

You can either use the ToString() or the String.Format() method to format date/time data as shown below:

Example:

```
DateTime dteNow = DateTime.Now;
MessageBox.Show(dteNow.ToString("d"));
MessageBox.Show(String.Format("{0:g}",dteNow));
```

Besides these standard or built-in formats you can create your custom formats using specific syntax symbols as shown in Table 13.

Code	Description	Code	Description
0	Zero placeholder	,	Thousands separator
#	Digit placeholder	%	Percentage placeholder
.	Decimal point	;	Section separator

Table 13: Custom Number Formatting Codes

Custom formats for numbers can be comprised of three parts, one for positive numbers, one for negative numbers, and one for zeros.

Example:

```
string strBalance = String.Format("{0:$#,##0.00;($#,##0.00);Zero}",-1267.88); ($1,267.88)
```

As you can imagine or may know from databases that Date/Time values can have a variety of custom formats. Table 14 shows the codes allowing for a nearly unlimited number of special formats for Date/Time formatting. Use those codes with the Format() or ToString() methods.

Code	Description	Code	Description	Code	Description
d	Day of the month without leading zero	y	Two-digit year without leading zero	m	Minutes with leading zero
dd	Day of the month with leading zero	yy	Two-digit year with leading zero	mm	Minutes without leading zeros
ddd	Abbreviated day name	yyyy	Four-digit year	s	Seconds without leading zeros
dddd	Full day name	/	Date separator	ss	Seconds with leading zeros
M	Month without leading zeros	h	Hour without leading zero	f	Fractions of seconds (one f for each decimal
MM	Month with leading zero	hh	Hour with leading zero	t	First character of AM/PM designator
MMM	Abbreviated month name	H	Hour on 24-hour format without leading	tt	Full AM/PM designator
MMMM	Full month name	HH	Hour on 24-hour format with leading	:	Time separator

Table 14: Date/Time Custom Formatting Codes

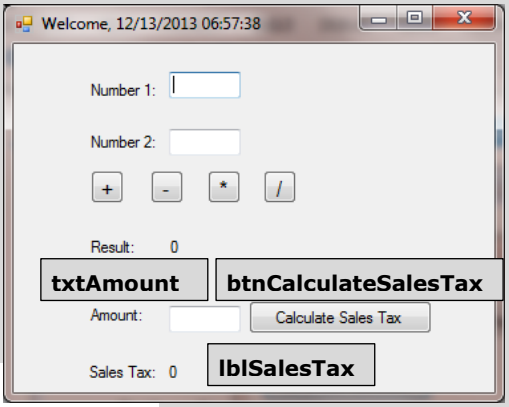
Warning: Note the capital M for formatting months and the lower case m for formatting minutes.

Example 2-8: Enhancing the Advanced Calculator Form

1. First of all, let us fix the issue of the validation from the previous example. Remember, when the Number 1 text box is empty, the validation method is triggered. We can now use the string property Length to verify that an entry exists in the textbox control and only then trigger the validation.
2. In the txtNumber1_Validating event, wrap an if structure around the validation code:

```
if (txtNumber1.Text.Length > 0)
{
    double Number1 = new double();
    bool Parse = double.TryParse(txtNumber1.Text, out Number1);
    if (!Parse)
    {
        MessageBox.Show("Invalid number");
        e.Cancel = true;
    }
}
```

3. Save and run the project. Test the validation to ensure that it still works properly. Then remove the value from the textbox control and close the form. The validation is not triggered anymore, the form actually closes now.
4. Now we want to also perform a sales tax calculation using this form, add the following controls below the current controls a shown on the right.
5. Set the Text properties as shown, and name the textbox, the button and the final label control as shown.
6. Double-click on the button to create the default click event and write the following code:




```
private void btnCalculateSalesTax_Click(object sender, EventArgs e)
{
    lblSalesTax.Text = (double.Parse(txtAmount.Text) * 0.0975).ToString("c");
}
```

7. The last thing we want to do is display the current date and time in the windows form's caption. We are going to create the form's load event by double-clicking on the form itself (not on any control!). This will create the form's load event, write the following code:

```
private void frmAdvancedCalculator_Load(object sender, EventArgs e)
{
    this.Text = "Welcome, " + DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss");
}
```

8. Save and run the project. Test the sales tax calculation and notice the current date and time in the window's title bar.

CLASS 3

8. Decision Structures

In general, a program or a sub unit is executed from top to bottom. There are many situations where you need to break this flow, in fact, almost always you need to direct the flow of your program. In this chapter and the following ones, we introduce the common programming constructs in C# to process the statements in the order necessary.

8.1 Program Control Flow Concepts

C# is a structured programming language. Structured programming essentially means that the programming language and its control flow structures determine the flow of the program. Structured programming does not use any Goto statements to force the flow of the program (= unstructured programming).

Although C# actually contains the Goto command, there is almost no reason to use it. In fact, C# (or better yet the creators of C#) has been criticized for the inclusion of the Goto statement. There is one possible situation where you may want to use a Goto statement, but in general, just stay away from it.

There are a couple of different programming constructs that affect program flow. One of them is the decision or conditional structure. The most common member is the If statement. It checks a condition and based on whether the condition is true or false, it directs the flow of the program.

Another important member is the iteration construct. This construct allows you to repeat blocks of code until a terminating condition has been met.

And lastly, there is another, and that is error or exception handling. You might ask, what does this have to do with control flow statements? The core idea is that you break your code into your execution section (this is where your code goes) and the exception section. If an error occurs in your execution section, then control is directed into the exception section where the error is handled in a certain way.

We will discuss these main program flow constructs in the following chapters in more detail.

8.2 Boolean Expressions

Control flow structures use conditional statements that either evaluates to true or false. An expression that evaluates to true or false is called a Boolean Expression. Boolean Expressions are used within control flow structures frequently to manage the flow of the program.

Bool Data Type

We have already introduced the main data types, such as number, string, and date and time in the previous chapters. The bool data type is another one that is useful when dealing with Boolean expressions.

To declare a Boolean variable, use the following syntax:

```
bool Invoiced = false;
```

Note: Use the C# keyword bool as opposed to the .NET underlying type of Boolean for Boolean variables.

A Boolean variable can only hold two values, true or false. You can also extend this data type to hold null values, if necessary. Suffix the type declaration with a question (?) mark.

```
bool? Invoiced = null;
```

Note: Using nulls in Boolean variables is particularly importing when reading data from a database since database columns can contain null values.

Using a Boolean variable is useful when the expression or expressions that evaluate to either true of false are lengthy. Rather than embedding the lengthy expression in the control flow structures (If, Switch, Loop, etc.) you assign the result to a boolean variable.

Then you use the Boolean variable in the control flow statements, such as an If statement. This is recommended practice and improves readability of your program.

Relational Operators

A relational expression is comprised of two operands and one operator. An operand can be an expression, such as a variable, a literal, an arithmetic expression, or a C# keyword such as null, true or false. A relational operator compares these two operands, the one to the left of the relational operator and the one to the right.

Table 15 shows the relational operators in C#.

Operator	Name	Description
==	Equality	Returns true if both operands are equal.
!=	Inequality	Returns true the operands are not equal.
>	Greater than	Returns true if the left operand is greater than the right operand.
<	Less than	Returns true if the left operand is less than the right operand.
>=	Greater than or equal	Returns true if the left operand is greater than or equal to the right operand.
<=	Less than or equal	Returns true if the left operand is less than or equal to the right operand.

Table 15: C# Relational Operators

 **Note:** C# uses the double equal sign for comparisons. This is similar to other programming languages. The single equal is the assignment operator, which assigns the right operand into the left operand. Some other languages use the same operator for assignment and comparison (VB, for example).

If a variable or an expression already evaluates to true or false, then there is no need to compare it to a Boolean value.

Example:

bool *invoiced*;
invoiced = **false**;

If (*invoiced* == **true**) is the same as **If** (*invoiced*) ….

 **Note:** When comparing numeric values, you usually compare values of the same data type. If you do not, then C# will cast the less precise data type to the more precise data type.

Logical Operators

Logical operators are used to connect multiple Boolean expressions together. Table 16 shows the logical operators that are defined in C#.

Operator	Name	Description
&&	Conditional And	Returns true if both expressions are true. This operator only evaluates the second expression if necessary.
||	Conditional Or	Returns true if either expression is true. This operator only evaluates the second if necessary.
&	And	Returns true if both expressions are true. Both expressions are evaluated.
|	Or	Returns true if either expression is true. Both expressions are evaluated.
!	Not	Reverses the value of the expression.

Table 16: C# Logical Operators

The first two operators are also referred to as short-circuit operators. They are more efficient in that they only evaluate the second Boolean expression if necessary. In the example below, you see that the first Boolean expression already evaluates to false, and since the logical operation is an And (&), there is no need to check the second expression since the entire expression will already be false.

Example:

int *Max* = 3;

Max **>=** 5 **&&** *Max* **<=** 10

Try to not use the Not (!) operator. This is simply for readability. The example below shows a situation where you can change an expression to avoid the Not(!) operator.

Example:

!(*sum* **<** 5000) can be rewritten as *sum* **>=** 5000

On the other hand, there may be situations where you want to evaluate the second expression, especially if you perform some kind of operations, such as incrementing a counter variable.

Example:

Active **== true &** *Loop***++ <** *Max*

In this example we are incrementing the counter variable Loop, by skipping this part of the expression we would not increment the counter variable.

If you carefully design expressions that use the conditional logical operators, you can boost the performance of your code by avoiding unnecessary work. Place simple Boolean expressions that can be evaluated easily on the left side of a conditional logical operator, and put more complex expressions on the right side. In many cases, you will find that the program does not need to evaluate the more complex expressions.

Order of Precedence

The order of precedence is important when you have more than two Boolean expressions. To change the order of precedence, use parentheses to group expressions in order to specify your customized order of precedence.

Table 17 summarizes the precedence and associativity of all the operators we have learned about so far. Operators in the same category have the same precedence. The operators in categories higher up in the table take precedence over operators in categories lower down.

Category	Operators	Description	Associativity
Primary	( ) ++ --	Precedence override Post-increment Post-decrement	Left
Unary	! + - ++ --	Logical NOT Addition Subtraction Pre-increment Pre-decrement	Left
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left
Additive	+ -	Addition Subtraction	Left
Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left
Equality	== !=	Equal to Not equal to	Left
AND	&& &	Logical Conditional AND Logical AND	Left
OR	|| |	Logical Conditional OR Logical OR	Left
Assignment	=		Right

Table 17: Operator Order of Precedence

8.3 If Statement

Shown below is the basic syntax of an If structure in C#. The Boolean expression must be enclosed in parentheses, each If block is enclosed in curly braces.

```
if (Boolean expression)
        {
                statements
        }
else if (Boolean expression)
        {
                statements
        }
else
        {
                statements
        }
```

 **Note:** The If statement in C# does not have an End If keyword. The curly braces enclosing the last If block are the substitute for the End If keyword.

Only one If block is processed, even though more than one Boolean expression evaluates to true. The statements in the first If block are executed where the Boolean expression evaluates to true. Any subsequent If blocks are skipped regardless of its Boolean value. If no condition evaluates to true, then the statements in the else block are executed. If there is no else block, then no statements are executed at all.

It is good coding practice to enclose each If block in curly braces. If you only have one statement, then no curly braces are needed. However, the majority of C# programmers still include the curly braces.

 **Warning:** Curly braces define a block of code. Declaring any variables within such a block are in scope only within this block of code. This is referred to as block scope.

Within each block of code in an If statement you can write additional If statements. This is known as nested If statements. It is good practice to indent the nested If statements for readability.

Example 3-1: Using If statement

1. Navigate to the form frmAdvancedCalculator.cs and add the following method.

```
private string performCalculation(string Number1, string Number2, string Operator)
{
    string ReturnResult = string.Empty;
    double Result = 0;
    if ("+-*/".IndexOf(Operator) > -1)
    {
        if (Operator == "+")
        {
            Result = Convert.ToDouble(Number1) + Convert.ToDouble(Number2);
        }
        else if (Operator == "-")
        {
            Result = Convert.ToDouble(Number1) - Convert.ToDouble(Number2);
        }
        else if (Operator == "*")
        {
            Result = Convert.ToDouble(Number1) * Convert.ToDouble(Number2);
        }
        else if (Operator == "/")
        {
            Result = Convert.ToDouble(Number1) / Convert.ToDouble(Number2);
        }
        ReturnResult = (Math.Round(Result,2,MidpointRounding.AwayFromZero)).ToString("#,##0.00")
            + " (" + Result.ToString() + ")";
    }
    else
    {
        ReturnResult = "Invalid Operator";
    }

    return ReturnResult;
}
```

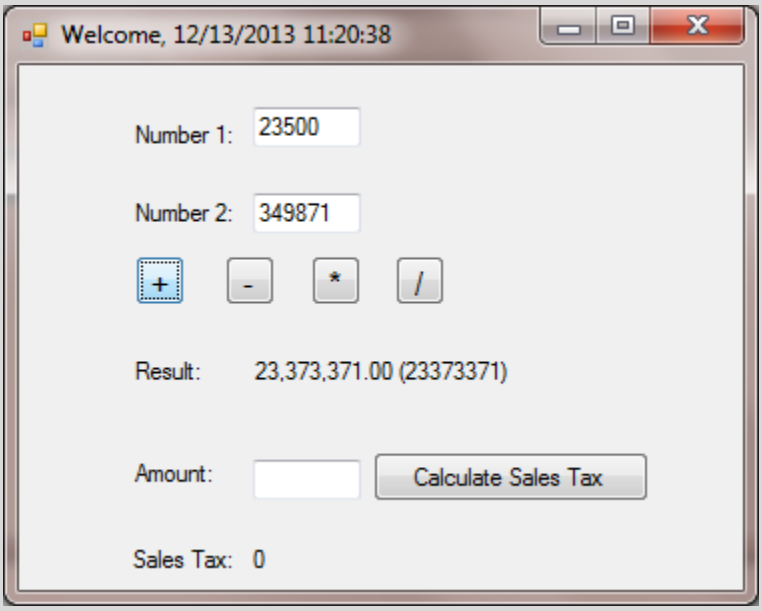
2. Let us take a close look at this code example. The string expression containing all four operators in one string uses the IndexOf method to find the position (or index) of the passed Operator parameter in the string expression "+-*/". For example, if the Operator parameter contains "+", then the index would be 0. If you pass in an invalid operator, then the IndexOf method returns a -1, meaning the substring was not found in the string. This ensures that we only operate using valid arithmetic operators.
3. If the operator is invalid, then we set the ReturnResult string to "Invalid Operator"
4. If the operator is valid, then we perform the arithmetic operation, use a rounding method from the static math class, display the raw result in parentheses, and format the final result as a number with two decimal digits.

Example 3-1(continued): Using If statement

5. Now call this new method from each of the operator button's event handler.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = performCalculation(txtNumber1.Text,txtNumber2.Text,"+");
}
private void btnSubtract_Click(object sender, EventArgs e)
{
    lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "-");
}
private void btnMultiply_Click(object sender, EventArgs e)
{
    lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "*");
}
private void btnDivide_Click(object sender, EventArgs e)
{
    lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
}
```

6. Save and test the project.
7. A sample output is shown below:



8.4 Switch Statement

The switch statement is C#'s implementation of the Case structure used in other languages. This kind of structure can be used instead of an If Else If statement where you have to match various different conditions. The syntax of a switch statement is shown below:

```
switch (expression)
{
        case value1:
                statements
                break;
        case value2:
                statements
                break;
        default:
                statements
                break;
}
```

The expression inside the parentheses of the switch keyword must use the built-in data types, such as int or string. You cannot use more complex user-defined types. Unlike Visual Basic, in C# the condition (values) must be a constant value as shown in Figure 42.

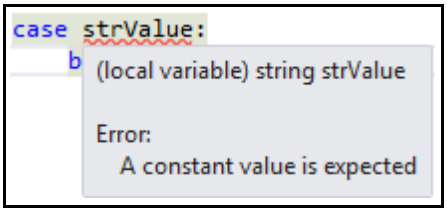


Figure 42: Error Message when using Variable in Switch Statement

The expression at the top of the switch structure is the left side of a Boolean expression. The value next to the case keyword is the right side of a Boolean expression.

The break statement is required for each label within a switch statement because C# does not allow to process multiple case blocks. Note that there are no curly braces for each case block like in the If statement.

Warning: The only relational operator allowed in a switch statement is the equal (=) operator and it is implied. The expression in the switch statement is compared to each of the case values based on equality.

The switch statement in C# is not as flexible as other similar structures in other languages. The only relational operator is the equal operator, and it is not permissible to compare the expression to a range of values. However, you can implement a workaround to compare to multiple values.

To code a comparison against multiple values you use the case label with no other statements and the break statement. This configuration allows C# to fall through this label and process another case block. This structure is shown below.

```
switch (expression)
{
        case value1:
        case value2:
        case value3:
                statements
                break;
        case value4:
                statements
                break;
        default:
                statements
                break;
}
```


Warning: The expression must evaluate to a string, char, long, sbyte, short, ushort, int, uint, or long data type.

In the example shown above, the switch expression is compared to value1, value2, and value3 and the statements in the case block 3 are executed for all three different values.


Note: A break statement is required in the default block even though there is no other block following. This is due to the fact that the default statement does not have to be at the end of the switch expression, although this is good practice.

Within each case block, you can include If statements or nested switch statements.

Example 3-2: Using Switch statement

1. Navigate to the form frmAdvancedCalculator.cs and replace the performCalculation method with the following code.

```
private string performCalculation(string Number1, string Number2, string Operator)
{
    string ReturnResult = string.Empty;
    double Result = 0;
    if ("+-*/".IndexOf(Operator) > -1)
    {
        switch (Operator)
        {
            case "+":
                Result = Convert.ToDouble(Number1) + Convert.ToDouble(Number2);
                break;
            case "-":
                Result = Convert.ToDouble(Number1) - Convert.ToDouble(Number2);
                break;
            case "*":
                Result = Convert.ToDouble(Number1) * Convert.ToDouble(Number2);
                break;
            case "/":
                Result = Convert.ToDouble(Number1) / Convert.ToDouble(Number2);
                break;
        }
        ReturnResult = (Math.Round(Result, 2, MidpointRounding.AwayFromZero)).ToString("#,##0.00")
            + " (" + Result.ToString() + ")";
    }
    else
    {
        ReturnResult = "Invalid Operator";
    }

    return ReturnResult;
}
```

2. Save the project and run it to test that it works exactly in the same way.

Note: Instead of a break statement within each case you could also use a return statement and return the final result directly from the case statement line. However, it is good practice to have one return statement in methods returning something back to the calling program.

9. Iteration Structures

The C# implementation of iteration structures results in four different loop statements.

- While loop
- Do/While loop
- For loop
- Foreach/ in loop

The While loop is probably the most common one. A variation exists in the Do While loop. And like many other programming languages, C# provides a For loop. The Foreach/in loop allows you to iterate through collection objects.

9.1 While Loop

The while looping construct is useful in case you want to execute a block of statements until some terminating condition has been reached.

The syntax of the while loop is shown below:

```
while (Boolean expression)
{
        statements
}
```

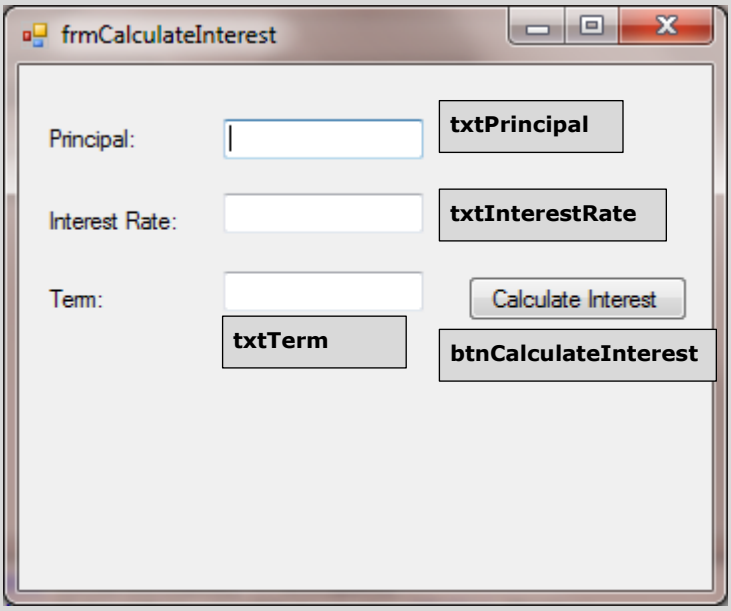
As you can see, at the top of the loop is a Boolean expression. While the expression evaluates to true, the statements inside the loop are executed. Once the Boolean expression evaluates to false, the iteration stops and control is directed to after the ending curly brace. If the Boolean expression is false from the very beginning, the statements are never executed and the enclosing block is basically skipped.

As shown already with the If statement, there is no ending or loop statement at the end. The curly braces define the block of the loop. Any variables declared inside the curly braces are of block scope and cannot be used outside the loop.

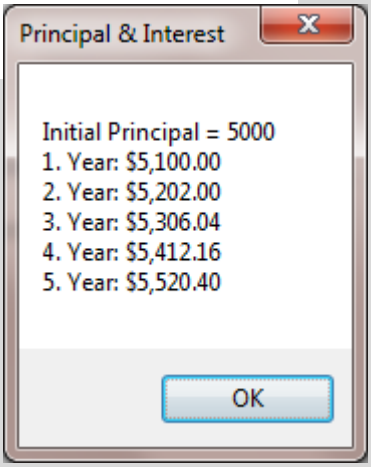
Warning: Within the scope of a while loop, you will need to ensure that a terminating event is established; otherwise, you will be stuck in an endless loop.

Example 3-3: Using While Loop statement

1. Add a new form to the project CourseExamples and name it frmCalculateInterest.cs.
2. Place three label and textbox controls and one button on the form as shown to the right.
3. Set the Text properties and name the textbox controls and the button as shown on the right.
4. Add a button on the MainMenu form, name it btnPrincipalInterest and use the usual two lines of code to show the form.
5. Double-click on the button btnCalculateInterest to create the default click event.
6. Write the following code to calculate the interest every year and add it to the principal. Output the compounded principal for every year in a messagebox.




```
private void btnCalculateInterest_Click(object sender, EventArgs e)
{
    int Year = 1, Term;
    double InterestRate, Principal;
    string OutputPrincipalInterest= string.Empty;
    InterestRate = double.Parse(txtInterestRate.Text) / 100.00;
    Term = int.Parse(txtTerm.Text);
    Principal = double.Parse(txtPrincipal.Text);
    while (Year <= Term)
    {
        //One line IF statement
        if (Year == 1) OutputPrincipalInterest = "Initial Principal = " + txtPrincipal.Text + "\n";
        Principal = Math.Round((Principal + (Principal * InterestRate)) * Math.Sign(Term), 2);
        OutputPrincipalInterest += Year.ToString() + ". Year: " + Principal.ToString("c") + "\n";
        Year++;
    }
    MessageBox.Show(OutputPrincipalInterest, "Principal & Interest");
}
```



7. Save the project and run it to test.
8. Enter the following values:

   Principal = 5000

   Interest = 2

   Term = 5

9.2 Do/While Loop

The Do While loop is a variation of the While loop. The While loop evaluations a Boolean expression at the top, and if the expression initially evaluates to false, then no statements are executed inside the block of this loop.

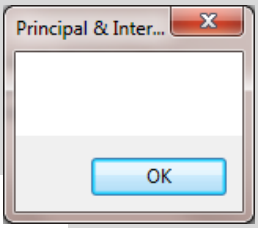
The Do While loop, on the other hand evaluates a Boolean expression at the bottom of the structure, that means the code block is at least executed once regardless of the outcome of the Boolean expression. The syntax is shown below:

```
do
{
        statements
}
while (Boolean expression);
```

Warning: You must place a semicolon at the end of the last statement since this is now outside the curly braces and functions like a regular C# code statement.

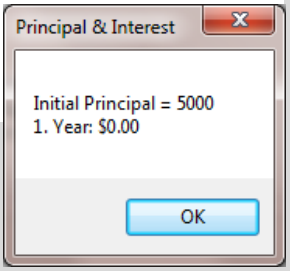
Most of the time you can use both loop structures to accomplish the same task.

Example 3-4: Using Do/While Loop statement



1. Run the previous example using the same values except use 0 for the term.
2. Replace the While loop from the previous example with a Do/While loop.

```
private void btnCalculateInterest_Click(object sender, EventArgs e)
{
    int Year = 1, Term;
    double InterestRate, Principal;
    string OutputPrincipalInterest= string.Empty;
    InterestRate = double.Parse(txtInterestRate.Text) / 100.00;
    Term = int.Parse(txtTerm.Text);
    Principal = double.Parse(txtPrincipal.Text);
    do
    {
        //One line IF statement
        if (Year == 1) OutputPrincipalInterest = "Initial Principal = " + txtPrincipal.Text + "\n";
        Principal = Math.Round((Principal + (Principal * InterestRate)) * Math.Sign(Term), 2);
        OutputPrincipalInterest += Year.ToString() + ". Year: " + Principal.ToString("c") + "\n";
        Year++;
    }
    while (Year <= Term);
    MessageBox.Show(OutputPrincipalInterest, "Principal & Interest");
}
```



3. Save and run the project.
4. Now repeat step 1 and notice the different output in the messagebox.

9.3 For Loop

For loops are useful when you know in advance how many times you have to loop. Furthermore, For loops are more efficient since the counter variable is built into this structure. The syntax is shown below:

```
for(initializing expression, ending expression, increment expression)
{
        statements
}
```

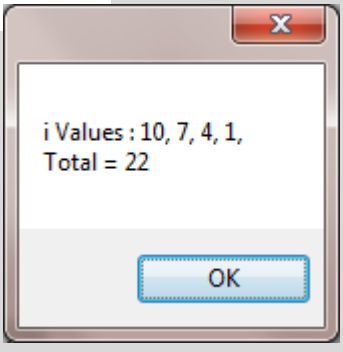
The initializing expressions handles two things: It declares a counter variable and sets its starting value. The ending expression determines when the loop is finished. The increment expression sets the increment of the counter variable. The increment expression can also be a decrement expression if the initializing expression is set up accordingly.

Example 3-5: Using For Loop statement

1. Add another button on the for frmCalculateInterest. Set its Text property to "For Loop Example" and name it btnForLoop.
2. Double-click on this button to create the default click event and write the following code.

```
private void btnForLoop_Click(object sender, EventArgs e)
{
    int Total = 0;
    string IValue = string.Empty;
    for (int i=10;i > 0; i -=3)
    {
        IValue += i.ToString() + ", ";
        Total += i;
    }
    MessageBox.Show("i Values : "  + IValue + "\n" + "Total = " + Total.ToString());
}
```

3. We are looping from an initial value of 10 and decrementing the counter variable i by 3 in every loop. The terminating condition takes place when i is less than zero.
4. Save the project and run it. Note the output in the message box.



In general, you would probably iterate upwards and increment your counter by 1. The for statement would look like this:

for (int i = 0; i < 10, i++) (This loop iterates 10 times)

9.4 Foreach/ In Loop

The C# foreach keyword allows you to iterate over all items within an array or a collection object without the need to test for an upper limit. The array or collection object knows how many items it contains and therefore terminates automatically. The syntax is shown below:

```
foreach (type variable_name in collection/array)
{
        statements
}
```

The foreach statement declares an iteration variable that automatically acquires the value of each element in the array or collection object .The type of this variable must match the type of the elements in the array/collection. The foreach statement is the preferred way to iterate through an array/collection; it expresses the intention of the code directly, and all of the for loop framework goes away.

Some important points to note here:

- The variable used to hold the individual elements of an array/collection in each iteration is read only. You cannot change the elements in the array/collection. This means that foreach will only allow you to iterate through the array or collection and not to change the contents of it. If you wish to perform some work on the array to change the individual elements, you should use a for loop.
- foreach can be used to iterate through arrays or collections. By a collection, we mean any class, struct or interface that implements the IEnumerable interface. (We will cover this later in this course)
- The string class is also a collection of characters (implements IEnumerable interface and returns char value in Current property).

Example:

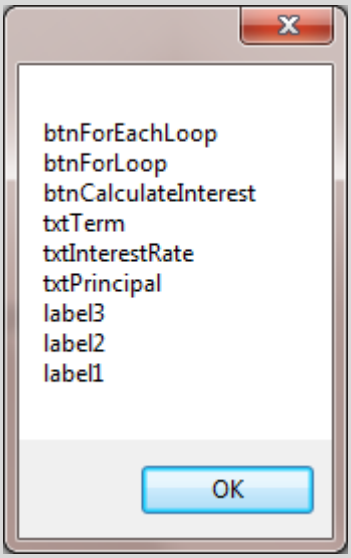
```
string FullName = string.Empty;
string Output;
int I = 0;
foreach(char Character in FullName)
{
        I++
        Output = I.ToString() + ". Letter: " + Character;
}
```

Example 3-6: Using Foreach Loop statement

1. Add another button on the for frmCalculateInterest. Set its Text property to "Foreach Loop Example" and name it btnForEachLoop.
2. Double-click on this button to create the default click event and write the following code.

```
private void btnForEachLoop_Click(object sender, EventArgs e)
{
    string OutputControls = string.Empty;
    foreach (Control ctl in this.Controls)
    {
        OutputControls += ctl.Name + "\n";
    }
    MessageBox.Show(OutputControls);
}
```

3. We are looping through the controls collection of the current form. Within each loop, we are collecting the name of the control and assign it into a string variable followed by a line break.
4. Save the project and run it. Note the output in the message box.



9.5 Break and Continue Statements

In most cases, you want your loops to execute "naturally", that means let the Boolean expression determine when the loop is finished. However, there are situations where you need conditional logic inside your loop and jump out of the loop, either jump to the end of the loop or to the beginning.

The break statement is designed for basically exiting a loop and transfer control of the program to after the loop. The continue statement is used for transferring the control of the program to the beginning of the loop.

The break statement is used mostly to avoid any unnecessary loops. If you have to loop 10,000 times, but after the third loop you have reached the goal, then you should exit the loop and not loop 9,997 times unnecessarily.

The continue statement allows you to stop the current iteration and start at the top of the loop again based on a condition.

Examples:

```
int I = 1;
while (I <= 10000)
{
        statements
        if (I > 3)
        {
                break;
        }
        I++;
}
```

```
int[] Total = {23,22,56,34,88};
for (int I = 0; I < 5; I++)
{
        if (Total[I] < 50)
                continue;
        MessageBox.Show("You passed with a score of " + Total[I].ToString());
}
```

In the above example, we have declared an array of 5 values. Inside the loop, if the score is less than 50, control is transferred back to the top, therefore the messagebox statement is not issued. Only if the score is greater than or equal to 50, the messagebox statement is executed.

10. Managing Errors and Exceptions

No matter how well your code is written, errors will occur. There are three types of errors:

- Syntax error (code does not compile, malformed statements)
- Run-time error (code compiles, error occurs while running the application)
- Logic error (code runs, but application logic is flawed, difficult to detect)

Syntax errors are fairly easy to detect, as the C# compiler reveals lots of information about the error. You must fix the problem before you can even run the application.

Run-time errors are obvious, once they occur. But you have to create the conditions for a run-time error to occur. This is where testing comes in, developer's own unit and integration testing, independent testing by another developer, and user acceptance testing.

Logic errors, the so-called bugs, are the hardest one to detect and fix. Mostly, your users will detect them during the lifetime of the application.

The only type of error we want to address here is the run-time error.

You, as a software developer, want to minimize the probability of errors being thrown. As a good programmer, you learn how to anticipate and handle errors. No matter how good of a programmer you are, you want to execute some sort of test plan to ensure that the code is ready for production. Thorough testing will reveal further possible run-time errors that lead to better and more bullet-proof code.

Whenever an error occurs in your code, it is said to throw an exception.

10.1 Overview of .NET Exceptions

Prior to .NET, error handling under the Windows operating system was a confused mishmash of techniques. Many programmers rolled their own error-handling logic within the context of a given application.

The obvious problem with these older techniques is the tremendous lack of symmetry. Each approach is more or less tailored to a given technology, a given language, and perhaps even a given project. To put an end to this madness, the .NET platform provides a standard technique to send and trap runtime errors: structured exception handling (SEH).

The beauty of this approach is that developers now have a unified approach to error handling, which is common to all languages targeting the .NET platform. Therefore, the way in which a C# programmer handles errors is syntactically similar to that of a VB programmer, or a C++ programmer using C++/CLI.

As an added bonus, the syntax used to throw and catch exceptions across assemblies and machine boundaries is identical. For example, if you use C# to build a Windows Communication Foundation (WCF) service, you can throw a SOAP fault to a remote caller, using the same keywords that allow you to throw an exception between methods in the same application.

Another bonus of .NET exceptions is that rather than receiving a cryptic numerical value that simply identifies the problem at hand, exceptions are objects that contain a human-readable description of the problem, as well as a detailed snapshot of the call stack that triggered the exception in the first place.

Furthermore, you are able to give the end user help-link information that points the user to a URL that provides details about the error, as well as custom programmer-defined data.

Programming with structured exception handling involves the use of four interrelated entities:

- A class type that represents the details of the exception
- A member that throws an instance of the exception class to the caller under the correct circumstances
- A block of code on the caller's side that invokes the exception-prone member
- A block of code on the caller's side that will process (or catch) the exception should it occur

The C# programming language offers four keywords (try, catch, throw, and finally) that allow you to throw and handle exceptions. The object that represents the problem at hand is a class extending System.Exception(or a descendent thereof).

When an unhandled exception occurs, you will see the exception dialog box shown in Figure 43.

You can view the exception settings directly from this dialog box by clicking on the link "Open Exception Settings".

To throw this kind of exception, using the Advanced Calculator form, enter an alphabetic character in the Number2 textbox and perform any arithmetic operation.

You have in general two options, to break and inspect the offending line of code, or to continue which normally will result in a termination of your application.

The goal of exception handling is to prevent any unhandled exception to bubble up to the end user, or in other words, to occur.

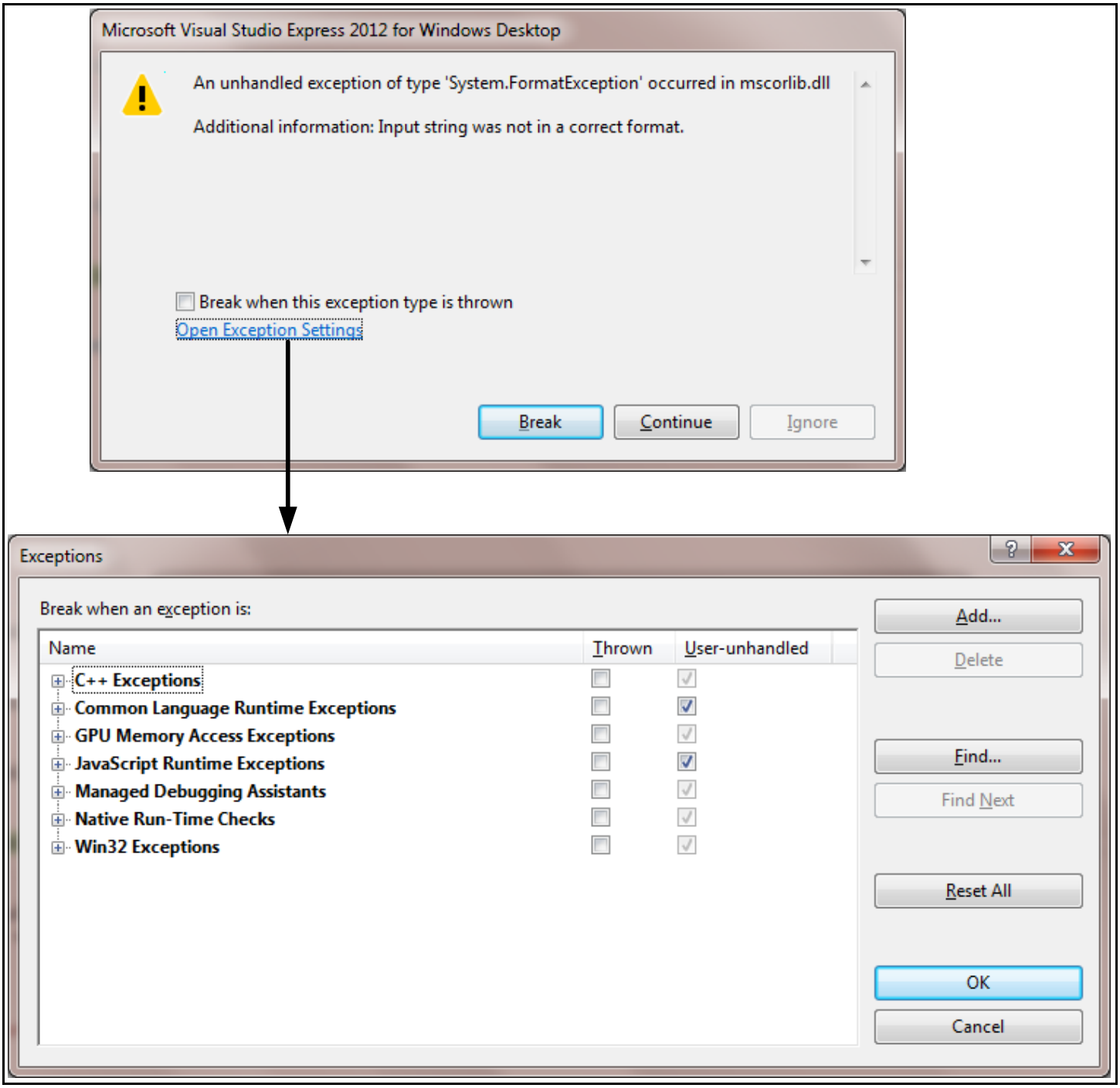


Figure 43: Unhandled Exception Dialog Box

All user- and system-defined exceptions ultimately derive from the System.Exception base class, which in turn derives from System.Object. Some important properties of this base class are shown in Table 18.

System.Exception Property	Description
Data	This read-only property retrieves a collection of key/value pairs (represented by an object implementing IDictionary) that provide additional, programmer-defined information about the exception. By default, this collection is empty.
HelpLink	This property gets or sets a URL to a help file or web site describing the error in full detail.
InnerException	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur. The previous exception(s) are recorded bypassing them into the constructor of the most current exception.
Message	This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter.
Source	This property gets or sets the name of the assembly, or the object, that threw the current exception.
StackTrace	This read-only property contains a string that identifies the sequence of calls that triggered the exception. As you might guess, this property is very useful during debugging or if you wish to dump the error to an external error log
TargetSite	This read-only property returns a MethodBase object, which describes numerous details about the method that threw the exception (invoking ToString()will identify the method by name).

Table 18: Select Properties of System.Exception

All exceptions are subclasses of the main Exception class. As you can see in Figure 44, there are two main exception classes, the System exception and the Application exception classes.

Under the application exception, you can create your own business rules exceptions. The System exception class is further subdivided into a whole hierarchy of subclasses.

Shown in Figure 44 is one example, the ArithmeticException class, and two subclasses, the Overflow and DivideByZero exception classes.

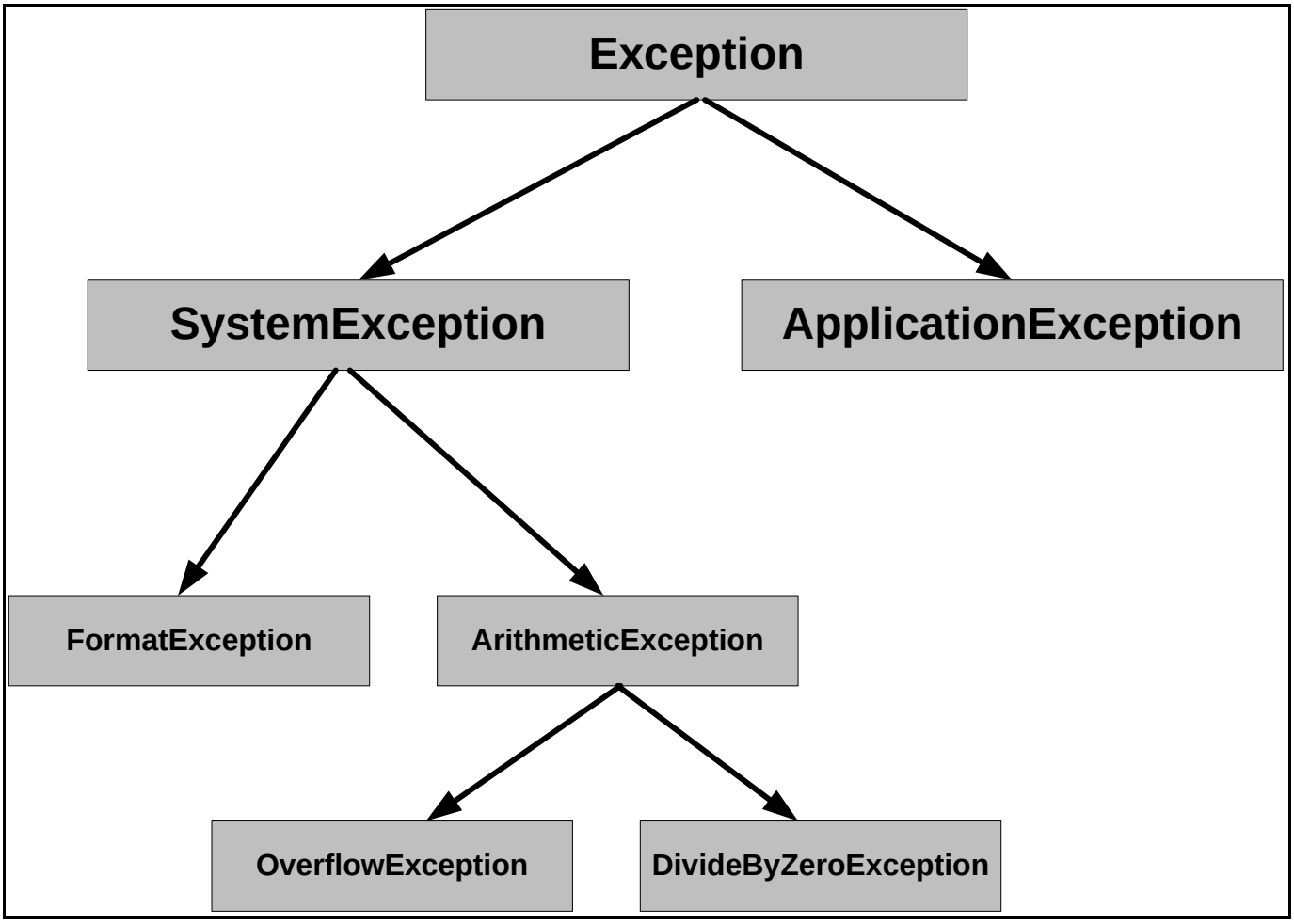


Figure 44: .NET Exception Hierarchy

10.2 .NET Structured Exception Handling (SEH)

To implement exception handling in your code, you have to use the exception structure as shown in Figure 45.

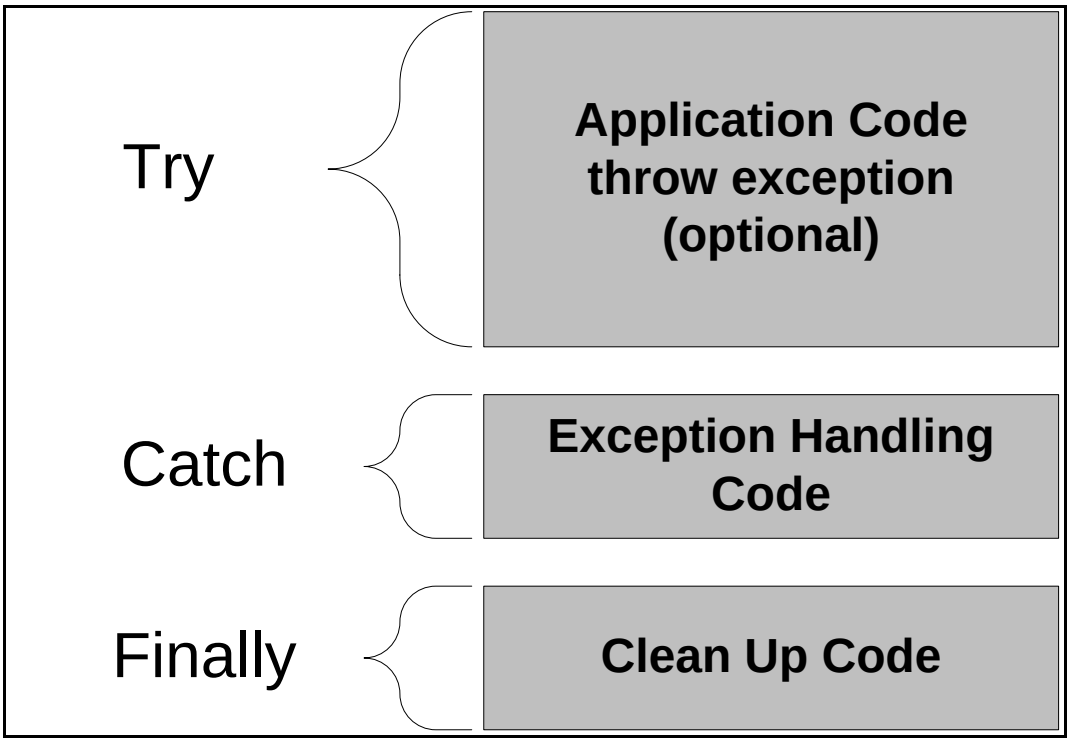


Figure 45: Structured Exception Handling

The C# programming language offers four keywords that allow you to throw and handle exceptions:

- try
    - throw
- catch
- finally

There are three blocks, two of them are required, the try and the catch block. The finally block is optional. First of all, you wrap your code inside the try block. Any error that occurs in this block will transfer control of the program into the catch block. Here you would handle the exception.

The finally block will be processed regardless of whether an exception occurred or not. This is useful to perform certain clean up actions before exiting the method.

Within the try block, you can also throw your own custom exception, based on business rules, for example. When you throw a custom exception, control will be also directed into the Catch block.

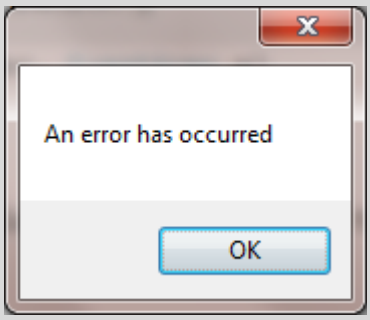
One important aspect of exception handling is that once control is directed into the catch block, you cannot go back to the try block. Again, once an exception is raised, the remaining code in the try block will not be executed. This is in contrast to other programming languages, where you can use a resume or resume next statement to continue the code in the try block.

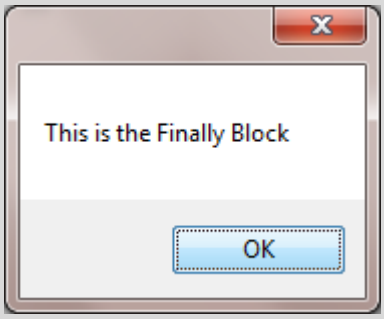
Example 3-7: Using Catch block

1. Navigate to the form frmAdvancedCalculator and inside the event procedure btnDivide_Click, add a try and catch block as shown below.

```
private void btnDivide_Click(object sender, EventArgs e)
{
    try
    {
        lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
    }
    catch
    {
        MessageBox.Show("An error has occurred");
    }
    finally
    {
        MessageBox.Show("This is the Finally Block");
    }
}
```

2. Save and run the project.
3. First, enter a zero into the Number2 text box control. Notice that no error occurred, in fact, the method returned an Infinity message. Kind of interesting, but the .NET framework takes care of this kind of error automatically
4. Now enter an alphabetic character into the Number2 textbox control. Now notice that the structured error handling was executed.
5. After dismissing the first messagebox, you should now see the finally block messagebox dialog box.





10.3 System-Level Exceptions (System.SystemException)

To display more specific information about the error, you need to create an exception variable in the catch block. After the keyword catch type a reference type variable declaration of type Exception and assign it a name. The syntax is shown below:

```
try
{
        code block
}
catch (exception ex)
{
        exception handler code
}
```

Using the reference variable ex (or however you name it) you can retrieve lots of information about the exception based on the select members of the exception class as shown in Table 18.

Exceptions that are thrown by the .NETplatform are (appropriately) called system exceptions. These exceptions are regarded as non-recoverable, fatal errors. System exceptions derive directly from a base class named System.SystemException, which in turn derives from System.Exception(which derives from System.Object).

Given that the System.SystemException type does not add any additional functionality beyond a set of custom constructors, you might wonder why SystemException exists in the first place. Simply put, when an exception type derives from System.SystemException, you are able to determine that the .NET runtime is the entity that has thrown the exception, rather than the code base of the executing application. You can verify this quite simply using the is keyword. The syntax is shown below using ex as a reference variable:

```
ex is SystemException
```

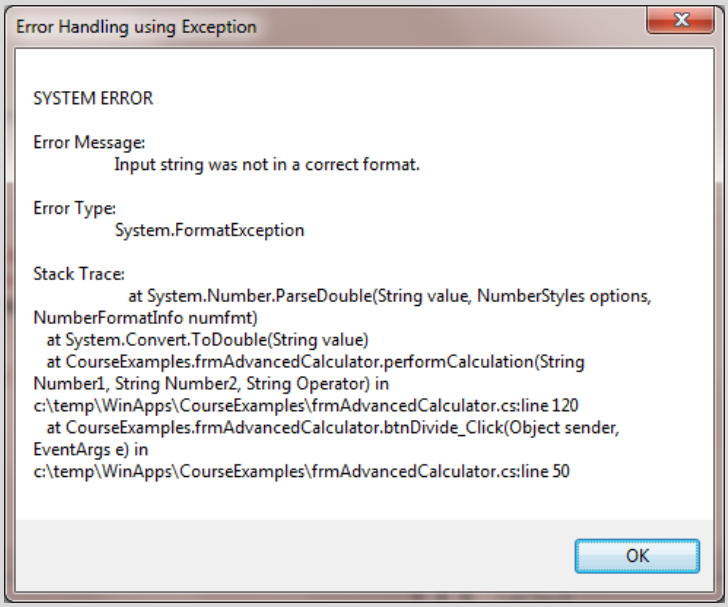
The above statement returns a Boolean, and you can use an if statement to check whether a certain exception is a system exception or a user-defined exception (Application-Level exception, see next section)

Example 3-8: Using Exception object

1. Navigate to the form frmAdvancedCalculator and inside the event procedure btnDivide_Click modify the code as shown below.

```
private void btnDivide_Click(object sender, EventArgs e)
{
    try
    {
        lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
    }
    catch(Exception ex)
    {
        string ErrorMessage = "APPLICATION ERROR\n\n";
        StringBuilder Sb = new StringBuilder();
        if (ex is SystemException) ErrorMessage = "SYSTEM ERROR\n\n";
        Sb.Append("Error Message:\n\t" + ex.Message + "\n\n");
        Sb.Append("Error Type:\n\t" + ex.GetType().ToString() + "\n\n");
        Sb.Append("Stack Trace:\n\t" + ex.StackTrace);
        ErrorMessage += Sb.ToString();
        MessageBox.Show(ErrorMessage,"Error Handling using Exception");
    }
    finally
    {
        MessageBox.Show("This is the Finally Block");
    }
}
```

2. First of all, note that we are using the StringBuilder class, this is an applicable example where you build up a string over a couple of lines. Also note we are testing whether the exception is a System Exception and using an If statement to set the first part of the string ErrorMessage to "SYSTEM ERROR". This string is initialized with "APPLICATION ERROR".
3. Furthermore, we are using some members of the Exception class, such as Message, GetType(), and StackTrace.
4. Save and run the project, and test the exception handler by again enter an alphabetic character into the Number2 textbox.
5. The following messagebox dialog box is displayed.



Catching Specific Types of Errors

A try block can throw various different exceptions, and it is useful to differentiate between those exceptions and issue either appropriate message boxes or handle the error in a specific way for a specific exception. In .NET you can issue multiple catch blocks based on the exception thrown in the try block.

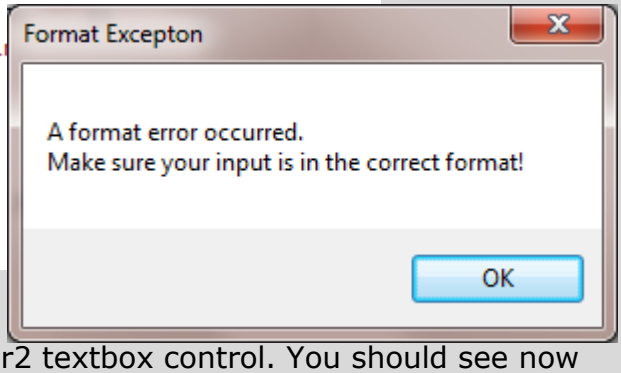
For example, based on the previous example, we could catch the format exception and issue a specific message box to the user. At the end, you want the catch all catch block to ensure that no thrown exception is missed. The syntax is shown below:

```
try
{
        statements
}
catch(FormatException)
{
        specific statements
}
catch(OverflowException)
}
        specific statements
}
catch(Exception ex)
{
        catch all statements
}
```

Example 3-9: Using Specific Types of Exceptions

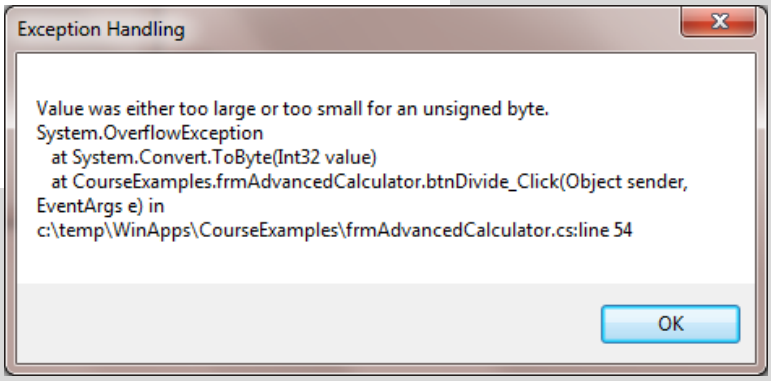
1. Navigate to the form frmAdvancedCalculator and inside the event procedure btnDivide_Click make the following modification:

```
private void btnDivide_Click(object sender, EventArgs e)
{
    try
    {
        lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
    }
    catch (FormatException)
    {
        MessageBox.Show("A format error occurred.\n" +
            "Make sure your input is in the correct format!", "Format Excepton");
    }
    catch (Exception ex)
    {
        string ErrorMessage = "APPLICATION ERROR\n\n";
        StringBuilder Sb = new StringBuilder();
        if (ex is SystemException) ErrorMessage = "SYSTEM ERROR\n\n";
        Sb.Append("Error Message:\n\t" + ex.Message + "\n\n");
        Sb.Append("Error Type:\n\t" + ex.GetType().ToString() + "\n\n");
        Sb.Append("Stack Trace:\n\t" + ex.StackTrace);
        ErrorMessage += Sb.ToString();
        MessageBox.Show(ErrorMessage, "Error Handling usi
    }
    finally
    {
        MessageBox.Show("This is the Finally Block");
    }
}
```



2. Save and run the project.
3. Again, type an alphabetic character into the Number2 textbox control. You should see now the following dialog box as shown above.
4. Add the following code to the try block to generate another exception type.

```
try
{
    lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
    byte test = 0;
    for (int i = 1; i < 300; i++)
    {
        test += Convert.ToByte(i);
    };
}
```



5. Save and run the project. Enter this time just numbers into both textbox controls and click on the divide button. You should see now another error dialog box.

10.4 Application Level Exceptions(System.ApplicationException)

Given that all .NET exceptions are class types, you are free to create your own application-specific exceptions. However, due to the fact that the System.SystemException base class represents exceptions thrown from the CLR, you may naturally assume that you should derive your custom exceptions from the System.Exception type. You could do this, but you could instead derive from the System.ApplicationException class.

Like SystemException, ApplicationException does not define any additional members beyond a set of constructors. Functionally, the only purpose of System.ApplicationExceptionis to identify the source of the error. When you handle an exception deriving from System.Application Exception, you can assume the exception was raised by the code base of the executing application, rather than by the .NET base class libraries or .NET runtime engine.

Using Application exceptions, you can create your own exception. This is kind of enforcing a validation or business rule. The descriptive text you provide when you throw the exception is then displayed in the Message property of the exception. The syntax is shown below:

throw new Exception (*descriptive string*)
or
throw new SystemException (*descriptive string*)
or
throw new ApplicationException (*descriptive string*)

In your exception handler the descriptive string can be called from the ex.Message property of the exception class. It is probably a good idea to instantiate an ApplicationException when creating customized exception handler. That allows you to differentiate your business rules exception from the SystemException (exception.getType()).

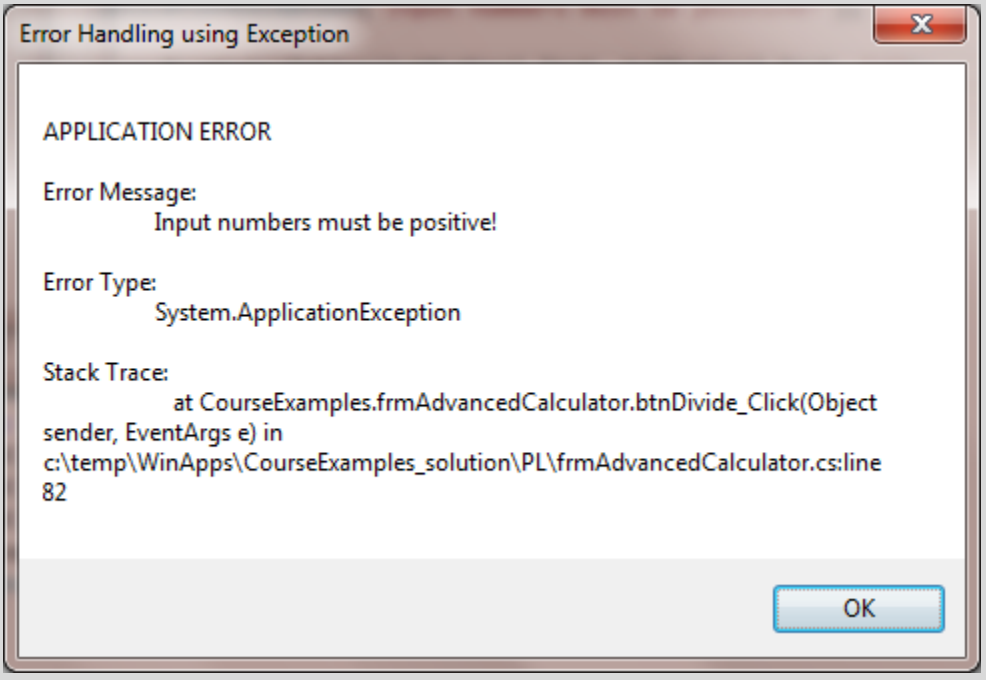
 **Note:** You can actually write your exception class and inherit from the ApplicationException class. It might be advantageous to build a strongly typed exception that represents the unique details of your current problem. This is beyond the scope of this course.

Example 3-10: Using Custom Exceptions

1. Navigate to the form frmAdvancedCalculator and inside the event procedure btnDivide_Click add the following code in the try block:

```
private void btnDivide_Click(object sender, EventArgs e)
{
    try
    {
        if ((double.Parse(txtNumber1.Text) < 0) || (double.Parse(txtNumber2.Text) <= 0))
        {
            throw new ApplicationException("Input numbers must be positive!");
        }
        lblResult.Text = performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
    }
    catch (FormatException)
```

2. The above code implements our custom exception, and if statement tests for the specific condition, in this case we want to ensure that numbers are positive only.
3. If you enter a negative number, the custom exception is raised. The exception handler will work without any further modification, the text "Input numbers must be positive!" will be added to the message property of the exception.
4. Save and run the project. Enter a negative number in one of the textbox controls. You should see the following dialog box.



10.5 Debugging Tools in Visual Studio

Now that we have discussed exceptions in detail, it is probably a good idea to introduce you to the tools to diagnose problems in your application code. Let us be honest, the .NET framework is a very comprehensive environment, and therefore you will experience errors that you might not immediately understand. Using the debugging tools in Visual Studio might help you discover and understand the underlying problem.

When you encounter an unhandled exception while testing your application code, you can enter break mode by clicking on the "Break" button in the exception dialog box, see Figure 43 on page 110.

For example, in break mode, you can simply hover you mouse over variables and objects and find out the current values. For objects, you see a little plus sign that allows you to expand on the object and dig deeper into the object and its values and settings as shown Figure 46

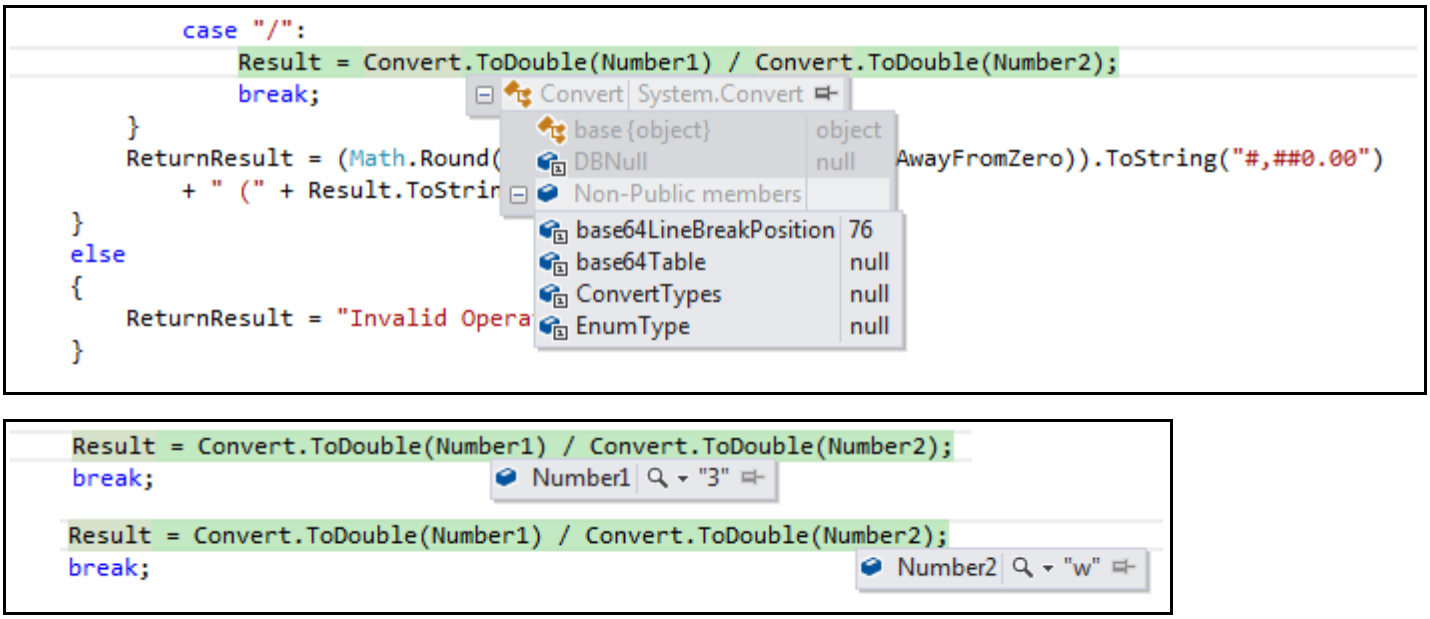


Figure 46: Data Tips in Visual Studio

If Data Tips did not help resolve the problem, then you can take advantage of even more investigative tools. Click on the pull-down menu Debug, then Windows and you see five different options:

- Output
- Locals
- Watch
- Immediate
- Call Stack

These are very powerful tools that should help you figuring out what is going on to resolve the run-time error.

To access these debug tool windows, click on the Menu DEBUG → Windows. In Figure 47 you see that there are actually more debug windows available.

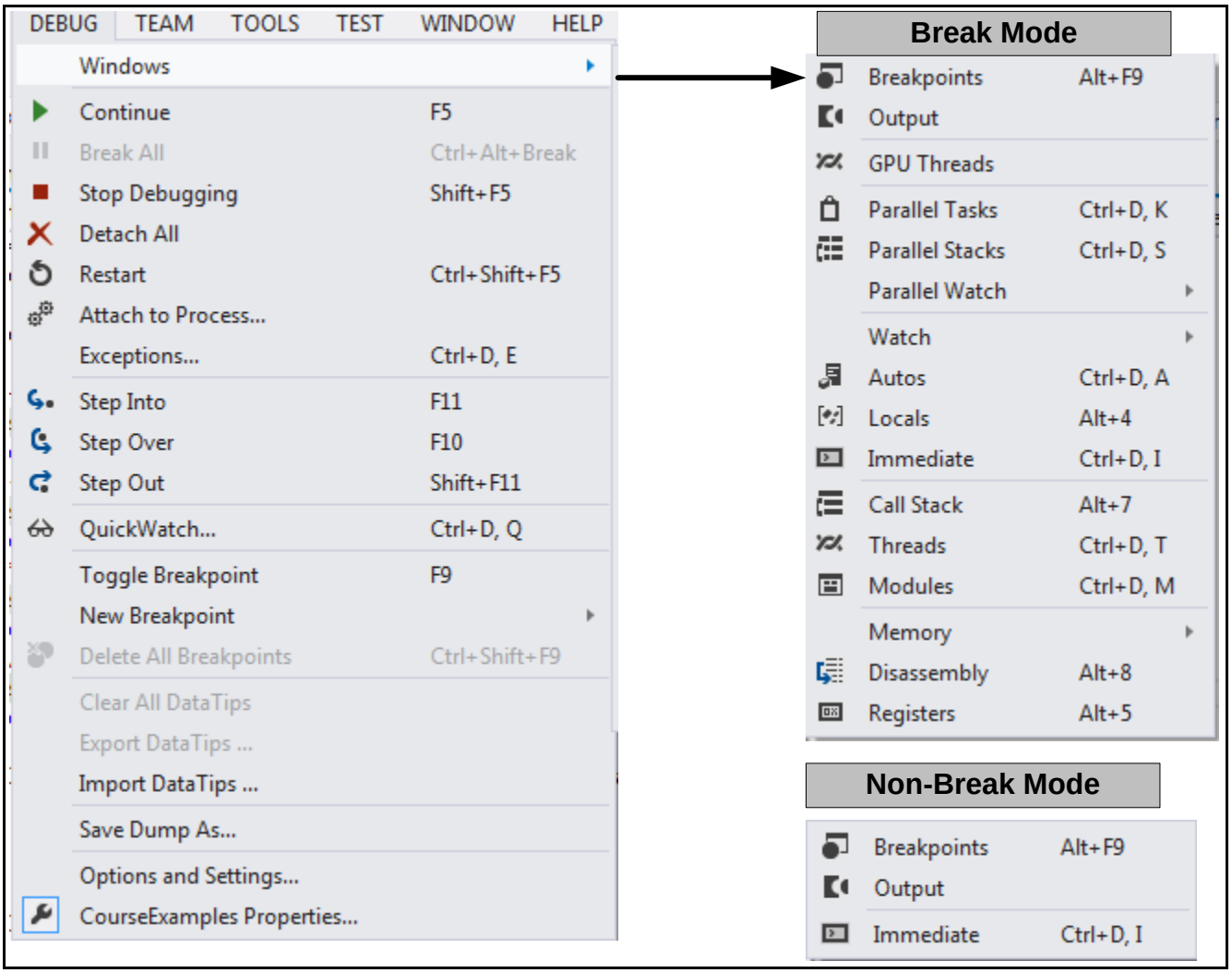


Figure 47: Debug Tool Windows

Warning: The Windows menu having a large selection of options as shown in Figure 47 is only available when you are in break mode. In non-break mode, there are only 3 windows available.

The output window in debug mode shows you the current processes in the .NET environment leading up to the run time error as shown Figure 48.

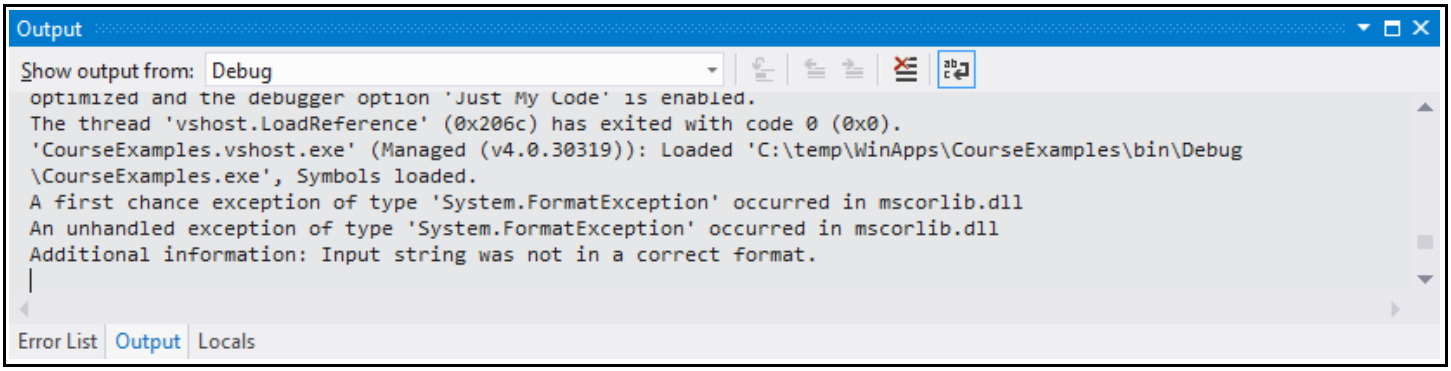


Figure 48: Output Window in Break Mode

The Locals Window as shown in Figure 49 displays you all objects and variables that are currently in scope based on the breakpoint. The plus signs next to the objects enable you to drill down to the methods and properties of these objects.

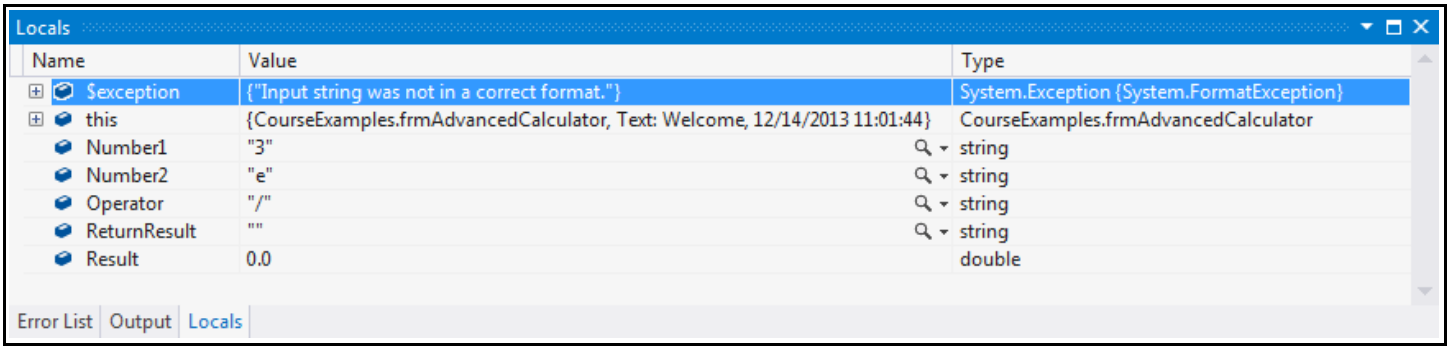


Figure 49: Locals Window in Break Mode

The Watch window is kind of like a sandbox. It allows you to add any expression, objects or variables so that you can see the values that they currently hold as shown in Figure 50

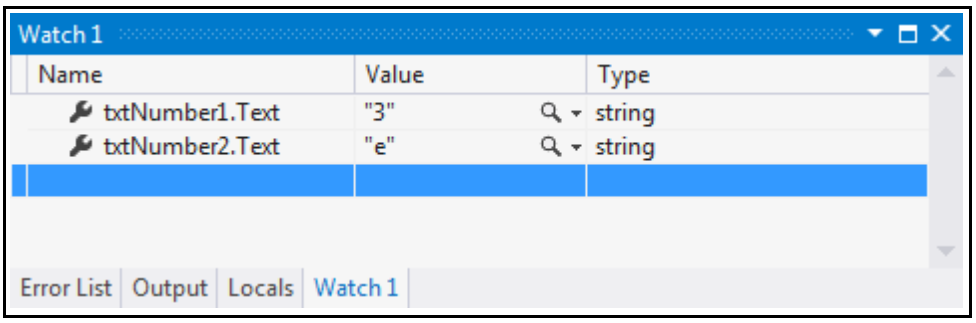


Figure 50: Watch Window in Break Mode

The Immediate Window is similar to the Locals or Watch window in that it allows you to type any valid expression, object or variable to check the current values. For example, if you type DateTime.Now.ToString() it shows you the result as shown Figure 51. You can also declare variable and perform any operations on these variables.

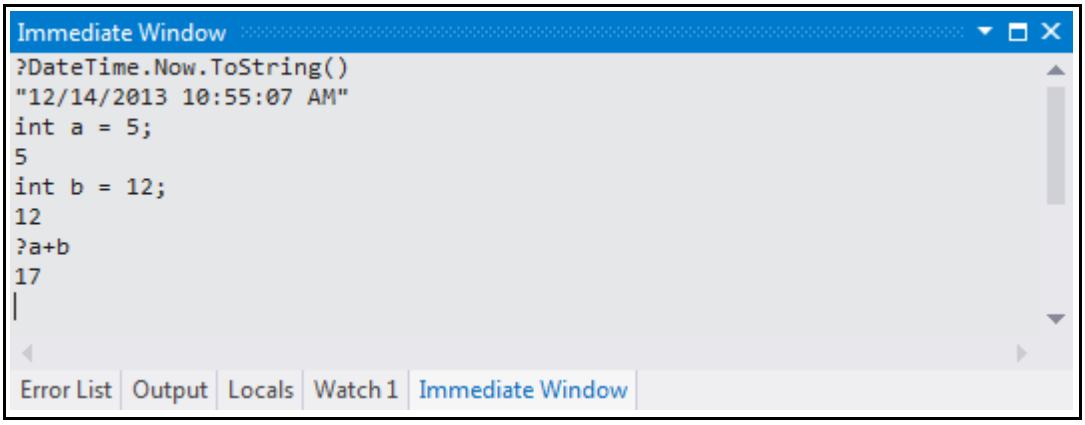


Figure 51: Immediate Window

Note: The Immediate Window is always available, not only in break mode.

The last window is the call stack, which simply shows you in descending order the line of calls until the exception occurred as shown in Figure 52.

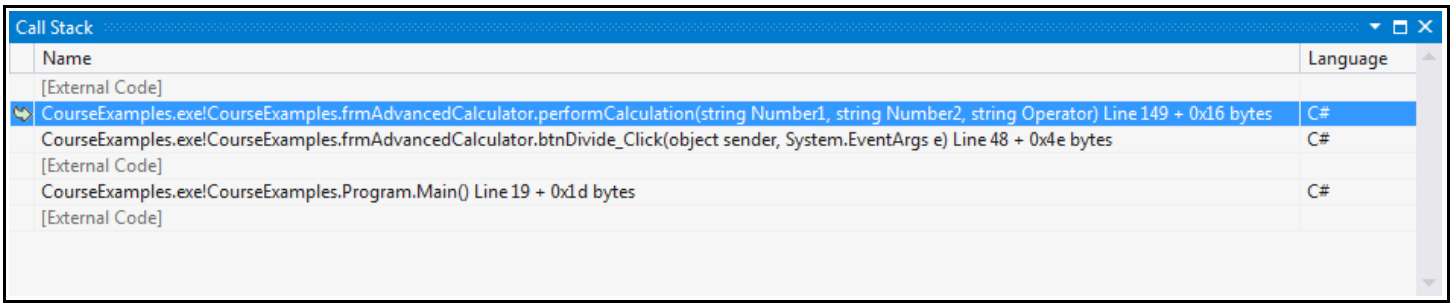


Figure 52: Call Stack Window in Break Mode

The last tool that I want to show you here is how to debug your application even before you encounter an exception. To test your application you can put a breakpoint on any executable line by clicking in the vertical, gray bar on the very left side of the code editor window in Visual Studio. A red circle is added to the bar and the line of code is highlighted with the same color.

When you run your project and the breakpoint line is executed the first time, Visual C# enters break mode, the same mode when an exception occurs. However, now you can use stepping tools to execute your code line by line and watch your objects and variables to further investigate the situation that leads to the exception.

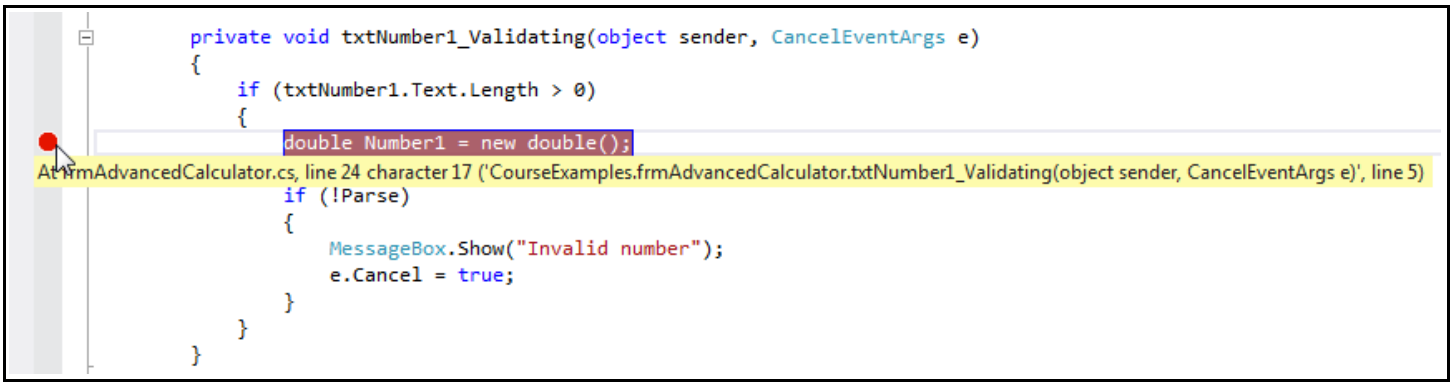


Figure 53: Setting Breakpoint in Visual Studio

Using the debug toolbar, you can step through the code in three modes as shown in Table 19.

Debug Toolbar Button	Description
Step Into (F11)	Steps through line by line, steps into methods or other calls
Step Over (F10)	Steps through line by line, steps over methods or other calls
Step Out (Shift + F11)	Moves one level up in the call stack to the calling line
Continue (F5)	Continues executing the code
Stop Debugging (Shift + F5)	Stops debugging mode and the application
Restart (Ctrl+Shift+F5)	Restarts the application

Table 19: Debug Toolbar Buttons

CLASS 4

11. C# Arrays

Arrays and Collections are objects that function as containers for structured and unstructured data. Whereas a value variable can hold one value, an array or a collection object can hold many data items having one name for the object. An index is associated with the object that allows you to tell the different items apart. Since arrays and collections are objects, which are instantiated through classes, they also possess properties and methods.

11.1 Introduction to C# Arrays

Our lives are all about data these days. In C#, you can create a value-type variable to hold a single value. On the other hand you can create classes and instantiate objects from them to hold complex data about an object, such as a student object. Creating and using classes is definitely quite complex and we will address the basics of classes and objects in this course.

Between these two extremes you have one other option to store data, arrays and collections. They allow you to hold multiple instances of data, but in a simpler way than a class does. We will cover arrays first and then in the next chapter we introduce you to collections.

An array is an object to hold a set of similar data, in fact, of one data type. Arrays can have many dimensions, however, most commonly arrays are either one or two dimensional. We will start out with a one-dimensional array and explore the basic features of an array.

Imagine having six different numbers where you need to calculate the average (and maybe the median, etc.). If you did not use an array object, you would have six value-type variables.

```
double d0=23, d1= 34, d2=36, d3=61, d4 = 96, d5 = 68;
double sum = d0 + d1 + d2 +d3 +d4 + d5;
double avg = sum / 6;
```

Using an array we can process data much for efficiently by utilizing iterations structures. Imagine if you had 1 million numbers.

```
double [] numbers = {23, 34, 36, 61, 96, 68}
double avg = 0;
avg = numbers.Sum() / numbers.Length;
or even simpler:
avg = numbers.Avg();
```

11.2 One-Dimensional Arrays

To create a one-dimensional array, use the following syntax:

```
data type[] array_name;                 //declaration statement
array_name = new data type[length];     //assignment statement
```

The two statements of declaring and initializing an array and their impact on the stack and heap memory are shown in Figure 54.

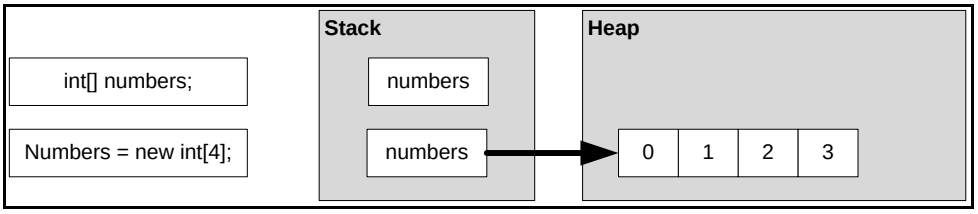


Figure 54: Value and Reference Type Components of an Array Object

You can combine both statements into one:

```
data type[] array_name = new data type[length];
```

Again, we see the new statement. This is used to create an object from the Array class. You will see this now more often, since C# is an object-oriented programming language.

The length specifies the size of an array, that means how many elements it contains. Arrays use zero-based indexes, that means a length value of 5 actually means that the array contains 5 elements (0, 1, 2, 3, 4).

If the elements of an array are not assigned any values, the default values depend on the data type. For any numeric data, the default value is 0. A Boolean is false, and a datetime value is set to 01/01/0001 00:00:00.

To assign values to an array, you simply use the array name followed by the index of element in square brackets you want to assign a value to. The syntax is shown below:

```
string[ ] astrNames = new string[3];
astrNames[0] = "Robert;"
astrNames[1] = "Peter";
astrNames[2] = "Mary";
```

Warning: If you reference element [3] in the above example, you will receive an IndexOutOfRange exception.

Now you can even declare, create, and assign values to an array in one statement as shown below:

```
string[ ] astrNames = [new string[3]] {"Robert","Peter", "Mary"}
```

or even simpler:

```
string[ ] astrNames = {"Robert","Peter", "Mary"}
```

You simply put the values in the order in curly braces after the declaration and creation of the array.

 **Note:** Many times you will use loops to assign values to an array and/or use loops to read values from an array. For example, you may loop through records of a database table, through controls on a form, or through an Excel spreadsheet file.

Up to now we have only covered fixed-size arrays, that is, used a literal value to size the array. Many times, you do not know the number of elements needed at run-time. Therefore, there is another option to create an array, the variable-length array. Instead of using a literal value to size the array you can use a variable in the declaration statement as shown below:

```
int size = 0;
//Perform logic to set the size
string astrNames = new string[size];
```

11.3 Array Properties and Methods

Every array you create gathers much of its functionality from the System.Array class. Using these common members, you are able to operate on an array using a consistent object model. Table 20 and Table 21 display of some of the more interesting members

Property	Description
IsFixedSize	Return a Boolean indicating if an array has a fixed size or not.
IsReadOnly	Returns a Boolean indicating if an array is read-only or not.
Length	Returns the total number of items in all the dimensions of an array.
Rank	Returns the number of dimensions of an array.

Table 20: Select Properties of Array Class

Static Method	Description
BinarySearch(array, value)	This method searches a one-dimensional sorted Array for a value, using a binary search algorithm.
Clear(array, index, length)	This method removes all items of an array and sets a range of items in the array to 0.
Copy(array1,array2,length)	This method copies a section of one Array to another Array and performs type casting and boxing as required.
Reverse(array)	This method reverses the order of the items in a one-dimensional Array or in a portion of the Array.
Sort(array)	This method sorts the items in one-dimensional Array objects.

Table 21: Select Static Methods of Array Class

The methods above are static methods of the array class, that means you have to reference the Array class and then issue the method as shown below:

Array.Sort(*array_name*)

 **Warning:** The Length property returns the number of all items in all dimensions. For a one-dimensional array, this works well for looping through the array. But for multi-dimensional arrays, this would not work since the Length property adds up the number of items in each dimension.

Therefore, it is good practice to check the Rank property. It returns the number 1 for a one-dimensional array. Once you know for sure that the array is one-dimensional, then you could use the Length property to retrieve the number of items in an array.

When you have instantiated an array object, you have additional methods available that operate on the specific instance of the array object as shown Table 22.

Instance Method	Description
Clone()	This method creates a shallow (=another reference pointing to the same object) copy of the Array.
CopyTo(array, index)	This method copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index.
GetLength(dimension)	This method returns the number of items in a specific dimension of an Array.
GetLowerBound(dimension)	This method returns the lower bound of an Array in a specific dimension.
GetUpperBound(dimension)	This method returns the upper bound of an Array in a specific dimension.

Table 22: Select Instance Methods of Array Class

To use the above methods, you use an existing array and then issue the method on that array as shown below:

```
array_name.GetLength(2)
```

There are two useful methods of the string class that are particularly useful when working with arrays, the Split and the Join methods as shown in Table 23.

String Method	Description
Join(array, joinCharacter)	Concatenates all the elements of a string array, using the specified joinCharacter between each element.
Split(splitCharacters)	Returns an array of strings where each element is a substring that is delimited by the specified character or characters.

Table 23: String Methods for Array Processing

 **Note:** You can specify more than one split character in the split method of the string class using a comma-separated list of delimiter characters.

Up to now, we have been able to declare arrays storing only one type of data, such as int or string, for example. In the majority of cases, this is desired and conforms to the strongly-typed nature of C#. One can use the keyword var to declare an implicitly-typed array, but still all elements must be eventually of the same type.

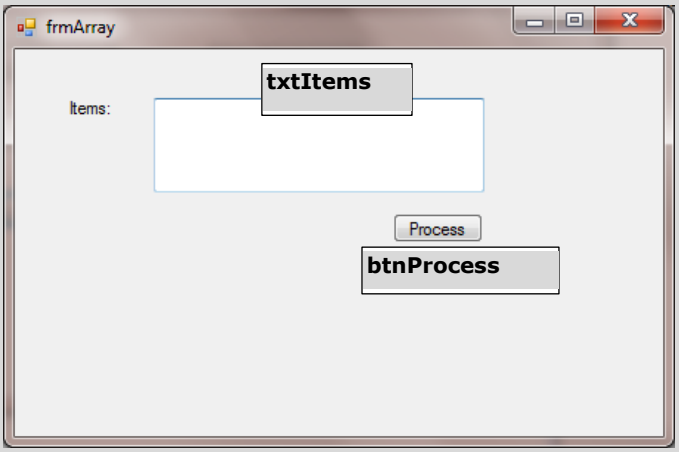
 **Note:** The **var** keyword causes the compiler to deduce the type of the variables from the types of the expressions used to initialize them. We will introduce this concept when we discuss LINQ, Language Integrated Query language.

Besides value-type variables we can also generate an array of objects, what an interesting concept. Remember, an object is instantiated from a class and can hold a variety of data (state and behavior). In an array, you can now store multiple instances of objects holding even different kind of data.

The System.Object is the ultimate base class to each and every type (including fundamental data types) in the .NET type system. Given this fact, if you were to define an array of objects, the sub items could be anything at all. We will demonstrate this in the next examples.

Example 4-1: Processing String Data using Arrays (Split, Join)

1. Add a new form to the project CourseExamples and name it frmArray.cs.
2. Place a label, textbox, and a button control onto the form. Set the Text properties for the controls as shown on the right.
3. Set the textbox MultiLine property to true and then size it to at least two lines tall. Name the textbox and the button control as shown on the right.
4. Double-click on the form itself (not any controls) to create the form's default load event. Add the following line of code:

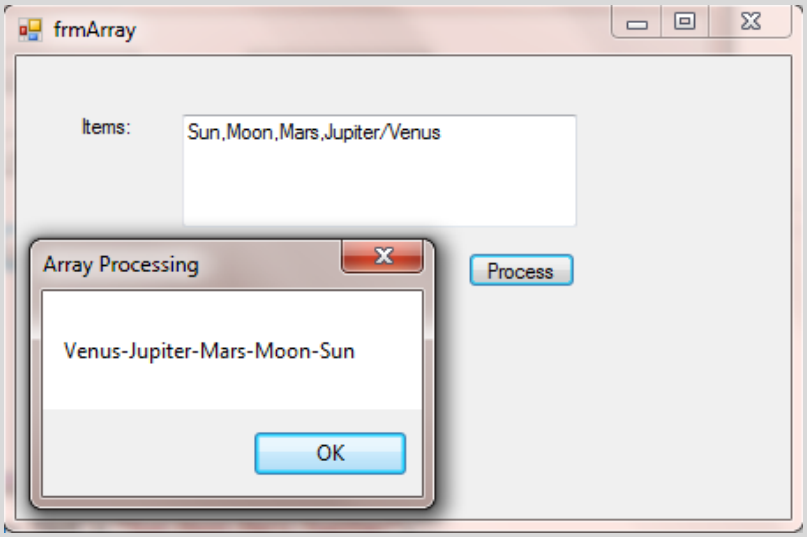



```
private void frmArray_Load(object sender, EventArgs e)
{
    txtItems.Text = "Sun,Moon,Mars,Jupiter";
}
```

5. Double-click on the button to create the button's default click event and type the following code:

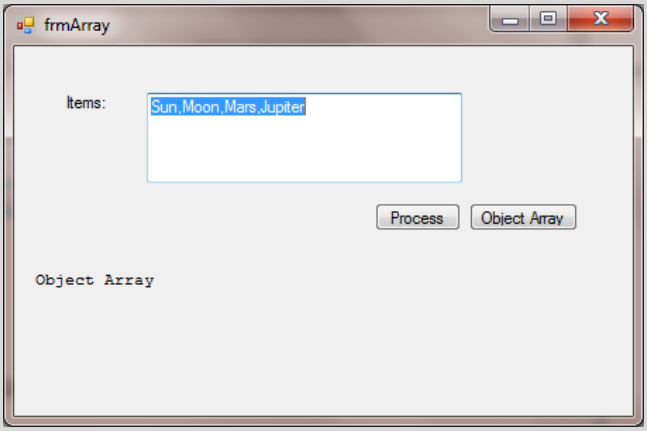
```
private void btnProcess_Click(object sender, EventArgs e)
{
    string Items = txtItems.Text;
    string[] ArrayItems = Items.Split(',','/');
    Array.Reverse(ArrayItems);
    MessageBox.Show(string.Join("-",ArrayItems),"Array Processing");
}
```

6. Add a button on the main menu form and name it btnArray, set its Text property to "Arrays" and create the usual two lines of code to show the form.
7. Save and run the project. You will notice that there is already an entry in the textbox control due to the form's load event. Add a forward slash at the end and type "Venus".
8. Now click on the button and notice the output in the message box as shown on the right.



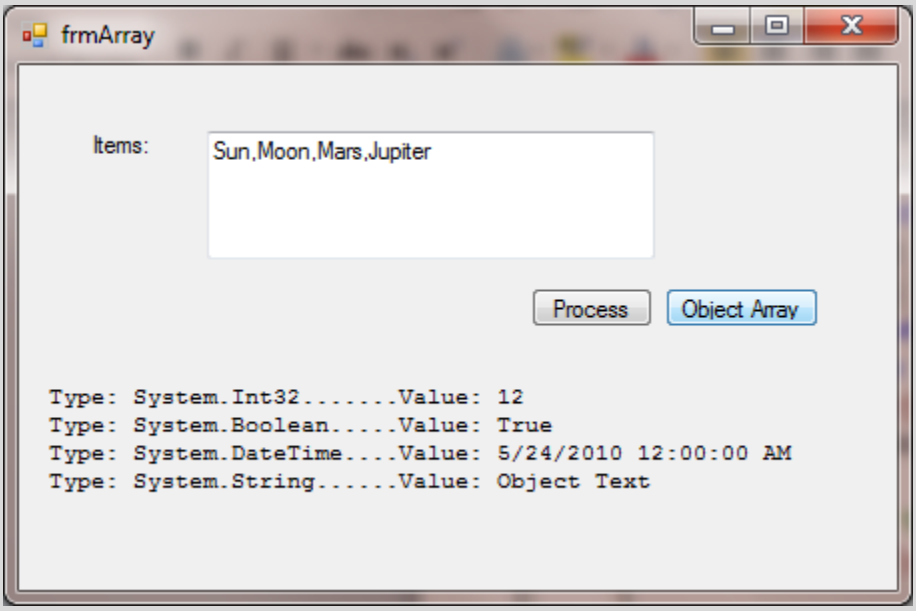
Example 4-2: Processing Object Data using Arrays

1. Add another button control to the form frmArray.cs and name it btnObjectArray. Set its Text property "Object Array".
2. Place a label control below the textbox control, name it lblOutput, and set its Text property to "Array Object". Change the font to Courier New and size it to 8.
3. Double-click on the Object Array button to create the default click event. Type the following code:



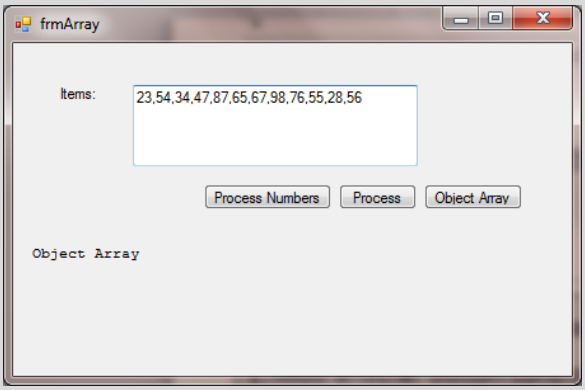

```
private void btnObjectArray_Click(object sender, EventArgs e)
{
    string Output = string.Empty;
    object[] myObject = new object[4];
    myObject[0] = 12;
    myObject[1] = true;
    myObject[2] = new DateTime(2010, 5, 24);
    myObject[3] = "Object Text";
    foreach (object obj in myObject)
    {
        // Print the type and value for each item in array.
        Output += ("Type: " + obj.GetType()).PadRight(25,'.') + "Value: " + obj + "\n";
    }
    lblOutput.Text = Output;
}
```

4. Save and run the project. Click on the button "Object Array" and notice the output in the label.



Example 4-3: Processing Number Data using Arrays

1. Add another button control to the form frmArray.cs and name it btnProcessNumbers. Set its Text property "Process Numbers".
2. In the load event, comment out the current line of code and add a line to assign some numbers in the textbox control.

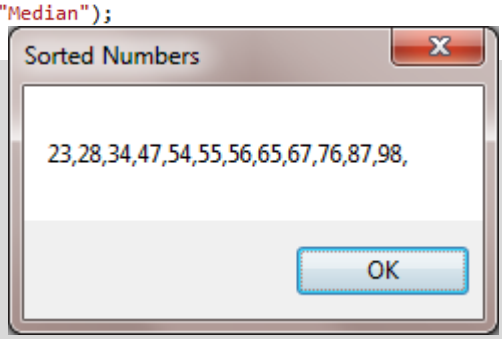



```
private void frmArray_Load(object sender, EventArgs e)
{
    //txtItems.Text = "Sun,Moon,Mars,Jupiter";
    txtItems.Text = "23,54,34,47,87,65,67,98,76,55,28,56";
}
```

3. Double-click on the new button to create the default click event and write the following code. We are going to convert a string array into a double array, sort the numbers, and also calculate the median.

```
private void btnProcessNumbers_Click(object sender, EventArgs e)
{
    string Items = txtItems.Text, Output = string.Empty;
    string[] ArrayItems = Items.Split(',', '/');
    double[] ArrayNumbers = new double[ArrayItems.Length];
    double Median = 0;
    //Convert array elements into double
    for (int i=0;i<ArrayItems.Length;i++)
    {
        ArrayNumbers[i] = Convert.ToDouble(ArrayItems[i]);
    }
    Array.Sort(ArrayNumbers);
    foreach (double dbl in ArrayNumbers)
    {
        Output += dbl.ToString() + ",";
    }
    MessageBox.Show(Output, "Sorted Numbers");
    //Calculate median
    if (ArrayNumbers.Length % 2 == 0) //even number of elements
    {
        Median = (ArrayNumbers[ArrayNumbers.Length / 2 - 1] + ArrayNumbers[ArrayNumbers.Length / 2]) / 2;
    }
    else //odd number of elements
    {
        Median = ArrayNumbers[Convert.ToInt32(ArrayNumbers.Length / 2)];
    }
    MessageBox.Show("Median = " + Median.ToString(),"Median");
}
```

4. Save and run the project. If needed, modify the numbers in the textbox.



11.4 Multi-Dimensional Arrays

Multi-dimensional arrays have two or more dimensions. Two-dimensional arrays are also referred to as rectangular arrays. These arrays are common in data processing where you store rows and columns of data in an array.

To declare a multi-dimensional array, you use commas in the type statement within the square brackets to separate the dimensions. The syntax is shown below:

Data type[,] *array_name* = **new data type**[*row_index, column_index*]

An example of a two-dimensional integer array is shown below:

Example:

int[,] Totals = **new int**[3,2];

or declaring and assigning in one step

int[,] Totals = { {1,4}, {3,8} {3,9} };

To assign or read values from a rectangular array you have to code a nested loop. The outer loop iterates through the row_index, whereas the inner loop iterates through the column index.

Example:

```
for (int i = 0; i < Totals.GetLength(0);i++)
{
      for (int j=0; j < Totals.GetLength(1); j++)
      {
            statements
      }
}
```

Here you see the GetLength method in action, the outer loop refers to the first dimension(0) of the array, whereas the inner loop refers to the second dimension(1).

Arrays of higher dimensions are simply called multi-dimensional arrays. In C#, you can also create Jagged Arrays, an array of arrays. These are arrays with varying number of columns.

12. C# Collections

Often times, an array is the most straightforward way to deal with list data. You may also encounter many places .NET framework class library that require the use of arrays.

However, there are also many other situations where an array does not fit the bill and other collection objects are needed. There are actually too many collection objects to cover in this course, I will focus on the most important ones.

12.1 Overview of C# Collections

The .NET Base Class Library (BCL) has a wide array of collection classes at your disposal which make it easy to manage collections of objects. While it is great to have so many classes available, it can be daunting to choose the right collection to use for any given situation. As hard as it may be, choosing the right collection can be absolutely key to the performance and maintainability of your application!

There are two main categories of collection classes as shown below:

- Associative collection classes, where a key/value pair is used to store data
- Non-associative collection classes, where simply the value is stored in a list

The generic collections were introduced in .NET 2.0 in the System.Collections.Generic namespace. This is the main body of collections you should tend to focus on first, as they will tend to suit 99% of your needs right up front.

 **Note:** The namespace System.Collections.Generic is normally automatically included.

Associative collections store a value in the collection by providing a key that is used to add/remove/lookup the item. Hence, the container associates the value with the key. These collections are most useful when you need to lookup/manipulate a collection using a key value. For example, if you wanted to look up an order in a collection of orders by an order id, you might have an associative collection where they key is the order id and the value is the order.

Non-associative collections do not use keys to manipulate the collection but rely on the object itself being stored or some other means (such as index) to manipulate the collection.

Collections used to be untyped (before .NET 2.0), that means they could hold any type of object within one collection. This lead to run-time errors when processing data of varying type. Since .NET 2.0 collections are typed, meaning they have to be declared as a type, such as int, for example, that is the collection could only hold integer values.

Table 24 shows the .NET 1.X collection classes and their counterpart in .NET 2.0 and higher.

.NET 2.0 and after	.NET 1.X	Description
List<type>	ArrayList	Uses an index to access each element. Very efficient for accessing elements sequentially.
SortedList<key type, value type>	SortedList	Acts like a lookup table, having a key and value pair to store data. This list is automatically sorted by the key value.
Queue<type>	Queue	Uses methods to add/remove elements (FIFO)
Stack<type>	Stack	Uses method to add/remove elements (LIFO)

Table 24: NET 1.X Collection Objects

 **Note:** The older, untyped collection classes reside in the System.Collections namespace, which is not automatically included anymore. You still might find old code that uses these collection classes.

What is all the fuzz about these collection objects, why not use an array? Again, if an array object meets your needs, please use it and keep it simple. However, there are two main areas where an array falls short:

- A program must declare the size of an array when it is created. Although you can use a variable to make the array sizing more dynamic, but the truth is, there has to be at some point a fixed size for an array.
- Inserting or removing an element in the middle of an array is very inefficient. You have to write quite some loops to move data around to make room for new data.

Table 25 shows the basic collection classes based on the System.Collections.Generics namespace.

Collection	Ordering	Contiguous Storage	Direct Access	Notes
Dictionary	Unordered	Yes	Via Key	Best for high performance lookups.
SortedDictionary	Sorted	No	Via Key	Compromise of Dictionary speed and ordering, uses binary search tree.
SortedList	Sorted	Yes	Via Key	Very similar to SortedDictionary, except tree is implemented in an array, so has faster lookup on preloaded data, but slower loads.
List	User has precise control over element ordering	Yes	Via Index	Best for smaller lists where direct access required and no sorting.
LinkedList	User has precise control over element ordering	No	No	Best for lists where inserting/deleting in middle is common and no direct access required.
HashSet	Unordered	Yes	Via Key	Unique unordered collection, like a Dictionary except key and value are same object.
SortedSet	Sorted	No	Via Key	Unique sorted collection, like SortedDictionary except key and value are same object.
Stack	LIFO	Yes	Only Top	Essentially same as List<T> except only process as LIFO
Queue	FIFO	Yes	Only Front	Essentially same as List<T> except only process as FIFO

Table 25: Summary of C# Collection Classes

12.2 Dictionary Object

Dictionary objects are probably the most commonly used collection object when it comes to storing a key value type pair of data. This is very similar to a database lookup table, where you have two columns, one is the key or ID, and the other one is the description column.

To declare a dictionary object, use the following syntax:

Dictionary<TKey, TValue**>** *variable_name* = **new Dictionary<**TKey, TValue**>()**;

TKey = type of the key values, TValue = type of the value associated with the key

The type for the key and the value can be chosen from all possible types within C#, even custom objects (when you later design your own classes).

You can now add data to this object by using the Add method. However, as with arrays, you can use a simpler syntax by declaring and initializing a dictionary object:

Dictionary<TKey, TValue**>** *variable_name*;

variable_name = **new Dictionary<**TKey, TValue**>**{ {key1,val1}, {key2, val2}, …}

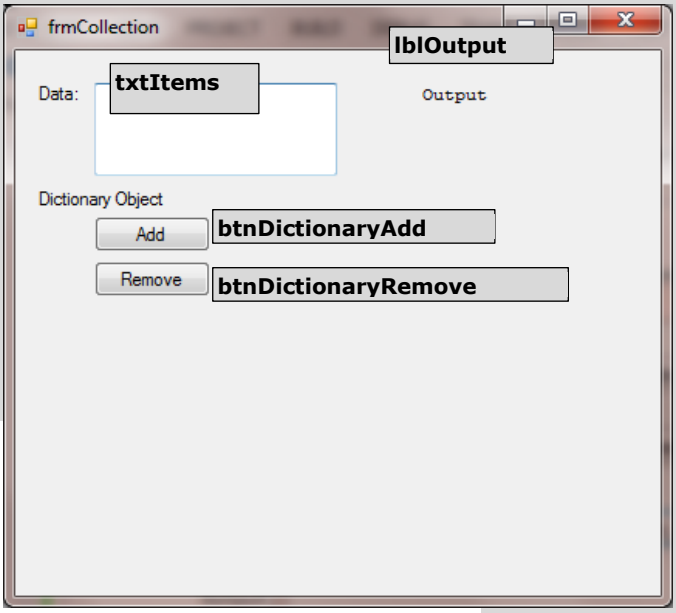
You can also combine these two lines into one statement. Some common methods and properties are shown in Table 26.

Indexer	**Description**
[index]	Gets or sets the value associated with the specified key.
Property	**Description**
Count	Gets the number of elements in the dictionary object.
Keys	Gets a collection containing the keys in the dictionary object.
Values	Gets a collection containing the values in the dictionary object.
Method	**Description**
Add(TKey,TValue)	Adds the specified key and value to the dictionary.
Clear()	Removes all elements from the list and sets its Count property to zero.
ContainsKey(TKey)	Returns a Boolean that indicates if the dictionary object contains the specified key.
Remove(Tkey)	Removes the value with the specified key from the dictionary object.

Table 26: Properties and Methods of Dictionary Object

Example 4-4: Using Dictionary Object

1. Add a new form to the project CourseExamples and name it frmCollections.cs.
2. Place the controls as shown on the right onto the form. Set the Text properties for the controls as shown on the right.
3. For the textbox control, set its Multiline property to True, and set the font for the Output label to Courier New.
4. Double-click on the form itself to create the form's load event and write the following statements:




```
Dictionary<int, string> Animals;
public frmCollection()
{
    InitializeComponent();
}

private void frmCollection_Load(object sender, EventArgs e)
{
    Animals = new Dictionary<int,string> { {0,"Monkey"},{1,"Elephant"}, {2,"Giraffe"} };
}
```

5. Create the following method to display the output in the output label:

```
private void OutputDictionary()
{
    string Output = "Dictionary Object Example \n" + "Key".PadRight(5) + "Value\n";
    foreach (var Pair in Animals)
    {
        Output += Pair.Key.ToString().PadRight(5) + Pair.Value + "\n";
    }
    Output += "Number of items: " + Animals.Count.ToString();
    lblOutput.Text = Output;
}
```

6. Double-click on the Add button to create the button's click event and write the following code:

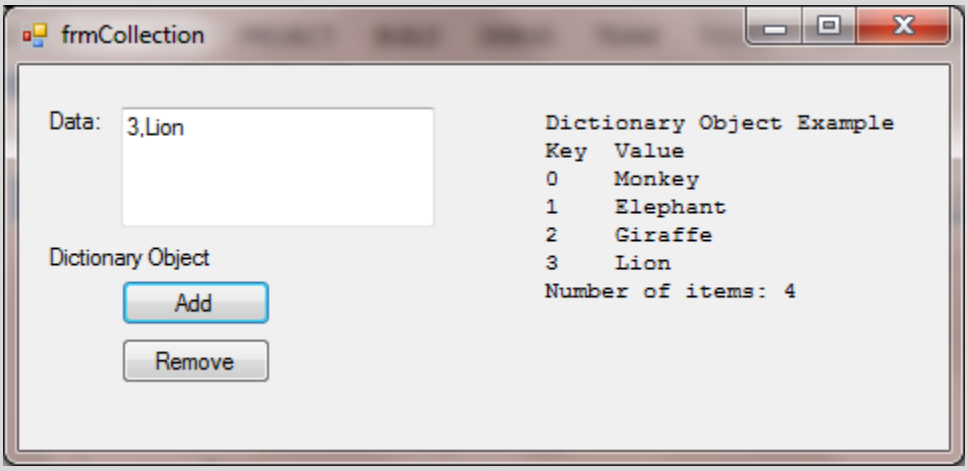
```
private void btnDictionaryAdd_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    Animals.Add(Convert.ToInt32(Input[0]), Input[1]);
    OutputDictionary();
}
```

Example 4-4(continued): Using Dictionary Object

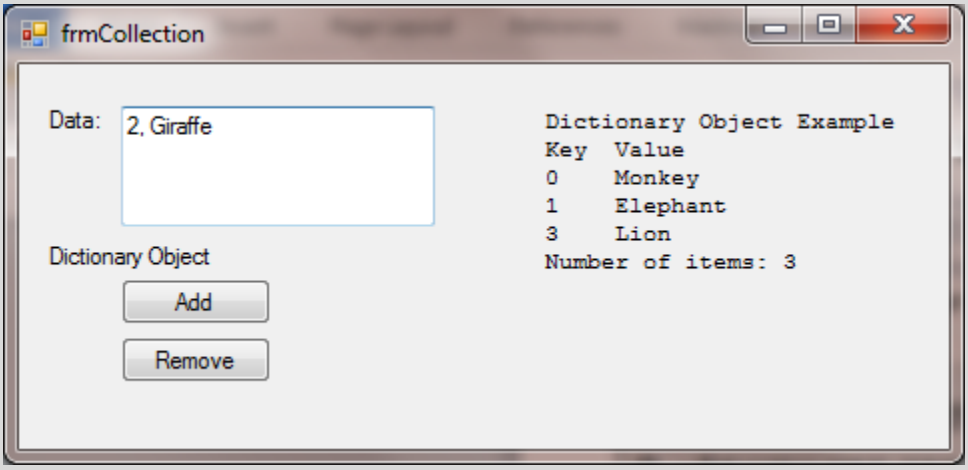
7. Double-click on the Remove button the create the button's click event and write the following code:

```
private void btnRemove_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    Animals.Remove(Convert.ToInt32(Input[0]));
    OutputDictionary();
}
```

8. Add a new button on the Main Menu form, name btnCollections and set its Text property to "Collections". Add the usual two lines of code to show this new form.
9. Save and run the project.
10. As you can see from the load method, we are initializing a Dictionary object (that is declared at the class level) with three animals. In the text box, enter a new key value, a comma, and a new animal. Then click on the Add button. You should see the following output:



11. Now change the entry in the textbox to an existing item in the Dictionary object to be removed, for example: 2,Giraffe. Then click on the Remove button. You should see the following output:



12.3 List Object

A list object is very similar to an array, it just stores one type (as opposed to Key/Value pair) using an index. But it overcomes the shortfalls of an array as stated earlier, for example, you do not dimension a list object, its space requirement is limited by memory resources only.

To create a new List object, use the following syntax:

List<type> *list_name* = **new List<type>()**

List<type> *list_name* = **new List<type>(***capacity***)**

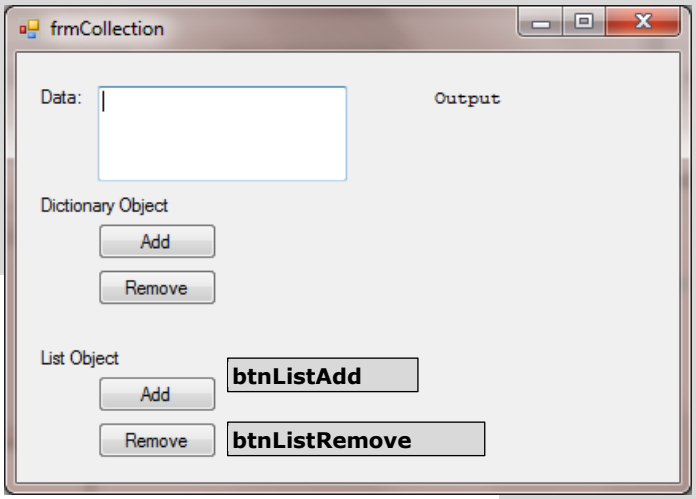
The type can be either a value or reference type. You can optionally set the initial capacity of the list object in the parentheses. If you omit the capacity, it is set to 16. Below is an example of a List object:

Table 27 shows the common properties and methods of the List class:

Indexer	**Description**
[index]	Gets or sets the elements at the specified index (zero-based).
Property	**Description**
Capacity	Gets or sets the number of elements the list can hold.
Count	Gets the number of elements in the list.
Method	**Description**
Add(object)	Adds an element to the end of a list and returns the element's index.
Clear()	Removes all elements from the list and sets its Count property to zero.
Contains(object)	Returns a Boolean that indicates if the list contains the specified object.
Insert(index,object)	Inserts an element into the list at the specified index.
Remove(object)	Removes the first occurrence of the specified object from the list.
RemoveAt(index)	Removes the element at the specified index of a list.
BinarySearch(object)	Searches a list for a specified object and returns the index for that object.
Sort()	Sorts the elements in a list into ascending order.

Table 27: Properties and Methods of List Collection Object

Example 4-5: Using List Object



1. Add another label and two button controls to the form frmCollections.cs and name the as shown to the right.
2. Modify the load event as follows:

```
Dictionary<int, string> Animals;
List<string> Countries;
public frmCollection()
{
    InitializeComponent();
}

private void frmCollection_Load(object sender, EventArgs e)
{
    Animals = new Dictionary<int,string> { {0,"Monkey"},{1,"Elephant"}, {2,"Giraffe"} };
    Countries = new List<string> { "U.S.", "Japan", "India", "France" };
}
```

3. Add the following output method to populate the label output:

```
private void OutputList()
{
    string Output = "List Object Example \n";
    foreach (string Item in Countries)
    {
        Output += Item + "\n";
    }
    Output += "Number of items: " + Countries.Count.ToString();
    lblOutput.Text = Output;
}
```

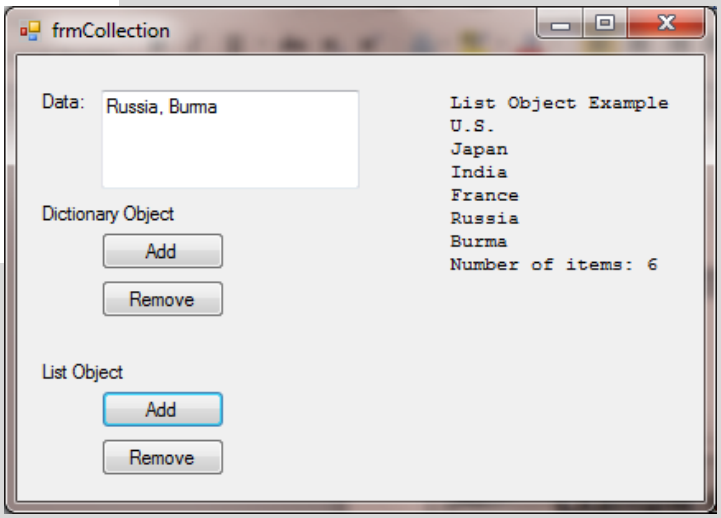
4. Double-click on the Add button to create the click event and write the following event handler:

```
private void btnListAdd_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    foreach (string Item in Input)
    {
        Countries.Add(Item.Trim());
    }
    OutputList();
}
```

Example 4-5 (continued): Using List Object

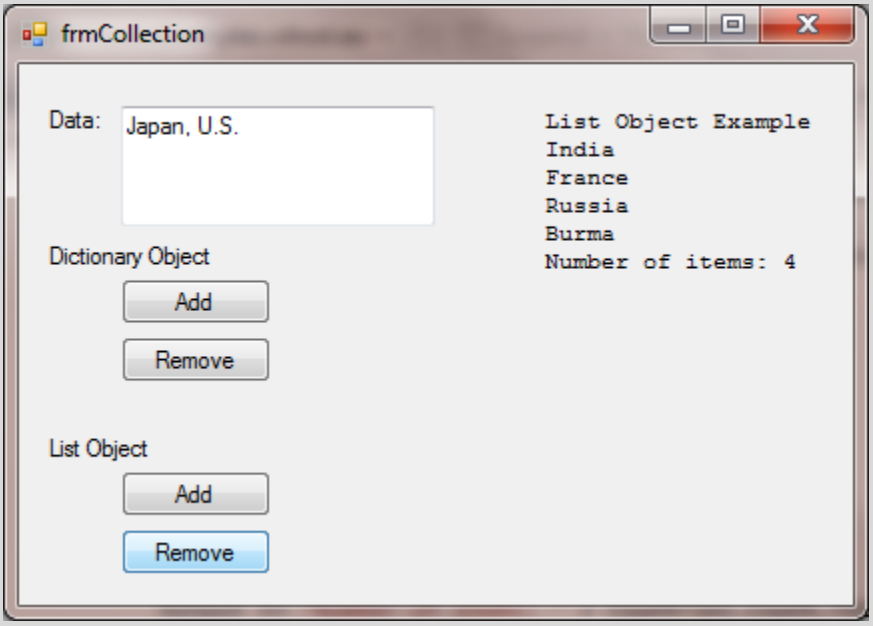
5. Double-click on the Remove button to create the click event and write the following event handler:

```
private void btnListRemove_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    foreach (string Item in Input)
    {
        Countries.Remove(Item.Trim());
    }
    OutputList();
}
```



6. Save and run the project. Enter one or more countries (comma-separated) in the textbox and click on the Add button to add these countries to the list. Note the output.

7. Type one or more existing countries (comma-separated) in the textbox and click on the Remove button. Note the output.



12.4 Queue and Stack Objects

Queues and Stacks objects are special objects in that they do not have an indexer to set or get items. Furthermore, they do not have an add method, instead, the Queue object has Enqueue and Dequeue methods to add and retrieve items and the Stack object has the Push and Pop methods.

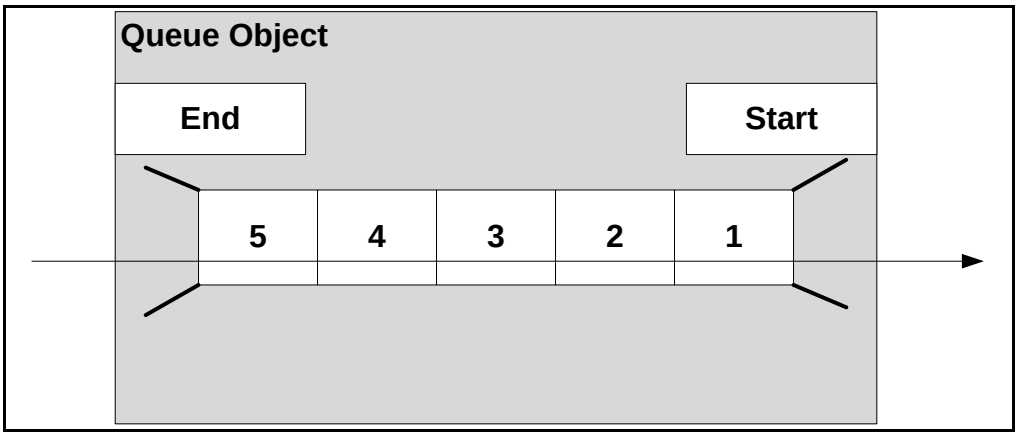


Figure 55: Queue Object

The Queue object processes items on a First In, First Out (FIFO) basis. As you can see from the Figure 55, item 1 was added first at the end of the Queue. As more and more items were added, item 1 was moved to the right towards the start. Item 5 was the last one that was added, and item I was the first one. Therefore, item 1 will be the first one that can be removed from the queue.

To add an item, you use the Enqueue method. To retrieve an item, you use the Dequeue method. The most commonly used properties and methods for the Queue object are shown in Table 28:

Property	**Description**
Count	Gets the number of items in the queue
Method	**Description**
Enqueue(object)	Adds the specified object to the end of the queue.
Dequeue()	Gets the object at the start and removes it from the queue.
Clear()	Removes all items from the queue.
Peek()	Retrieves the next item in the queue without deleting it.

Table 28: Properties and Methods of Queue Object

A Stack object operates on the opposite principle compared to the Queue object. Its items are processed based on Last In, First Out (LIFO) concept. As you can see in Figure 56, items are added and removed only through the top. A stack works like a barrel or any other top-opened container.

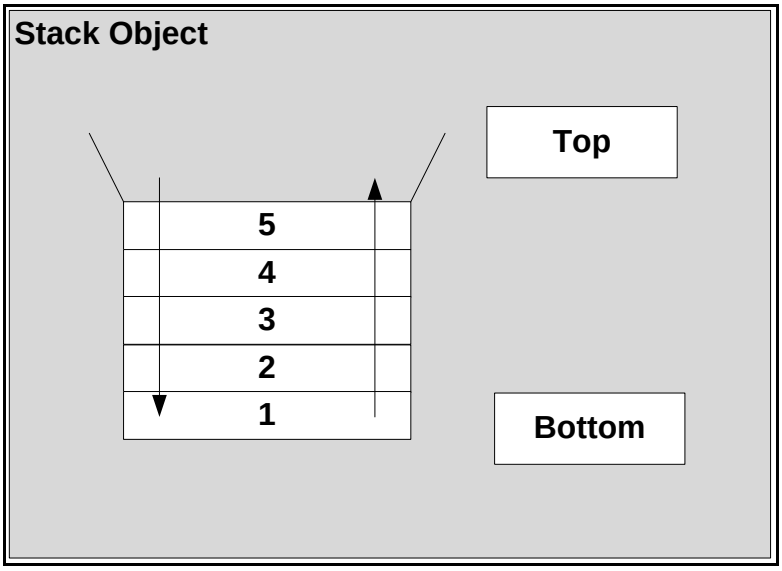


Figure 56: Stack Object

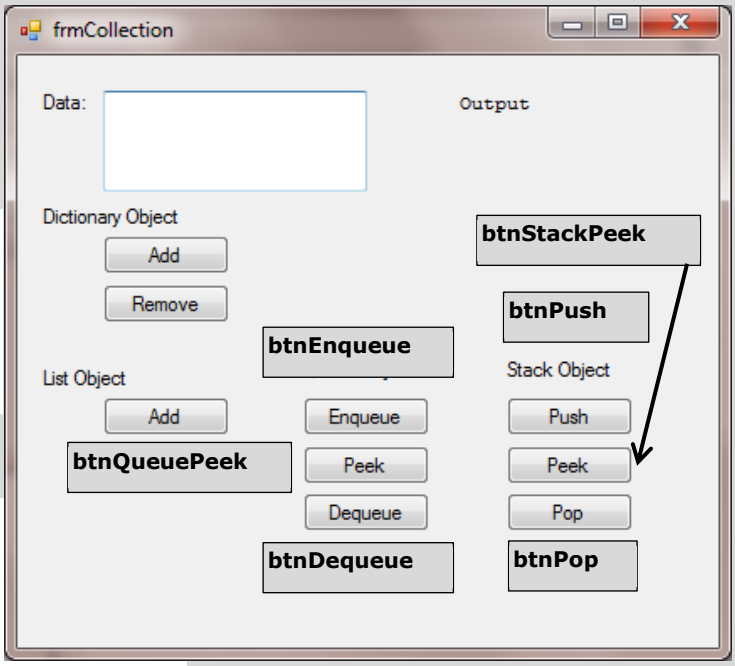
The most common properties and methods are shown in

Property	**Description**
Count	Gets the number of items in the queue
Method	**Description**
Push(object)	Adds the specified object to the end of the stack.
Pop()	Gets the object at the top and removes it from the stack.
Clear()	Removes all items from the stack.
Peek()	Retrieves the next item in the stack without deleting it.

Figure 57: Properties and Methods of Stack Object

Example 4-6: Using Queue and Stack Objects

1. Add the controls shown on the right side for processing Queue and Stack objects.
2. Add a Queue and Stack declaration at the top of the class as shown below:




```
public partial class frmCollection : Form
{
    Dictionary<int, string> Animals;
    List<string> Countries;
    Queue<string> Waitlist = new Queue<string>();
    Stack<string> Bin = new Stack<string>();
```

3. Write the following output method:

```
private void OutputQueue()
{
    string Output = "Queue Object Example \n";
    foreach (string Item in Waitlist)
    {
        Output += Item + "\n";
    }
    Output += "Number of items: " + Waitlist.Count.ToString();
    lblOutput.Text = Output;
}
```

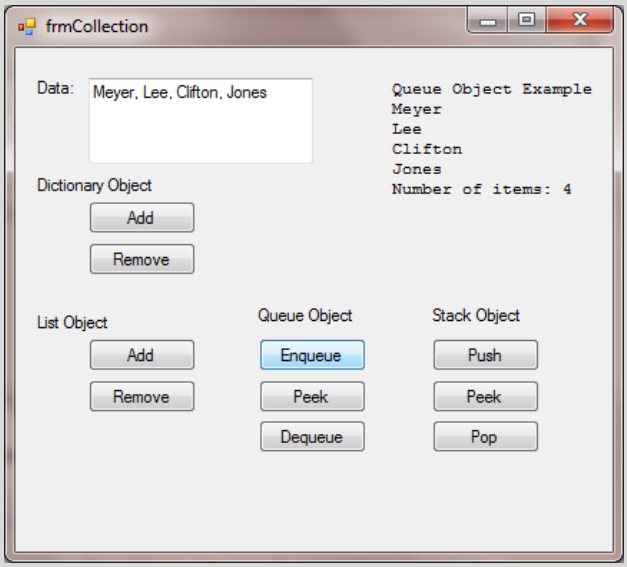
4. Double-click on the Enqueue button to create the click event as shown below:

```
private void btnEnqueue_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    foreach (string Item in Input)
    {
        Waitlist.Enqueue(Item.Trim());
    }
    OutputQueue();
}
```

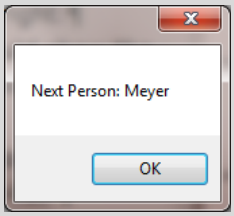
5. Double click on the Dequeue and Peek buttons to create the respective click events and write the following event handlers:

```
private void btnQueuePeek_Click(object sender, EventArgs e)
{
    MessageBox.Show("Next Person: " + Waitlist.Peek());
}

private void btnDequeue_Click(object sender, EventArgs e)
{
    Waitlist.Dequeue();
    OutputQueue();
}
```

Example 4-6(continued): Using Queue and Stack Objects



6. Save and run the project. Add some last names into the Data textbox. Now click on the Enqueue button. The output is shown on the right:
7. Now click on the Peek button, it should show the next person to be processed in the waitlist, which is the first person (FIFO).

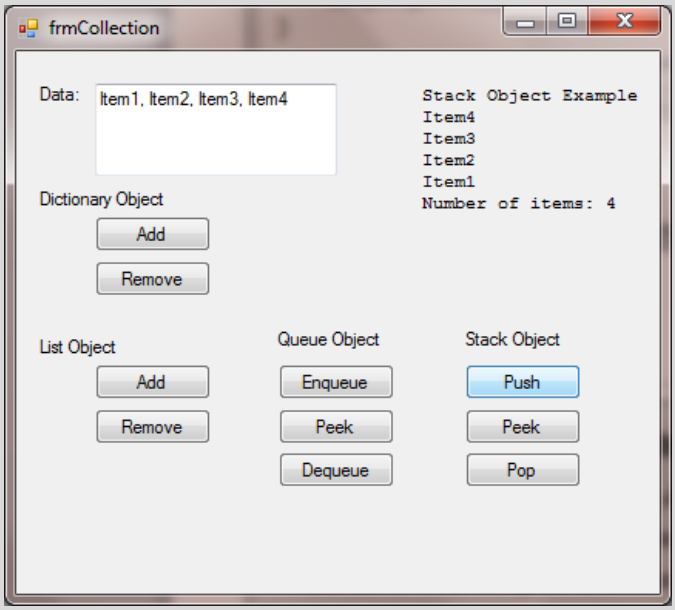


8. Now click on the Dequeue button. Now the next person is actually removed from the list.
9. Continue to click on the Dequeue button and you will see that every person is eventually removed from the list.
10. Now let us code the corresponding Stack object. All three event handlers and the output method are shown below:

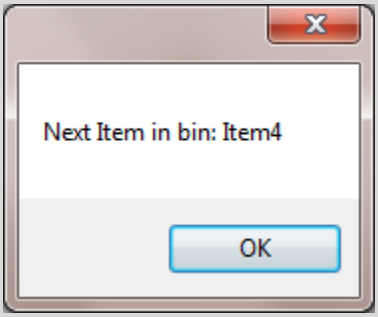
```
private void OutputStack()
{
    string Output = "Stack Object Example \n";
    foreach (string Item in Bin)
    {
        Output += Item + "\n";
    }
    Output += "Number of items: " + Bin.Count.ToString();
    lblOutput.Text = Output;
}

private void btnPush_Click(object sender, EventArgs e)
{
    string[] Input = txtInput.Text.Split(',');
    foreach (string Item in Input)
    {
        Bin.Push(Item.Trim());
    }
    OutputStack();
}

private void btnStackPeek_Click(object sender, EventArgs e)
{
    MessageBox.Show("Next Item in bin: " + Bin.Peek());
}

private void btnPop_Click(object sender, EventArgs e)
{
    Bin.Pop();
    OutputStack();
}
```

Example 4-6(continued): Using Queue and Stack Objects

11. Save and run the project. Add some last items into the Data textbox. Now click on the Push button. The output is shown on the right:



12. Click on the Peek button to see which item is the next item to be processed:



13. Now click on the Pop button to actually remove the item from the Stack.
14. Continue to click on the Pop button to eventually remove all items from the Stack.

13. Classes and Methods

Now that we have covered enough basic C# skills, it is time to put it to use. The big question remains, where are we going to put our code? So far we have mainly used event handlers, which are special methods in C#. What is called procedures, functions, sub routines, etc. in other programming languages is called collectively methods in C#. And methods are placed in classes, containers for your code.

13.1 Classes

Classes are simply contained in files. There can be more than one class in one file. Furthermore, you may spread one class among separate physical files and call them partial classes. The compiler will merge them together into one class.

Class Architecture

So far, we have used only class files that are associated with the Windows form. This class file is inherently linked to the Windows Form file. In addition to the form class file, you can create individual class files that hold code that you want to share in your application.

When building a database application, you actually want to structure your code in a distinct model as shown in Figure 58.

The presentation layer represents all the forms classes to support basic form functionality. It may also include validation methods in separate, standalone classes. The business layer may contain classes to support special business logic and business rules, such as discount or credit policies. The database layer takes care of connection to the database and saving and retrieving data.

Warning: The term layer and tier are sometimes used interchangeably. However, layer refers to the logical division and tier to the physical division. For example, we are currently developing and running the entire application on one workstation, yet we still divide up our code into different layers. In larger enterprise applications, code modules are actually distributed across physical computers, such as application server and database server. In this case you refer to tiers.

Note: Sometimes the business layer is divided up between the presentation layer and the database layer to form a two-tier architecture.

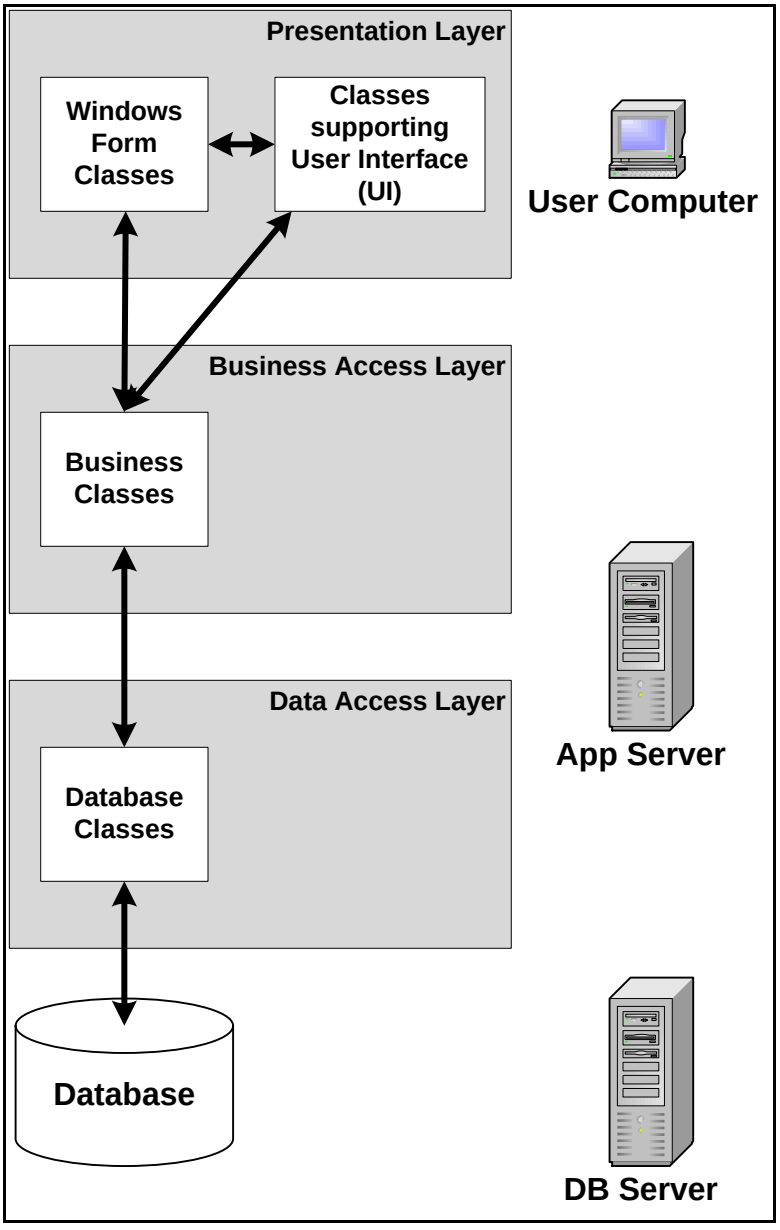


Figure 58: Three-Layer Application Architecture

The main advantage of separating the three different functionalities is maintenance purposes. All this logic could reside in one class file, but for multi-person teams it is a good idea to have separate class files. Furthermore, it supports the concept of code abstraction. Code abstraction means to break up spaghetti code into smaller manageable pieces and to move repeating logic into one central place.

Another reason is that the individual classes can be used in other applications. For example, the database class can be written a generic and dynamic way that it can be used with any application.

The last main advantage is that these classes can be placed on different servers for performance reasons. In this case, the three-layered architecture becomes a three-tier architecture.

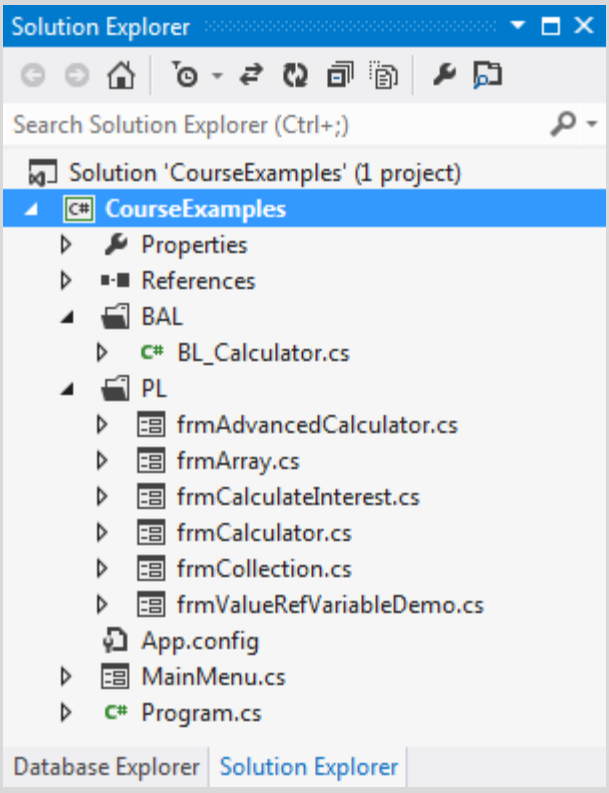
Example 4-7: Creating Business Layer Class File

1. Using the CourseExamples project, right-click on the project root in Solution Explorer and select Add → New Folder.Type PL (for Presentation Layer).
2. Repeat step 1 and create another folder and type BAL (for Business Access Layer).
3. Move all windows forms files except for the MainMenu.cs into the PL folder.
4. Right-click on the BAL folder and select Add →Class. Type BL_Calculator for the file name. The empty class file is shown on the right.
5. Your Solution Explorer should look like the one shown below.

BL_Calculator.cs | frmAdvancedCalculator.cs*

CourseExamples.BAL.BL_Calculator

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CourseExamples.BAL
{
    class BL_Calculator
    {
    }
}
```



Object-Oriented Programming in Classes

C# is an object-oriented programming language. To create an object-oriented application or program, the programmer uses C# and its concepts to build object-oriented code. The main idea is to model the real world, and the real world is made up of objects that relate to each other. We will discuss the object-oriented concepts in the following chapters.

At this point, we simply use the Classes that we create as storage containers. That means, we do not take advantage of the object-oriented features of classes yet!

Remember, we have already introduced the math class. This is a class that contains many methods to perform mathematical operations. We do not need to create an instance (=object) in order to use the methods contained in the class. This class simply acts as a storage container for defining common functionality. Such a class is called a static class.

A class definition can become quite complex. We will explore some of these class types later in the course. For the discussion of this chapter, the following class syntax will suffice:

{public | internal | static } **class** *class_name*

Classes that you declare directly within a namespace, not nested within other classes, can be either public or internal. Classes are internal by default, internal means accessible within the same assembly. Public means within the entire solution. If you entire solution is comprised of one assembly (= one project), then internal and public have the same meaning.

A static class means that only static members can be created within the class. Each individual member still must be marked as static as well.

Example:

Non-Static: **class** *Calc*{ **int** Add(**int** num1, **int** num2) }

To call the method Add you have to write the following lines of code:

Calc *Calculator* = **new** Calc();
Calculator.Add(5,3);

Static: **static class** *Calc*{ **static int** Add(**int** num1, **int** num2) }

To call the method Add you have to write only one line of code:

Calc.Add(5,3);

13.2 Methods

Methods are the lowest structures to hold your code. Methods are like procedures and functions in other programming languages, they perform some kind of evaluation and may or may not return a value to the calling program.

Methods are used to break larger program units into smaller, more manageable units.

Method Definition

There are a few modifiers for methods we need to discuss since they will impact your program significantly. Let us first look at the syntax as shown below:

public|private [static] return_type *method_name*(**type** *param_name*, ..)

First of all, there is the access modifier, that means whether the method is public or private. If you code a method as public, it will be available in your entire program. A private method is only available within the class.

The static modifier is an important one and is probably difficult to understand in the beginning. As you have already seen, objects are created from classes using the new keyword. An object is an instance of the class, one of many possible members of a class. A class can have instance methods and static methods. An instance method can only be used with an instance of the class, that means with an object. A static method can be used at the class level and does not depend on a specific instance.

Example:

Assume we have a class named Student that tracks students in a school.

Instance method:

Student student1 = new Student(); → Instantiating an object

student1.Add(name) → The Add method is an instance method, and can only be called from an object.

Static method:

Student.Count() → The Count method is a static method, it keeps track of all instances of students and knows at any point in time the number of students.

If you omit the keyword static from your method, the method becomes automatically an instance method. Adding the keyword static makes it a static method.

The return modifier void makes this method act like a procedure that does not return a value to the calling environment. If you instead use a return type, such as string or int, then the method is designed as a function and will return a value of that type.

When using a non-void method, meaning a method that returns a value, you have to use the return keyword followed by an expression inside the method to return a value or reference based on the specified type you created in the method definition.

 **Note:** It is good practice to implement the return keyword at the end of your method. Use variables to store the returning result, if needed, and then use the variable in conjunction with the return keyword at the end of the method.

You can use the return keyword alone to simply exit a method, for example, based on a certain condition. You can use the return keyword by itself even in a void method.

 **Example 4-8:** Creating a Static Method

1. Using the new BL_Calculator class file, copy the performCalculation method from the frmAdvancedCalculator.cs into the BL_Calculator file.
2. Comment out the performCalculation method in the frmAdvancedCalculator.cs file.
3. Change the access modifier private to public as shown below:

```
namespace CourseExamples.BAL
{
    class BL_Calculator
    {
        public string performCalculation(string Number1, string Number2, string Operator)
        {
            string ReturnResult = string.Empty;
            double Result = 0;
            if ("+-*/".IndexOf(Operator) > -1)
            {
                switch (Operator)
                {
                    case "+":
                        Result = Convert.ToDouble(Number1) + Convert.ToDouble(Number2);
                        break;
                    case "-":
                        Result = Convert.ToDouble(Number1) - Convert.ToDouble(Number2);
                        break;
                    case "*":
                        Result = Convert.ToDouble(Number1) * Convert.ToDouble(Number2);
                        break;
                    case "/":
                        Result = Convert.ToDouble(Number1) / Convert.ToDouble(Number2);
                        break;
                }
                ReturnResult = (Math.Round(Result, 2, MidpointRounding.AwayFromZero)).ToString("#,##0.00")
                    + " (" + Result.ToString() + ")";
            }
            else
            {
                ReturnResult = "Invalid Operator";
            }

            return ReturnResult;
        }
    }
}
```

Example 4-8 (continued): Creating a Static Method

4. Navigate back to the frmAdvancedCalculator.cs and notice the red-wavy underlined method class in the event handlers. We would like to call the performCalculation method in our BAL.
5. Click inside the bntAdd_Click event handler and create a blank line. Then type the same left side (lblResult.Text =) and then type the beginning string BL_. You may notice that the BL_Calculator class is not available.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_
    lblResult.Text =
}
private void btnSubtr
{
```

BackgroundImageLayout
BackgroundImageLayoutChanged
BAL

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using CourseExamples.BAL;
```

6. One reason is that the namespace that this method is enclosed in is not referenced in the form's class file. At the top of the class file type using "CourseExamples.BAL;"
7. Now repeat step 5, and now we can actually see and use our class BL_Calculator.
8. Now Type a dot to access the method, but we cannot see the method performCalculation.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.
    lblResult.Text = performCalcula
}
```

Equals
ReferenceEquals

9. Our method is not static, therefore we cannot simply access the method by typing the class. We need to create an instance of the class first. Type the following code as shown on the right:

```
private void btnAdd_Click(object sender, EventAr
{
    BL_Calculator Test = new BL_Calculator();
    lblResult.Text = Test.
    lblResult.Text = perfo
}
private void btnSubtract_C
{
    lblResult.Text = perfo
```

Equals
GetHashCode
GetType
performCalculation

10. But we really do not need an instance method. Let us make the method a static method. In the BL-Calculator class file, insert Static after the Public keyword for the performCalculation method.
11. Now back to the form file frmAdvamcedCalculator.cs file, modify each event handler by inserting "BL_Calculator." before the method name.
12. Save and test your project.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(txtNumber1.Text,txtNumber2.Text,"+");
}
private void btnSubtract_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(txtNumber1.Text, txtNumber2.Text, "-");
}
private void btnMultiply_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(txtNumber1.Text, txtNumber2.Text, "*");
}
private void btnDivide_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(txtNumber1.Text, txtNumber2.Text, "/");
}
```

Method Arguments/Parameters

Methods may have parameters, meaning values or references that are passed into the method. But you can also code methods having no parameters. However, you always have to code parentheses after the method name, even if you do not use any parameters.

When using parameters, you code a parameter list inside the parentheses. A parameter list is comprised of the type of the parameter followed by the name of the parameter. When using multiple parameters, you simply comma-separate the parameters. When you code the name of the method and the parameter list, you form the signature of the method.

We have already used methods with arguments in the previous examples. The method performCalc for our calculator is a good example.

public static string performCalculation(**string** Number1, **string** Number2, **string** Operator)

There are a few keywords and options before and after the method name which we will now discuss in more detail:

Before the method name:

There a couple of modifiers before the method name, which we have already discussed in the earlier section. In this particular example, the method is public and static, meaning it is accessible in the entire project and the method does not need an instance of the class in order to be called. The string keyword is the return type, meaning this method returns a string once it is done with its work.

After the method name in parentheses:

Inside the parentheses you can list the parameters. This is optional, if no parameters are needed, you still have to include the empty parentheses. For each parameter, at a minimum you have to declare the type and the name of the parameter.

 **Note:** The variable names of arguments and the names of parameters do not have to be the same. In fact, in most cases they are different.

When you call a method from within the same class, you simply use the method name followed by possible arguments. When you call a method that resides in a different class, then you use namespace, a dot operator, followed by the method name.

 **Warning:** In the method definition, the variables inside the parentheses of the method are called parameters. When you call this method and use values or references to pass into the method, they are called arguments.

The arguments are passed by ordinal position into the method, that means the first arguments is mapped to the first parameter, the second argument to the second parameter, and so on as shown in Figure 59.

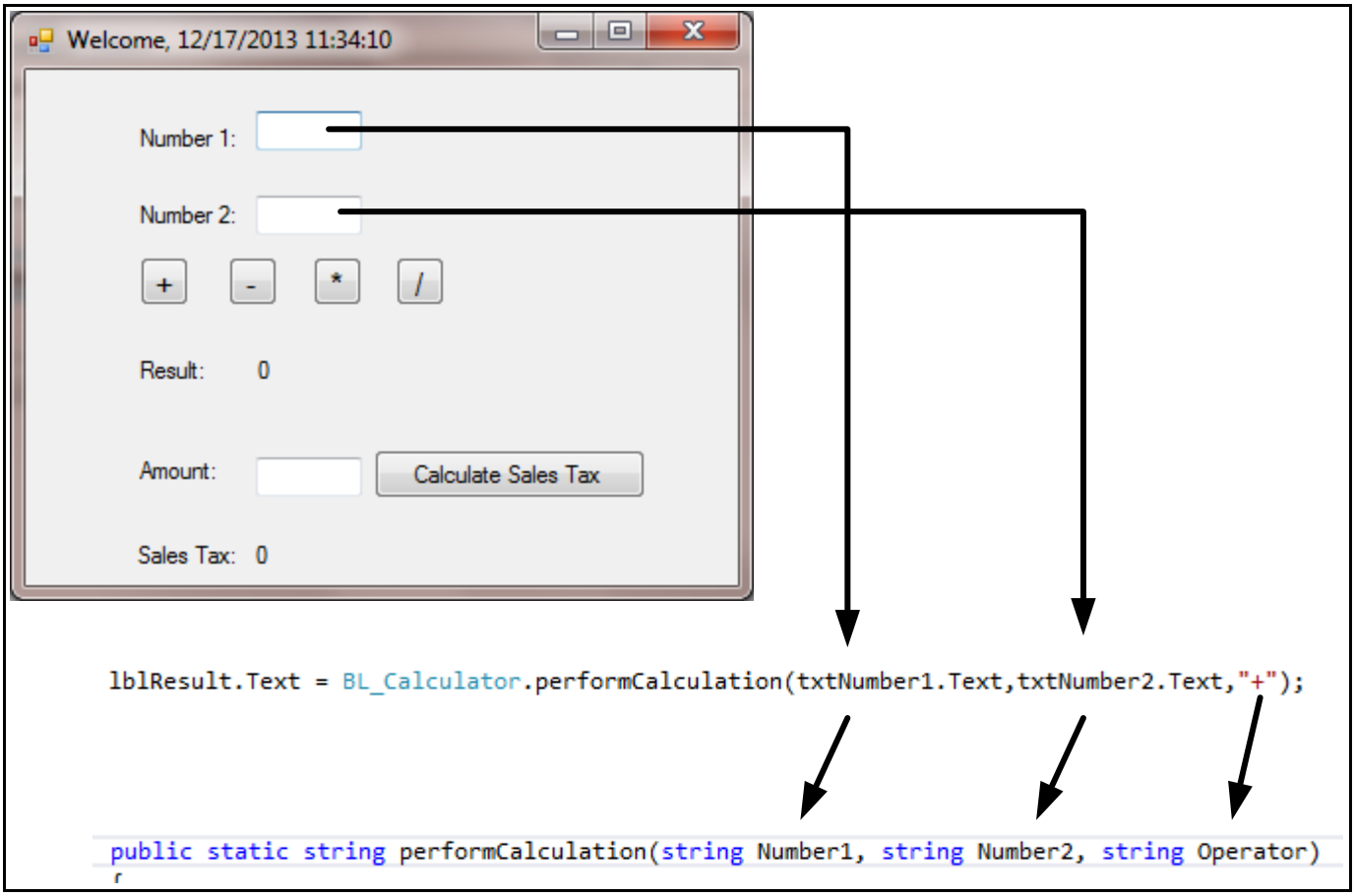


Figure 59: Passing Arguments into Method Parameters

Figure 59 shows the method call within an expression. The first two arguments are references to values in textbox controls, the third one is simply a literal value (hard-coded). These arguments are passed by ordinal position into the method. The method then uses these values and then returns a value back to the calling environment

There are two new features in C# since .NET version 4.0 in terms of parameter definition, one is optional arguments and the other one is named arguments. These improvements were added to bring C# in line with other programming languages.

You can define default values for the parameters in a method. When you call the method and do not supply a value for the default argument, the default value will be used. If you provide a value for the default argument, then the passed value will be used in the method. The syntax is shown below:

method_name (**type** param1 = default_value, **type** param1 = default_value,..)

Instead of using the positional notation of assigning arguments to parameters, you can also use the named notation. When using the named notation, the arguments passed do not have to be in the same order as the parameter list of the method. The syntax is shown below:

```
method_name (param2: arg1, param1: arg2, .. )
```

You type the parameter name (as defined in the method signature) followed by a colon, then the argument(literal value or variable). As you can see you can now refer to a parameter by using the name, so C# exactly knows for which parameter you want to assign the argument value.

 **Note:** In general, named arguments are really not needed, why would you ever want to switch the order of your arguments compared to the order of the parameters. However, when using default arguments it may make sense since you only want to possibly supply values for the non-default parameters.

There are a few additional parameter modifiers as shown in Table 29. When not specifying any parameter modifiers (None), the value variables are passed by value and not by reference, meaning you are actually copying the value into a new variable in the method. This is the default and in most cases the desired behavior. However, you can also pass value variables by reference, meaning the method points to the same value in memory as the calling method and can therefore potentially change the value.

Parameter Modifier	Description
(None)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
Out	Output parameters must be assigned by the method being called, and therefore are passed by reference. If the called method fails to assign output parameters, you are issued a compiler error.
Ref	The value is initially assigned by the caller and may be optionally reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a ref parameter.
params	This parameter modifier allows you to send in a variable number of arguments as a single logical parameter. A method can have only a single params modifier, and it must be the final parameter of the method. In reality, you may not need to use the params modifier all too often, however be aware that numerous methods within the base class libraries do make use of this C# language feature.

Table 29: Parameter Modifiers

Overloading Methods

You can code methods having the same name but different number or type of parameters. This concept is known as overloading. Other programming languages implement this concept by using optional parameters. The end result is basically the same, you want to be able to pass different types or number of parameters into a method.

When you code the calling part of an overloaded method, the tooltip box actually shows that the method is overloaded as shown in Figure 60. It further allows you to scroll through the various method definitions (method signatures) by clicking on the up and down arrows.

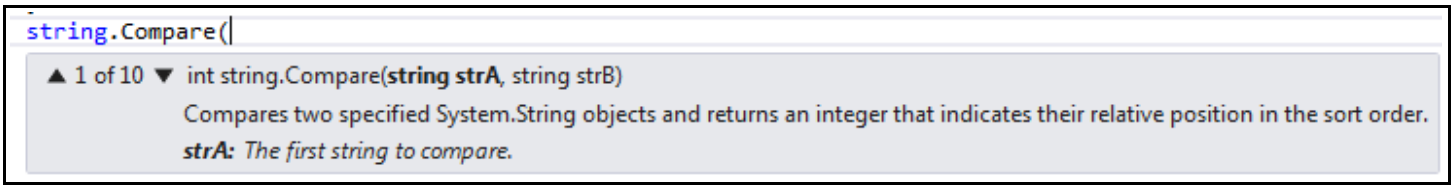


Figure 60: Tooltip Box showing Overloaded Method Signatures

Example 4-9: Creating an overloaded Method

1. Using the new BL_Calculator class file, create another "version" of the performCalculation method as shown below.

```
public static string performCalculation(string Number1, string Number2)
{
    double Result = Convert.ToDouble(Number1) + Convert.ToDouble(Number2);
    return (Math.Round(Result, 2, MidpointRounding.AwayFromZero)).ToString("#,##0.00")
           + " (" + Result.ToString() + ")";
}
```

2. We are creating a "default" operation for the method performCalculation where we do not pass the Operator into the method. This method will automatically perform an addition.
3. Go back to the frmAdvancedCalculato.cs file to the btnAdd_Click event handler. Remove the performCalc method and code it again. You see now the tool tip having two methods with the same name.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(;
}
```

▲ 1 of 2 ▼ string BL_Calculator.performCalculation(**string Number1**, string Number2)

4. The final event handler is shown below:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    lblResult.Text = BL_Calculator.performCalculation(txtNumber1.Text, txtNumber2.Text);
}
```

CLASS 5

14. Introduction to Object-Oriented Programming

This and the next chapter will cover the basic concepts of object-oriented programming. As you know by now, C# is an object-oriented programming language. That means, the language itself is based on object-oriented concepts. However, you can code and create an application using the old-fashioned procedural concepts with no object-oriented features except again for the language C# itself. C# allows you to code your application using object-oriented concepts since C# fully supports it. In the following chapters we will see the reasons for doing so and cover some examples to fully explore this new paradigm.

Note: In this chapter, I will explain the concepts of object-oriented programming using the metaphor of a microwave oven for making some Nachos (tortilla chips, cheese, beans, etc.).

14.1 Class Concept

A class defines the characteristics of an object. Characteristics include: Attributes (fields or properties), and behaviors (methods or operations). The "Oven" class could have properties such as: make, model, color, and power consumption. Behaviors of the "Oven" class include: On, Off, Bake, Keep Warm, and Set Temperature as shown in Figure 61.

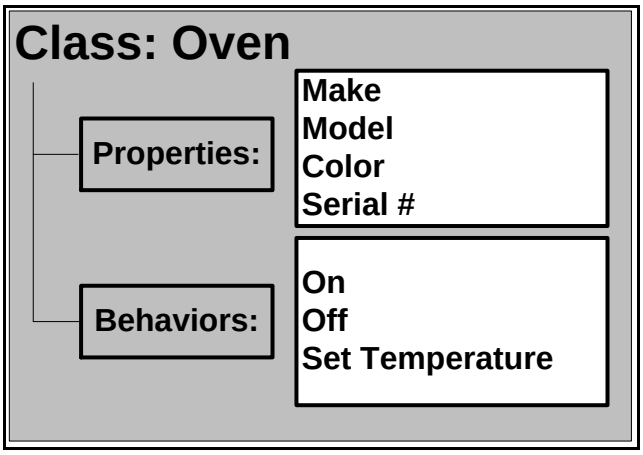


Figure 61: Class Concept

A class acts like an architectural blueprint. You form or instantiate objects from those blueprints, and they may vary in terms of color or some other properties. However, they are the same in terms of classification and behavior.

14.2 Object Concept

An object is an instance of a class. Creating an object is also known as instantiation. For example, the object "MyOven" is an instance of the Oven class. This object is determined by specific values of the properties defined in the class, such as Make and Model as shown in Figure 62.

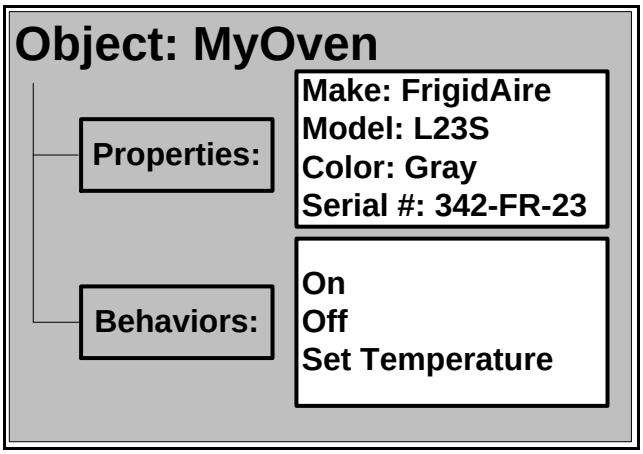


Figure 62: Object Concept

14.3 Method Concept

A method is a behavior of an object. Within a program, a method usually affects only one particular object. In our example, all ovens can bake, but the program only needs to make "MyOven" bake. This is also called an instance method.

There can be also class methods, also called static methods. In our example, the manufacturer probably wants to know how many ovens of a particular model have been produced.

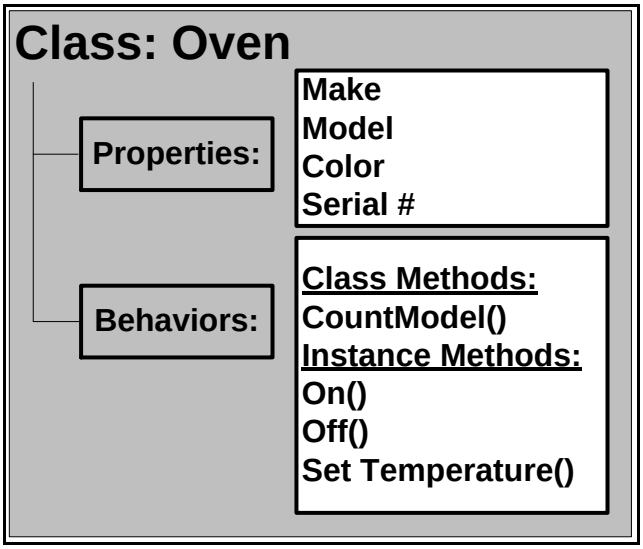


Figure 63: Method Concept

14.4 Message Passing Concept

Message passing (or method calling) is the process where an object sends data to another object to invoke a method. For example, when the object called "Mary" (an instance of the user class), presses the On button, she literally passes a turn on message to object "MyObject", which in turn, invokes the "MyOven" On() method.

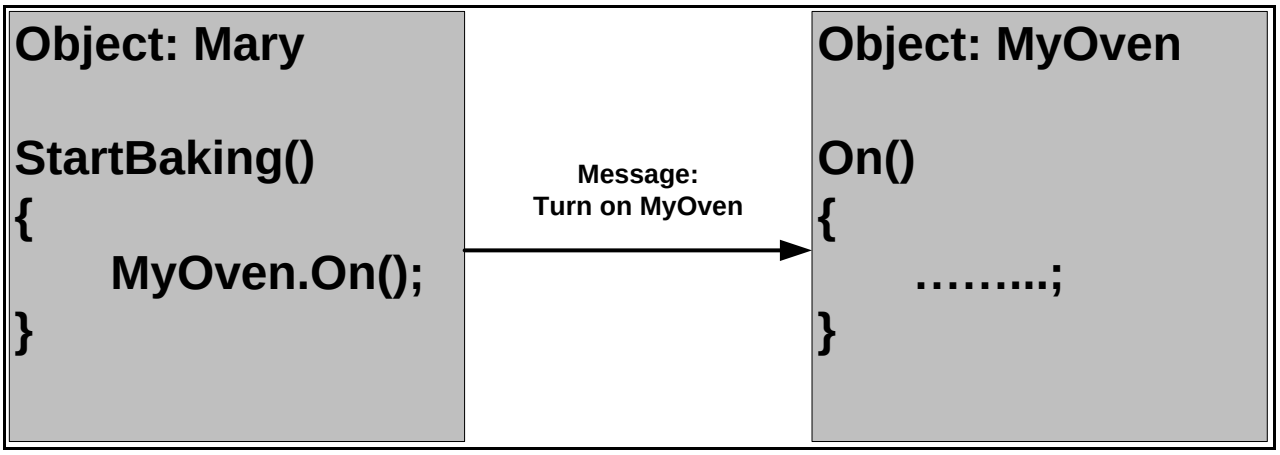


Figure 64: Message Passing Concept

In Figure 64 an instance of the User class named "Mary" sends a message to the instance "MyOven" of the Oven class. The User class contains a method StartBaking() which in turn contains a call to the method On which is part of the Oven class.

14.5 Classification Concept

Critical to the concept of abstraction is that of classification. "What is a microwave?" well it is an oven that. . . . Now you can go on, "What is an oven?" It is a kitchen appliance that. . . .And what is a kitchen appliance? And so on and so on as shown in Figure 65.

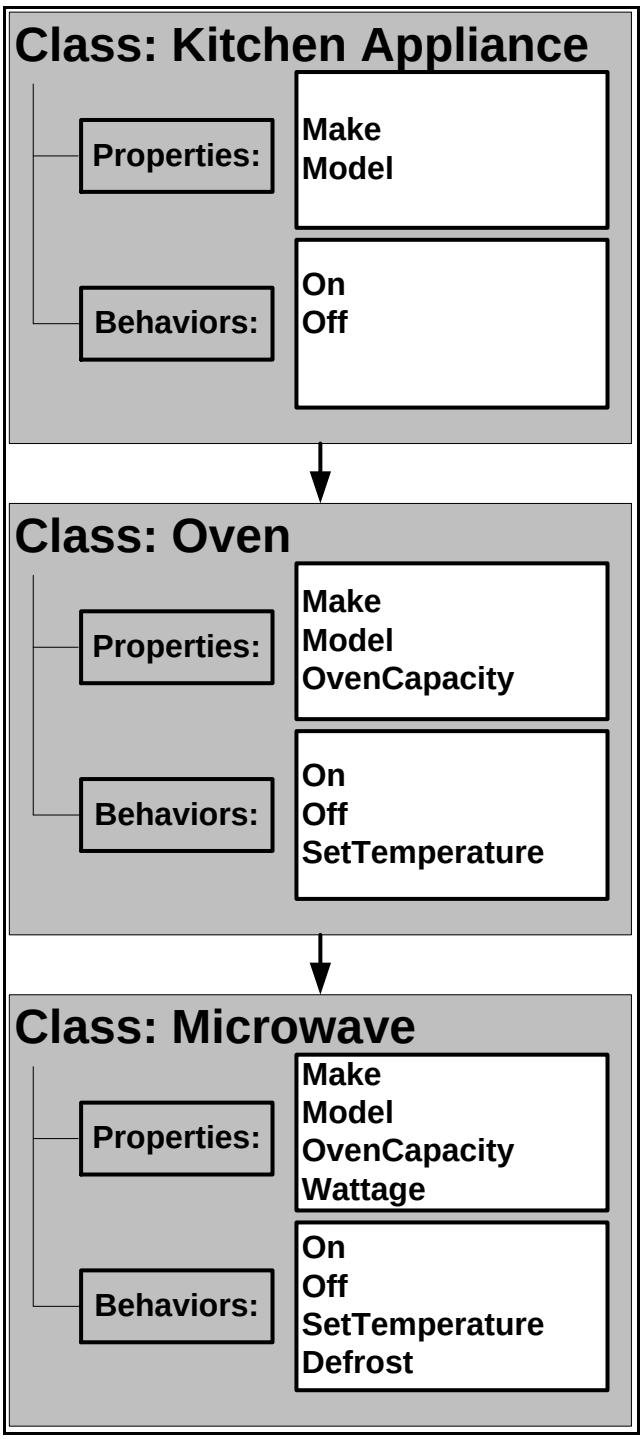


Figure 65: Classification Concept

As you can see, this particular microwave is an example of the type (=class) of item known as a microwave oven. In addition, a microwave oven is just a special type of oven, which itself is just a special type of kitchen appliance.

Humans classify. Everything about our world is ordered into taxonomies. We do this to reduce the number of items we have to remember. When thinking or dealing with a microwave oven, you just want to use the particular features of a microwave oven, and not the common features of kitchen appliances or ovens. You see here the close connection between abstraction and classification.

Why classify? It sounds like a lot of trouble. Besides, people have been using the procedural approach for a long time — why change now? Designing and building a microwave oven specifically for this problem may seem easier than building a separate, more generic oven object. Suppose that you want to build a microwave oven to cook only nachos. You would not need to put a front panel on it, other than a Start button. You probably always cook nachos for the same length of time. You could dispense with all that Defrost and Temp Cook nonsense in the options. The oven needs to hold only one flat, little plate. Three cubic feet of space would be wasted on nachos.

For that matter, you can dispense with the concept of "microwave oven." All you need is the guts of the oven. Then, in the recipe, you put the instructions to make it work: "Put nachos in the box. Connect the red wire to the black wire. Bring the radar tube to about 3,000 volts. And so on and on.

But the procedural approach has these problems:

- **It is too complex.** You do not want the details of oven-building mixed into the details of nacho-building. If you cannot define the objects and pull them from the morass of details to deal with separately, you must deal with all the complexities of the problem at the same time.
- **It is not flexible.** Someday, you may need to replace the microwave oven with another type of oven. You should be able to do so as long as the two ovens have the same interface. Without being clearly delineated and developed separately, one object type cannot be cleanly removed and replaced with another.
- **It is not reusable.** Ovens are used to make lots of different dishes. You do not want to create a new oven every time you encounter a new recipe. Having solved a problem once, you want to be able to reuse the solution in other places within my program. If you are lucky, you may be able to reuse it in future programs as well.

14.6 Abstraction Concept

Sometimes I feel like making some nachos (I know that this not very healthy). I put some tortilla chips on a plate, throw on some beans and cheese and lots of jalapeños, and cook the whole mess in the microwave oven for a few minutes.

To use my microwave, I open the door, throw in the plate of food, and punch a few buttons on the front. After a few minutes, the nachos are done. So, what is the point? Let us think what I do not have to do to operate the microwave oven:

- Rewire or change anything inside the microwave to get it to work. The microwave has an interface comprised of the front panel with all the buttons and the timer display
- Reprogram the software used to drive the little processor inside the microwave, even if I cooked a different dish the last time I used the microwave.
- Look inside the microwave's case.

Even if I were a microwave designer and knew all about the inner workings of a microwave, including its software, I still would not think about all those concepts while using it to heat nachos.

Let us translate this into an object-oriented programming context: To reduce the complexity (humans can only deal with a few number of things at once), you work at a certain level of detail. In object-oriented (OO) context, the level of detail at which you are working is the level of abstraction.

 **Note:** Abstraction is actually a concept that has already been used in the computer programming world for some time, it is called modularization (to break up a large program into smaller components). The object-oriented programing framework enforces abstraction at a stricter level.

Using powerful abstractions makes the job simpler and far less error-prone than it used to be. In a sense, that is what the past half-century or so of computing progress has been about: managing ever more complex concepts and structures with ever fewer errors.

When I am making my nachos, I consider the microwave oven as a box. As long as I use the microwave only by way of its interface (the keypad), nothing I can do should cause the microwave to enter an inconsistent state and crash (turn the nachos into a black mass).

In an object-oriented approach to making nachos, you first identify the types of objects in the problem: chips, beans, cheese, jalapeños, and an oven. Then you begin the task of modeling those objects in software, without regard for the details of how they might be used in the final program.

For example, you can model cheese as an object in isolation from the other objects and then combine it with the beans, the chips, the jalapeños, and the oven and make them interact. (And you might decide that some of these objects do not need to be objects in the software: cheese, for instance.)

While you do that, you are said to be working (and thinking) at the level of the basic objects. You need to think about making a useful oven, but you do not have to think about the logical process of making nachos — yet. After all, the microwave designers did not think about the specific problem of you making a snack. Rather, they set about solving the problem of designing and building a useful microwave.

After you successfully code and test the objects you need, you can move up to the next level of abstraction and start thinking at the nacho-making level rather than at the microwave-making level.

Procedural programmers live in a world devoid of objects such as microwave ovens and other appliances. They tend to worry about flowcharts with their complex procedural paths. In a procedural solution to the nachos problem, the flow of control would pass through my finger to the front panel and then to the internals of the microwave. Then, flow would pass through complex logic paths about how long to turn on the microwave tube and eventually to sound a tone when the nachos are done.

In that world of procedural programming, you cannot easily think in terms of levels of abstraction. You have no objects and no abstractions behind which to hide inherent complexity.

14.7 Interfaces

An object must be able to project an external interface that is sufficient and as simple as possible. If it is not, users may start ripping the top off the device. Not only is the warranty voided, most likely the device will break and not function properly. Furthermore, if the interface is too complex, no one will buy the device or will not use all the features.

The remote control of our electronic devices found in many homes is a good example. It is pretty complex (too many buttons, multi-function buttons, and small buttons) and different products have differently designed remote controls. For whatever reason, today's remote controls project an interface that is too difficult and too nonstandard for most people to use beyond the basics.

On the other hand, a good example of a usable interface is the automobile. In the old days, we had two or three pedals (stick shift), steering wheel, ignition, levers on the left and right side of the steering column, and various other controls such as buttons. Now beam yourself to today, and sit in a modern car (hybrid, electric). You most likely would be able to operate the car since you are used to the interface. This is pretty amazing and powerful, think about it. Granted, there are lots of new controls and new features nowadays, but you would be able to drive the car safely knowing the interface of a VW Bug or a Ford Model T.

The same is true in the object-oriented programming world. You want to design interfaces that are simple and do not change. An interface of an object is for example the signature of method, that is the method name and its parameters. The inner workings might change, but as long as the interface does not change, your program will still work.

14.8 Encapsulation

A microwave oven must be built so that no combination of keystrokes that you can enter on the front keypad can cause the oven to hurt you. Certainly, some combinations do not do anything. However, no sequence of keystrokes should:

- Break the device: You may be able to put the device into a strange state in which it does not do anything until you reset it (say, by throwing an internal breaker). However, you should not be able to break the device by using the front panel.
- Cause the device to catch fire and burn down the house: As bad as it may be for the device to break itself, catching fire is much worse.

However, to enforce these two rules, you as a user have to take some responsibility. You cannot make modifications to the inside of the device. Almost all kitchen devices of any complexity, including microwave ovens, have a small seal to keep consumers from reaching inside them. If the seal is broken, indicating that the cover of the device has been removed, the manufacturer no longer bears responsibility. If you modify the internal workings of an oven, you are responsible if it subsequently catches fire and burns down the house.

Encapsulation is a way of organizing data and methods into a structure by concealing the way the object is implemented, i.e. preventing access to data by any means other than those specified. Encapsulation therefore guarantees the integrity of the data contained in the object.

The user of a particular class does not need to know how the data in that object is structured, this means that a user does not need to know how implementation takes place. By preventing the user from directly modifying attributes, and forcing the user to use defined functions in order to modify them (called interfaces), data integrity is thus ensured (for example, one can ensure that the data type given matches expectations, or is returned within the expected time interval).

Encapsulation defines the access levels for elements of that class. These access levels define the access rights to the data, allowing us to access the data by a method of that particular class itself, from an inheritance class, or even from any other class.

Encapsulation protects the integrity of an object by preventing users from changing internal data into something invalid. Encapsulation reduces system complexity and thus increases robustness, by limiting inter-dependencies between components.

Encapsulation lets you control the data and operations within a class that are exposed to other classes. The data of a class is usually encapsulated within a class using data hiding. Furthermore, code that performs operations within the class is encapsulated so it can be changed without changing the way other classes use it.

14.9 Polymorphism

Polymorphism is closely related to Inheritance. Using inheritance, the subclass (derived class) inherits or adopts the members of the superclass (base class). Using Polymorphism, the subclass that inherits can now override or further define one or more members of the superclass.

Back to our microwave metaphor, a microwave oven is a type of oven, not because it looks like an oven but, rather, because it performs the same functions as an oven. A microwave oven may perform additional functions (classification, abstraction), but it performs, at the least, the base oven functions — most importantly, making my nachos when I press the appropriate button. I do not particularly care what the oven must do internally to make that happen, any more than I care what type of oven it is, who made it, or whether it was on sale when my wife bought it.

From our human vantage point, the relationship between a microwave oven and a conventional oven does not seem like such a big deal, but consider the problem from the oven's point of view. The steps that a conventional oven performs internally are completely different from those that a microwave oven may take.

Polymorphism enables us to "program in the general" rather than "program in the specific. This means, in the base class we program a general method of cooking/baking. In the derived class, in our example the microwave oven, we can now provide the details how the microwave performs the cooking/baking operation.

14.10 C#'s Implementation of Object-Oriented Concepts

How does C# implement object-oriented programming? In a sense, this is the wrong question. C# is an object-oriented language; however, it does not implement object-oriented programming — the programmer does. You can certainly write a non-object-oriented program in C# or any other language.

Warning: The object-oriented approach to solving common real world or business problems is not always appropriate. There are different kind of approaches (Procedure-Oriented(Algorithmus), Logic-Oriented(Goals, often expressed in a predicate calculus), Rule-Oriented, Constraint-Oriented)

These C# features are necessary for writing object-oriented programs:

Encapsulation: C# controls the way in which class members can be accessed. C# keywords enable you to declare some members wide open to the public whereas internal members are protected from view and some secrets are kept private.

Classification: C# supports classification (specialization) through a mechanism known as class inheritance. One class inherits the members of another class. For example, you can create a Car class as a particular type of Vehicle. The Vehicle class is the base or super class, whereas the Car class is the derived or sub class.

Warning: In C#, you can only inherit from one class. However, you can implement multiple interfaces.

Polymorphism: This feature enables an object to perform an operation the way it wants to. The Rocket type of Vehicle may implement the Start operation much differently from the way the Car type of Vehicle does. At least, I hope it does every time I turn the key in my car. But all Vehicles have a Start operation, and you can rely on that. In C#, you design a virtual method in the base class which you can then override in the derived class.

Interfaces: Objects frequently use the services of other objects — by calling their public methods. But classes can "know too much" about the classes they use. The two classes are then said to be "too tightly coupled," which makes the using class too dependent on the used class. The design is too brittle — liable to break if you make changes. But change is inevitable in software, so you should find more indirect ways to connect the two classes. That is where the C# interface construct comes in.

15. Object-Oriented Programming in C#

An object-oriented program may be considered a collection of interacting objects. Each object is capable of sending and receiving messages, and processing data.

15.1 Classes in C#

We have already discussed the basic structure and syntax of a class, simply to be able to write some code using static methods. Now we will look at the class structure in more detail with regards to object-oriented programming.

Classes contain members, such as properties, methods, and events. However, there a quite a few more class members as shown in Table 30.

Class Member	Description
Field	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class.
Method	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Constructor	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Property	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Destructor	Destructors are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.
Event	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Constant	Constants are fields or properties whose value is set at compile time and cannot be changed.
Indexer	A special type of property that allows individual items within the class to be accessed by index values. Used for classes that represents collections of objects.
Operator	Overloaded operators are considered class members. When you overload an operator, you define it as a public static method in a class.
Nested Class	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

Table 30: Class Members

The bolded members in Table 30 (first four) are the most common ones.

The constructor and property member are specialized versions of a method, we will discuss this in the coming chapter. The core syntax for a class is shown below:

```
[attributes] [class modifiers] [partial] class class_name [(parameters)][: base class[(parameters]]
{
        //Fields
        //Constructors
        //Methods
}
```

An attribute is a declarative tag that is used to convey information to the runtime engine about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc., in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for. There are built-in attributes and you can also build custom attributes. For example, the Obsolete attribute allows you to mark a class or method as obsolete, the compiler will generate a warning.

Classes in C# have various modifiers to set a specific behavior for a given glass. There are access and other modifiers. The access modifiers (encapsulation) and are shown in Table 31.

Class Access Modifier	Description
public	Unlimited access, the class is available in the entire project.
private	This applies only to nested or inner classes, the class is only accessible from within the main class.
protected	The class is protected from other classes, only classes that inherit from this class have access.
internal	The class may only be accessed from the same assembly
protected internal	The only allowed combination of access modifiers, access through the same assembly or parent and child classes.

Table 31: Class Access Modifiers

The additional class modifiers are shown in Table 32.

Class Modifier	Description
abstract	An abstract class acts only as a base class for other subclasses. This type of class can never be instantiated into an object. (Example Geometric Shape class → Rectangle class, Circle Class, etc.)
sealed	A sealed class cannot have any subclasses, meaning no other class can inherit from a sealed class. Based on this definition a sealed class can never be also an abstract class.
static	A static class can never be instantiated. You simple use the class name, dot operator, followed by its properties or methods (Example: Math class, Math.Round())

Table 32: Class Modifiers

15.2 Objects in C#

An object is a self-contained unit that has properties, methods, and other members. An object is an instance of a class, and the process of creating an object is called instantiation. To instantiate an object in C#, you use the new keyword as shown below:

class_name object_name = **new** *class_name*(*[parameters]*);

Example:

To create an object of type Windows Form(in our example form Calculator), write the following code:

frmCalculator *frm* = **new** frmCalculator();
frm.Show();

Using the object variable frm, which points to the form frmCalculator in memory, you can now use all of the properties and methods that are defined in the frmCalculator class. The Calculator class, in turn, is derived from the generic Form class.

Warning: You can only instantiate non-static classes.

The intelli-sense window now shows all the properties and methods (and more) that are available to you based on the general windows form class as shown in Figure 66.

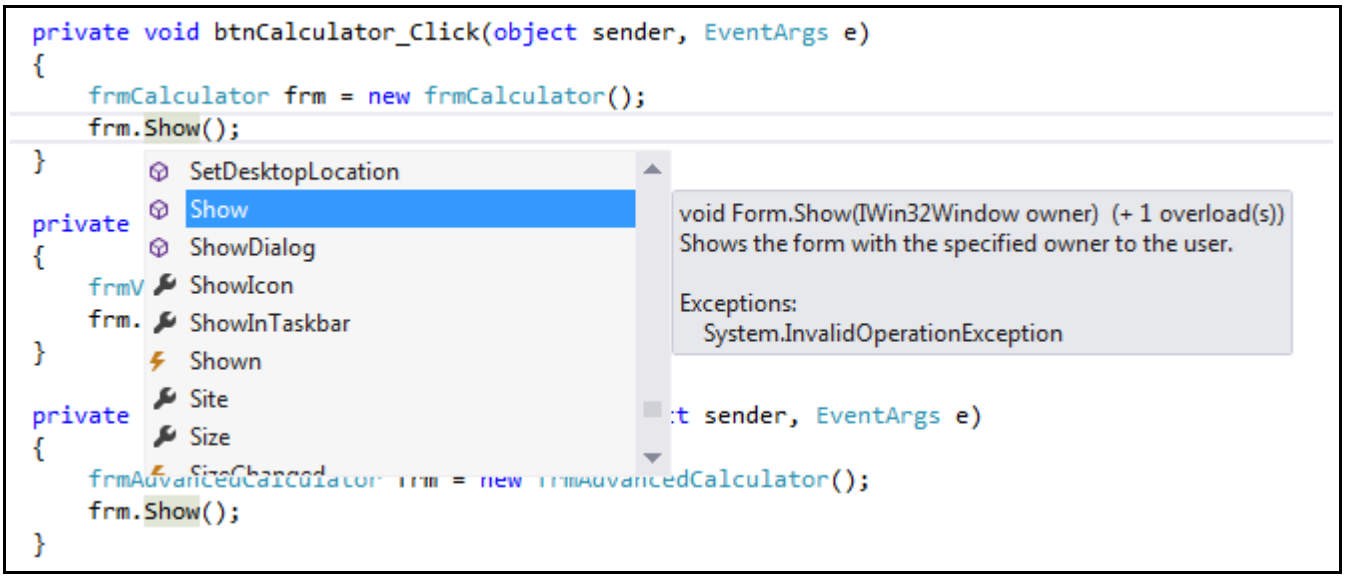


Figure 66: Form's Properties, Methods, and Events

15.3 Constructors in C#

A constructor works exactly like a method, the only difference is that you cannot call it directly. It is inherently called when you instantiate an object from a class using the new keyword. The name of the constructor method is the same name as the class name, that inherently makes it a constructor method. In general, you initialize the fields in this method, but you could also perform other functional logic here.

Constructors work together with the private fields in a class. Private fields are variables inside a class. Shown below is the basic syntax:

```
class class_name
{
        private type  _variable_name1
        private type  _variable_name2

        class_name(){}  //This is an empty constructor

        class_name(type variable1, type variable2) //custom constructor
        {
                _variable_name1 = variable1;
                _variable_name2 = variable2
        }
}
```

 **Note:** If you do not provide a constructor method, the compiler will automatically add a default constructor and initialize all fields to their appropriate values based on their type.

The syntax example above shows two constructor methods. The first one is an empty constructor, it really does not do anything except for providing memory space on the heap. Depending on the type of the class variables the default values will be assigned to the private fields.

The second constructor is a custom constructor where you pass in values when you instantiate an object from this class as shown below:

```
class_name object_name = new class_name(variable1_value, variable2_value);
```

A constructor is a special method exhibiting the following characteristics:

- The constructor always carries the same name as the class
- The constructor can take parameters
- The constructor never has a return type, not even void
- The constructor cannot be invoked directly, it is executed when an object is instantiated

Example 5-1: Creating a Rectangle Class

1. Using the CourseExamples project, right-click on the BAL folder and add a new class. Name the class file Rectangle.cs.
2. Rename the namespace to CourseExamples.Shapes.Rectangles and the class to clsRectangle.
3. Add the namespace System.Drawing at the top of the file.
4. Create the following members as shown below:

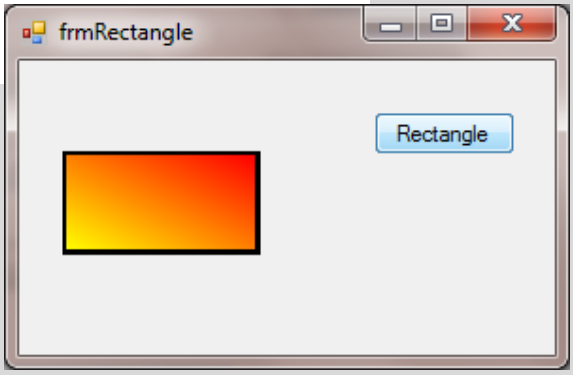
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace CourseExamples.Shapes.Rectangles
{
    public class clsRectangle
    {
        //Private Fields
        private Size _size;
        private Point _xy;
        // Default Constructor
        public clsRectangle()
        {
        }
        //Overloaded Constructor
        public clsRectangle(int x, int y, int width, int height)
        {
            _xy.X = x;
            _xy.Y = y;
            _size = new Size(Math.Abs(width), Math.Abs(height));
        }
        //Methods
        public void SetCoordinatesSize(int x, int y, int width, int height)
        {
            _xy.X = x;
            _xy.Y = y;
            _size.Width = width;
            _size.Height = height;
        }
        public Rectangle Render()
        {
            Rectangle rect = new Rectangle(_xy.X, _xy.Y, _size.Width, _size.Height);
            return rect;
        }
    }
}
```

5. Note that the type Rectangle is a built-in class in C# to handle a rectangle object.

Example 5-1(continued): Creating a Class

6. The fields (private fields) are all prefixed with an underscore, and they represent the coordinates (_xy) using a C# Point struct representing the top left corner and the dimensions (_size) of the rectangle using a C# Size struct.
7. The default constructor is overloaded with additional parameters to set the coordinates and the dimensions of the rectangle.
8. There are two methods, one to set the coordinates and the dimensions and the other one to create a C# rectangle object and return it to the caller.
9. Add a new form to the project and name it frmRectangle. Place a button in the right, top corner and set its Text property to Rectangle. Name the button btnRectangle.
10. Double-click on the button and add the following code to draw rectangle. Also make sure to add the namespaces CourseExamples.Shapes.Rectangles and System.Drawing.Drawing2D with a Using statement and to remove the .PL extension from the namespace (just CourseExamples)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using CourseExamples.Shapes.Rectangles;

namespace CourseExamples
{
    public partial class frmRectangle : Form
    {
        public frmRectangle()
        {
            InitializeComponent();
        }

        private void btnRectangle_Click(object sender, EventArgs e)
        {
            //Setting up graphing objects
            Graphics canvas = this.CreateGraphics();
            Pen pen = new Pen(Color.Black, 5);
            //Creating C# rectangle object
            Rectangle rectangle;
            //Instantiating clsRect object
            clsRectangle rect = new clsRectangle(25,50,100,50);
            //Using render method to return C# rectangle object
            rectangle = rect.Render();
            //Drawing Rectangle
            canvas.DrawRectangle(pen, rectangle);
            //Filling Rectangle
            LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
            canvas.FillRectangle(lBrush, rectangle);
        }
    }
}
```



11. Using the main menu form, add a button and name it btnRectangle. Set its Text property to Draw Rectangle. Add the usual two lines of code to show the form frmRectangle.
12. Save and run the project. Test the button to create a rectangle object and to draw it onto the form.

15.4 Methods and Properties

We have already covered methods, but mostly for simply writing some code to explore the C# language. Methods in classes are not different, however, there are a few details we need to cover, especially when it comes to properties.

First of all, the constructor and the destructor are special methods that cannot be invoked directly.

Other methods perform operations inherently tied to the class, meaning to the real-world object.

Finally, there are methods that either read and return the values of the private fields (get methods) or write (assign) new values to the private fields. The reason being for the convoluted way of having private fields and creating methods to either read and/or write to the private fields is part of the encapsulation concept.

Before blindly accepting values from outside the class you want to perform some validation. In the previous example, you can see that the width and length values are "converted" to absolute values in case a negative value was passed (which would not make sense).

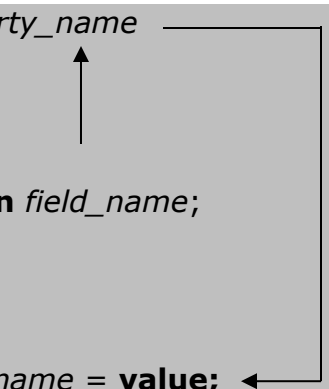
When reading values from private fields, you may want to format the data for output. For example, when performing calculations, you may want to round or display only a certain number of digits. Internally, you want to use the raw number, but for output purposes you want to format the data.

The private fields are the data fields of a class that make up the characteristics of the class. The data that determine the characteristics of a class are also called properties. The properties are exposed to the outside world using special methods, the accessor (get method or "getter") and mutator (set method or "setter") methods.

The role of a "get" method is to return to the caller the current value (maybe formatted) of the underlying state data. A "set" method allows the caller to change the current value of the underlying state data, so long as the defined business rules are met.

Since a class may contain many properties, writing a get and set method for each property would result in a huge number of such methods. Therefore, C# has defined special construct known as property which contains set and get methods for the private fields. The syntax is shown below:

```
public type property_name
{
        get
        {
                return field_name;
        }
        set
        {
                field_name = value;
        }
}
```



This is the simplest way to code a property. As you can see, there are two parts within a property, a get accessor and a set mutator. The get accessor simply returns the current value of its current field. The set mutator uses the implicit parameter value, which represents the value of the property that was passed into this property to assign a value.

The field name (see Example 5-1) usually starts with an underscore and then contains the name of the property. This convention is used to make it easier to differentiate between the property, which is public, and the field, which is private. The field is only exposed through the means of a property, meaning no one can access or manipulate (mutate) the fields directly from the outside but through properties.

The set mutator can be more elaborate, for example, it could include data validation or some other processing of data. However, in many situations it is simply just this one line, setting the value. The get accessor is usually just this one statement, to return the current value of the private field. Therefore, these statements can be much more simplified in C#. This comes in handy, especially when you have many properties to code. The syntax is shown below:

```
public type property_name
{
        get{ return field_name;}
        set {field_name = value;}
}
```

Since C# 2008 (.NET 3.5) a yet simpler approach was introduced to code properties:

public type *property_name* { **get**; **set**;}

Warning: Using the auto-implemented property causes C# to create the fields internally. Therefore, you can only access these fields through the property and cannot add any additional logic or processing. Furthermore, you must include both get and set accessor.

A property that has both get and set accessor is called a read/write property. A property that just has the get accessor is named a read-only property, and correspondingly, a property that just has the set accessor is called a write-only property.

To interact with these properties, you simply assign these properties in an expression to another variable of form control. Or you set the property value by assigning it a value in your code as shown below:

Setting a property value:

class_name.*property_name* = *variable_name/expression*;

Reading a property value:

type *variable_name* = *class_name*.*property_name*;

In the next example, we will rewrite the method SetCoordSize into properties. Furthermore, we will make some other modifications as listed below:

- Add four textbox and label controls to the form to interactively set the x and y coordinates and the width and the height of the rectangle.
- Add a RectangleShape to the form in which we will draw the rectangle rather than the entire form area.

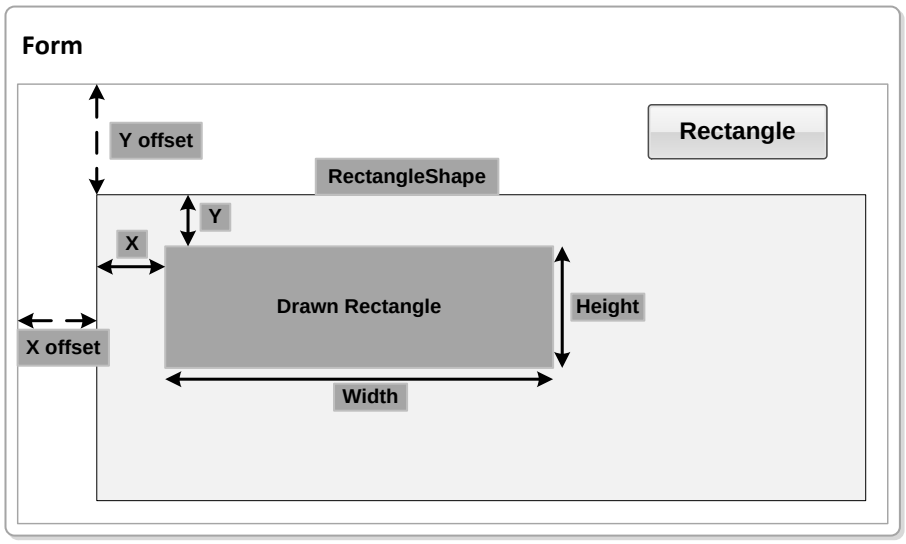


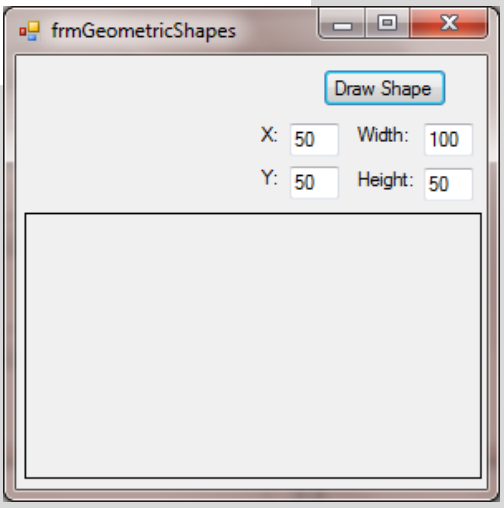
Figure 67: Rectangle Form Drawing Coordinates and Dimensions

Example 5-2: Creating Properties for Rectangle Class

1. Add a new class file in the BAL folder and name it GeometricShapes.cs.
2. Add the System.Drawing and System.Drawing.Drawing2D namespaces.
3. Rename the namespace to CourseExamples.Shapes.Classes.
4. Based on the previous rectangle class, write a new, modified rectangle class as shown below.

```
public class clsRectangle
{
    //Private Fields
    private Size _size;
    private Point _xy;
    // Default Constructor
    public clsRectangle()
    {
    }
    //Overloaded Constructors
    public clsRectangle(int x, int y)
    {
        _xy.X = x;
        _xy.Y = y;
    }
    public clsRectangle(int x, int y, int width, int height)
    {
        _xy.X = x;
        _xy.Y = y;
        _size = new Size(Math.Abs(width), Math.Abs(height));
    }
    //Properties
    public Point XY
    {
        get { return _xy; }
        set { _xy = value; }
    }
    public Size RectangleSize
    {
        get { return _size; }
        set { _size = new Size(Math.Abs(value.Width), Math.Abs(value.Height)); }
    }
    //Methods
    public Rectangle Render(Point offset)
    {
        Rectangle rect = new Rectangle(_xy.X + offset.X, _xy.Y + offset.Y, _size.Width, _size.Height);
        return rect;
    }
}
```

5. Add a new Windows Form to the PL folder and name it frmGeometricShapes. Rename the namespace to CourseExamples.
6. Add the namesapace CourseExamples.Shapes.Classes and System.Drawing.Drawing2D.
7. Add four textbox and label controls as shown on the right. Name the textbox controls txtX, txtY, txtWidth, and txtHeight. Set the Text properties as shown.



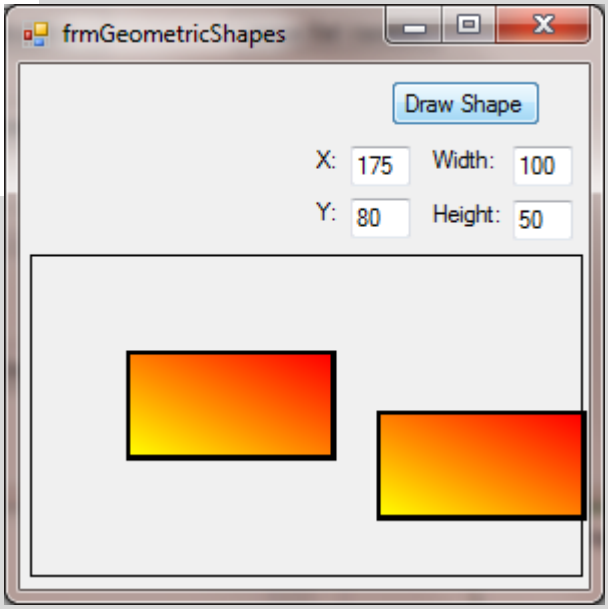
Example 5-2 (continued): Creating Properties for Rectangle Class

8. Add a RectangleShape from the Visual Basic PowerPacks category. Set the location X property to 5 and the Y property to 95 (approximately). Name it rctForm.
9. In the frmGeometricShapes.cs file, create a method named DrawRectangle.

```
private void DrawRectangle(Graphics canvas, Pen pen)
{
    //Creating C# rectangle object
    Rectangle rectangle;
    //Instantiating clsRect object
    clsRectangle rect = new clsRectangle(Convert.ToInt32(txtX.Text), Convert.ToInt32(txtY.Text));
    rect.RectangleSize = new Size(Convert.ToInt32(txtWidth.Text), Convert.ToInt32(txtHeight.Text));
    //Using render method to return C# rectangle object
    rectangle =  rect.Render(rctForm.Location);
    //Drawing Rectangle
    canvas.DrawRectangle(pen, rectangle);
    //Filling Rectangle
    LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
    canvas.FillRectangle(lBrush, rectangle);
}
```

10. Double-click on the Draw Shape button and create the following event handler.

```
private void btnDrawShape_Click(object sender, EventArgs e)
{
    //Setting up graphing objects
    Graphics canvas = this.CreateGraphics();
    Pen pen = new Pen(Color.Black, 5);
    DrawRectangle(canvas, pen);
}
```

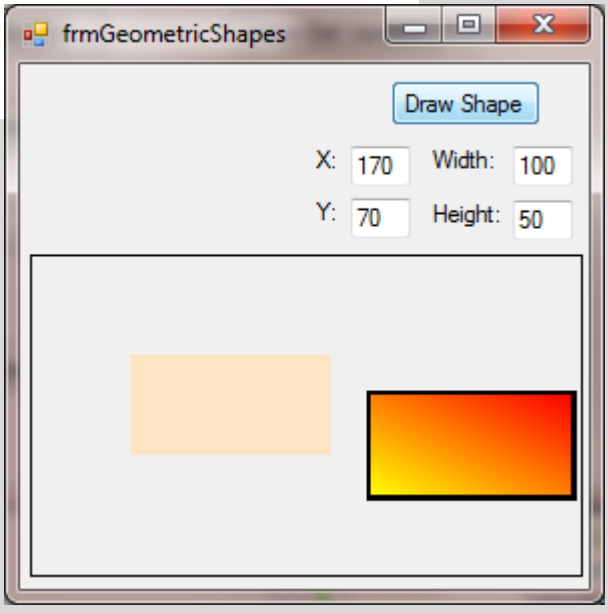


11. Save and run the project.
12. Click on the Rectangle button to create one rectangle using the default values (x,y, width and height).
13. Change the default values and then click on the Rectangle button again. Notice that a second rectangle has been drawn.
14. Whenever you click on the Rectangle button you create another instance of the rectangle class. And since you only have one reference variable (rect) you can not point anymore to the previous instances. But they still exist in memory, yet they are unreachable.
15. Even the current instance is unreachable since the reference variable is declared inside the method DrawRectangle(). Once the method is finished executing, the variable is out of scope.

Example 5-2 (continued): Creating Properties for Rectangle Class

16. In order to keep track of all the various instances of rectangles (whenever the button is clicked), we will create a list collection object to store all instances of rectangles. We then change the fill color of all "old" rectangles to a solid color and just use the gradient fill color for the latest instance. Modify the frmGeometricShapes.cs as follows.

```
public partial class frmGeometricShapes : Form
{
    //Collection object to store all shape instances
    List<clsRectangle> Rectangles = new List<clsRectangle>();

    private void DrawPreviousObjects(Graphics canvas)
    {
        //Loop through all existing shape objects and change fill color
        foreach (clsRectangle rct in Rectangles)
        {
            canvas.FillRectangle(new SolidBrush(Color.Bisque), rct.Render(rctForm.Location));
        }
    }

private void DrawRectangle(Graphics canvas, Pen pen)
{
    //Creating C# rectangle object
    Rectangle rectangle;
    //Instantiating clsRect object
    clsRectangle rect = new clsRectangle(Convert.ToInt32(txtX.Text), Convert.ToInt32(txtY.Text));
    rect.RectangleSize = new Size(Convert.ToInt32(txtWidth.Text), Convert.ToInt32(txtHeight.Text));
    //Using render method to return C# rectangle object
    rectangle =  rect.Render(rctForm.Location);
    //Drawing Rectangle
    canvas.DrawRectangle(pen, rectangle);
    //Filling Rectangle
    LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
    canvas.FillRectangle(lBrush, rectangle);
    //Draw previous objects and change fill color
    DrawPreviousObjects(canvas);
    //Adding rectangle object to collection
    Rectangles.Add(rect);
}
```



17. Save and run project. Test this new feature by generating at least three rectangles.
18. When you close the form, note that all rectangle instances still exist in memory (heap) but are again unreachable!
19. Eventually the Garbage Collection will clean up the heap memory.

Example 5-2 (continued): Creating Properties for Rectangle Class

20. To count the number of instances of the clsrectangle class we will create a static property that will be incremented whenever either the default constructor or the overridden constructors are invoked. Modify the class clsRectangle as follows.

```
public class clsRectangle
{
    //Private Fields
    private Size _size;
    private Point _xy;
    private static int _count=0;
    // Default Constructor
    public clsRectangle()
    {
        _count += 1;
    }
    //Overloaded Constructors
    public clsRectangle(int x, int y)
    {
        _xy.X = x;
        _xy.Y = y;
        _count += 1;
    }
    public clsRectangle(int x, int y, int width, int height)
    {
        _xy.X = x;
        _xy.Y = y;
        _size = new Size(Math.Abs(width), Math.Abs(height));
        _count += 1;
    }
    //Properties
    public static int Count
    {
        get { return _count; }
    }
    public Point XY
    {
        get { return _xy; }
        set { _xy = value; }
    }
    public Size RectangleSize
    {
        get { return _size; }
        set { _size = new Size(Math.Abs(value.Width), Math.Abs(value.Height)); }
    }
    //Methods
    public Rectangle Render(Point offset)
    {
        Rectangle rect = new Rectangle(_xy.X + offset.X, _xy.Y + offset.Y, _size.Width, _size.Height);
        return rect;
    }
}
```

21. Add a label on the form frmGeometricShapes in the top left corner, name it lblCount and set its Text property to 0. Add the following line of code at the end of the method DrawRectangle: `lblCount.Text = clsRectangle.Count.ToString();`
22. Save and run the project. Note the count of instances whenever the button is clicked.
23. You will also noticed that even when you close the form frmGeometricShapes and launched it again from the MainMenu form the count continues since the instances still exist.

Now we are at the point where we can actually use the clsRectangle class. Let us review the concepts of object-oriented programming using this specific example:

Class:
We have created a class named clsRectangle that serves as a specific class for basic operation of a rectangle. This class defines the characteristics of an object and has properties and methods.

Object:
We have instantiated an object from the class in the btnRectangle_click event handler. We have set properties and methods at the instance level (non-static) and at the class level (static).

Properties:

We have created properties, both static (class level) and non-static (instance level)

Method:
We have created methods that allowed us to interact with the class.

Message Passing:
We have created an event handler that send messages to the method in the clsRectangle class.

Abstraction:
Think about it, we simply have focused on the Rectangle object, without thinking how are we going to use or integrate this class into our program. We only thought about how this object should behave. This is a very important concept.

Encapsulation:
When we interact with the Rectangle class, we simply use its methods and pass the required arguments. We do not need to know any of the internal workings of the class. The other aspect associated with this concept is that we do not have direct access to the internal fields of this class. We can only use the methods or properties to change the values of the private fields.

We have not covered the concepts of Classification (=Inheritance), Interfaces and Polymorphism in this example, this is covered in the next chapter.

CLASS 6

16. Advanced Object-Oriented Programming in C#

This second part of object-oriented programming deals with two more advanced, but important concept, inheritance and polymorphism.

16.1 Class Inheritance

Inheritance in the object-oriented world promotes simpler code, but it also models real-world situations. Furthermore, it supports the concept of abstraction.

First of all, let us take a look at a real-world example. And we use the microwave example that we have been following through this course. Ask yourself what microwave do you own right now? Now try to classify your microwave oven, whether it is a countertop, integrated into a range, a convection microwave oven etc. And then continue the hierarchy ladder up until you reach the oven class. So you can see that we are great on taxonomies, we like to classify things. The process of classifying helps us make things easier by factoring out commonalities. This is the real world, and this actually helps us in the software world as well.

If the general characteristics of a microwave oven are already modeled and encapsulated in a software class, then I do not have to concern myself with that part. The make, model, color, etc. are already taken care of in terms of collecting, processing, and storing. And if not, then I have to write it myself. But at that level of abstraction, I do not have to worry or even think about the Oven class, and the properties and methods that come along with this specific instance of a microwave oven.

Class Relationships

There are two main relationships when it comes to class inheritance, and it is important to recognize them in the real world in order to model them correctly in the software world.

One is the IS_A relationship between classes. In our oven example, a Microwave IS_A (n) Oven, and an Oven IS_A Kitchen Appliance. In the object-oriented world, you say that the class Microwave inherits from the class Oven. And the Oven class inherits from the Kitchen Appliance class. You also say that the Kitchen Appliance class is the base class, and the Oven class is the subclass.

Note that the IS_A relationship is not reflexive, that means an Oven IS_A Kitchen Appliance, but an Kitchen Appliance is not (necessarily) an Oven. The last statement really refers to the general

case, a particular Kitchen Appliance in the Kitchen Appliance class might be an Oven, but in general this does not have to be true.

On the other hand, the inheritance relationship is transitive. That means because a Microwave is an Oven, and an Oven is a Kitchen Appliance, a Microwave is therefore also a Kitchen Appliance.

An inherited class gets all the members of the base class as if they were its own, plus any new members that are defined in the inherited class.

The other relationship is the HAS_A relationship. In our car example, a microwave oven HAS_A magnetron (radiation source), but it does not inherit from the magnetron class. Obviously, a microwave needs a magnetron to be able to generate heat, but a microwave is not a type of magnetron. It is said that a microwave contains a magnetron. In the software world, you have access to the magnetron class through its public properties and methods so that the microwave class can use it to meet its operational needs, in this case to generate heat.

There is a more fine-grained differentiation of the HAS_A relationship:

- A microwave has a magnetron, meaning a microwave is composed of a magnetron (of course this is not the only part) (composition)
- A microwave has an oven lamp, meaning a microwave possesses or can have a lamp (aggregation)

The difference between these two relationships can be boiled down to the life time of the child objects. Most likely, a magnetron lives as long as the microwave oven. Once the microwave breaks down due to age, the magnetron goes with it. In the software world, you would instantiate the magnetron object in the constructor of a microwave class, and when an instance of a microwave is destroyed the magnetron is destroyed as well.

On the other hand, lamp bulbs are parts that do not outlive the microwave, a microwave will use most likely many light bulbs over its lifetime. In the software world, you create an instance of a light bulb class somewhere in the methods of the microwave object.

UML, or Unified Modeling Language is used to graphically model those relationships. In UML, classes are represented by a box divided vertically into three sections. The name of the class is displayed in the uppermost section. In the middle section you find the data members. A plus (+) signifies a public member, whereas a minus sign (-) represents a private member. The bottom portion is reserved for methods. Most methods are public, in general.

The IS_A relationship is represented by a line connecting the classes and a triangular arrowhead as shown in Figure 68 on the left side. The arrow points up the hierarchy to the base class.

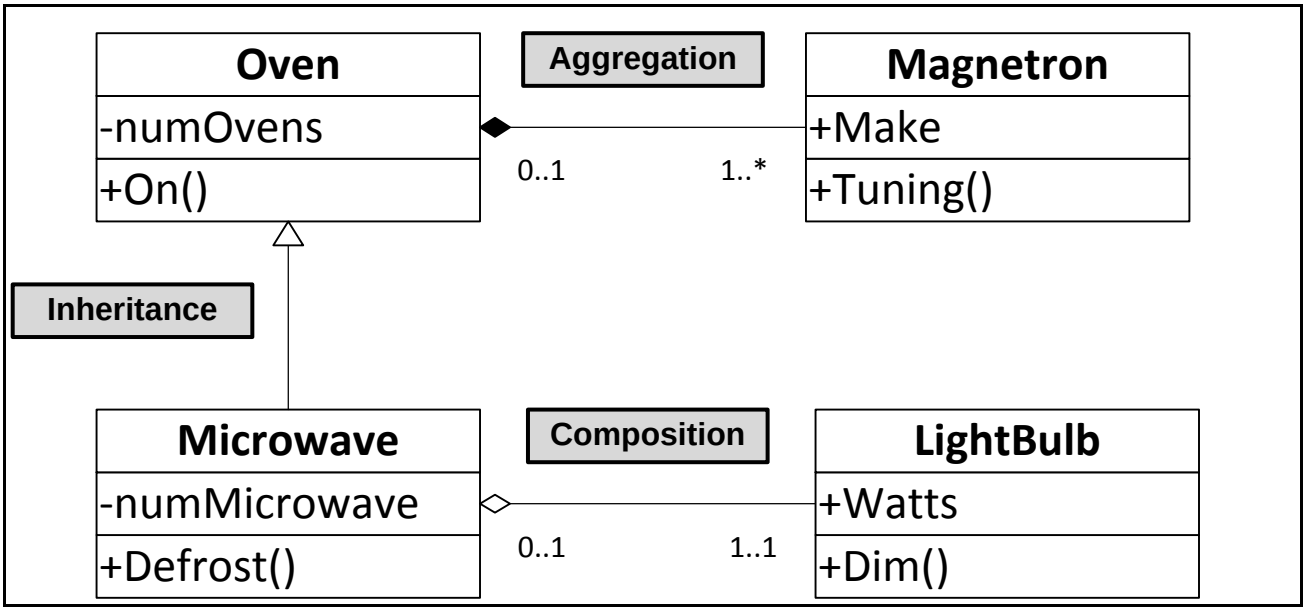


Figure 68: UML, Class Relationships

A HAS_A relationship is also expressed by a line connecting the classes together. A diamond symbol located closer to a class signifies that that class is composed or contains the other class. A black diamond symbol represents composition. A white diamond symbol means aggregation.

Similar to database Entity-Relationship diagrams (ERD), the minimum and maximum cardinality is denoted at both ends of the relationship line. In Figure 68, one (1) magnetron maybe associated with zero(0) to at most one(1) microwave. The zero means that a magnetron can function without a microwave. On the other hand, one microwave must be associated with at least one (1) and at most one (1) magnetron. A microwave without a magnetron is not a functioning microwave.

In Figure 68 you see both types of HAS_A relationships. A magnetron is an integral part of a microwave, therefore a microwave is composed of a magnetron. Light bulbs on the other hand, are parts that have a lifetime in comparison to the lifetime of a microwave, therefore a microwave possesses (or aggregates) light bulbs.

Class Inheritance (IS_A)

To implement an IS_A inheritance relationship, you create a subclass name followed by a colon and then by the base class name as shown below:

```
public subclass_name : baseclass_name
{
        statements
}
```

Now the subclass inherits all public members from the base class in addition to the newly defined members in the sub class.

We will explore this concept now using the example of the rectangle class. We need other geometric shapes, such as a circle and an ellipses. These geometric shapes have some properties and methods in common, such as the x and y coordinates and calculating the area.

Therefore, we are going to create a class named TwoDShapes where we abstract out the commonalities of two-dimensional geometric shapes. We will use this new base class to create a new rectangle class which will inherit from the TwoDShapes class.

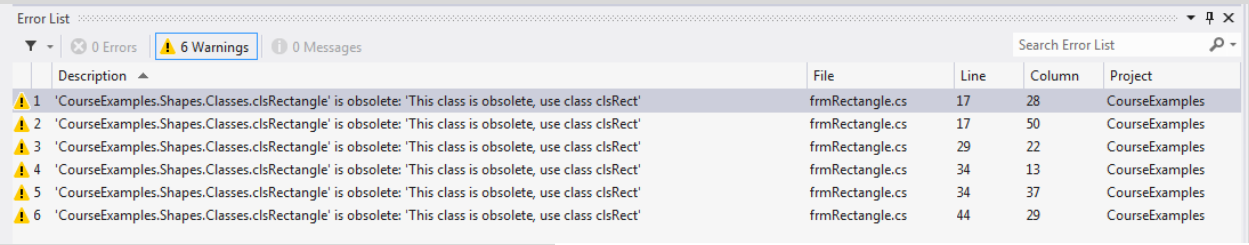
Warning: In C#, a subclass can only inherit from one base class, which is different from other programming languages. However, you can implement multiple interfaces (covered later in this chapter).

Example 6-1: Creating Base and Sub Class

1. First of all, we will add the attribute [Obsolete] to the existing class clsRectangle and add a comment as shown below.

```
[Obsolete("This class is obsolete, use class clsRect",false)]
public class clsRectangle
{
```

2. Now click on the pull-down Build → Build Solution and notice the generated warning message in the output window as shown below.



3. The new base class TwoDShapes is shown on the right. It is very similar to the old class clsRectangle, but it is more generic.
4. There are three internal, instance fields:
   _position, _color, _shapeName
5. There is one static field:
   _count
6. The color property is defined as type Color based on the namespace System.Drawing.
7. A new method is defined as ToString to return the shapeName and the position of the shape.

```
public abstract class TwoDShapes
{
    //Internal Fields
    internal Point _position;
    internal Color _color;
    internal string _shapeName;
    static int _count=0;
    //Constructor
    public TwoDShapes()
    {
        _shapeName = "Two-dimensional Shape";
    }
    public TwoDShapes(int x, int y)
    {
        _position = new Point(x, y);
        _shapeName = "Two-dimensional Shape";
    }
    //Properties
    public Color color
    {
        get { return _color; }
        set { _color = value; }
    }
    public Point Position
    {
        get { return _position; }
        set { _position = new Point(value.X, value.Y); }
    }
    public static int Count
    {
        get { return _count; }
    }
    //Methods
    public string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(_shapeName + "\n");
        sb.Append("Coordinates: " + _position.X + ", " + _position.Y + "\n");
        return sb.ToString();
    }
}
```

Example 6-1 (continued): Creating Base and Sub Class

8. The new sub class, clsRect is shown below.

```
public class clsRect : TwoDShapes
{
    internal Size _size;
    //Constructor
    public clsRect()
    {
        _shapeName = "Rectangle";
        _count += 1;
    }
    public clsRect(int x, int y)
    {
        _shapeName = "Rectangle";
        _count += 1;
    }
    public clsRect(int x, int y, int width, int height)
    {
        _size.Width = width;
        _size.Height = height;
        _shapeName = "Rectangle";
        _count += 1;
    }
    //Properties
    public Size RectangleSize
    {
        get { return _size; }
        set { _size = new Size(Math.Abs(value.Width), Math.Abs(value.Height)); }
    }
    //Methods
    public Rectangle Render(Point offset)
    {
        Rectangle rect = new Rectangle(_position.X + offset.X, _position.Y + offset.Y, _size.Width, _size.Height);
        return rect;
    }
}
```

9. Notice that this class is derived from the class TwoDShapes (super class). All internal fields defined in the super class are available in this class.
10. There are three constructors, the default one without any parameters, then one with the x and y coordinates as parameters, and the last one with the x and y coordinates as well as the width and height as parameters.
11. We have removed the offset values as properties (since they are really not properties of a rectangle). We simply pass the offset values as a parameter of type Point into the Render method.
12. The shapeName is now “Rectangle” since this is the rectangle class.

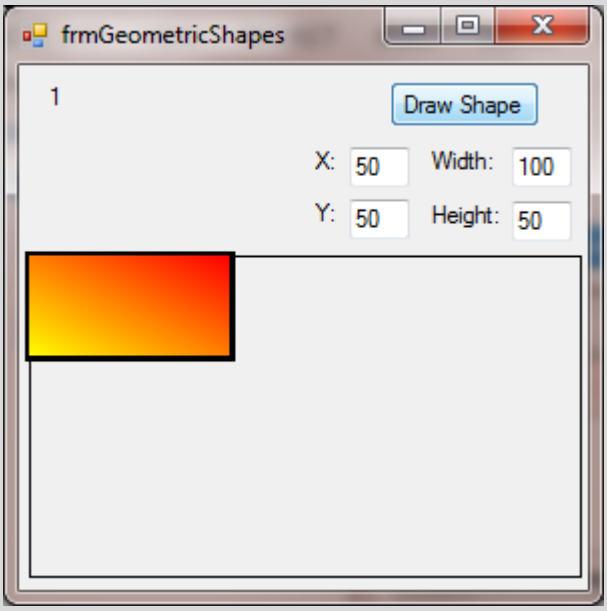
Example 6-1 (continued): Creating Base and Sub Class

13. Replace the class clsRectangle with clsRect in frmGeometricShapes.cs is shown below:

```
//Collection object to store all shape instances
List<clsRect> Rectangles = new List<clsRect>();

private void DrawPreviousObjects(Graphics canvas)
{
    //Loop through all existing shape objects and change fill color
    foreach (clsRect rct in Rectangles)
    {
        canvas.FillRectangle(new SolidBrush(Color.Bisque), rct.Render(rctForm.Location));
    }
}
```

```
private void DrawRectangle(Graphics canvas, Pen pen)
{
    //Creating C# rectangle object
    Rectangle rectangle;
    //Instantiating clsRect object
    clsRect rect = new clsRect(Convert.ToInt32(txtX.Text), Convert.ToInt32(txtY.Text));
    rect.RectangleSize = new Size(Convert.ToInt32(txtWidth.Text), Convert.ToInt32(txtHeight.Text));
    //Using render method to return C# rectangle object
    rectangle =  rect.Render(rctForm.Location);
    //Drawing Rectangle
    canvas.DrawRectangle(pen, rectangle);
    //Filling Rectangle
    LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
    canvas.FillRectangle(lBrush, rectangle);
    //Draw previous objects and change fill color
    DrawPreviousObjects(canvas);
    //Adding rectangle object to collection
    Rectangles.Add(rect);
    lblCount.Text = clsRect.Count.ToString();
}
```

14. Note we are instantiating the rectangle object using the second constructor by passing the x and y coordinates.
15. Save and run the project. Create some rectangles. Note that now setting the coordinates does not work anymore. The rectangle is always created in the top left corner of the drawing area.



16. We were expecting that the constructor method of the superclass would be called as well where the coordinates are set. We will cover this problem in the next section.

Inheritance and Constructors

Based on the last example, we know that something is not working correctly. We have to address the issue of constructors in inherited classes. When an inherited class is instantiated, the constructor for this class is executed. Because this class is inherited, all base class constructors are executed as well in the order of the hierarchy chain going down. That actually means that the base class constructor is executed first followed by the subclass constructor.

So what happens if the base class contains two or more constructors? Which constructor is executed? Well, that depends what arguments you pass down to the base constructors.

In Example 6-1, we have not specifically addressed anything in the base class. That means the default constructor in the base class is being executed, the one having no parameters at all. In the TwoDShapes class the default constructor creates a new object using the default values (0 for number fields) for the internal fields and sets the shapeName.

We need the second constructor in the base class to be executed. So how do we do that? Well, the base keyword allows us to do just that. We add the base keyword to the constructor in the subclass and pass any required arguments into it. That way the base class constructor matching the number and type of arguments will be invoked. The syntax is shown below:

```
Class MainClass
{
        public MainClass() //Default constructor
        {
        }
        public MainClass(type param1, type param2, ..)   //Additional constructor
        {
                statements
        }
        properties and methods
}
Class SubClass : MainClass
{
        public SubClass()  //Default constructor
        {
        }
        public SubClass(type param1, type param2, ..) : base(param1, param2)
        {
                statements
        }
        properties and methods
}
```

As you can see here, we have added the base keyword to the constructor of the subclass. A colon separates the base keyword from the constructor signature. In parentheses, you add any necessary arguments. Most likely, these arguments come from the parameters of the constructor of the subclass, where they have been declared as a specific type.

When instantiating the subclass using arguments, these arguments are then passed into the corresponding constructor method in the base class by using the base keyword.

SubClass *test* = **new** SubClass(5,20)

The call above will first execute the base class constructor using the provided arguments, then the subclass constructor.

Example 6-2: Calling Base Class Constructor

1. In order to call the appropriate base class constructor, we need to pass the parameters of the x and y coordinates into the base class constructor using the base keyword as shown below.

```
public clsRect(int x, int y) :base(
{                                          ▲ 2 of 2 ▼ TwoDShapes.TwoDShapes(int x, int y)
    _shapeName = "Rectangle";
}
```

2. The complete class is shown below.

```
public class clsRect : TwoDShapes
{
    internal Size _size;
    //Constructor
    public clsRect()
    {
        _shapeName = "Rectangle";
        _count += 1;
    }
    public clsRect(int x, int y) :base(x,y)
    {
        _shapeName = "Rectangle";
        _count += 1;
    }
    public clsRect(int x, int y, int width, int height) :base(x,y)
    {
        _size.Width = width;
        _size.Height = height;
        _shapeName = "Rectangle";
        _count += 1;
    }
    //Properties
    public Size RectangleSize
    {
        get { return _size; }
        set { _size = new Size(Math.Abs(value.Width), Math.Abs(value.Height)); }
    }
    //Methods
    public Rectangle Render(Point offset)
    {
        Rectangle rect = new Rectangle(_position.X + offset.X, _position.Y + offset.Y, _size.Width, _size.Height);
        return rect;
    }
}
```

3. Save and run the project. It should now work correctly again.

Class Substitution, Class Casting

When you have inherited subclasses, the interesting fact is that you can declare an object variable as type BaseClass, but you can also use any inherited subclasses for this object variable (besides, of course, the BaseClass itself).

When you think about it a bit longer, it makes kind of sense. The base class is the lowest, common denominator. Any subclass derived from the base class inherits its properties and methods, so even the subclasses are able to complete any operation that a base class can perform. We will demonstrate this with the following example.

But first of all, let us add another shape class, the circle class, to demonstrate this behavior.

Example 6-4: Creating a circle class

1. Besides the rectangle class (clsRect) we also want to create another class to draw circle. This new class is named clsCircles and will inherit from the rectangles class. (A circle is a specialized object from an rectangle)
2. The complete code of clsCircle is shown below, add it below the class clsRect.

```
public class clsCircle : clsRect
{
    //Private field
    private int _radius;
    //Constructor
    public clsCircle()
    {
        _shapeName = "Circle";
    }
    public clsCircle(int x, int y, int radius) : base(x, y)
    {
        _radius = radius;
        _size.Width = radius * 2;
        _size.Height = radius * 2;
    }
    //Properties
    public int radius
    {
        get { return _radius; }
        set { _radius = value; }
    }
    //Methods
}
```

3. The only "addition" is the concept of a radius. A circle will be placed inside a square (equal width and height) where the width and height is two times the radius.
4. The constructor method asks for a radius only, the width and height can be derived from the radius.
5. There are currently no methods.

Example 6-5: Substitutable Classes

1. In the frmGeometricShapes.cs class file, create the following method.

```
private void DisplayShapeInfo(clsRect shape)
{
        MessageBox.Show(shape.ToString());
}
```

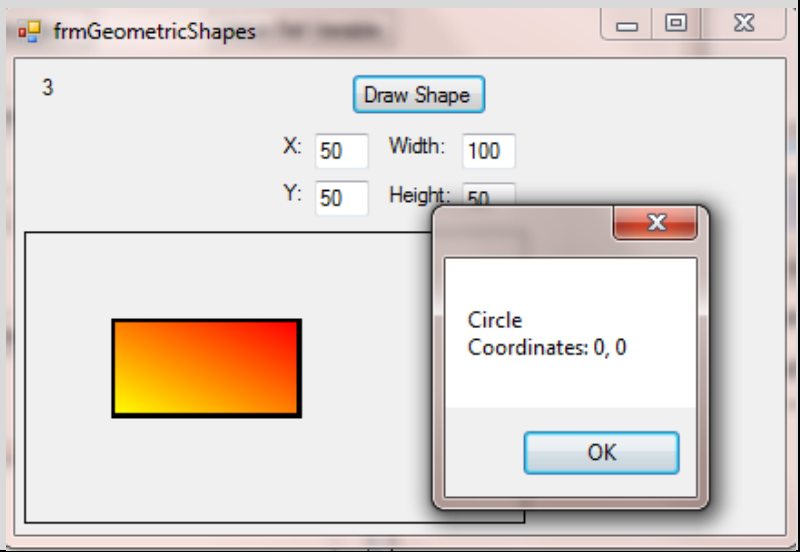
2. Note that we declare a parameter based on the base class clsRect. We can pass into this method the base class or any sub class.
3. In the method DrawRectangle, create a test instance of the circle class and add the following call DisplayShapeInfo at the end.

```
lblCount.Text = clsRect.Count.ToString();
clsCircle test = new clsCircle();
DisplayShapeInfo(test);
```

4. Note that we pass a clsCircle object into the method DisplayShapeInfo although this methods expects an object of type clsRect. Since clsCircle is a derived, inherited class, this is a valid, implicit cast.
5. Save and run the project. Note the message box after clicking the "Draw Shape" button.
6. It might be better, to explicitly cast the test object into a rectancle object just for readability as shown below.

```
clsCircle test = new clsCircle();
DisplayShapeInfo((clsRect)test);
```

7. Place the clsRect class name in front of the test object in parentheses.
8. Save and run the project. There should be no changes from the previous version.



Casting the other way around is a dangerous operation for the very reason why casting a subclass into a base class is ok. If you cast a rectangle object into a circle object, you may cause run-time errors if any subclass methods are invoked. The base class is not aware of any subclass methods, and therefore will throw a run-time error.

To avoid these kinds of errors, you can use the following operators to test whether a cast is valid or not.

The Is operator compares an object on the left side of the operator to a type on the right side and returns a true if the object is compatible with the type.

if (*object* **is** *type*) → returns true of false

The As operator works differently, and is more efficient. It actually converts the object on the left to the type on the right without causing a run-time error if the conversion fails. It simply sets the object to null, so you have to test whether the object is null or not.

```
type object = object_to_cast as type
if (object != null)
{
        statements
}
```

The following example will show first the run-time error, and then demonstrates the two methods above.

Example 6-6: Casting BaseClass to SubClass

1. In the frmGeometricShapes.cs class file, change the type to clsCircle in the DisplayShapeInfo method as shown below.

```
private void DisplayShapeInfo(clsCircle shape)
{
        MessageBox.Show(shape.ToString());
}
```

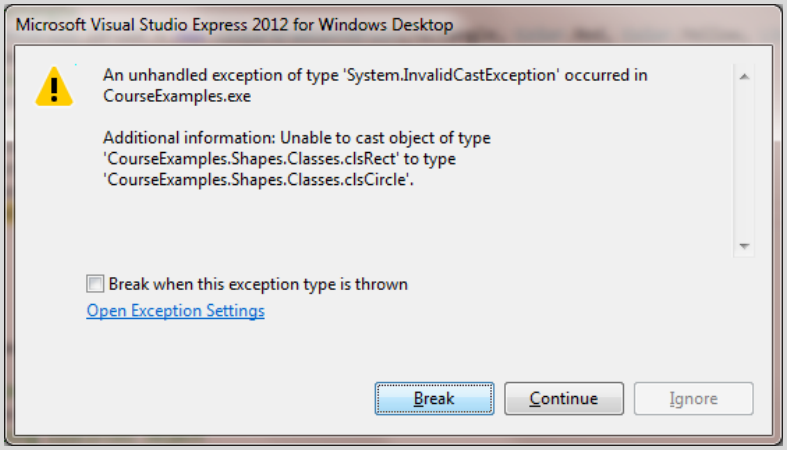
2. Now change the call to this method in the method DrawRectangle to pass the rect object.

```
clsCircle test = new clsCircle();
DisplayShapeInfo(rect);
```

3. You will notice a compiler error indicating that you cannot implicitly convert a base class (rect) into a sub class (circle). You have to explicitly cast the base class to the sub class.

```
clsCircle test = new clsCircle();
DisplayShapeInfo((clsCircle)rect);
```

4. This works for the compiler, but when you run it, you will receive an error message.



Example 6-6 (continued): Casting BaseClass to SubClass

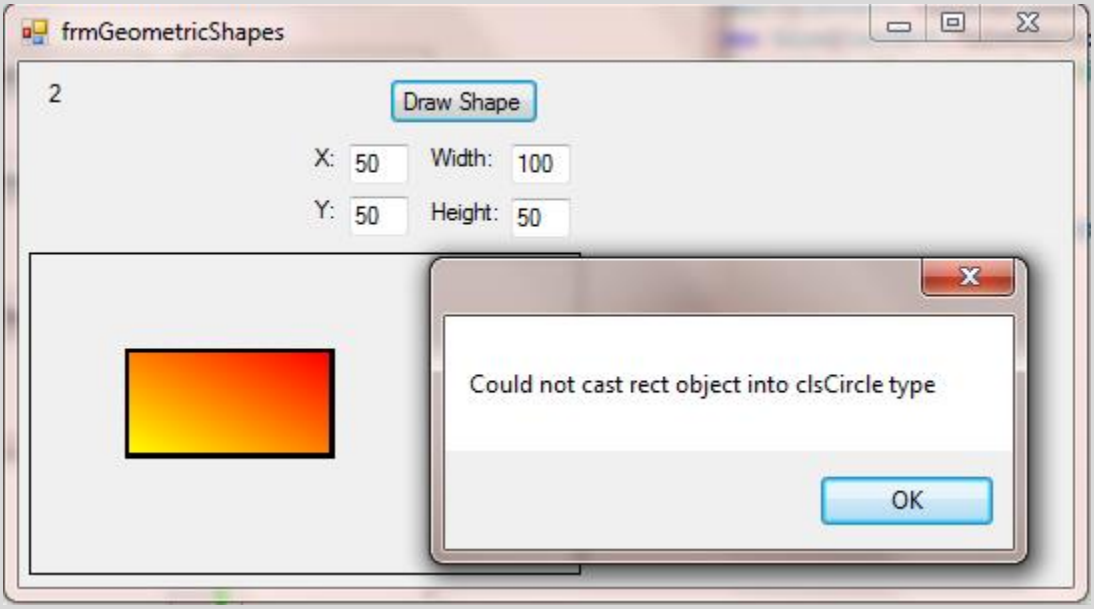
5. Now modify the call in the frmGeometricShapes.cs class file to use the Is operator.

```
if (rect is clsCircle)
{
    DisplayShapeInfo((clsCircle)rect);
}
```

6. Save and run the project. You will notice that no error message is displayed due to the fact that the method is not being called because a rect object is just not of type clsCircle.
7. Now modify the call in the fromGeometricShapes.cs file to use the As operator.

```
clsCircle test = rect as clsCircle;
if (test != null)
{
    DisplayShapeInfo((clsCircle)rect);
}
else
{
    MessageBox.Show("Could not cast rect object into clsCircle type");
}
```

8. Save and run the project.



Class Containment (HAS_A)

Up to now we have only discussed and demonstrated the inheritance (IS_A) relationship. And this is by far the more complex one in terms of implementing this kind of relationship in your program.

The HAS_A relationship is fundamentally different. It really does not use any new code constructs or commands. You simply use properties and methods of the other class in your class.

Shown in the next example is a class that shows how a HAS_A relationship is implemented in C# code. An object variable is declared as type of the class to be contained in this class. This object variable allows you access to all public properties and methods of that class.

We define a point class that handles storing the x and y coordinates of a point. We also create a ToString() method to output the characteristics of a point. Suppose that we need a new class called Line, we can reuse the Point class via composition. We say that "A line is composed of two points", or "A line has two points". Composition exhibits a "has-a" relationship.

 **Note:** C# already has a Point struct (which is simple class), therefore in the next example we name our point class MyPoint, simply to differentiate and to avoid ambiguity.

Example 6-7: Point and Line classes demonstrating Has_A relationship

1. In the BAL folder, add a new class file and name it Point_Line.cs.
2. Change the namespace to CourseExamples.PointLine.
3. Type the class MyPoint as shown on the right.
4. Note we defined two integer private fields _x and _y for the coordinates.
5. We have two constructors, one where you can specify the coordinates and the other one where the coordinates are set to 0.
6. Then there are the public properties to set and get the x and y coordinates.
7. Last but not least there is a ToString() method to return the coordinates in a comma-separated format.

```
namespace CourseExamples.PointLine
{
    public class MyPoint
    {
        // Private member variables
        private int _x, _y;   // (x, y) co-ordinates

        // Constructors
        public MyPoint(int x, int y)
        {
            _x = x;
            _y = y;
        }
        public MyPoint()
        {    // default (no-arg) constructor
            _x = 0;
            _y = 0;
        }

        // Public getter and setter for private variables
        public int X
        {
            get { return _x; }
            set { _x = value; }
        }
        public int Y
        {
            get { return _y; }
            set { _y = value; }
        }
         // toString() to describe itself
        public String ToString()
        {
            return "(" + _x + "," + _y + ")";
        }
    }
```

Example 6-7(continued): Point and Line classes demonstrating Has_A relationship

8. Now add a second class named Line below the MyPoint class.
9. There are two constructors, one using integers for defining the begin and end point and the other one using MyPoint types for the begin and end points.
10. Then there are many properties, for the points Begin and End and for the individual coordinates for the Begin and End point.
11. There is also a ToString() method to return the line information in a text format.
12. And finally a method to calculate the length of a line.
13. Note that we have not used any special C# constructs to implement a Has_A relationship. We are simply using the MyPoint class in the line class.

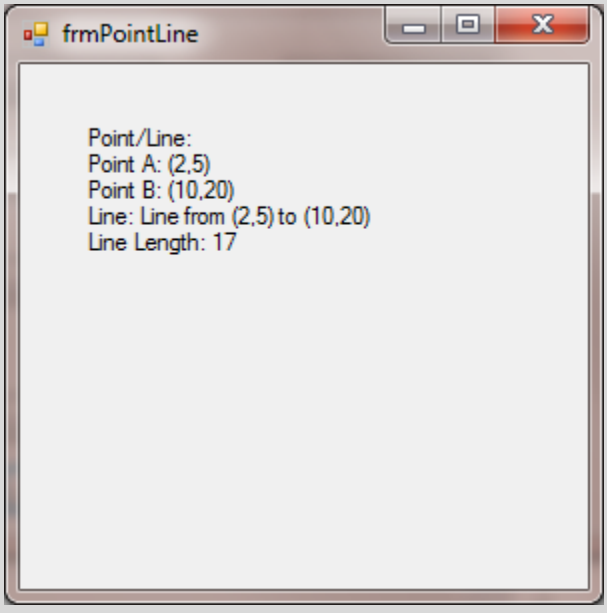
```
public class Line
{   // Private member variables
    private MyPoint _begin, _end;   // Declare begin and end as instances of Point
    // Constructors
    public Line(int x1, int y1, int x2, int y2)
    {
        _begin = new MyPoint(x1, y1);  // Construct Point instances
        _end = new MyPoint(x2, y2);
    }
    public Line(MyPoint begin, MyPoint end)
    {
        _begin = begin;  // Caller constructed Point instances
        _end = end;
    }
    // Properties
    public MyPoint Begin
    {
        get { return _begin; } set { _begin = value; }
    }
    public MyPoint End
    {
        get { return _end; } set { _end = value; }
    }
    public int BeginX
    {
        get { return _begin.X; } set { _begin.X = value; }
    }
    public int BeginY
    {
        get { return _begin.Y; } set { _begin.Y = value; }
    }
    public int EndX
    {
        get { return _end.X; } set { _end.X = value; }
    }
    public int endY
    {
        get { return _end.Y; } set { _end.Y = value; }
    }
    //Public Methods
    public String ToString()
    {
        return "Line from " + _begin.ToString() + " to " + _end.ToString();
    }
    public double GetLength()
    {
        int xDiff = _begin.X - _end.X;
        int yDiff = _begin.Y - _end.Y;
        return Math.Sqrt(xDiff * xDiff + yDiff * yDiff);
    }
}
```

Example 6-7(continued): Point and Line classes demonstrating Has_A relationship

14. Now add a new Windows form into the PL folder and name it frmPointLine.
15. Add a label control to the form and place it in the top left corner. Set its Text Property to "Point/Line:" and name it lblOutput.
16. Double-click on the form to create the Load event.
17. Rename the namespace to just CourseExamples (you may also have to rename it in the designer file).
18. Add the namespace CourseExample.PointLine at the top of the class file.
19. Write the following code into the Load event:

```
private void frmPointLine_Load(object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder();
    MyPoint pointA = new MyPoint(2, 5);
    MyPoint pointB = new MyPoint(10, 20);
    sb.Append(lblOutput.Text + "\n");
    sb.Append("Point A: " + pointA.ToString() + "\n");
    sb.Append("Point B: " + pointB.ToString() + "\n");
    Line line = new Line(pointA, pointB);
    sb.Append("Line: " + line.ToString() + "\n");
    sb.Append("Line Length: " + line.GetLength());
    lblOutput.Text = sb.ToString();
}
```

20. On the main menu form, add another button. Set its Text property to Point/Line and name it btnPointLine.
21. Write the usual two lines of code to show the form frmPointLine.
22. Save and run the project. Test the form frmPointLine.



16.2 Polymorphism

As covered in the previous chapters, inheritance means that the subclass adopts the properties and methods of the base class. But what if a method needs to be implemented in a slightly different way? The method in its essence is still the same, however, some internal details are different.

For example, in our rectangle class we have a ToString() method. The ToString() method in a circle is virtually the same. However, there may be some different string representation of the data between the rectangle class and the circle class.

This is where polymorphism comes in. A class can inherit properties and methods from the base class but it can also override a method that is inherited from the base class.

Overloading an Inherited Method

We have already discussed the concept of overloading a method. Overloading a method means having the same method name but different types and/or number of parameters.

Example:

```
class Test
{
        public void method1(int intI)
        {
                statements
        }
        public void method1(string strName)
        {
                statements
        }
}
```

Warning: You cannot overload a method by implementing a different return type, which is not allowed in C#.

As part of the signature of a method for overloading purposes is the class name in which the method is defined counts. That means that you can write a method having the same name and methods in a subclass and a base class, for example. This is also known as hiding the method.

Example 6-8: Hiding a method

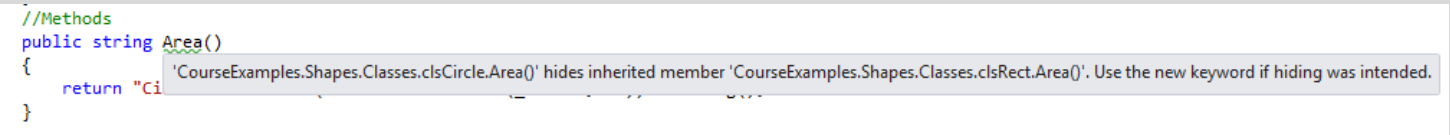
1. In the class clsRect, add the following method to calculate the area of a rectangle.

```
public string Area()
{
    return "Rectangle Area: " + (_size.Width * _size.Height).ToString();
}
```

2. In the class clsCircle, add the following method to calculate the area of a circle.

```
//Methods
public string Area()
{
    return "Circle Area: " + (Math.PI * Math.Pow(_radius, 2d)).ToString();
}
```

3. Notice the green underlined method name, when you hover your mouse over it, you will see the following message:



4. Add the new keyword in the method Area in the circle class.

```
//Methods
public new string Area()
{
    return "Circle Area: " + (Math.PI * Math.Pow(_radius, 2d)).ToString();
}
```

If you inherit from another class and you have a method that shares the same signature you can define it as "new" so that it is independent from the parent class. This means that if you have a reference to the "parent" class then that implementation will be executed, if you have a reference to the child class then that implementation will be executed.

Using the "new" keyword works great as long as you always use the declared type of the class as shown below:

clsRect rect = new clsRect(50,50);
rect.Area(); → using the base class method Area

clsCircle circle = new clsCircle(50,50,25);
circle.Area(); → using the sub class method Area

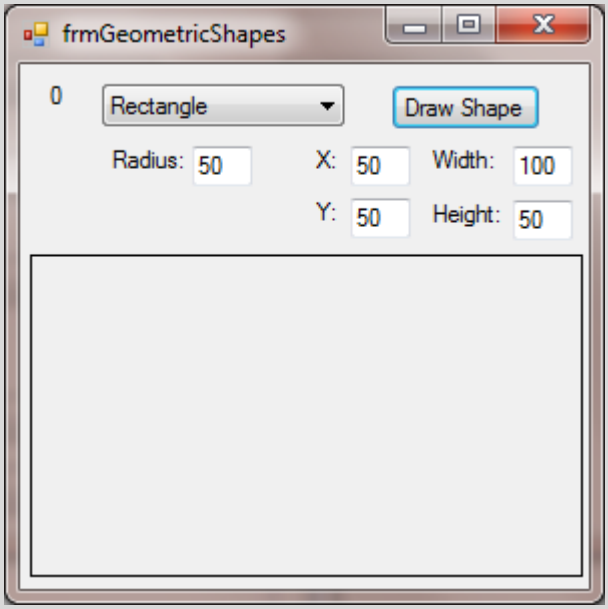
But the following call to the method Area will call the base method Area:

clsRect test = new clsCircle(50,50,25)
test.Area(); → using the base method Area

Warning: Using the new modifier is not polymorphic as shown above.

Example 6-8: Using a hidden method

1. First of all, we also want to display the circle class on the form frmGeometricShapes. Add a combo box control from the toolbox onto the form as shown on the right.
2. Name the combo box cmbShapeType and set its DropDownStyle to DropDownList.
3. Add a label and text box control as shown on the right. Name the text box control txtRadius.and set its Text property to 50.
4. Double-click on the form surface (not any control) to create the form's load event. Write the following code snippet to populate the combo box and select "Rectangle" as the default value.




```
private void frmGeometricShapes_Load(object sender, EventArgs e)
{
    //Populating drop-down list
    cmbShapeType.Items.Add("Rectangle");
    cmbShapeType.Items.Add("Circle");
    cmbShapeType.SelectedIndex = 0;
```

5. Write the following DrawCircle method.

```
private void DrawCircle(Graphics canvas, Pen pen)
{
    //Creating C# rectangle object
    Rectangle rectangle;
    //Instantiating clsCircel object
    clsCircle circle = new clsCircle(Convert.ToInt32(txtX.Text), Convert.ToInt32(txtY.Text), Convert.ToInt32(txtRadius.Text));
    //Using render method to return C# rectangle object
    rectangle = circle.Render(rctForm.Location);
    canvas.DrawEllipse(pen, rectangle);
    //Filling Circle
    LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
    canvas.FillEllipse(lBrush, rectangle);
    //Adding rectangle object to collection
    Rectangles.Add(circle);
    DisplayShapeInfo(circle);
}
```

6. Modify the DrawRectangle method as shown below.

```
private void DrawRectangle(Graphics canvas, Pen pen)
{
    //Creating C# rectangle object
    Rectangle rectangle;
    //Instantiating clsRect object
    clsRect rect = new clsRect(Convert.ToInt32(txtX.Text), Convert.ToInt32(txtY.Text));
    rect.RectangleSize = new Size(Convert.ToInt32(txtWidth.Text), Convert.ToInt32(txtHeight.Text));
    //Using render method to return C# rectangle object
    rectangle =  rect.Render(rctForm.Location);
    //Drawing Rectangle
    canvas.DrawRectangle(pen, rectangle);
    //Filling Rectangle
    LinearGradientBrush lBrush = new LinearGradientBrush(rectangle, Color.Red, Color.Yellow, LinearGradientMode.BackwardDiagonal);
    canvas.FillRectangle(lBrush, rectangle);
    //Adding rectangle object to collection
    Rectangles.Add(rect);
    DisplayShapeInfo(rect);
}
```

Example 6-8 (continued): Using a hidden method

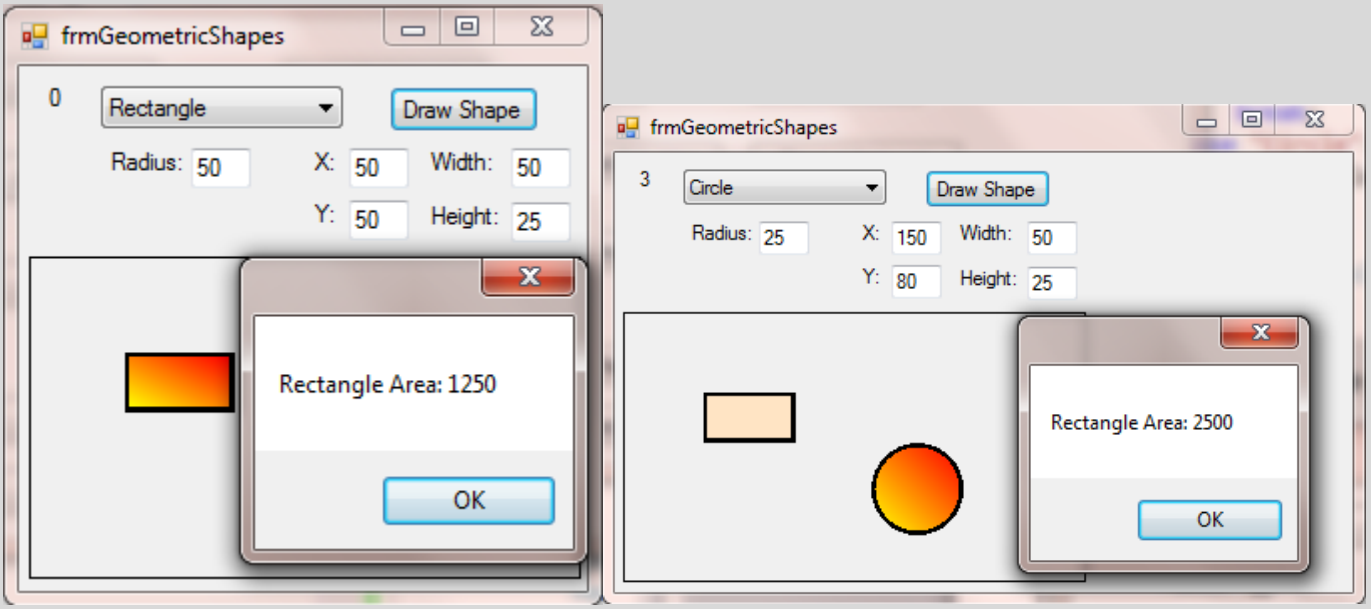
7. Modify the event handler btnDrawShape_click as shown below:

```
private void btnDrawShape_Click(object sender, EventArgs e)
{
    //Setting up graphing objects
    Graphics canvas = this.CreateGraphics();
    Pen pen = new Pen(Color.Black, 5);
    //Draw previous objects and change fill color
    DrawPreviousObjects(canvas);
    switch (cmbShapeType.SelectedItem.ToString())
    {
        case "Rectangle":
            DrawRectangle(canvas, pen);
            break;
        case "Circle":
            DrawCircle(canvas, pen);
            break;
    }
    lblCount.Text = clsRect.Count.ToString();
}
```

8. Modify the DisplayShapeInfo method to display the area:

```
private void DisplayShapeInfo(clsRect shape)
{
    MessageBox.Show(shape.Area().ToString());
}
```

9. Save and run the project. Create at least one rectangle shape and one circle shape.



10. Notice that the DrawCircle method calls the DisplayShapInfo passing a Circle object into it. However, since the parameter in the DisplayShapeInfo was defined as a type clsRect and the Area method was hidden, the base method was called (meaning calculate the area of a rectangle) using the circle's width and height (2 times the radius).

Implementing Polymorphism in C#

We know now how to hide and effectively "override" a method in a base class. This works great as long as you always use the declared type of the class as demonstrated in the previous example.

If you instantiate an object of its own class (and not the base class) hiding a method works very well. However, if you instantiate an object based on its base class, the base class method is the one being called. This may be very well what you need, and there are scenarios that require such a behavior.

However, if you need the method being called the one of its appropriate class, then you need to employ a slightly different C# construct. We basically need to find a way to have C# detect automatically whether the run-time class is the base or the subclass and then execute the corresponding method.

We could again the Is or As operator and check whether the passed object is the base class or the subclass. But what if we add other classes later, we have to continuously modify our code. It would be better if we could instruct C# to detect at run-time what class is being passed.

The solution is the virtual keyword. This keyword is used for the base class method. Each subclass method that overrides this method needs to use the keyword override. The syntax is shown below:

```
class MainClass
{
        public virtual type method(type param1, type param2, ..)
        {
                statements
        }
}
class Subclass : MainClass
{
        public override type method(type param1, type param2, ..)
        {
                statements
        }
}
```

Example 6-9: Using a hidden method

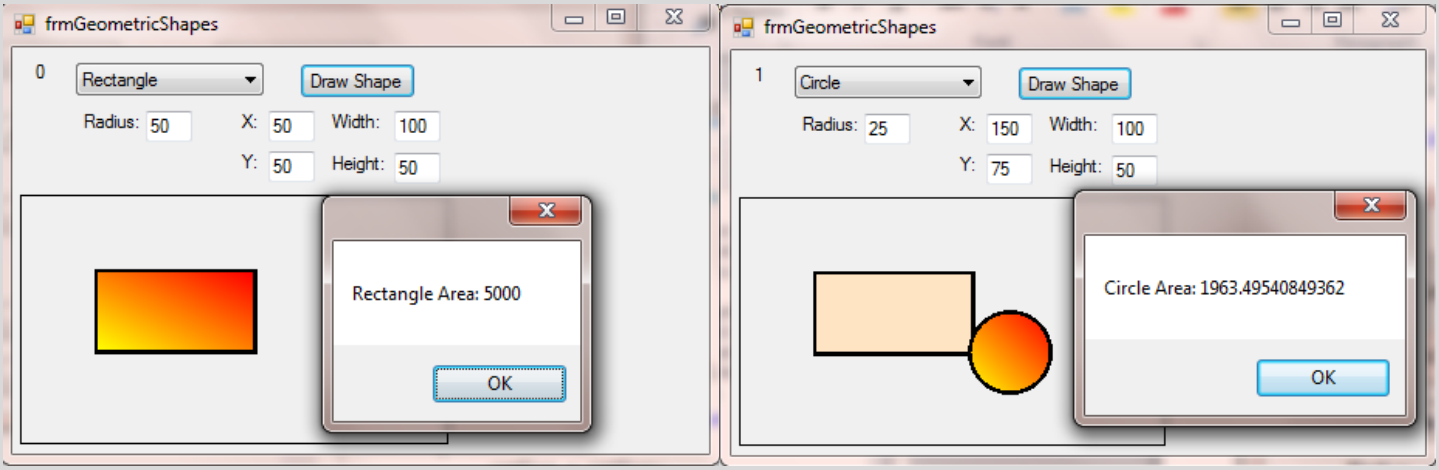
1. In the class file GeometricShapes.cs, make the following changes to the classes clsRectangle and clsCircle.
2. First, add the virtual keyword for the method Area in the class clsRect.

```
public virtual string Area()
{
    return "Rectangle Area: " + (_size.Width * _size.Height).ToString();
}
```

3. Secondly, remove the new and add the override keyword for the method Area in the class clsCircle.

```
//Methods
public override string Area()
{
    return "Circle Area: " + (Math.PI * Math.Pow(_radius, 2d)).ToString();
}
```

4. Save and run the project again and create a least one rectangle and one circle object.



5. Notice now that the correct method Area is called for the circle object.

CLASS 7

17. Introduction to Database Access

The .NET platform defines a number of namespaces that allow you to interact directly with machine local and remote relational databases. Collectively speaking, these namespaces are known as ADO.NET. We will discuss ADO.NET in the following chapters in more detail as it relates to database application programming.

17.1 Introduction to ADO.NET

If you have worked before with ADO (ActiveX Data Objects) in Microsoft's COM technology, you need to understand that ADO.NET has little to do with ADO beyond the letters A, D, and O. While it is true that there is some relationship between the two systems (e.g., each has the concept of connection and command objects), some familiar ADO types (e.g., the Recordset) no longer exist.

Furthermore, you can find many new ADO.NET types that have no direct equivalent under classic ADO (e.g., the data adapter). Unlike classic ADO, which was primarily designed for tightly coupled client/server systems, ADO.NET was built with the disconnected world in mind, using DataSets. This type represents a local copy of any number of related data tables, each of which contains a collection of rows and column.

Using the DataSet, the calling assembly (such as a web page or desktop executable) is able to manipulate and update a DataSet's contents while disconnected from the data source and send any modified data back for processing using a related data adapter.

Perhaps the most fundamental difference between classic ADO and ADO.NET is that ADO.NET is a managed library of code; therefore, it plays by the same rules as any other managed library. The types that make up ADO.NET use the CLR memory management protocol, adhere to the same type system (e.g., classes, interfaces, enums, structures, and delegates), and can be accessed by any .NET language.

From a programmatic point of view, the bulk of ADO.NET is represented by a core assembly named System.Data.dll. Within this binary, you find a good number of namespaces, many of which represent the types of a particular ADO.NET data provider as shown in Figure 69.

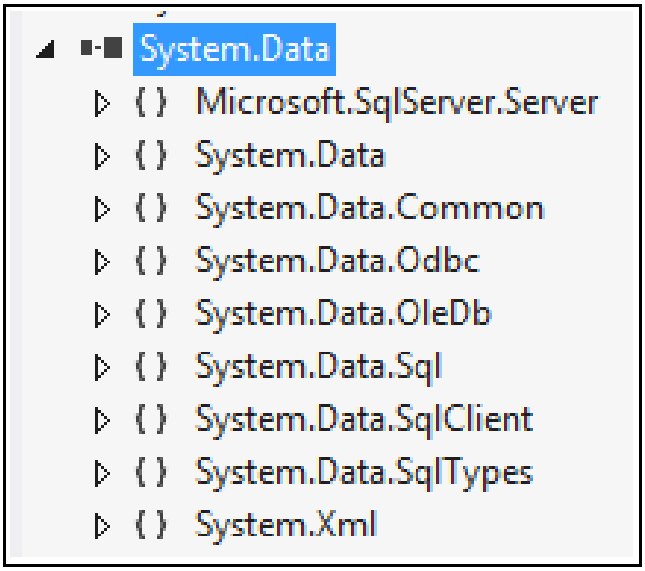


Figure 69: System.Data.dll Namespaces

Most Visual Studio project templates automatically reference this key data access assembly. However, you do need to update your code files to import the namespaces you wish to use, such as individual sub namespaces, for example System.Data.SqlClient for working specifically with a SQL Server database.

You can use the ADO.NET libraries in three conceptually unique manners: connected, disconnected, or through the Entity Framework. The basic architecture is shown in Figure 70.

When you use the connected layer, your code base explicitly connects to and disconnects from the underlying data store. When you use ADO.NET in this manner, you typically interact with the data store using connection objects, command objects, and data reader objects.

The disconnected layer allows you to manipulate a set of DataTable objects (contained within a DataSet) that functions as a client-side copy of the external data. This is an in-memory representation of the data queried from the database. When you obtain a DataSet using a related data adapter object, the connection is automatically opened and closed on your behalf. As you would guess, this approach helps free up connections for other callers quickly and goes a long way toward increasing the scalability of your systems.

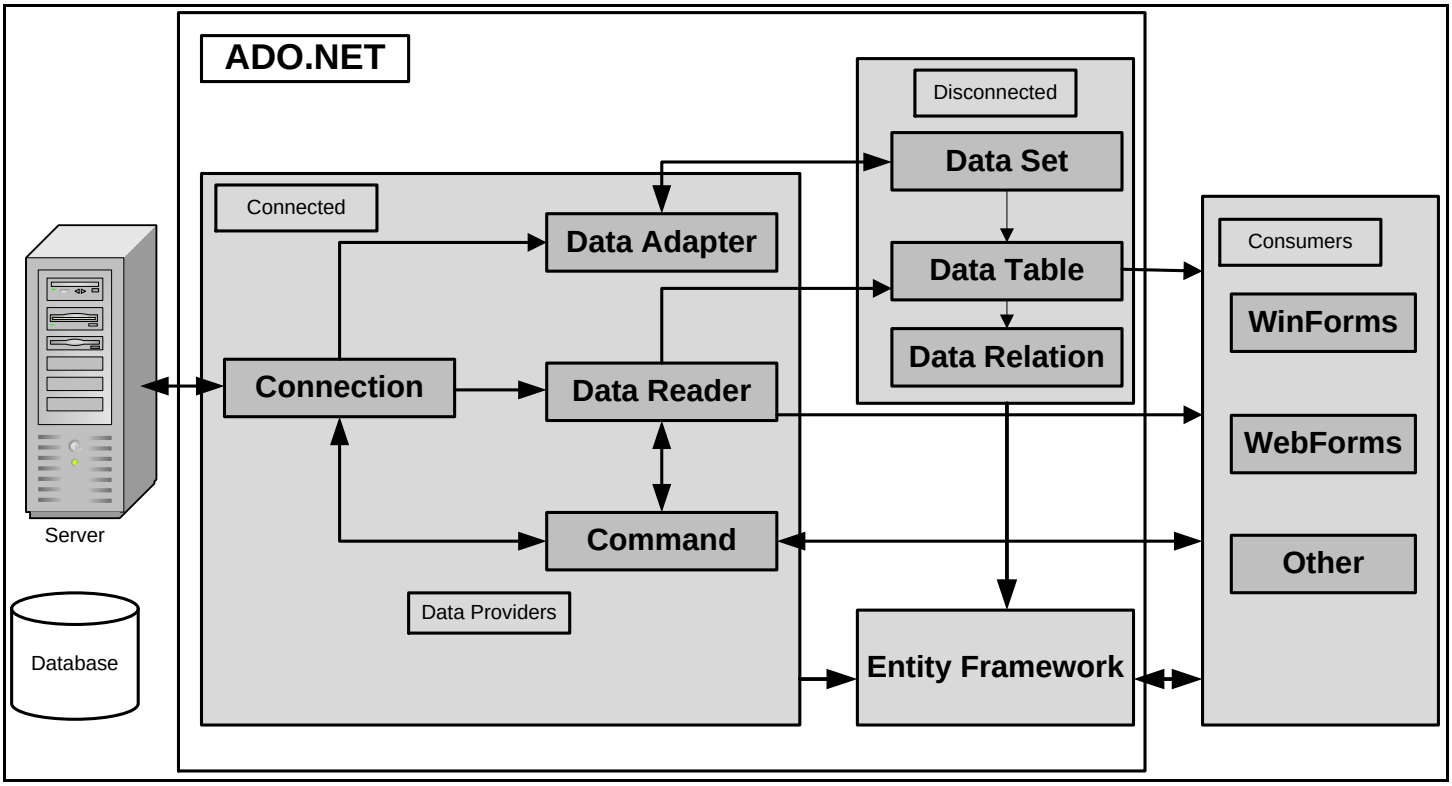


Figure 70: ADO.NET Basic Architecture

Once a caller receives a DataSet, it is able to traverse and manipulate the contents without incurring the cost of network traffic. Also, if the caller wishes to submit the changes back to the data store, the data adapter (in conjunction with a set of SQL statements) is used to update the data source; at this point the connection is reopened for the database updates to occur, and then closed again immediately. But be aware that there are potential issues of concurrency control, meaning if multiple callers request the same kind of data at the same time.

With the release of .NET 3.5 SP1, ADO.NET was enhanced to support a new API termed the Entity Framework (or simply, EF). Using EF, you will find that many of the low-level database specifics (such as complex SQL queries) are hidden from view and authored on your behalf when you generate a fitting LINQ query (e.g., LINQ to Entities). You can use EF in a connected or disconnected fashion.

ADO.NET EF entity is an ORM (object relational mapping) which creates a higher abstract object model over ADO.NET components. So rather than getting into dataset, datatables, command, and connection objects, you work on higher level domain objects like customers, suppliers, etc.

We know that ADO.NET allows us to interact with different types of data sources and different types of databases. However, there is not a single set of classes that allow you to accomplish this universally. Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol. Some older data sources use the ODBC protocol, many newer data sources use the OleDb protocol, and there are more data sources every day that allow you to communicate with them directly through ADO.NET class libraries.

ADO.NET provides a relatively common way to interact with data sources, but comes in different sets of libraries for each way you can talk to a data source. These libraries are called Data Providers and are usually named for the protocol or data source type they allow you to interact with as shown Table 33.

Provider	Namespace	Description
SQL Server	System.Data.SqlClient	Access to SQL Server databases
OLE DB	System.Data.OleDb	Access to databases that support OLE DB
ODBC	System.Data.Odbc	Access to databases that support ODBC
Oracle	System.Data.OracleClient	Access to Oracle databases

Table 33: ADO.NET Data Providers

ADO.NET is a set of classes that expose data access services to the .NET developer. The ADO.NET classes are found in System.Data.dll and are integrated with the XML classes in System.Xml.dll. There are two central components of ADO.NET classes: the DataSet, and the .NET Framework Data Provider.

Data Provider is a set of components including:

- Connection object (SqlConnection, OleDbConnection, OdbcConnection, OracleConnection)
- Command object (SqlCommand, OleDbCommand, OdbcCommand, OracleCommand)
- DataReader object (SqlDataReader, OleDbDataReader, OdbcDataReader, OracleDataReader)
- DataAdapter object (SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter, OracleDataAdapter).

The data provider classes can be used for a connected architecture directly or for the disconnected architecture either through the DataAdapter to fill a DataSet or the DataReader to fill a DataTable object.

18. Database Programming, Connected Architecture

This chapter will focus on the connected architecture, which is commonly used in client-server applications. Web applications, on the other hand, use the disconnected architecture due to the stateless http protocol and scalability requirements.

18.1 Data Provider Objects

ADO.NET includes many objects you can use to work with data. This section introduces some of the primary objects you will use. The objects below are the ones you must know. Learning about them will give you an idea of the types of things you can do with data when using ADO.NET.

Connection Object

To interact with a database, you must have a connection to it. The connection helps identify the database server, the database name, user name, password, and other parameters that are required for connecting to the data base. A connection object is used by command objects so they will know which database to execute the command on.

Command Object

The process of interacting with a database means that you must specify the actions you want to occur. This is done with a command object. You use a command object to send SQL statements to the database. A command object uses a connection object to figure out which database to communicate with. You can use a command object alone, to execute a command directly, or assign a reference to a command object to an SqlDataAdapter, which holds a set of commands that work on a group of data as described below.

DataReader Object

Many data operations require that you only get a stream of data for reading. The data reader object allows you to obtain the results of a SELECT statement from a command object. For performance reasons, the data returned from a data reader is a fast forward-only stream of data. This means that you can only pull the data from the stream in a sequential manner. This is good for speed, but if you need to manipulate data, then a DataSet is a better object to work with.

DataAdapter Object

Sometimes the data you work with is primarily read-only and you rarely need to make changes to the underlying data source. Some situations also call for caching data in memory to minimize the number of database calls for data that does not change. The data adapter makes it easy for you to accomplish these things by helping to manage data in a disconnected mode.

The data adapter fills a DataSet object when reading the data and writes in a single batch when persisting changes back to the database. A data adapter contains a reference to the connection object and opens and closes the connection automatically when reading from or writing to the database.

Additionally, the data adapter contains command object references for SELECT, INSERT, UPDATE, and DELETE operations on the data. You will have a data adapter defined for each table in a DataSet and it will take care of all communication with the database for you. All you need to do is tell the data adapter when to load from or write to the database.

 **Note:** Although the DataAdapter is mainly used for the disconnected architecture, it falls under the connected architecture since it actually opens a connection, retrieves the data, and then immediately closes the connection. We will discuss the DataAdapter in the Disconnected Architecture chapter.

18.2 Connection Object

The connection object is the very first piece in the chain that you must create. Like any other .NET class, you instantiate an object from the connection class and provide certain parameters to establish a connection.

Although working with connections is very easy in ADO.NET, you need to understand connections in order to make the right decisions when coding your data access routines. Understand that a connection is a valuable resource. Sure, if you have a stand-alone client application that works on a single database on one machine, you probably do not need to care about this.

However, think about an enterprise application where hundreds of users throughout a company are accessing the same database. Each connection represents overhead and there can only be a finite amount of them. To look at a more extreme case, consider a Web site that is being hit with hundreds of thousands of hits a day. Applications that grab connections and do not let them go can have seriously negative impacts on performance and scalability.

To create a connection object, use the following syntax(using SQL Server provider):

```
SqlConnection cnn = new SqlConnection(connection_string);
```

Note: You must include the following namespace directive in order to reference the SqlConnection object: using System.Data.SqlClient;

The connection string for SQL Server Express 2012 and using LocalDB is shown below:

```
@"Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;Integrated
Security=True;Connect Timeout=30"
```

```
@"Data
Source=(LocalDB)\v11.0;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True"
```

Note: The connection string for SQL Server vary depending on the kind of the SQL Server. Shown above is the SQL Server Express 2012 Database File version.

As you can see, the connection string is made up of many parts separated by semicolons. And again, the different parts of the connection string depend on the database you are connection to.

Most modern OLE DB providers (including SQL Server provider) implement connection pooling. If you create database connections, they are held in a pool. When you want a connection for an application, the provider extracts the next available connection from the pool. When your

application closes the connection, it returns connection to the pool and makes itself available for the next application that wants a connection.

This means that opening and closing a database connection is no longer an expensive operation. If you close a connection, it does not mean you disconnect from the database. It just returns the connection to the pool. If you open a connection, it means it is simply a matter of obtaining an already open connection from the pool. It is recommended not to keep the connections longer than you need to. Therefore, you should:

- Open a connection when you need it
- Close it as soon as you have finished with it.

Once you have established a database connection object, you can use various methods and properties of this object. The usual process for working with a connection is shown below:

1. Instantiate the connection.
2. Open the connection.
3. Pass the connection to other ADO.NET objects.
4. Perform database operations with the other ADO.NET objects.
5. Close the connection.

18.3 Command Object

A SqlCommand object allows you to specify what type of interaction you want to perform with a database. For example, you can do select, insert, update, and delete commands on rows of data in a database table. The SqlCommand object can be used to support disconnected data management scenarios, but here we will only use the SqlCommand object alone.

To create a command object, use the following syntax:

```
SqlCommand cmd = new SqlCommand();
```

Note: You must include the following namespace directive in order to reference the SqlCommand object: using System.Data.SqlClient;

There are 3 additional overloaded constructors for the SqlCommand object:

```
SqlCommand cmd = new SqlCommand(command_text);
SqlCommand cmd = new SqlCommand(command_text, connection_object);
SqlCommand cmd = new SqlCommand(command_text, connection_object,
transaction_object);
```

The SqlCommand object is a fairly complex object. Shown in Table 34 and Table 35 are the most common properties and methods that you will work with.

Property	Description
Connection	The connection used to connect to the database.
CommandText	A SQL statement or the name of a stored procedure.
CommandType	A member of the CommandType enumeration. Valid values are: Text, StoredProcedure, TableDirect.
Parameters	The collection of parameters for the command object.

Table 34: SqlCommand Object Properties

Method	Description
ExecuteScalar()	Executes the query identified by the CommandText property and returns the first column of the first row.
ExecuteReader()	Executes the query identified by the CommandText property and returns the result as a SqlDataReader object.
ExecuteNonQuery()	Executes the query identified by the CommandText property and returns an integer that indicates the number of rows

Table 35: SqlCommand Object Methods

Example 7-1: Using ExecuteScalar Method

1. In Windows Explorer, navigate to C:\temp\WinApps and create a new folder named Database under it. Either create a new SQL Server database named Bank (in SQL Server Management Studio) using the script Bank_DB_Script.txt provided by the Instructor or downloadable from the instructor web site. Or simply download the created Bank.mdf file and place it in the Database.
2. Add a new windows form to the project in the PL folder and name it frmDatabase and place one button on it. Name the button btnScalar and set its Text property to "Scalar". Change the namespace to CourseExamples (remove the PL), you may also need to do this in the designer.cs file.
3. Add a button on the MainMenu form and name it btnDatabase. Set its Text property to "Database". Add the usual two lines of code to open the form frmDatabase.
4. Create the following class level variables: strConn, cnn, and cmd. Also add the namespace System.Data.SqlClient

CourseExamples.frmDatabase — btnScalar_Click(object sender, EventArgs

```
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.SqlClient;

namespace CourseExamples
{
    public partial class frmDatabase : Form
    {
        string strConn = @"Data Source=(LocalDB)\v11.0;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;"
                         + "Integrated Security=True;Connect Timeout=30";
        //string strConn = @"Data Source=.\SQLEXPRESS;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;"
        //                 +Integrated Security=True;Connect Timeout=30; User Instance=True;";
        SqlConnection cnn;
        SqlCommand cmd;

        public frmDatabase()
        {
            InitializeComponent();
        }
```

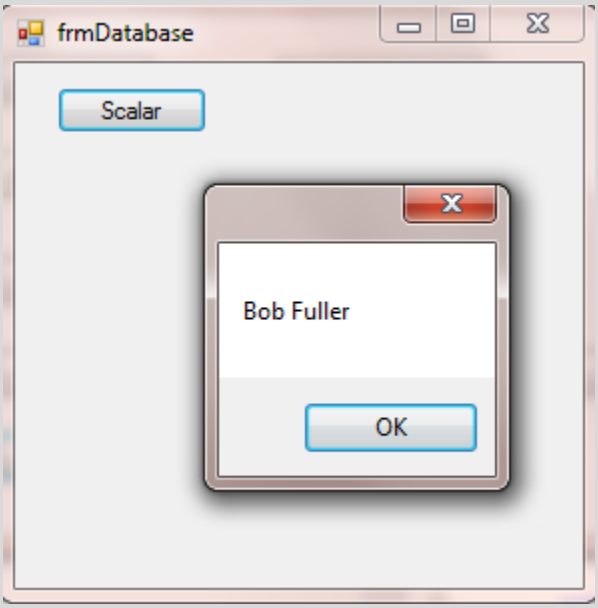
5. The string strConn is shown here in both implementations, for SQL Server Express or for LocalDB.

Example 7-1 (continued): Using ExecuteScalar Method

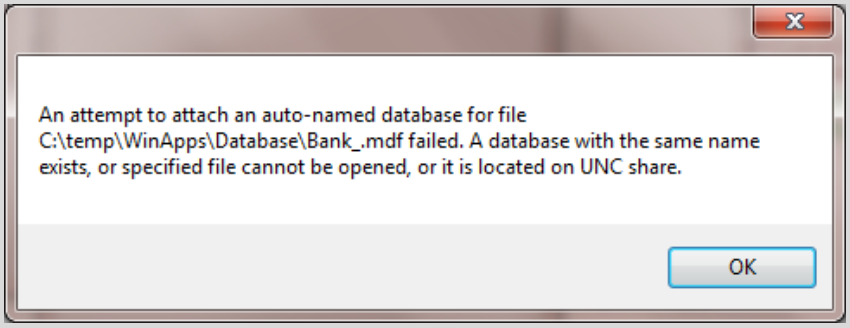
6. Now double-click on the button btnScalar to create its default click event. Write the following code:

```
private void btnScalar_Click(object sender, EventArgs e)
{
    try
    {
        cnn = new SqlConnection(strConn);
        cnn.Open();
        cmd = new SqlCommand();
        cmd.Connection = cnn;
        cmd.CommandText = "SELECT FName + ' ' + Lname AS Name FROM Customer";
        string strScalar = cmd.ExecuteScalar().ToString();
        cnn.Close();
        MessageBox.Show(strScalar);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

7. Save the project and run it. Test the database connection by clicking on the Scalar button.



8. If the database connection does not work, then take a note of the error message.



Example 7-1 (continued): Using ExecuteScalar Method

9. You can also easily connect to a Microsoft Access database. Place the instructor provided Bank.accdb database file into the database folder.
10. Then modify the reference class variables and the event method as shown below.

```
string strConn = @"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\temp\WinApps\Database\Bank.accdb";
//SqlConnection cnn;
//SqlCommand cmd;
OleDbConnection cnn;
OleDbCommand cmd;
```

```
//cnn = new SqlConnection(strConn);
cnn = new OleDbConnection(strConn);
cnn.Open();
//cmd = new SqlCommand();
cmd = new OleDbCommand();
cmd.Connection = cnn;
cmd.CommandText = "SELECT FName + ' ' + Lname AS Name FROM Customer";
string strScalar = cmd.ExecuteScalar().ToString();
cnn.Close();
MessageBox.Show(strScalar);
```

11. These are the only changes, save and run the project. It should work exactly like the SQL Server version.

When working with resources, such as the connection object, it is advisable to use the using resource acquisition statement. This construct ensures automatically that the object is closed and disposed of at the end of the using block.

Note: Any type that implements the IDisposable interface is a resource in C#.

In fact, all objects that implement the IDisposable interface should be coded using the using resource acquisition statement. The syntax is shown below:

```
using (type name = new type())
{
        Statements
}
```

Internally in C# the using block is transformed into a Try …Finally structure where in the Finally block the Dispose method is automatically called.

The SqlConnection and the SqlCommand types are resources, therefore we can rewrite the event method as shown in the next example.

Example 7-2: Using Resource Acquisition Statement

1. Based on the previous example, implement two using resource acquisition statements for the SqlConnection and the SqlCommend type as shown below.

```
private void btnScalar_Click(object sender, EventArgs e)
{
    try
    {
        using (cnn = new SqlConnection(strConn))
        {
            //cnn = new OleDbConnection(strConn);
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                //cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandText = "SELECT FName + ' ' + Lname AS Name FROM Customer";
                string strScalar = cmd.ExecuteScalar().ToString();
                MessageBox.Show(strScalar);
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

2. Note that the cnn.Close() statement is now removed. It is taken care of by the using block.
3. Save and run the project.

The ExecuteScalar method retrieves only the first column of the first row as specified by the SQL statement. If no sort order is specified when retrieving multiple rows, the first row is a random row.

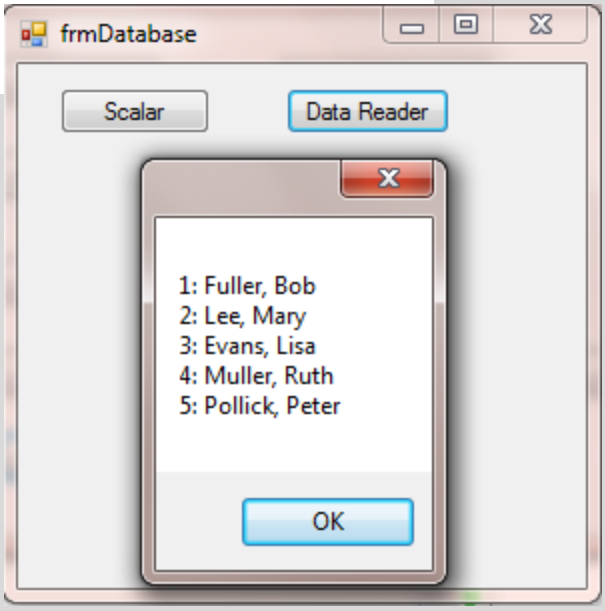
Now we ready for the next step. We want to query an entire table and display the result in a messagebox. The table customer is small and currently only holds 5 records. We are going to use the ExecuteReader method to read the data and assign the result into a string variable.

Example 7-3: Using ExecuteReader Method

1. Add a new button on the form frmDatabase and name it btnDataReader. Set its Text property to "Data Reader".
2. Double-click on the button to create its default click event and write the following code.

```
private void btnDataReader_Click(object sender, EventArgs e)
{
    try
    {
        using (cnn = new SqlConnection(strConn))
        {
            //cnn = new OleDbConnection(strConn);
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                //cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandText = "SELECT CustomerID, LName + ', ' + Fname AS Name FROM Customer";
                using (SqlDataReader sdr = cmd.ExecuteReader())
                {
                    StringBuilder sb = new StringBuilder();
                    while (sdr.Read())
                    {
                        sb.Append(sdr["CustomerID"].ToString() + ": " + sdr["Name"].ToString() + "\n");
                    }
                    MessageBox.Show(sb.ToString());
                } //closing Data Reader
            } //closing Sql Command
        } //closing Sql Connection
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

3. Save and run the project.
4. Note that the SqlDataReader was not created as another instance since the command object is already instantiated. You are simply assigning the return type of the ExecuteReader method of the command object into a SqlDataReader object.
5. Since the SqlDataReader also implements the IDisposable interface, we can again using the using resource acquisition statement.



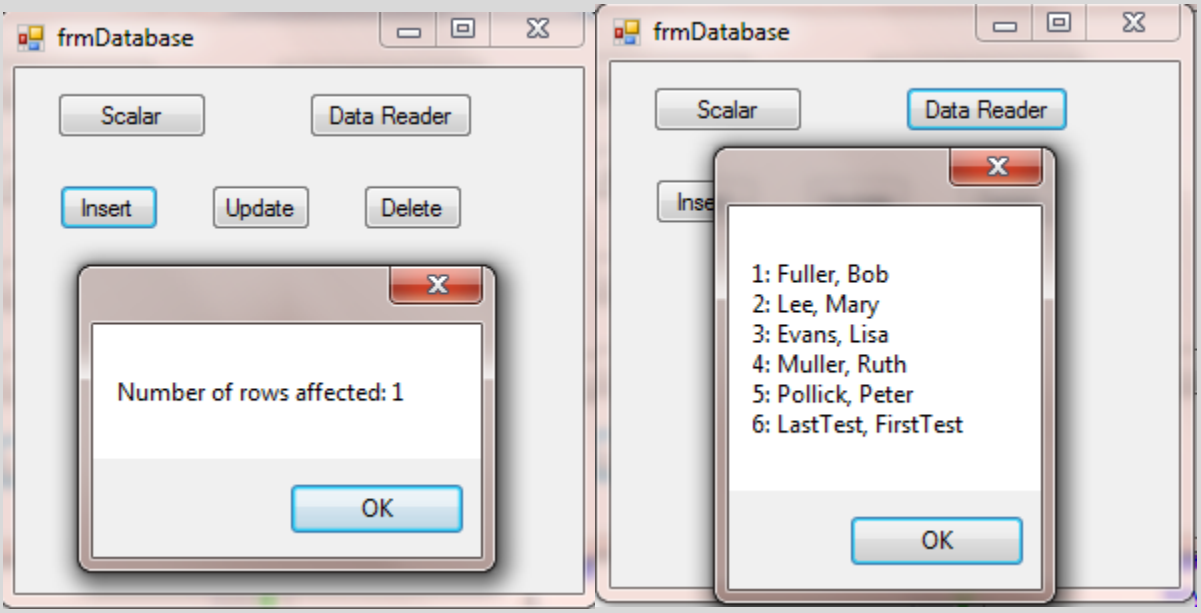
The last method of the command object is the ExecuteNonQuery method. This method allows you to issue SQL Data Manipulation Language (DML) commands against the database. In the next example, we will demonstrate the Insert, Update, and Delete statements.

Example 7-4: Using ExecuteNonQuery Method

1. Add three new buttons on the form frmDatabase and name them btnNonQueryInsert, btnNonQueryUpdate and btnNonQueryDelete. Set their respective Text properties to "Insert", "Update", and "Delete".
2. Double-click on the button btnNonQueryInsert to create the default click event. Write the following event handler.

```
private void btnNonQueryInsert_Click(object sender, EventArgs e)
{
    int intRowsAffected = 0;
    try
    {
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandText = "INSERT INTO Customer(Fname, LName) VALUES('FirstTest','LastTest')";
                intRowsAffected = cmd.ExecuteNonQuery();
            }
        }
        MessageBox.Show("Number of rows affected: " + intRowsAffected.ToString());
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

3. Save and run the project. Click on the Delete button to insert the new record. Then click on the Data Reader button to verify that the record has been inserted.



Example 7-4 (continued): Using ExecuteNonQuery Method

4. In coding the event handlers for the Update and Delete operation you may notice that the procedure is very similar to the Insert event handler. The only difference is the actual SQL statement. Therefore, let us abstract this method by creating a new method accepting a string SQL parameter as shown below.

```
private void NonQueryMethod(string strSQL)
{
    int intRowsAffected = 0;
    try
    {
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandText = strSQL;
                intRowsAffected = cmd.ExecuteNonQuery();
            }
        }
        MessageBox.Show("Number of rows affected: " + intRowsAffected.ToString());
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

5. This method is then called from each of the individual button click event procedures as shown below.

```
private void btnNonQueryInsert_Click(object sender, EventArgs e)
{
    NonQueryMethod("INSERT INTO Customer(FName, LName) VALUES('FirstTest','LastTest')");
}

private void btnNonQueryDelete_Click(object sender, EventArgs e)
{
    NonQueryMethod("DELETE FROM Customer WHERE CustomerID > 5");
    NonQueryMethod("DBCC CHECKIDENT (Customer, RESEED, 5)"); //Reseed Identity value back to 5
}
private void btnNonQueryUpdate_Click(object sender, EventArgs e)
{
    NonQueryMethod("UPDATE Customer Set FName = 'FirstUpdate' WHERE FName = 'FirstTest'");
}
```

6. Save and run the project. Test the individual DML statements and verify them using the DataReader button feature.

18.4 DataReader Object

We have already used the DataReader object in the previous examples, especially when using the ExecuteReader method of the command object. We will here discuss the DataReader in more detail.

The third component of a data provider, after connections and commands, is the data reader. Once you have connected to a database and queried it, you need some way to access the result set. This is where the data reader comes in.

Data readers are objects that implement the System.Data.IDataReaderinterface. A data reader is a fast, unbuffered, forward-only, read-only connected stream that retrieves data on a per-row basis. It reads one row at a time as it loops through a result set.

You cannot directly instantiate a data reader; instead, you create one with the ExecuteReader method of a command. The syntax for creating a SQLClient data reader is shown below:

```
SqlDataReader sdr = cmd.ExecuteReader();
```

You can now use this data reader to access the query's result set. The SqlDataReader is an abstract class and cannot be instantiated explicitly. For this reason, you obtain an instance of a SqlDataReader by executing the ExecuteReader method of SqlCommand. The ExecuteReader() does not just create a data reader; it sends the SQL to the connection for execution, so when it returns, you can loop through each row of the result set and retrieve values column by column.

To do this, you call the Read method of SqlDataReader, which returns true if a row is available and advances the cursor (the internal pointer to the next row in the result set) or returns false if another row is not available. Since Read() advances the cursor to the next available row, you have to call it for all the rows in the result set, so you call it as the condition in a while loop. The syntax is shown below:

```
while (sdr.Read())
{
        statements
}
```

Once you call the Read method, the next row is returned as a collection and stored in the DataReader object. To access data from a specific column, you can use a number of methods. One method is to use the ordinal indexer lookup method, giving the column number to the reader to retrieve values (just as you would specify an index for an array).

To use the connection for another purpose or to run another query on the database, it is important to call the Close method of DataReader to close the reader explicitly.

```
sdr.Close();
```

There are a number of methods to retrieve individual column values from a data reader row:

- Ordinal Indexer method
- Column Name Indexer method
- Typed Accessor method

Ordinal Indexer Method

You use an ordinal indexer to retrieve column data from the result set. The syntax is shown below:

```
sdr[0].ToString()
```

sdr[0] is a reference to the data reader's Item property and returns the value in the column specified for the current row. The value is returned as an object. Therefore, you use the ToString method to retrieve the actual value.

Column Name Indexer Method

Most of the time, we do not really keep track of column numbers and prefer retrieving values by their respective column names, simply because it's much easier to remember them by their names, which also makes the code more self-documenting.

You use column name indexing by specifying column names instead of ordinal index numbers. This has its advantages. For example, a table may be changed by the addition or deletion of one or more columns, upsetting column ordering and raising exceptions in older code that uses ordinal indexers.

Using column name indexers would avoid this issue, but ordinal indexers are faster, since they directly reference columns rather than look them up by name. The syntax is shown below:

```
sdr.["column_name"].ToString();
```

Typed Accessor Method

When a data reader returns a value from a data source, the resulting value is retrieved and stored locally in a .NET type rather than the original data source type. This in-place type conversion feature is a tradeoff between consistency and speed, so to give some control over the data being retrieved, the data reader exposes typed accessor methods that you can use if you know the specific type of the value being returned.

Typed accessor methods all begin with Get, take an ordinal index for data retrieval, and are type safe; C# will not allow you to get away with unsafe casts. These methods turn out to be faster than both the ordinal and the column name indexer methods. Being faster than column name

indexing seems only logical, because the typed accessor methods take ordinals for referencing; however, we need to explain how it is faster than ordinal indexing. This is because even though both techniques take in a column number, the conventional ordinal indexing method needs to look up the data source data type of the result and then go through a type conversion. This overhead of looking up the schema is avoided with typed accessors.

.NET types and typed accessor methods are available for almost all data types supported by SQL Server and OLE DB databases. The syntax is shown below:

sdr.**GetString(**1**)** for a SQL Server char data type, second column in row
sdr.**GetInt32(**3**)** for a SQL Server int data type, fourth column in row

Table 36 shows some typed access methods.

SQL Server Data Types	.NET Type	.NET Typed Accessor
bigint	Int64	GetInt64
binary	Byte[]	GetBytes
bit	Boolean	GetBoolean
char	String or Char[]	GetString or GetChars
datetime	DateTime	GetDateTime
decimal	Decimal	GetDecimal
float	Double	GetDouble
imageor long varbinary	Byte[]	GetBytes
int	Int32	GetInt32

Table 36: SQL Server Typed Accessors

19. Using Stored Procedures

In general, it is not a good idea to write any SQL into your application code due to security concerns (possible SQL injection attacks), maintenance issues when performing database schema changes, and SQL Server's compilation and performance-based reuse. It is common practice to perform all of your database operation in the database itself using stored procedures. That way, your application pages do not contain any SQL code.

19.1 Coding Storing Procedures

Writing stored procedures varies significantly across the various database platforms. Unfortunately, there is no defined standard yet for procedural SQL. Stored procedure programming is not part of this course, however, we will take a look at one already written stored procedure and execute it through ADO.NET.

All of your data operations should be performed in stored procedures. You may want to group stored procedures by business module or even by table. In Oracle, you can create packages that group many stored procedures into one package. Another way of grouping is to write all data operations for one table in one stored procedure and pass a parameter into the stored procedure indicating which operation to perform.

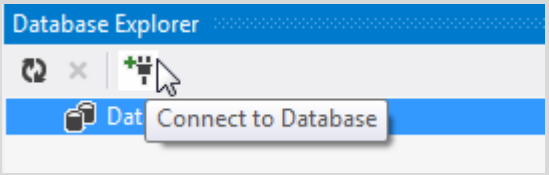
In general, for each table, you may need the following operations:

- List all data (SELECT * FROM table ORDER BY ..)
- List a specific record (SELECT * FROM table WHERE primary key = parameter)
- Count of records (Select count(*) FROM table)
- Insert operation (INSERT INTO table Values(parameters))
- Update operation (UPDATE table Set col1=param1, col2=param2, …)
- Delete operation (DELETE FROM table WHERE primary key = parameter)

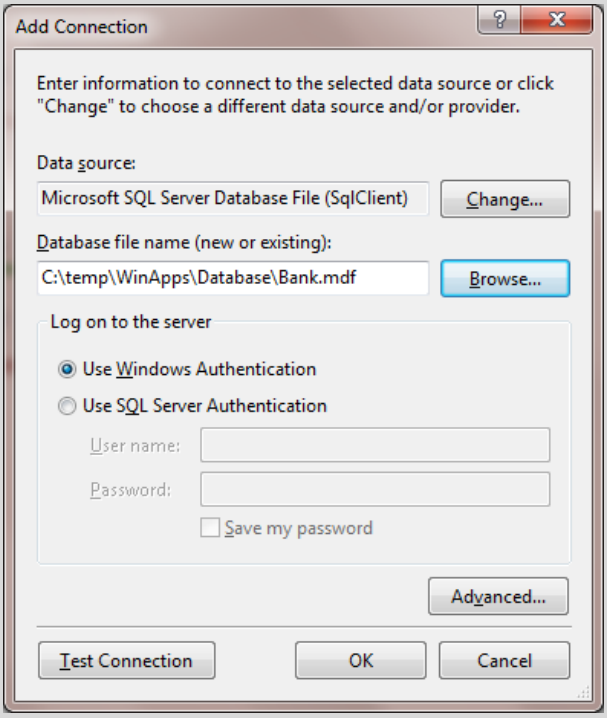
In addition to those basic statements, you may need some special operations based on business rules, for example a count of all active Customers.

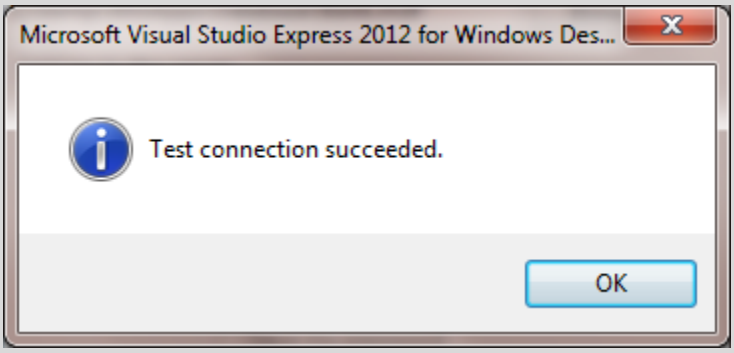
Example 7-5: Stored Procedure for table Customer

1. First of all, let us explore how we can view and modify database information through Visual Studio.
2. Display the Database Explorer window by navigating to View →Other Windows → Database Explorer (Ctrl+W, L).
3. Click on the button "Connect to Database" as shown on the right.

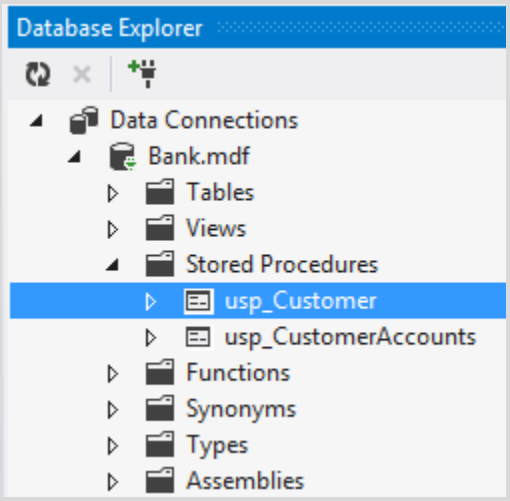


4. In the Add Connection Dialog box, select MS SQL Server Database File and navigate to the Bank.mdf file as shown on the right.
5. Click on Test Connection to verify the connection.





6. If the test is successful, you will see the above shown dialog box. Otherwise an error message is diplayed.
7. Then click on OK.



8. Expand the Bank.mdf object to view the navigation tree. Expand on Stored Procedures.
9. The Bank database already comes with a two stored procedures usp_Customer and usp_CustomerAccounts.
10. Double-click on this on usp_Customer procedure to view its content.

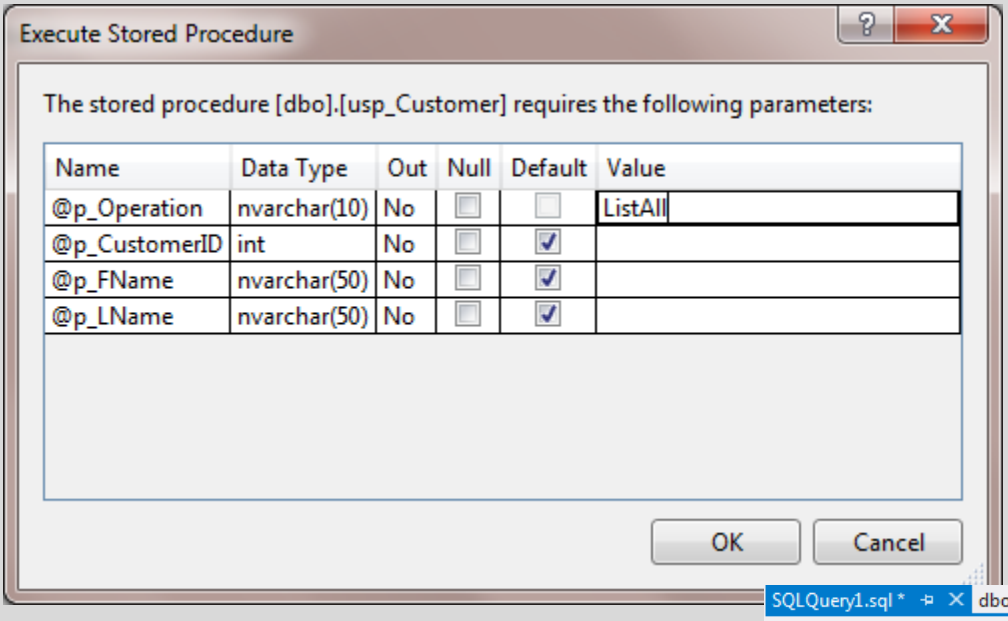
Example 7-5 (continued): Stored Procedure for table Customer

11. This procedure takes four input parameters. The first one, @p_Operation is the indicator for the kind of operation to perform. The other three parameters are simply table column values.
12. The first If statement processes a SELECT statement. It also includes an additional, derived column where the last and first name is concatenated with a comma and a space.

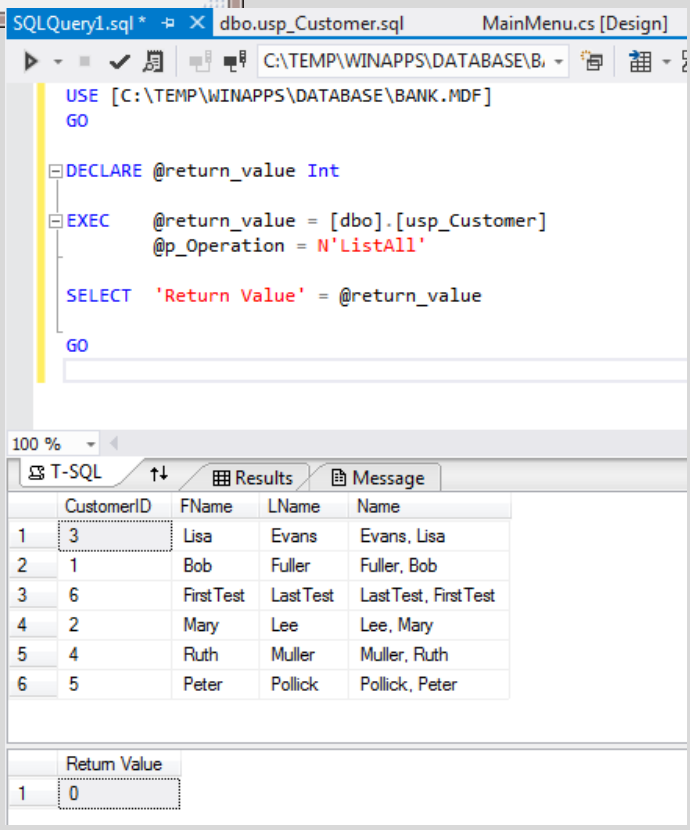
```
CREATE PROCEDURE [dbo].[usp_Customer]
    (
    @p_Operation  nvarchar(10),
    @p_CustomerID int = null,
    @p_FName      nvarchar(50)= null,
    @p_LName      nvarchar(50) = null
    )
AS
IF      @p_Operation = 'ListAll' --DQL Operations
    BEGIN
        SELECT CustomerID, FName, LName,LName + ', ' + FName as Name
        FROM Customer
        ORDER BY LName;
    END
ELSE IF @p_Operation = 'ListCust'
    BEGIN
        SELECT CustomerID, FName, LName
        FROM Customer
        WHERE CustomerID = @p_CustomerID;
    END
ELSE IF @p_Operation = 'Count'
    BEGIN
        SELECT count(*)
        FROM Customer;
    END
ELSE                            --DML Operations
    Begin
        If @p_CustomerID is null --Insert
            Begin
                INSERT INTO Customer(FName,LName)
                Values(@p_Fname,@p_LName);
            End
        ELSE IF @p_Operation='Update' --Update
            BEGIN
                UPDATE Customer
                Set FName =@p_Fname,LName =@p_LName
                WHERE CustomerID = @p_CustomerID;
            END
        ELSE
            BEGIN
                DELETE FROM Customer
                WHERE CustomerID = @p_CustomerID;
            END
    END
RETURN
```

Example 7-5 (continued): Stored Procedure for table Customer

13. The remaining part is comprised of multiple If statements for listing or counting customer records or for inserting and updating customer records.
14. Right-click in the procedure and select Execute. This allows you to run the procedure by itself. The Run Stored Procedure dialog box is displayed for specifying parameters. Type ListAll for the @p_Operation parameter.



15. When you run the procedure (click on OK), the output is displayed in a new tabbed pane.
16. In the top portion, you see the actual SQL script to run the stored procedure.
17. In the middle section, you will see the output result data.
18. At the bottom, you see a return value, which in general returns an error code (0 = no error).



19.2 Parameters and Returning Data

When calling stored procedures from ADO.NET, you need to provide the required parameters. This is done through the parameters collection of the command object. The parameters collection is also used for parameterized queries as shown below:

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Customer WHERE CustomerID = " + txtCustomerID.Text, cnn)
```

Warning:. Do not ever build a query this way! The input variable, txtCustomerID, is typically retrieved from a TextBox control on either a Windows form or a Web Page. Anything placed into that TextBox control will be put into your SQL string. This situation invites a hacker to replace that string with something malicious. In the worst case, you could give full control of your computer away.

Instead, you should parameterize the query in the following way:

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Customer WHERE CustomerID = @CustomerID", cnn)
```

Instead of dynamically building a string, as shown in the bad example above, use parameters. Anything placed into a parameter will be treated as field data, not part of the SQL statement, which makes your application much more secure.

Using parameterized queries is a three step process:

1. Construct the SqlCommand command string with parameters.
2. Declare a SqlParameter object, assigning values as appropriate.
3. Assign the SqlParameter object to the SqlCommand object's Parameters property.

Use the following syntax to create and add a parameter to the command object:

```
SqlParameter p = cmd.CreateParameter();
p.ParameterName = name;
p.SqlDbType = datatype;
p.Value = value;
cmd.Parameters.Add(p);
```

For multiple parameters, you have to repeat the steps shown above for each parameter. It is a good idea to abstract this code out into a separate method and call it repeatedly.

Example 7-6: Creating Method for Adding Parameters to Command Object

1. Create the following method in the form frmDatabase.cs file.

```
private void AddParameters(string strParam, string strValue, SqlDbType datatype)
{
    SqlParameter p = cmd.CreateParameter();
    p.ParameterName = strParam;
    p.SqlDbType = datatype;
    p.Value = strValue;
    cmd.Parameters.Add(p);
}
```

2. This method enables us to add parameters to the current command object cmd.
3. The parameter name (strParam), the value (strValue), and the database data type (datatype) is passed into this method.
4. A SqlParameter p is then created, and the name, the value, and the data type are assigned.
5. The last step is to add the parameter to the Parameters collection.

Now we call this method every time we need to add one or more parameters to the command object.

Note: Since we have declared the command object variable cmd at the class level, we have access to this variable in all methods in this class. This makes it easier than declaring an additional parameter in the AddParameters method (Command cmd). Another option would be to pass a command object (cmd) into this method.

19.3 Executing Stored Procedures using ADO.NET

Up to know we have used hard-coded SQL statements inside our application code. The command object's command type by default is Text. Therefore, we did not have to specify this value since this is the correct setting when using passing SQL statements into the database. However, as mentioned earlier, it is more advantageous to move the database related code into the database and encapsulating it in a stored procedure. The command type value now needs to be set to StoredProcedure.

The different command type values are shown in Table 37.

CommandType	Description
Text	The command will execute a direct SQL statement. The SQL statement is provided in the CommandText property. This is the default value.
StoredProcedure	The command will execute a stored procedure in the data source. The CommandText property provides the name of the stored procedure.
TableDirect	The command will query all the records in the table. The CommandText is the name of the table from which the command will retrieve the records. (This option is included for backward compatibility with certain OLE DB drivers only. It is not supported by the SQL Server data provider, and it will not perform as well as a carefully targeted query.)

Table 37: CommandType Settings

In the previous example, the CommandText property contained the actual SQL statement. When calling stored procedures, the CommandText property now needs to be set to the name of the stored procedure.

The command type values are encapsulated in a C# enumeration object, also called enum. An enum type is a distinct value type that declares a set of named constants. To set the command type use the following syntax:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = cnn;
cmd.CommandType = CommandType.StoredProcedure;
cmd.CommandText = usp_name;
```

If you need to pass parameters into a stored procedure, you need to add them to the command object as shown in the previous chapter. Finally, you call the ExecuteNonQuery method on the command object.

Example 7-7: Executing a Stored Procedure using Parameters

1. Add a new method named NonQueryMethodSP to the file frmDatabase.cs as shown below.

```
private void NonQueryMethodSP(string strOperation, string strCustomerID, string strFName, string strLName)
{
    int intRowsAffected = 0;
    try
    {
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.CommandText = "usp_Customer";
                AddParameters("@p_Operation", strOperation, SqlDbType.Text);
                AddParameters("@p_CustomerID", strCustomerID,SqlDbType.Int);
                AddParameters("@p_FName", strFName, SqlDbType.Text);
                AddParameters("@p_LName", strLName, SqlDbType.Text);
                intRowsAffected = cmd.ExecuteNonQuery();
            }
        }
        MessageBox.Show("Number of rows affected: " + intRowsAffected.ToString());
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

2. Comment out the previous calls to method NonQueryMethod in all three button event handlers. Add the new calls to this new method as shown below.

```
private void btnNonQueryInsert_Click(object sender, EventArgs e)
{
    //NonQueryMethod("INSERT INTO Customer(FName, LName) VALUES('FirstTest','LastTest')");
    NonQueryMethodSP("Insert",null, "FirstTest","LastTest");
}

private void btnNonQueryDelete_Click(object sender, EventArgs e)
{
    //NonQueryMethod("DELETE FROM Customer WHERE CustomerID > 5");
    NonQueryMethodSP("Delete", "6","FirstTest","LastTest");
    NonQueryMethod("DBCC CHECKIDENT (Customer, RESEED, 5)"); //Reseed Identity value back to 5
}
private void btnNonQueryUpdate_Click(object sender, EventArgs e)
{
    //NonQueryMethod("UPDATE Customer Set FName = 'FirstUpdate' WHERE FName = 'FirstTest'");
    NonQueryMethodSP("Update","6","FirstUpdate","LastUpdate");
}
```

3. Save and run the project. Test the insert, update, and delete functionality.

19.4 Returning Data from a Stored Procedure

A stored procedure in SQL Server can return data in three different ways:

- Through an In/Out or Out parameter
- Through the return variable (using RETURN keyword)
- Through a result set object (SELECT statement)

 **Note:** A result set object is automatically created when a SQL Select statement is created in a stored procedure.

 **Warning:**. Returning a result set from a stored procedure in other database platforms is fundamentally different, such as in Oracle. You have to return a result set in Oracle through an Out parameter return a ref sys cursor.

In order to use out parameters in a stored procedure, we have to first adjust the method for adding parameters to the command object. In addition to parameter name, value and data type we now also have to account for the parameter direction. The four different values for the parameter direction in SQL Server are shown in Table 38.

Parameter Direction	Description
Input	The parameter is an input parameter. (default)
InputOutput	The parameter is capable of both input and output.
Output	The parameter is an output parameter.
ReturnValue	The parameter represents a return value from an operation such as a stored procedure, built-in function, or user-defined function using the RETURN keyword. This is an integer data type only!

Table 38: Parameter Direction Enumeration Values

 **Note:** The Input direction is the default direction, therefore for input parameters you do not have to explicitly set the direction.

Example 7-8: Modifying AddParameters Method to Account for Direction

1. Add another parameter for the method AddParameters and use the type ParameterDirection.
2. Modify the method AddParameters as shown below.

```
private void AddParameters
    (
        string strParam,
        string strValue,
        SqlDbType datatype,
        ParameterDirection paramDirection
    )
{
    SqlParameter p = cmd.CreateParameter();
    p.ParameterName = strParam;
    p.SqlDbType = datatype;
    p.Direction= paramDirection;
    p.Value = strValue;
    cmd.Parameters.Add(p);
}
```

3. Modify all existing calls (ExecuteReaderMethodSP and NonQueryMethodSP) to this method and add the parameter direction Input as shown below.

```
AddParameters("@p_Operation", strOperation, SqlDbType.Text, ParameterDirection.Input);
AddParameters("@p_CustomerID", null, SqlDbType.Int, ParameterDirection.Input);
AddParameters("@p_FName", null, SqlDbType.Text, ParameterDirection.Input);
AddParameters("@p_LName", null, SqlDbType.Text, ParameterDirection.Input);
```

4. Save and run the project. Test all existing functionality in order to ensure that it is working correctly.

Using this modified AddParameters method we can now explore the different options of returning data from a stored procedure.

To assign the value of an output/return type parameter back into a variable in C#, you need to cast the value into the correct data type since the parameter.value is of type object. The syntax is shown below:

type *variable_name* = **(type)** *cmd*.**Parameters**[“*parameter_name*”].**Value**;

In the following examples, we will use the stored procedure usp_CustomerAccounts. This procedure returns either a count of all accounts having a balance of greater than 0 and a result set showing the customers and their respective account numbers and type.

Example 7-9: Returning Data from SP using OUT parameter

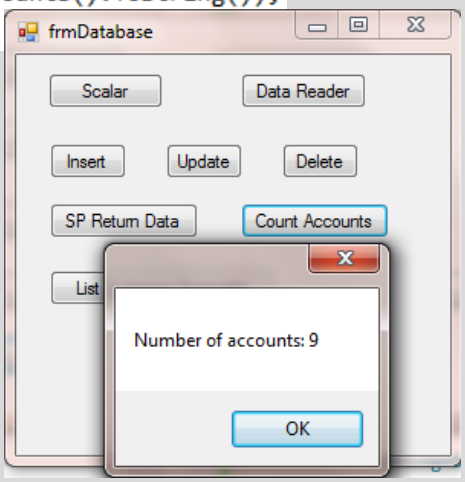
1. Add the following method to the form frmDatabase.cs file.

```
private int CountAccounts()
{
    try
    {
        int intCountAccounts;
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.CommandText = "usp_CustomerAccounts";
                AddParameters("@p_Operation", "ListCount", SqlDbType.Text, ParameterDirection.Input);
                AddParameters("@p_CountAccounts", null, SqlDbType.Int, ParameterDirection.Output);

                cmd.ExecuteNonQuery();
                intCountAccounts = (int)cmd.Parameters["@p_CountAccounts"].Value;
            }
        }
        return intCountAccounts;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        return -1;
    }
}
```

2. This method returns an integer showing the number of accounts. If an error occurs, a value of -1 is being returned.
3. Add another button on the form frmDatabase and name it btnCountAccounts. Set its Text property to Count Accounts.
4. Double-click on this button to create the default click event. Write the following statement.

```
private void btnCountAccounts_Click(object sender, EventArgs e)
{
    MessageBox.Show("Number of accounts: " + CountAccounts().ToString());
}
```

5. Save and run the project.



Example 7-10: Returning Data from SP using Return value

1. Add the following method to the form frmDatabase.cs file.

```
private string ReturnValueError()
{
    string strReturnValue = string.Empty;
    try
    {
        int intReturnValue = 0;
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.CommandText = "usp_CustomerAccounts";
                AddParameters("@p_Operation", "ListCount", SqlDbType.Text, ParameterDirection.Input);
                AddParameters("@p_CountAccounts", null, SqlDbType.Int, ParameterDirection.Output);
                AddParameters("@Return_Value", null, SqlDbType.Text, ParameterDirection.ReturnValue);
                cmd.ExecuteNonQuery();
                intReturnValue = (int)cmd.Parameters["@Return_Value"].Value;
                if ( intReturnValue == -1)
                {
                    strReturnValue = "Invalid Operation Parameter";
                }
                else if (intReturnValue == 0)
                {
                    strReturnValue = "No Stored Procedure Error";
                }
                //throw new System.DivideByZeroException();
            }
        }
    }
    catch (Exception ex)
    {
        strReturnValue = ex.Message;
    }
    return strReturnValue;
}
```

2. Notice the parameter direction of ReturnValue, this is the SQL Server return value from a stored procedure.
3. Add another button on the form frmDatabase and name it btnReturnValue. Set its Text property to “Return Value”.
4. Double-click on this button to create the default click event. Write the following statement.

```
private void btnReturnValue_Click(object sender, EventArgs e)
{
    MessageBox.Show(ReturnValueError());
}
```

5. Save and run the project. Change the value for the parameter @p_Operation to an invalid choice to test the functionality. Also uncomment out the throw new System.DivideByZeroException() line to test the C# error handler.

Example 7-11: Returning Data from SP using Result Set

1. Add the following method to the form frmDatabase.cs file.

```
private string ListCustomerAccounts()
{
    string strReturnData = string.Empty;
    try
    {
        string strCustomerName = string.Empty;
        StringBuilder sb = new StringBuilder();
        using (cnn = new SqlConnection(strConn))
        {
            cnn.Open();
            using (cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.CommandText = "usp_CustomerAccounts";
                AddParameters("@p_Operation", "ListCustomerAccounts", SqlDbType.Text, ParameterDirection.Input);
                AddParameters("@p_CountAccounts", null, SqlDbType.Int, ParameterDirection.Output);
                SqlDataReader sdr = cmd.ExecuteReader();
                while (sdr.Read())
                {
                    if (strCustomerName != sdr["CustomerName"].ToString())
                    {
                        sb.Append(sdr["CustomerName"].ToString().PadRight(25));
                    }
                    else
                    {
                        sb.Append(string.Empty.PadRight(40)) ;
                    }
                    sb.Append(sdr["AccountID"].ToString() + "(" + sdr["AccountType"].ToString() + ")\n");
                    strCustomerName = sdr["CustomerName"].ToString();
                }
            }
        }
        strReturnData = sb.ToString();
    }
    catch (Exception ex)
    {
        strReturnData = ex.Message;
    }
    return strReturnData;
}
```

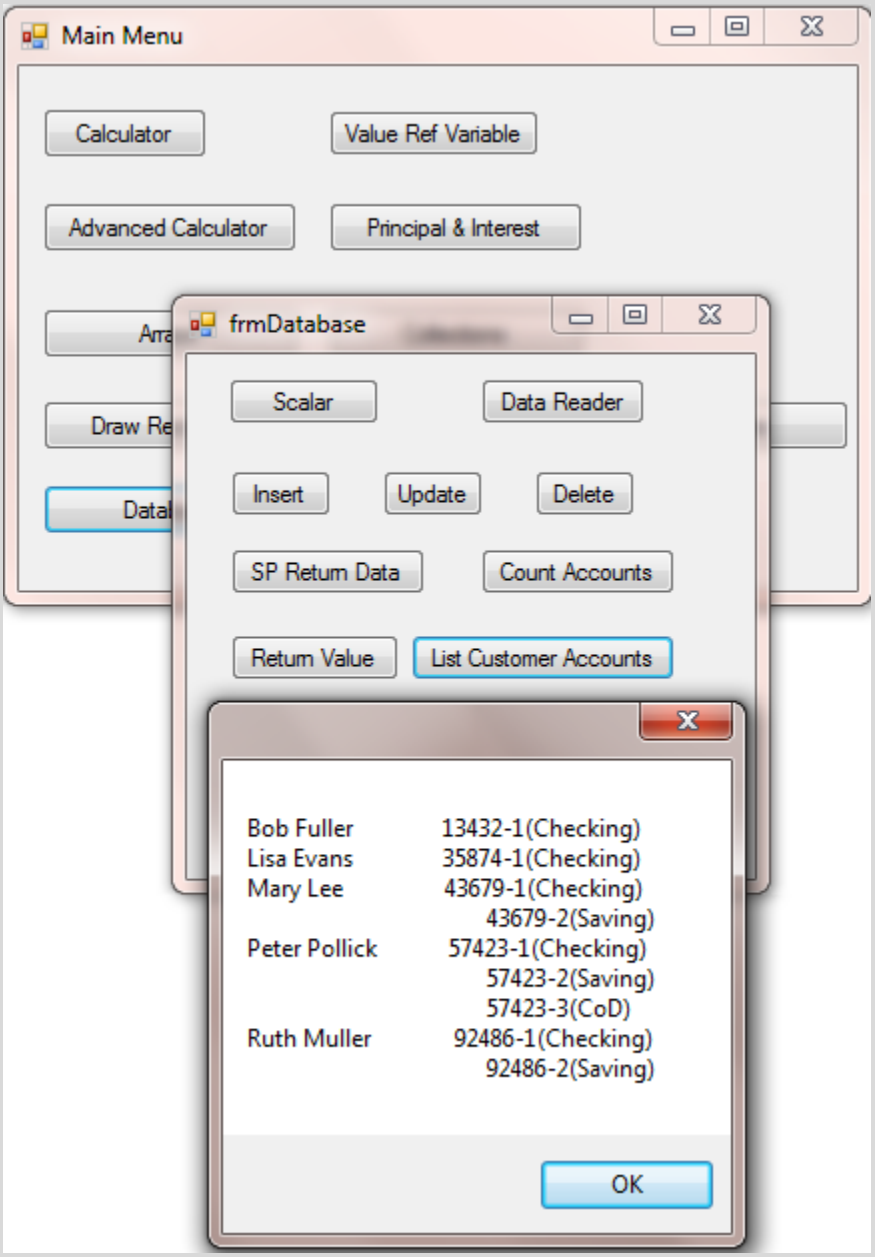
2. Note that we are using the ExecuteReader method to assign the result of a SQL statement (Operation = ListCustomerAccounts) into a SqlDataReader object.
3. We then loop through the SqlDataReader to assemble a string of customer name and the respective accounts. If a customer has more than one account, we are suppressing the customer name on the following lines using the string strCustomerName. This variable holds the previous value of the customer name and we compare it to the current one. If they equal, then do not show the customer name again, otherwise show it.

Example 7-11 (continued): Returning Data from SP using Result Set

4. Add another button on the form frmDatabase and name it btnCustomerAccounts. Set its Text property to "List Customer Accounts".
5. Double-click on this button to create the default click event. Write the following statement.

```
private void btnCustomerAccounts_Click(object sender, EventArgs e)
{
    MessageBox.Show(ListCustomerAccounts());
}
```

6. Save and run the project.



CLASS 8

19. Database Programming, Disconnected Architecture

We discussed a fully connected mode of operation for interacting with a data source by using the SqlConnection and the SqlCommand object. We learned to read data quickly and let go of the connection with the SqlDataReader. In this chapter we see how to accomplish something in-between SqlConnection and SqlDataReader interaction by using the DataSet and SqlDataAdapter objects.

19.1 DataSet Object

Unlike DataReaders, which are objects of data provider–specific classes that implement the System.Data.IDataReader interface, data sets are objects of the class System.Data.DataSet, a distinct ADO.NET component used by all data providers. Data sets are completely independent of and can be used either connected to or disconnected from data sources. Their fundamental purpose is to provide a relational view of data stored in an in-memory cache.

DataReader vs. DataSet

If you simply want to read and display data, then you need to use only a data reader, particularly if you are working with large quantities of data. In situations where you need to loop through thousands or millions of rows, you want a fast sequential reader (reading rows from the result set one at a time), and the data reader does this job in an efficient way.

If you need to manipulate the data in any way and then update the database, you need to use a data set. A data adapter fills a data set by using a data reader; additional resources are needed to save data for disconnected use. You need to think about whether you really need a data set; otherwise, you will just be wasting resources. Unless you need to update the data source or use other data set features such as reading and writing to XML files, exporting database schemas, and creating XML views of a database, you should use a data reader.

Overview of DataSet Object

A DataSet is an in-memory data store that can hold numerous tables. DataSets only hold data and do not interact with a data source. It is the DataAdapter that manages connections with the data source and gives us disconnected behavior.

A couple scenarios illustrate why you would want to work with disconnected data: people working without network connectivity and making Web sites more scalable. Consider sales people who need customer data as they travel. At the beginning of the day, they will need to sync up with the main database to have the latest information available. During the day, they will make modifications to existing customer data, add new customers, and input new orders. This is okay because they have a given region or customer base where other people will not be changing the same records. At the end of the day, the sales person will connect to the network and update changes for overnight processing.

Another scenario is making a Web site more scalable. With a DataReader, you have to go back to the database for records every time you show a page. This requires a new connection for each page load, which will hurt scalability as the number of users increase. One way to relieve this is to use a DataSet that is updated one time and stored in cache. Every request for the page checks the cache and loads the data if it is not there or just pulls the data out of cache and displays it. This avoids a trip to the database, making your application more efficient.

Exceptions to the scenario above include situations where you need to update data. You then have to make a decision, based on the nature of how the data will be used as to your strategy.

Use disconnected data when your information is primarily read only, but consider other alternatives (such as using Command object for immediate update) when your requirements call for something more dynamic.

Also, if the amount of data is so large that holding it in memory is impractical, you will need to use DataReader for read-only data. Really, one could come up with all kinds of exceptions, but the true guiding force should be the requirements of your application which will influence what your design should be.

DataSet Object Architecture

A DataSet is comprised of the following objects as shown in Figure 71:

- DataTable (→ Rows, Columns, Constraints)
- DataRelation

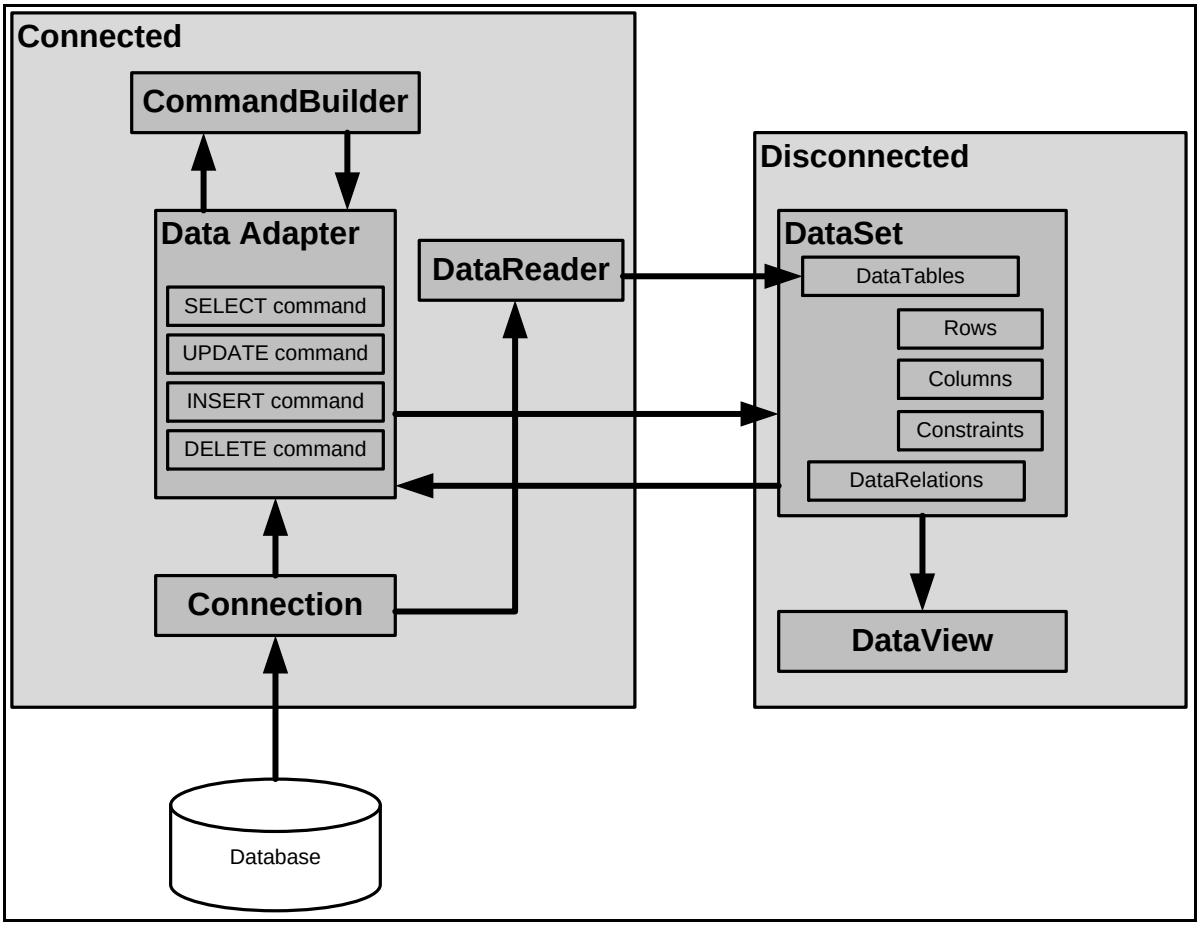


Figure 71: Disconnected Architecture

The "Connected" part in the above diagram only shows component needed to manage the "Disconnected" part of ADO.NET.

A DataSet can contain one or more tables depending on how you fill the dataset. Each table object has data column, data row, and a constraint object that you access to. If you have more than one table in the DataSet, then the Data relation object is filled as well.

To create a dataset object, use the following syntax:

```
DataSet ds = new DataSet();
```

The DataSet constructor does not require any parameters. However there is one overload that accepts a string for the name of the DataSet, which is used if you were to serialize the data to XML. Right now, the DataSet is empty and you need a SqlDataAdapter to load it.

19.2 DataTable Object

A data table is an instance of the class System.Data.DataTable. It is conceptually analogous to a relational table. A data table has collections of data rows and data columns. You can access these nested collections via the Rows and Columns properties of the data table. A data table can be created in three different ways:

- As an independent table object (without a data set) created manually in C#
- As an independent table object (without a data set) created through a data reader object
- As part of data set object created through the data adapter object

We will demonstrate these different ways in the following examples.

The data table object is a powerful one as most controls on a Windows form can be bound to a data table object. If you connect to a database, you can create a data set and then use the data table object to refer to the database tables or views.

To create a table object manually, use the following syntax:

```
DataTable dt = new DataTable();
DataRow dr;
dt.Columns.Add(column_name, data_type);
dr = dt.NewRow();
dr[column1_name] = value1;
dr[column2_name] = value2;
……
dt.Rows.Add(dr);
```

To create a data table using a data reader object, use the Load method of the data table.

```
dt.Load(sdr)
```

To create a data table from a data set (ds), you would first load the data set using data adapter (da) and then use the following syntax to point to a data table:

```
DataSet ds = new DataSet();
da.Fill(ds,"table_name");
dt = ds.Tables["table_name"];
```

A data column represents the schema of a column within a data table and can then be used to set or get column properties. You obtain the collection of data columns using the data table's Columns property, whose indexer accepts either a column name or a zero-based index.

```
DataColumn col = dt.Columns["ContactName"];
```

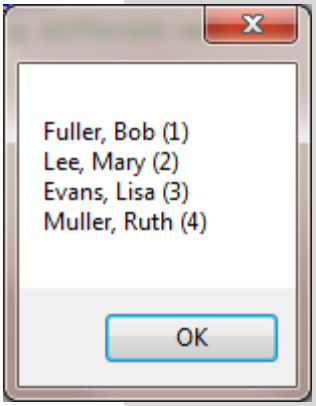
A data row represents the data in a row. You can programmatically add, update, or delete rows in a data table. To access rows in a data table, use its Rows property using a zero-based index:

```
DataRow row = dt.Rows[2];
```

Example 8-1: Creating DataTable manually

1. Add a new windows form to the project in the PL folder and name it frmDataSet and place one button on it. Name the button btnDataTableManual and set its Text property to "Data Table Manual". Change the namespace to CourseExamples (remove the PL), you may also need to do this in the designer.cs file.
2. Add a button on the MainMenu form and name it btnDataSet. Set its Text property to "DataSet/DataTable". Add the usual two lines of code to open the form frmDataSet.
3. Double-click on the button btnDataTableManual to create the default click event. Write the following code.

```
private void btnDataTableManual_Click(object sender, EventArgs e)
{
    //Setting up datatable and datarow object
    DataTable dt = new DataTable();
    DataRow dr;
    //Defining datatable
    dt.Columns.Add("CustomerID", typeof(int));
    dt.Columns.Add("FName", typeof(string));
    dt.Columns.Add("LName", typeof(string));
    //Populating datatable with data using different methods
    //1st row
    dr = dt.NewRow();
    dr[0] = 1;
    dr[1] = "Bob";
    dr[2] = "Fuller";
    dt.Rows.Add(dr);
    //2nd row
    dr = dt.NewRow();
    dr["CustomerID"] = 2;
    dr["FName"] = "Mary";
    dr["LName"] = "Lee";
    dt.Rows.Add(dr);
    //3rd row
    dr = dt.NewRow();
    object[] obj = { 3, "Lisa", "Evans" };
    dt.Rows.Add(obj);
    //4th row
    dt.Rows.Add(4, "Ruth", "Muller");
    //Looping through database and displaying values
    StringBuilder sb = new StringBuilder();
    foreach (DataRow datarow in dt.Rows)
    {
        sb.Append(datarow["LName"] + ", " + datarow["FName"] + " (" + datarow[0] + ")\n");
    }
    MessageBox.Show(sb.ToString());
}
```



4. Save and run the project. Note the output in the windows message box.

Note: Every row in the above example was created using a different method.

In the next example, we will need the connection string to connect to the database again. Since we will be using a new form, we would have to create the same logic over again. Now the time has come to implement some application-wide settings. I will present two main possibilities here, but keep in mind there are many more:

- Create a static public class and expose the connection string as a get accessor property
- Add the connection string to the app.config file and use the ConfigurationManager class to access it

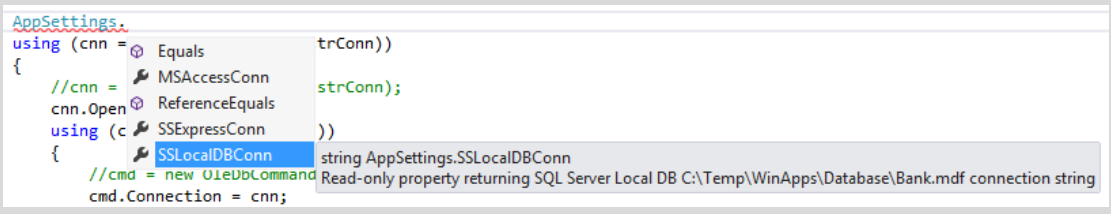
The next example will show a new static class to expose the connection string.

Example 8-3: Creating Static Class to Store Application Settings

1. Add a new class into the BAL folder and name it AppSettings.cs
2. Write the following code (copy the connection strings from form frmDatabase.cs).

```
AppSettings.cs | frmDatabase.cs* | frmDatabase.cs [Design]* | frmDataSet.cs | frmDataSet.cs [Design]
CourseExamples.BAL.AppSettings                                  MSAccessConn
 7  namespace CourseExamples.BAL
 8  {
 9      public static class AppSettings
10      {
11          private static string _SSLocalDBConn = @"Data Source=(LocalDB)\v11.0;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;"
12                                  + "Integrated Security=True;Connect Timeout=30";
13          private static string _SSExpressConn = @"Data Source=.\SQLEXPRESS;AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;"
14                                 + "Integrated Security=True;Connect Timeout=30; User Instance=True;";
15          private static string _MSAccessConn = @"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\temp\WinApps\Database\Bank.accdb";
16
17          /// <summary>
18          /// Read-only property returning SQL Server Local DB C:\Temp\WinApps\Database\Bank.mdf connection string
19          /// </summary>
20          public static string SSLocalDBConn
21          {
22              get { return _SSLocalDBConn; }
23          }
24          /// <summary>
25          /// Read-only property returning SQL Server Express C:\Temp\WinApps\Database\Bank.mdf connection string
26          /// </summary>
27          public static string SSExpressConn
28          {
29              get { return _SSExpressConn; }
30          }
31          /// <summary>
32          /// Read-only property returning MS Access C:\Temp\WinApps\Database\Bank.accdb connection string
33          /// </summary>
34          public static string MSAccessConn
35          {
36              get { return _MSAccessConn; }
37          }
38      }
39  }
```

3. Save and close this class.
4. Navigate to the file frmDatabase.cs and add the namespace CourseExamples.BAL.
5. Inside any method, type AppSettings and the dot and notice the Intelli-Sense displaying the three different read-only properties.
6. Click on one of the properties to see the tooltip info.



Example 8-4: Storing Application Settings in App.config file

1. Double-click on the App.config file in Solution Explorer. Add an <AppSettings> section as shown.

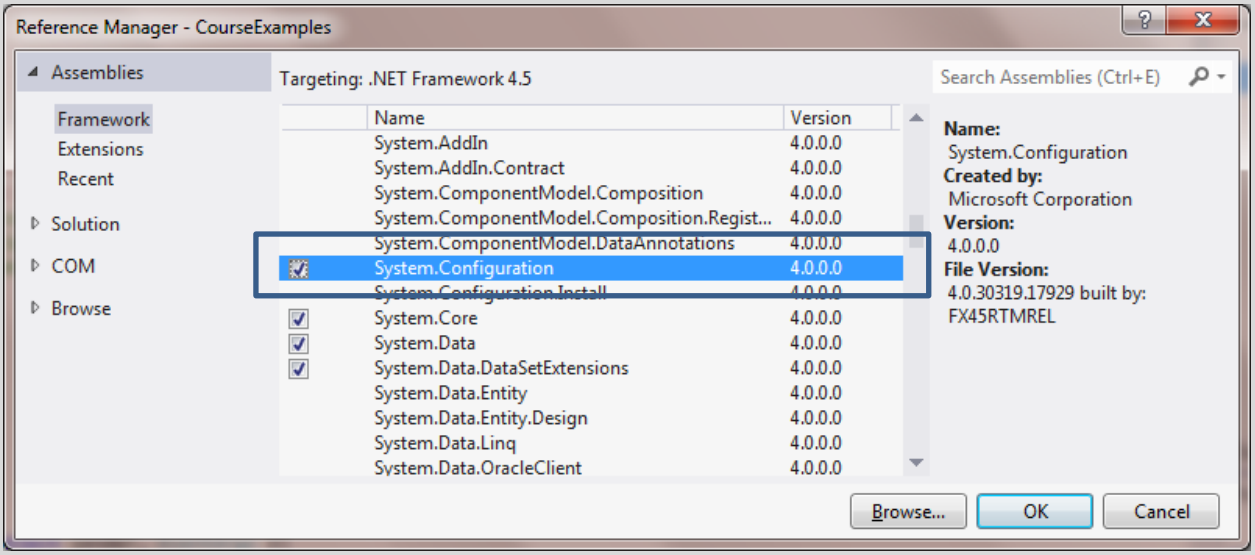
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <appSettings>
    <add key="SSLocalDBConnectionString" value="Data Source=(LocalDB)\v11.0;
        AttachDbFilename=C:\temp\WinApps\Database\Bank.mdf;Integrated Security=True;Connect Timeout=30"
  </appSettings>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
</configuration>
```

2. Save and close the file. Navigate to the file frmDatabase.cs file and add the namespace System.Configuration.
3. In the btnScalar_click methods define a string variable strSSLocalDB and assign it the following expression.

```
private void btnScalar_Click(object sender, EventArgs e)
{
    string strSSLocalDB = ConfigurationManager.AppSettings["SSLocalDBConnectionString"];
    try
    {
```

The name 'ConfigurationManager' does not exist in the current context

4. You noticed the red, wavy underline under ConfigurationManager and the error message.
5. It turns out that the namespace System.Configuration exists in two assemblies, in System.dll and also in System.Configuration.dll. The configuration.dll assembly is not included by default in a windows form project.
6. To add the reference to System.Configuration.dll, right-click on the folder References in Solution Explorer and select Add Reference. Place a checkmark in the square next to SystemConfiguration and click on OK.



7. Now you notice that the ConfigurationManager class is recognized.
8. Replace the string variable strConn with strSSLocalDB, save and run the project.
9. Click on the Scalar button to test the database connection.

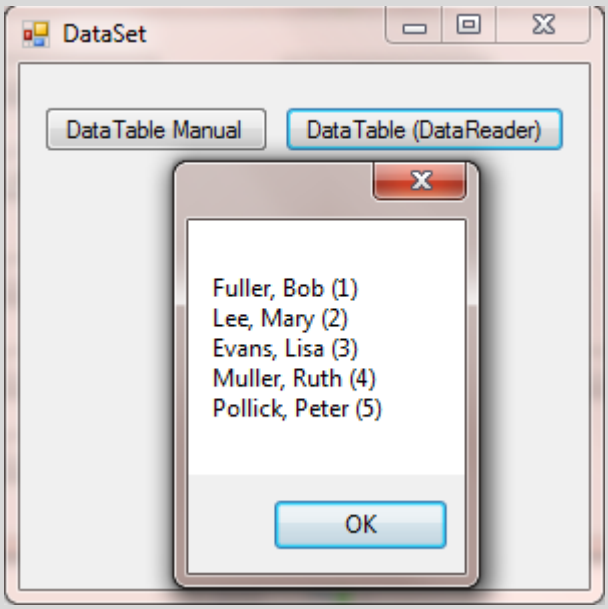
The next example will use a DataReader object to fill a data table object.

Example 8-5: Creating DataTable using a DataReader

1. Add a new button on frmDataSet.cs and name it btnDataTableDataReader. Set its Text property to "Data Table (DataReader)".
2. Double-click on the button to create the default click event.
3. Add namespaces System.Data.SqlClient, CourseExamples.BAL and write the following code.

```
private void btnDataTableDataReader_Click(object sender, EventArgs e)
{
    string strSSLocalDB = AppSettings.SSLocalDBConn;
    try
    {   //connecting to db
        using (SqlConnection cnn = new SqlConnection(strSSLocalDB))
        {   //Establishing command object and SQL statement
            cnn.Open();
            using (SqlCommand cmd = new SqlCommand())
            {
                cmd.Connection = cnn;
                cmd.CommandText = "SELECT * FROM customer";
                SqlDataReader sdr = cmd.ExecuteReader();
                DataTable dt = new DataTable();
                //Loading datatable through datareader
                dt.Load(sdr);
                //Looping through database and displaying values
                StringBuilder sb = new StringBuilder();
                foreach (DataRow datarow in dt.Rows)
                {
                    sb.Append(datarow["LName"] + ", " + datarow["FName"] + " (" + datarow[0] + ")\n");
                }
                MessageBox.Show(sb.ToString());
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

4. Save and run the project.
5. Click on the button "Data Table (DataReader)". The result should be the same as clicking the first button (Data Table Manual) (except for some data differences).



19.3 DataAdapter Object

The SqlDataAdapter holds the SQL commands and connection object for reading and writing data. You initialize it with a SQL select statement and connection object. The syntax is shown below:

```
SqlDataAdapter da = new SqlDataAdapter(SQL_SELECT_statement, connection_object);
```

The code above creates a new SqlDataAdapter, da. The SQL select statement specifies what data will be read into a DataSet. The connection object should have already been instantiated, but not opened. It is the SqlDataAdapter's responsibility to open and close the connection during Fill and Update method calls.

As indicated earlier, the SqlDataAdapter contains all of the commands necessary to interact with the data source. The code showed how to specify the select statment, but did not show the insert, update, and delete statements. These are added to the SqlDataAdapter after it is instantiated.

There are two ways to add insert, update, and delete commands:

- via SqlDataAdapter properties
- via SqlCommandBuilder

The syntax how to add commands to the SqlDataAdapter with the SqlCommandBuilder is shown below:

```
SqlCommandBuilder cb = new SqlCommandBuilder(da);
```

Notice in the code above that the SqlCommandBuilder is instantiated with a single constructor parameter of the SqlDataAdapter (da) instance. This tells the SqlCommandBuilder what SqlDataAdapter to add commands to. The SqlCommandBuilder will read the SQL select statement (specified when the SqlDataAdapter was instantiated), infer the insert, update, and delete commands, and assign the new commands to the Insert, Update, and Delete properties of the SqlDataAdapter, respectively.

The SqlCommandBuilder has limitations. It works when you do a simple select statement on a single table. However, when you need a join of two or more tables or must do a stored procedure, it does not work. In that case, you have to manually add the statements to the SqlAdapter insert, update, and delete properties.

To add the commands manually to the SqlDataAdapter object, use the the following syntax:

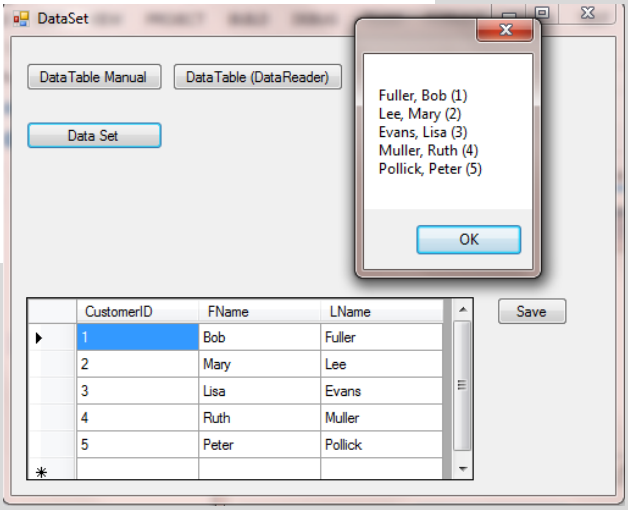
```
da.UpdateCommand = new SqlCommand("SQL update statement", cnn);
da.InsertCommand = new SqlCommand("SQL insert statement", cnn);
da.DeleteCommand = new SqlCommand("SQL delete statement", cnn);
```

Example 8-6: Creating DataTable using DataAdapter and DataSet

1. Add a new button on frmDataSet.cs and name it btnDataSet. Set its Text property to "Data Set".
2. Add a DataGridView control (under Data) to the form (resize form if necessary). Name the control dgvCustomer.
3. Add another button on the right side of the grid view control. Name it btnSave and set its Text property to "Save". Double-click on this button to create the default click event.
4. At the top of the class, create the following reference variables as shown on the right.
5. Now double-click on the button btnDataSet to create the default click event. Write the following code.

```
public partial class frmDataSet : Form
{
    SqlDataAdapter da;
    DataSet ds;
    SqlConnection cnn;
    DataTable dt;
    public frmDataSet()
    {
        InitializeComponent();
    }
```

```
private void btnDataSet_Click(object sender, EventArgs e)
{
    try
    {   //Instantiate DataSet object
        ds = new DataSet();
        //Creating connection
        cnn = new SqlConnection(AppSettings.SSLocalDBConn);
           //Note we did not open connection!!
            //Instantiate Data Adapter
            da = new SqlDataAdapter("SELECT * FROM customer", cnn);
            //Instantiate CommandBuilder to generate DML statements for Data Adapter
            SqlCommandBuilder cb = new SqlCommandBuilder(da);
            //Fill Data Set with data
            da.Fill(ds, "DS_Customer");
            dt = new DataTable();
            dt = ds.Tables["DS_Customer"];
            //Looping through database and displaying values
            StringBuilder sb = new StringBuilder();
            foreach (DataRow datarow in dt.Rows)
            {
                sb.Append(datarow["LName"] + ", " + datarow["FName"] + " (" + datarow[0] + ")\n");
            }
            MessageBox.Show(sb.ToString());
            dgvCustomer.DataSource = dt;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```



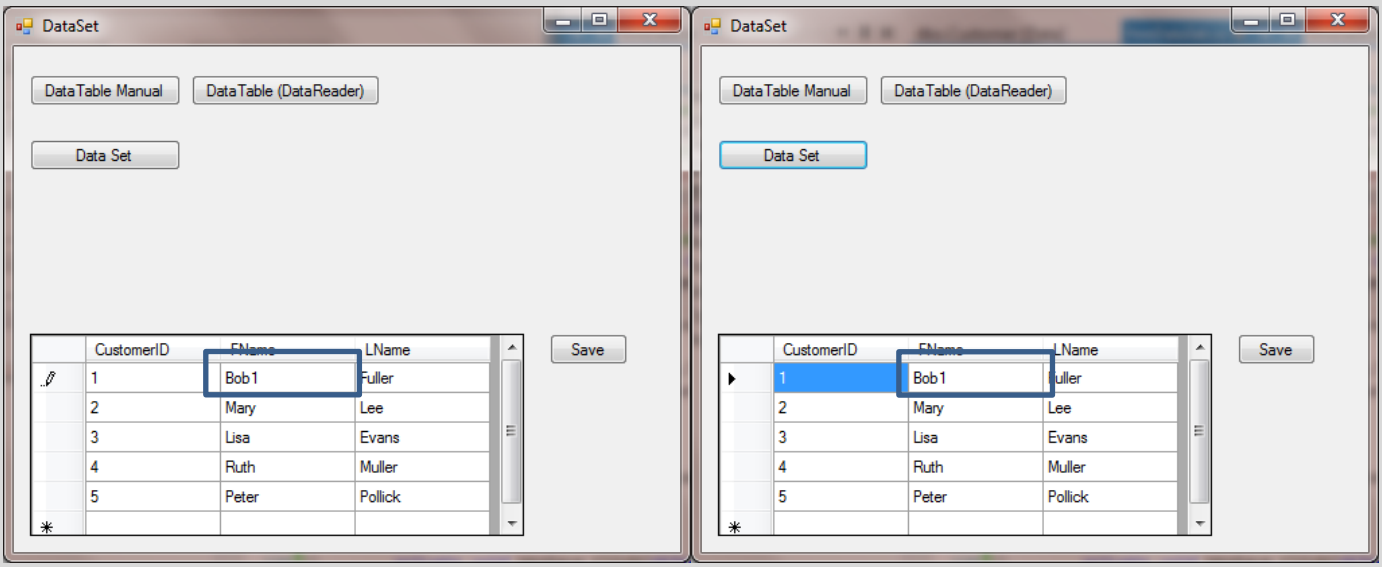
6. Save and run the project.

Example 8-6 (continued): Creating DataTable using DataAdapter and DataSet

7. Code the event handler for the button btnSave as shown below.

```
private void btnSave_Click(object sender, EventArgs e)
{
    try
    {
        da.Update(ds, "DS_Customer");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

8. Save and run the project.
9. Test the update capability. Once the data grid is populated, make a change to either the first or last name. Then click on the "Data Set" button to verify that the data changes actually were propagated back to the database.



Warning:. In the above example we did not use the using resource acquisition statement for the connection object. While this construct is in most cases desired, in this situation it does not work. Remember, the data adapter takes care of opening and closing a connection, there is no need to use additional programming constructs. For example, in the save button event handler, the Update method of the data adapter automatically opens the connection, saves the data back to the database, and also closes the connection.

You have seen that updating a database with data sets and data adapters is relatively straightforward. However, we have oversimplified things; we have been assuming that no other changes have been made to the database while we are working with disconnected data sets.

Imagine two separate users trying to make conflicting changes to the same row in a data set and then trying to propagate these changes to the database. What happens? How does the database resolve the conflicts? Which row gets updated first, second, or at all?

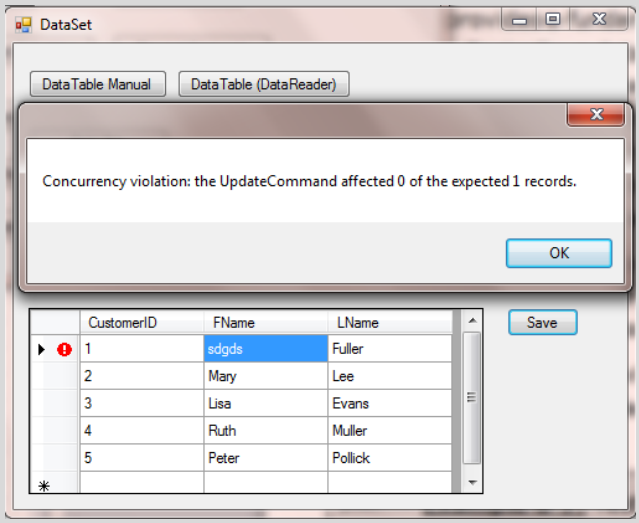
ADO.NET provides a fundamental level of concurrency control that is designed to prevent update anomalies. Basically, a data set marks all added, modified, and deleted rows. If a row is propagated to the database but has been modified by someone else since the data set was filled, the data manipulation operation for the row is ignored (in fact an error is thrown). This technique is known as optimistic concurrency and is essentially the job of the data adapter. When the Update method is called, the data adapter attempts to reconcile all changes. This works well in an environment where users seldom contend for the same data.

This type of concurrency is different from what is known as pessimistic concurrency, which locks rows upon modification (or sometimes even on retrieval) to avoid conflicts. Most database management system use some form of locking to guarantee data integrity.

Disconnected processing with optimistic concurrency is essential to successful multitier systems. How to employ it most effectively given the pessimistic concurrency of DBMSs is a thorny problem.

Example 8-7: Testing Concurrency of DataAdapter

1. In the database explorer, right-click on table Customer and select Show Table Data.
2. Now run the project, click on the Data Set/Data Table button, then on the Data Set button.
3. In the data grid, make a change to the data.
4. Go back to Visual Studio, and in the Customer [data] tab make a change to the same column and row.
5. Go back to the form and click on the Save button. Notice the error message displayed.



20. Entity Framework

With the release of .NET Framework 4.5, a new version of ADO.NET Entity Framework 5.0 was introduced. This chapter will introduce you to the ADO.NET Entity Framework 5.0 data model, also known as the Entity Data Model (EDM). EDM is Microsoft's way of implementing object-relational mapping (ORM). ORM is a way of processing and mapping relational datainto a collection of objects, called entities.

20.1 Introduction to Entity Framework

The vision behind ADO.NET Entity Framework (EF) 5.0 is to extend the level of abstraction for database programming and completely remove the impedance mismatch between data models and development languages that programmers use to write database-oriented software applications.

ADO.NET EF 5.0 allows developers to focus on data through an object model instead of through the traditional logical/relational data model, helping abstract the logical data schema into a conceptual model, a mapping layer, and a logical layer to allow interaction with that model through a new data provider called EntityClient.

ADO.NET EF 5.0 allows developers to write less data access code, reduces maintenance, and abstracts the structure of the data into a more business-friendly manner. It can also help reduce the number of compile-time errors since it generates strongly typed classes from the conceptual model.

ADO.NET EF 5.0 generates a conceptual model that developers can write code against using a new data provider called EntityClient, as mentioned previously. EntityClient follows a model similar to familiar ADO.NET objects, using EntityConnection and EntityCommand objects to return an EntityDataReader.

The core of ADO.NET EF is its Entity Data Model. ADO.NET EF supports a logical store model that represents the relational schema from a database.

A relational database often stores data in a different format from what the application can use. This typically forces developers to retrieve the data in the same structure as that contained in the database. Developers then often feed the data into business entities that are more suited for handling business rules.

ADO.NET EF bridges this gap between data models using mapping layers. There are three layers active in ADO.NET EF's model.

- Conceptual layer
- Mapping layer
- Logical layer

These three layers allow data to be mapped from a relational database to a more object-oriented business model. ADO.NET EF defines these layers using XML files. These XML files provide a level of abstraction so developers can program against the OO conceptual model instead of the traditional relational data model.

The conceptual model is defined in an XML file using Conceptual Schema Definition Language (CSDL). CSDL defines the entities and the relationships as the application's business layer knows them.

The logical model, which represents the database schema, is defined in an XML file using Store Schema Definition Language (SSDL). The mapping layer, which is defined using Mapping Schema Language (MSL), maps the other two layers. This mapping is what allows developers to code against the conceptual model and have those instructions mapped into the logical model.

20.2 Implementing Entity Framework

To implement the entity framework, you add a new item to your project and select the Entity Framework template as shown in Figure 72.

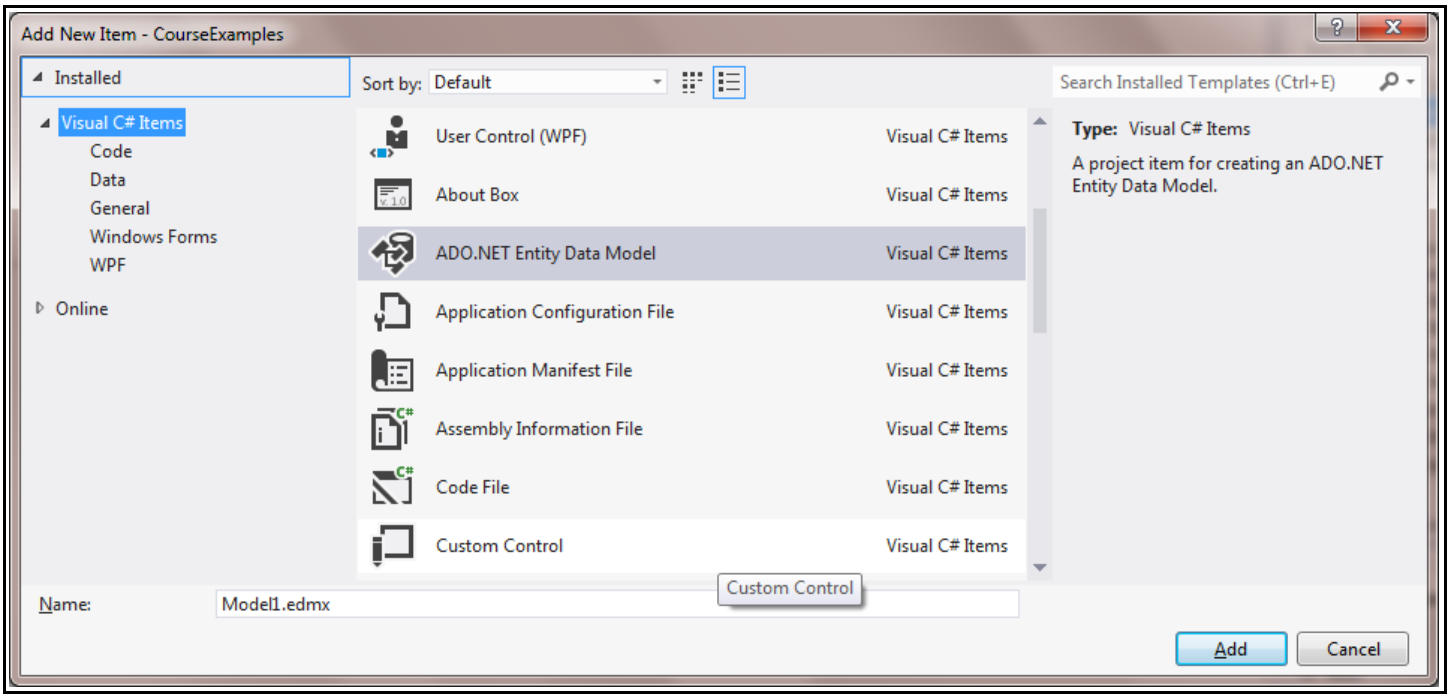


Figure 72: ADO.NET Entity Data Model Template

Then a wizard guides you through the process of setting up the EF, mainly connecting to the database, selecting the object to map to, and finally creating the entire framework. In the end all the three different layers are created through xml files as shown in Figure 73.

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="3.0" xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema Namespace="BankModel.Store" Alias="Self" Provider="System.Data.SqlClient" ProviderManifestToken="20
        <EntityContainer Name="BankModelStoreContainer">
          <EntitySet Name="Account" EntityType="BankModel.Store.Account" store:Type="Tables" Schema="dbo" />
          <EntitySet Name="Customer" EntityType="BankModel.Store.Customer" store:Type="Tables" Schema="dbo" />

    <!-- CSDL content -->
    <edmx:ConceptualModels>
      <Schema Namespace="BankModel" Alias="Self" p1:UseStrongSpatialTypes="false" xmlns:annotation=
        <EntityContainer Name="BankEntities" p1:LazyLoadingEnabled="true">
          <EntitySet Name="Accounts" EntityType="BankModel.Account" />
          <EntitySet Name="Customers" EntityType="BankModel.Customer" />
          <AssociationSet Name="FK_Account_Customer" Association="BankModel.FK_Account_Customer">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Account" EntitySet="Accounts" />
          </AssociationSet>
        </EntityContainer>
<!-- C-S mapping content -->
<edmx:Mappings>
  <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
    <EntityContainerMapping StorageEntityContainer="BankModelStoreContainer" CdmEntityContainer="BankEntities">
      <EntitySetMapping Name="Accounts">
        <EntityTypeMapping TypeName="BankModel.Account">
          <MappingFragment StoreEntitySet="Account">
            <ScalarProperty Name="AccountID" ColumnName="AccountID" />
            <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
            <ScalarProperty Name="AccountType" ColumnName="AccountType" />
            <ScalarProperty Name="Balance" ColumnName="Balance" />
          </MappingFragment>
        </EntityTypeMapping>
```

Logical Model

Conceptual Model

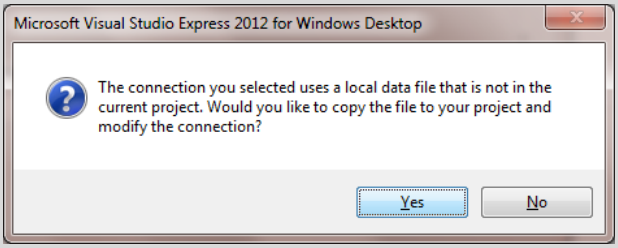
Mapping Layer

Figure 73: EF Generated Layer Files

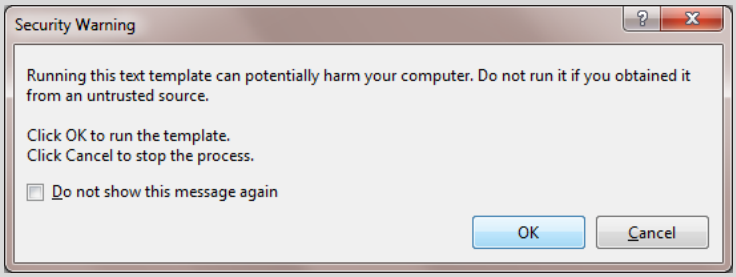
Once the EF model is setup, programming against this framework is fairly simple and does not involve any connection, command or data reader objects. You simply deal with entities in an object-oriented layer.

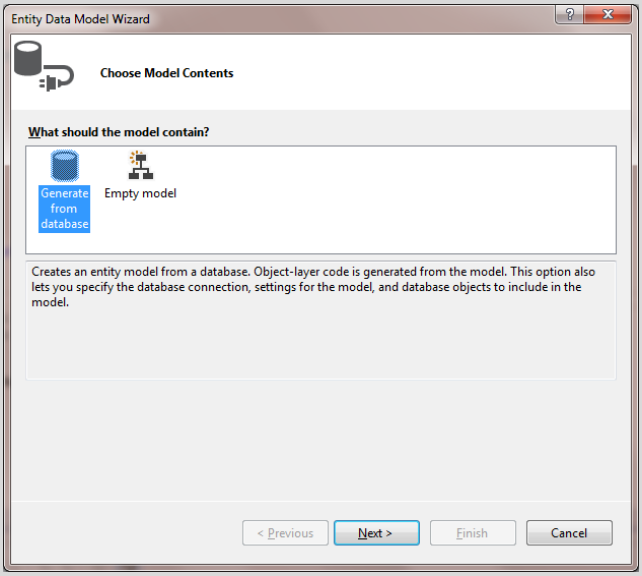
Example 8-8: Implementing EF Model

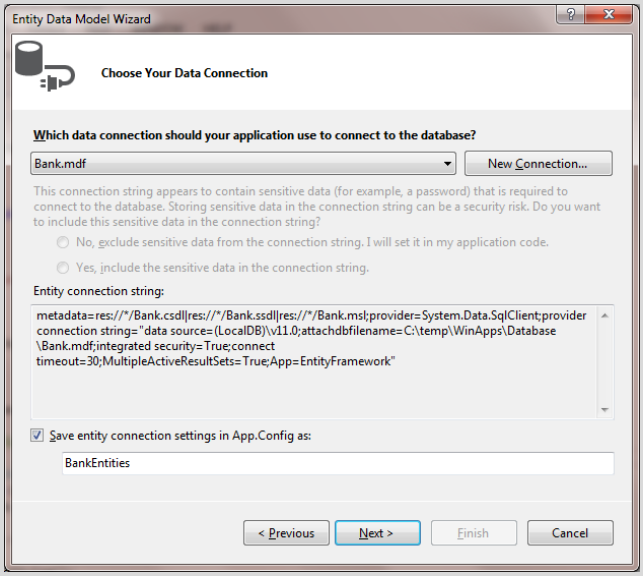
1. Right-click on the project and select New Item.
2. Select the ADO.NET Entity Data Model. Name it Bank.edmx. Click on Add.
3. In the next step, select Database. Click on Next.
4. The next step entails connecting to a database.Since we already have a connection string setup in the App.config file, the wizard picks up this connection. Otherwise you would click on New Connection and go through the steps to establish a connection to the database.
5. Leave everything as is and click on Next.
6. You will see a messagebox informing you that you have a local database and whether you want to include it in the current project. Click on NO, since we want the database to be separated from this project.

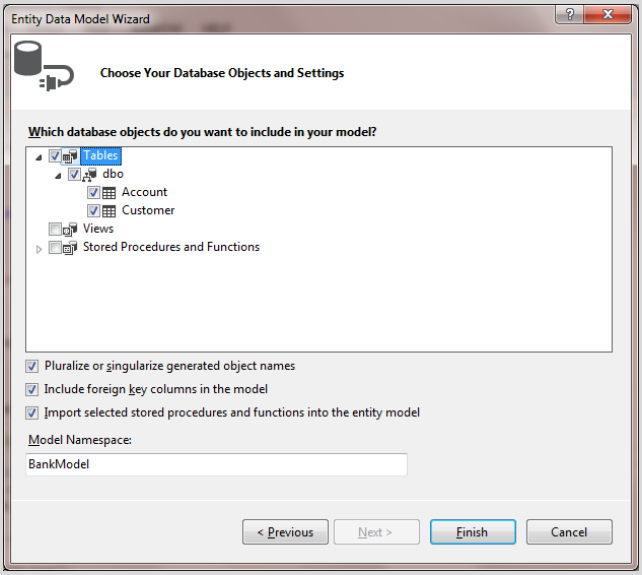


7. In the next step you select the database objects to be included. Select Customer and Account tables as shown.
8. Click on Finish.
9. Now it is actually running the process. You might see a warning message box as shown below. Click on OK.



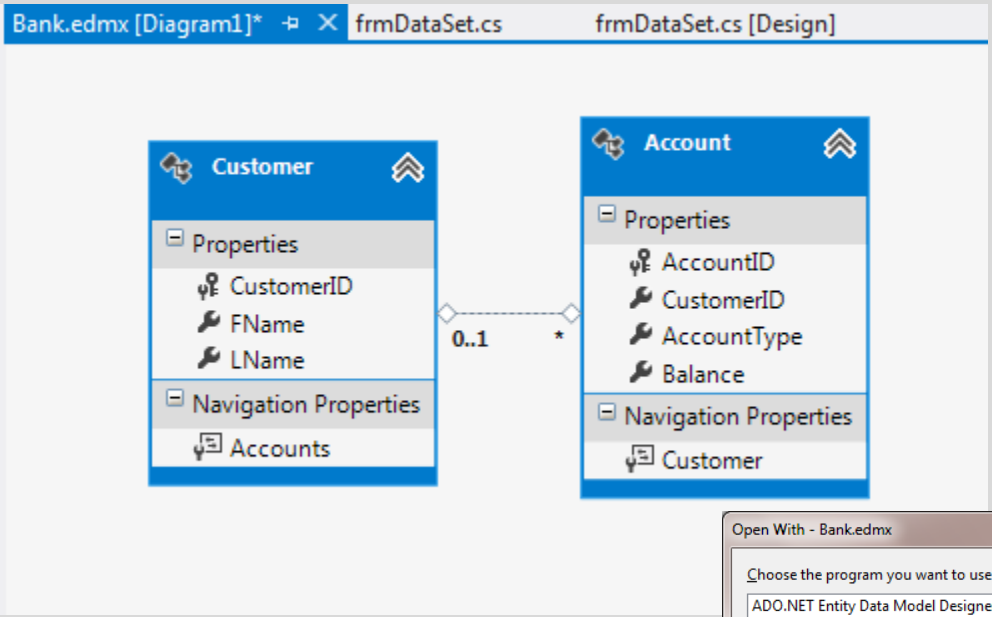




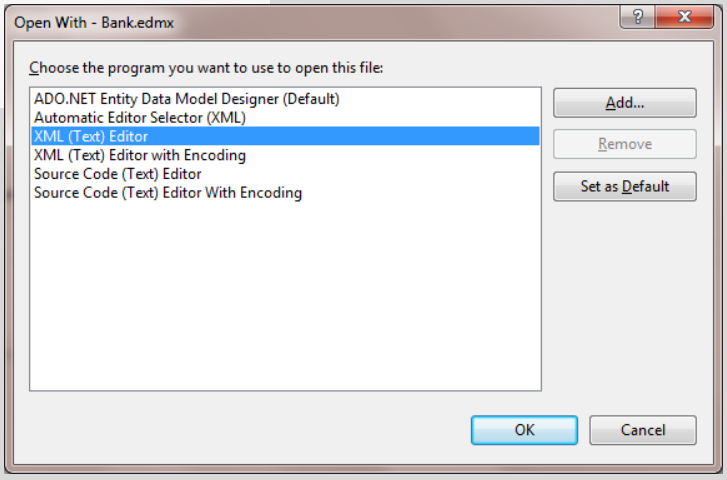


Example 8-8 (continued): Implementing EF Model

10. Now you see a diagram that looks like a database entity relationship diagram. This is now the mapped representation of the database in C#.



11. Close this diagram and right-click on the Bank.edmx file in Solution Explorer and select Open With. From the dialog box, select XML (Text) Editor. Click OK



12. Now you see the XML file behind for the EF model. When you scroll through, you should see all three layers (SSDL, CSDL,C-S mapping).
13. Add a new button to the form frmDataSet and name it bntEntityFramework. Set its Text property to Entity Framework.
14. Double-click on this button to create the default click event. Write the following code.

```
private void btnEntityFramework_Click(object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder();
    BankEntities bank = new BankEntities();
    foreach (var row in bank.Customers)
    {
        sb.Append(row.LName + ", " + row.FName + "(" + row.CustomerID + ")\n");
    }
    MessageBox.Show(sb.ToString());
}
```

15. Save and run the project. You should see the familiar messagebox.

21. Binding Data to Form Controls

We are finally at the point where we are going to cover some user interface elements. Although we have already used some in a limited fashion, we are going to discuss in this chapter some controls and how to bind them to database data. In the next class, we will cover forms and controls in a comprehensive manner.

21.1 Overview of Data Binding

There are two main methods of data binding as shown below:

- Simple Data Binding (single value)
- Complex Data Binding (multiple value)

These two methods map to certain user interface controls. Some controls are designed to display only a single value, such as text box. Other controls are geared towards displaying complex data, such as drop-down or list box controls.

Using Visual Studio you again have quite a few options to implement data binding. There are automatic methods where you use objects that handle most of the work for you. These objects are:

- Data Source, Binding Source, Navigation Control
- Entity Framework

And then you can write your own customized code to implement data binding. In this class we will go the harder way and write our own code.

The most commonly used controls for single binding are listed below:

- Label (read-only)
- Textbox, Rich Textbox
- Checkbox, Radiobutton
- DateTimePicker, MonthCalendar

The controls for complex binding are listed below:

- Listbox
- Drop-down, Combobox
- DataGridView

21.2 Simple Data Binding

For a textbox control, you can use the Databindings.Add method or directly assign a value to the Text property of a textbox control from a data table object as shown below.

```
textbox_name.Databindings.Add(property_name, datatable, column_name)
textbox_name.Text = datatable.Rows[0][1].ToString();
```

In the add method of the databindings object only the first row is referenced. In the second example you can exactly specify which row of the datatable you want to reference.

To specify a specific row in the data table for the Databindings.Add method create a DataView object from the data table and apply a filter as shown below.

```
DataView dv = new DataView(dt);
dv.Filter = "column_name = value";
textbox_name.Databindings.Add(property_name, dv, column_name)
```

You could theoretically run the ExecuteScalar method of the command object and assign it directly to the Text property. This, of course, works if you only need to retrieve one column.

```
textbox_name.Text = cmd.ExecuteScalar().ToString();
```

In most cases, you want to retrieve a row or even rows of data at once and assign the data to multiple controls. In this case you would use a data table object to store the returned result set from the database.

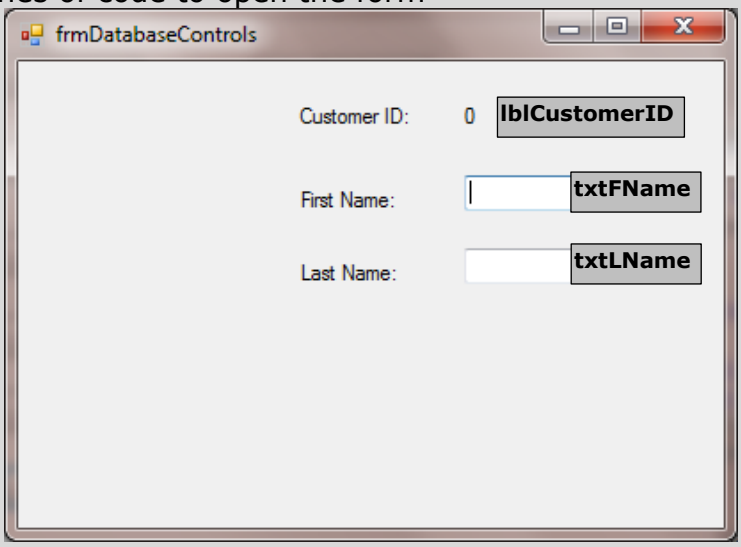
.NET has expanded the scope of possible data providers. In .NET any class or component that implements the IList interface is a valid DataSource. If a component implements the IList interface then it is transformed into an index based collection.

Some of the classes that support the IList interface in the NET framework are given below. Please note that any class that implements the IList interface is a valid data provider.

- Arrays
- DataColumn
- DataTable
- DataView
- DataSet

Example 8-9: Simple Data Binding

1. Add a new windows form to the project in the PL folder and name it frmDatabaseControls. Change the namespace to CourseExamples (remove the PL), you may also need to do this in the designer.cs file.
2. Add a button on the MainMenu form and name it btnDatabaseControls. Set its Text property to "Database Controls". Add the usual two lines of code to open the form frmDatabaseControls.
3. Place four label and two textbox controls as shown on the right. The left side labels do not need to be named. The top right side label and the two textboxes should be named as shown.
4. Double-click on the form surface (not on any control) to create the form's default load event.
5. Add the namespaces System.Data.SqlClient and CourseExamples.BAL.




```
using System.Data.SqlClient;
using CourseExamples.BAL;
```

6. Copy and paste the method AddParameters from the form frmDatabase.cs into this form.
7. Create the following class level reference variables.

```
public partial class frmDatabaseControls : Form
{
    SqlConnection cnn;
    SqlCommand cmd;
    DataTable dt;
    public frmDatabaseControls()
    {
        InitializeComponent();
    }
```

8. Create the following method to establish a connection

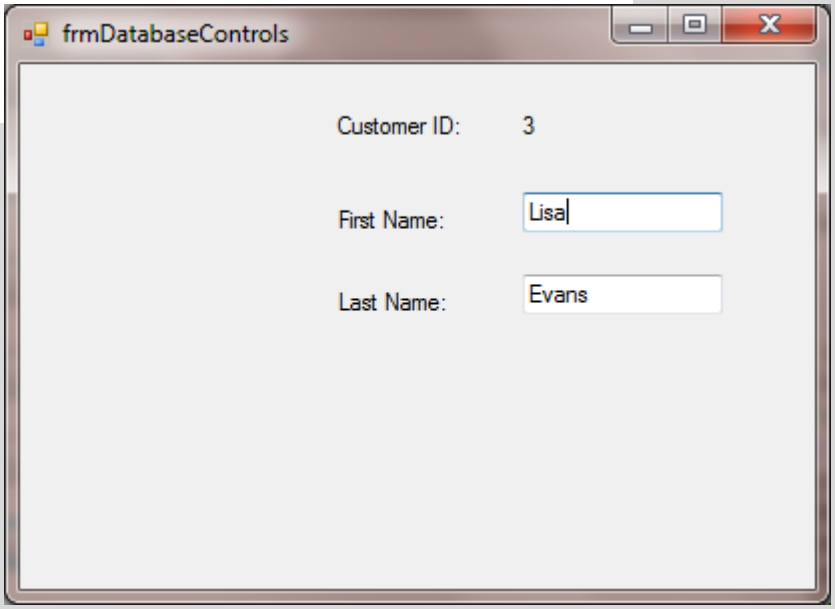
```
private void DbConnection()
{
    cnn = new SqlConnection(AppSettings.SSLocalDBConn);
    cnn.Open();
}
```

Example 8-9 (continued): Simple Data Binding

9. Write the following code in the load event.

```
private void frmDatabaseControls_Load(object sender, EventArgs e)
{
    try
    {
        //Create and open connection
        DbConnection();
        //Establishing command object
        cmd = new SqlCommand();
        cmd.Connection = cnn;
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.CommandText = "usp_Customer";
        AddParameters("@p_Operation", "ListAll", SqlDbType.Text, ParameterDirection.Input);
        //Creating and loading datatable from database
        dt = new DataTable();
        dt.Load(cmd.ExecuteReader());
        //Assigning first row of datatable to controls
        //Using Text property assignment
        //lblCustomerID.Text = dt.Rows[0][0].ToString();
        //txtFName.Text = dt.Rows[0][1].ToString();
        //txtLName.Text = dt.Rows[0][2].ToString();
        //Using DataBinding.Add method
        DataView dv = new DataView(dt);
        dv.RowFilter = "[CustomerID]=3";
        lblCustomerID.DataBindings.Add("Text", dv, "CustomerID");
        txtFName.DataBindings.Add("Text", dv, "FName");
        txtLName.DataBindings.Add("Text", dv, "LName");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

10. Note that the event handler contains both methods of assigning values to label and textbox controls. Comment and uncomment the sections to test both methods.
11. Save and run the project.



21.2 Complex Data Binding

Complex binding always involves at least multiple column/values, or at most an entire result set consisting of many rows and columns. There are a couple of controls that can consume complex data as shown at the beginning of this chapter.

Let us focus first on the drop-down/combobox and listbox controls since these controls are very similar. These controls can be bound to two columns, one column that is hidden (not displayed to the user) which is normally a key column or other unique value, and the other column being the displayed value.

The syntax of binding data to these controls is shown below. The variable ctl can either refer to a listbox or drop-down/combobox control.

```
ctl.DataSource = datatable
ctl.ValueMember = datatable.Columns[0].ToString()
ctl.DisplayMember = datatable.Columns[1].ToString()
```

Note: You can also use a data reader, data view, or an array as a data source.

A data grid view control is rich data control that allows you to manage the entire data set using the control. It looks like a spreadsheet, where each individual cell can be edited. To bind a data grid view control, use the following syntax:

```
dgv.DataSource = datatable;
```

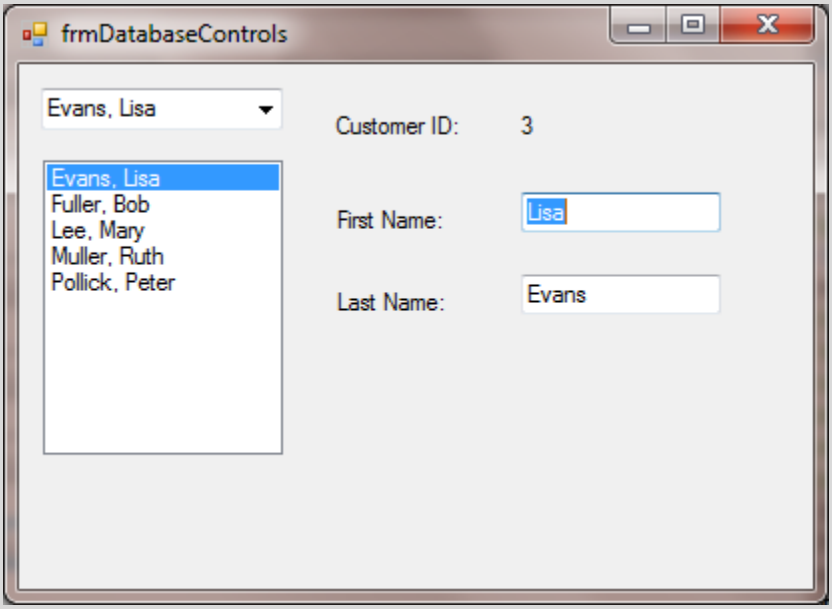
There are no other settings needed, the control will read the data source and determine the number of columns and column headers and render the control accordingly.

Example 8-10: Complex Data Binding

1. Add a combobox control to the top left side of the form frmDatabaseControls. Name this control cmbCustomer.
2. Add a listbox control below the combobox and name it lstCustomer.
3. In the form's load event method, add the following code at the end of the current code.

```
//Binding Listbox
lstCustomer.DataSource = dt;
lstCustomer.ValueMember = dt.Columns[0].ToString();
lstCustomer.DisplayMember = dt.Columns[3].ToString();
//Binding Combobox
cmbCustomer.DataSource = dt;
cmbCustomer.ValueMember = dt.Columns[0].ToString();
cmbCustomer.DisplayMember = dt.Columns[3].ToString();
```

4. Save and run the project. Test the combobox and listbox controls.



CLASS 9

22. Windows Forms

At this point we have already designed and developed quite a few forms and controls. This and the next chapter will go into more detail of form design and also cover more kinds of controls.

22.1 Windows Forms Architecture

You can create Windows Forms using the designer interface in Visual Studio. As you set properties for the form and add controls to the form, Visual Studio automatically writes C# Code that creates this form when the application is run.

A Windows Form is comprised of three files as shown in Figure 74:

- *form_name*.Designer.cs
- *form_name*.cs
- *form_name*.resx

Figure 74: Windows Form Files

The Designer.cs file contains auto-generated code that stores the visual representation of the form. A sample file is shown in Figure 75. In general, you do not have to edit this file at all; in fact, a warning is included in this file as comment not to edit this file with the code editor.

When you double-click on the form.cs file, you actually see first a graphical representation of the form which is read from the form.Designer.cs file. So what you actually see is the form.Designer.cs file.

Every action that is performed manually in the designer interface, such as adding or modifying a control is automatically recorded in the form.Designer.cs file. A Windows form can be completely build using code, it is based on .NET classes. The Designer interface is simply a tool for the developer to quickly build forms graphically.

```
namespace CourseExamples
{
    partial class frmDatabaseControls
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.txtFName = new System.Windows.Forms.TextBox();
            this.lstCustomer = new System.Windows.Forms.ListBox();
            this.txtLName = new System.Windows.Forms.TextBox();
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.label3 = new System.Windows.Forms.Label();
            this.lblCustomerID = new System.Windows.Forms.Label();
            this.cmbCustomer = new System.Windows.Forms.ComboBox();
            this.SuspendLayout();
```

Figure 75: Form's Designer File

The form_name.cs file is a complimentary C# code file for the form where custom code is stored. Both .cs form files are merged at compile time into one .cs form class file. You may notice that both files are partial classes of the form. This is the C# compiler directive to merge those two files into one class. This makes it easier to separate class files for various reasons. In case of a Windows form, it separates user code from system generated code.

The third file, the .resx file, is a resource file that mostly stores globalization and localization information (for multi-language support). It further stores additional properties of the form, such as icon files, for example. The x at the end of the file extension stands for xml, this file is a xml file. In general, you do not have to maintain this file manually, Visual Studio automatically maintains this file.

The bottom line is that you have to toggle only between two interfaces, the designer interface and the custom code file in the code editor. You can use the Project Explorer to quickly toggle between these two interfaces or right-click in form in design mode. Select the form file in Project Explorer and either right-click on it and select View Designer or View Code. Or click at that the top of the Project Explorer window on the appropriate buttons displaying the same icons as in the shortcut menu shown in Figure 76.

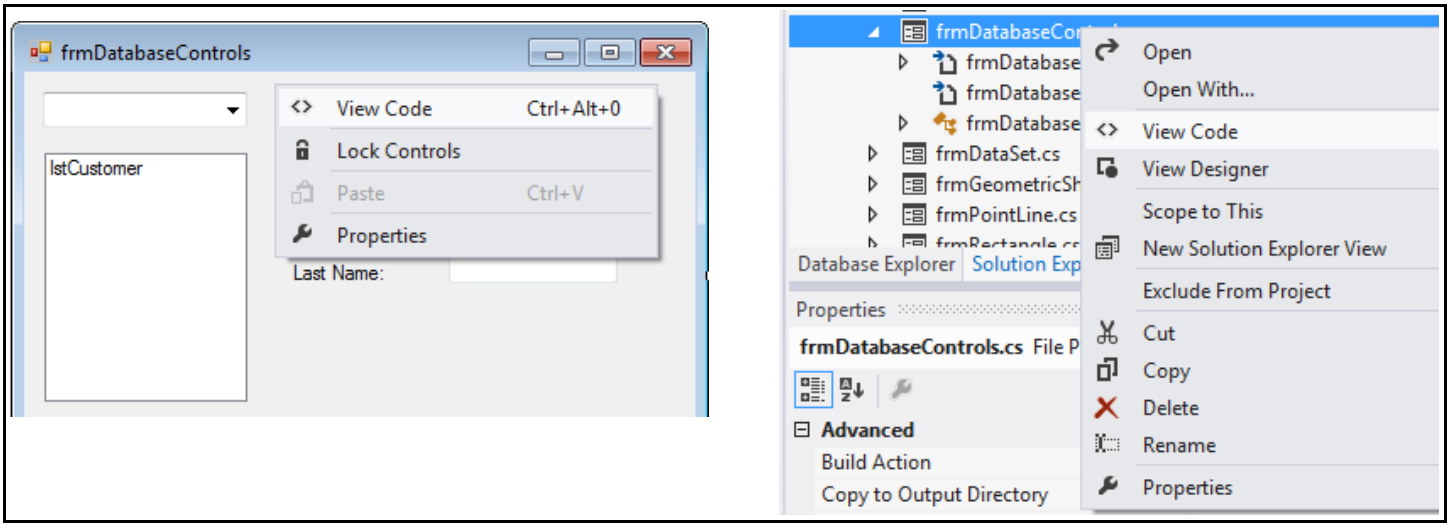


Figure 76: Switching between Design and Code Files

22.2 Forms Properties and Methods

Setting properties for a form can be performed by using the property sheet in designer view. The property sheet is shown in Figure 77. The categories are:

- Accessibility
- Appearance
- Behavior
- Data
- Design
- Focus
- Layout
- Misc
- Window Style

Many of the properties are self-explanatory, and a short description is displayed at the bottom of the property sheet when the property is selected. The most common properties are the Text and the Name property. The text property is displayed in the title bar of the form. The Name property is the unique identifier within your project, you use the name property to refer to this form in C# code. In fact, any object has a name property so that you can refer to it in code.

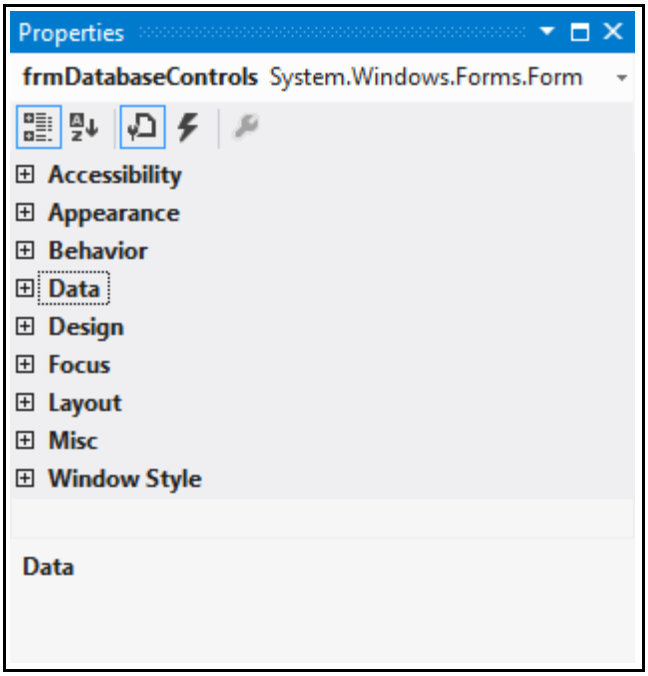


Figure 77: Form's Property Categories

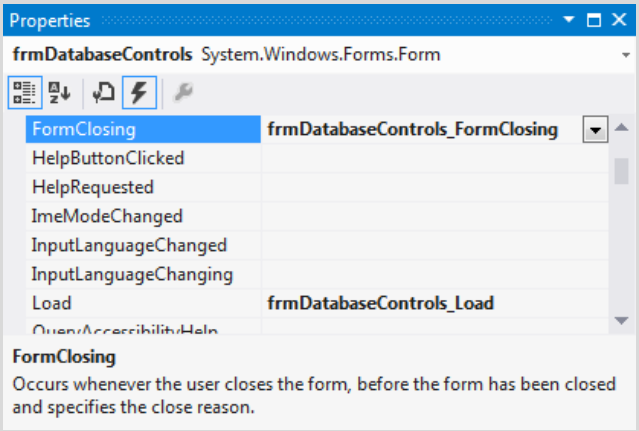
Some of the most commonly used methods of a form are Show() and Hide(). We will use those in the next example.

Example 9-1: Using Form Methods

1. Open form MainMenu.cs and edit the click event method for the button btnDatabaseControls.

```
private void btnDatabaseControls_Click(object sender, EventArgs e)
{
    frmDatabaseControls frm = new frmDatabaseControls();
    frm.Show();
    this.Hide();
}
```

2. Besides opening the form we have added one line of code. The keyword this refers to the current object, which is the form MainMenu.
3. Now we want to make sure to show the form MainMenu again when we close the form frmDatabaseControls. We are going to create an event method for the form's closing event.
4. Open the form frmDatabaseControls and select the form itself. In the property sheet, click on the lightning bolt and navigate to category Behavior and double-click inside the FormClosing property to create this event method as shown below.



5. The form MainMenu is still open, we have just hidden it. We need to find a way to point to this instance to show this form again.
6. At the application level, we have a collection object OpenForms that keeps track of all open forms. Since the main menu is always the first form opened, the index to refer to this form is 0.
7. Write the following line of code in the FormClosing event.

```
private void frmDatabaseControls_FormClosing(object sender, FormClosingEventArgs e)
{
    Application.OpenForms[0].Show();
}
```

8. Save and run the project. Test this new functionality.

22.3 Creating Forms Dynamically using C#

We have seen earlier the architecture of a Windows Form when creating a form in a declarative manner. We can also create a Windows Form in a non-declarative way, that means create a class file and write code that makes up our form. The following example will demonstrate that concept.

Example 9-2: Creating a Form Programmatically

1. Right-click on the PL folder and select Add Item. Select class as a template and name it MyForm.cs. Click on Add.
2. Add the namespace System.Windows.Forms.
3. Type the code shown on the right.
4. Notice we are creating a Textbox, Label, and a Button control.
5. We are also adding an event to this form (functionality from previous example).
6. On the MainMenu form add another button and name it btnMyForm. Set its Text property to "My Form".
7. Double-click on this button to create the default click event.
8. Write the following code.

```
class MyForm :Form
{
    private Button btnClick;
    private TextBox txtName;
    private Label lblName;
    public MyForm()
    {
        this.Text = "My Own Form";
        this.Width = 250;
        this.Height = 250;
        //Create and Configure Button
        btnClick = new Button();
        btnClick.Left = 150;
        btnClick.Top = 150;
        btnClick.Width = 50;
        btnClick.Height = 25;
        btnClick.Text = "Click";
        //Create and Configure TextBox
        txtName = new TextBox();
        txtName.Left = 100;
        txtName.Top = 50;
        txtName.Width = 50;
        txtName.Height = 25;
        //Create and Configure Label
        lblName = new Label();
        lblName.Left = 25;
        lblName.Top = 50;
        lblName.Width = 50;
        lblName.Height = 25;
        lblName.Text = "Name:";
        //Add controls to controls collection
        this.Controls.Add(btnClick);
        this.Controls.Add(txtName);
        this.Controls.Add(lblName);
        //Add event delegate
        this.FormClosing += new FormClosingEventHandler(this.MyFormClosing);
    }
    private void MyFormClosing(object sender, FormClosingEventArgs e)
    {
        Application.OpenForms[0].Show();
    }
}
```

```
private void btnMyForm_Click(object sender, EventArgs e)
{
    MyForm frm = new MyForm();
    frm.Show();
    this.Hide();
}
```

9. Save and run the project. Test the programmatically created form.

23. Basics of Form Controls

Controls are the elements on a form. Each of the controls is based on classes, no surprise here anymore. We will cover the most important controls in this chapter and explain the most important properties and methods.

23.1 Overview of Controls

Some controls are unbound controls whereas others are bound controls. Bound controls can be linked to or bound to data, from a database for example.

 **Note:** Using C# code, you can even "bind" unbound controls to data, such as a label control. Bound controls are those controls that you can bind to data in a declarative way through the property sheet.

To create controls on a form, you have to first display the Toolbox in the Visual C# IDE. Click on pull-down menu, then Toolbox. The toolbox is divided into multiple sections. The ones we need for Windows Forms are the All Windows Forms and Common Controls. The Common Controls is just a subsection of the All Windows Forms section.

There are three different ways to add a control:

- Double-click on the control in the toolbox. This will place the control in top left corner.
- Select a control in the toolbox, hold the mouse button and drag it onto the form.
- Select a control in the toolbox, then on the design surface drag a rectangle to size the control.

Once a control is placed on the form, you can use the property sheet to set its properties. Double-clicking on the control will create the default event handler for that control.

The toolbox is divided into categories of controls to easier find a control as shown below:

- All Windows Forms
- Common Controls
- Containers
- Menu & Toolbars
- Data
- Components
- Printing
- Dialogs
- WPF Interoperability
- Visual Basic Power Packs

When Microsoft Visual Studio is set up, it installs in the Toolbox the most regularly used controls. If you are working in an environment that creates only a particular group of applications and there are controls you hardly use, if you want, you can remove them from the list. To remove a control, right-click it and click Delete.

Besides the objects in the Common Controls section, other controls are left out but are still available. Some of the left out controls were created with the .NET Framework but are not installed by default because they are judged hardly used. To add one or more of these left out controls, right-click anywhere in the Toolbox and click Choose Items... In the Choose Toolbox Items dialog box, click the .NET Framework Components tab, then click the check box of the desired control before clicking OK as shown in Figure 78.

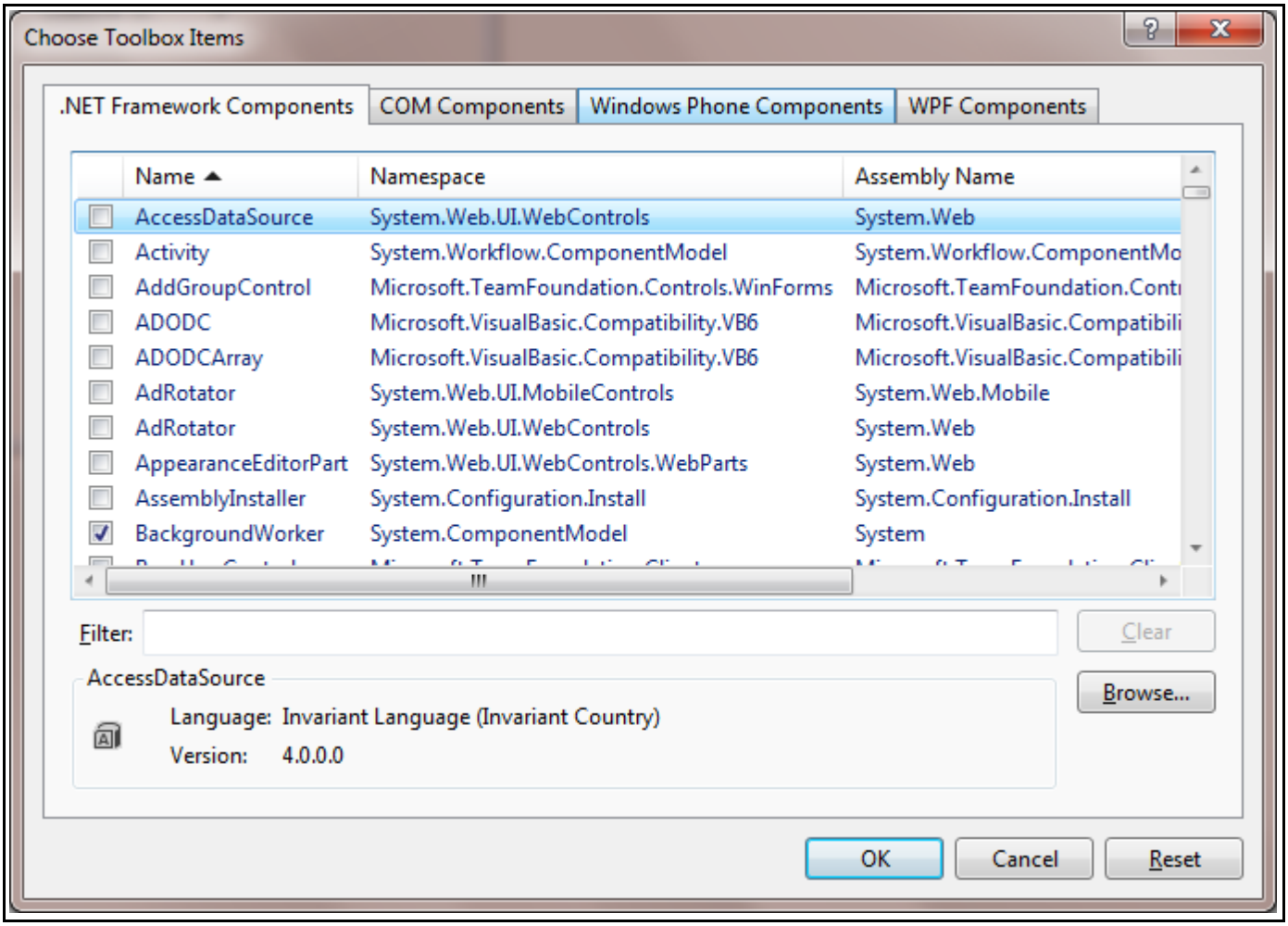


Figure 78: Add Toolbox Items

If the available list of categories is not enough, you can add a new section of your choice. By default, Visual Studio hides some categories because they are judged hardly used. To display these additional sections, you can right-click anywhere in the Toolbox and click Show All as shown in Figure 79.

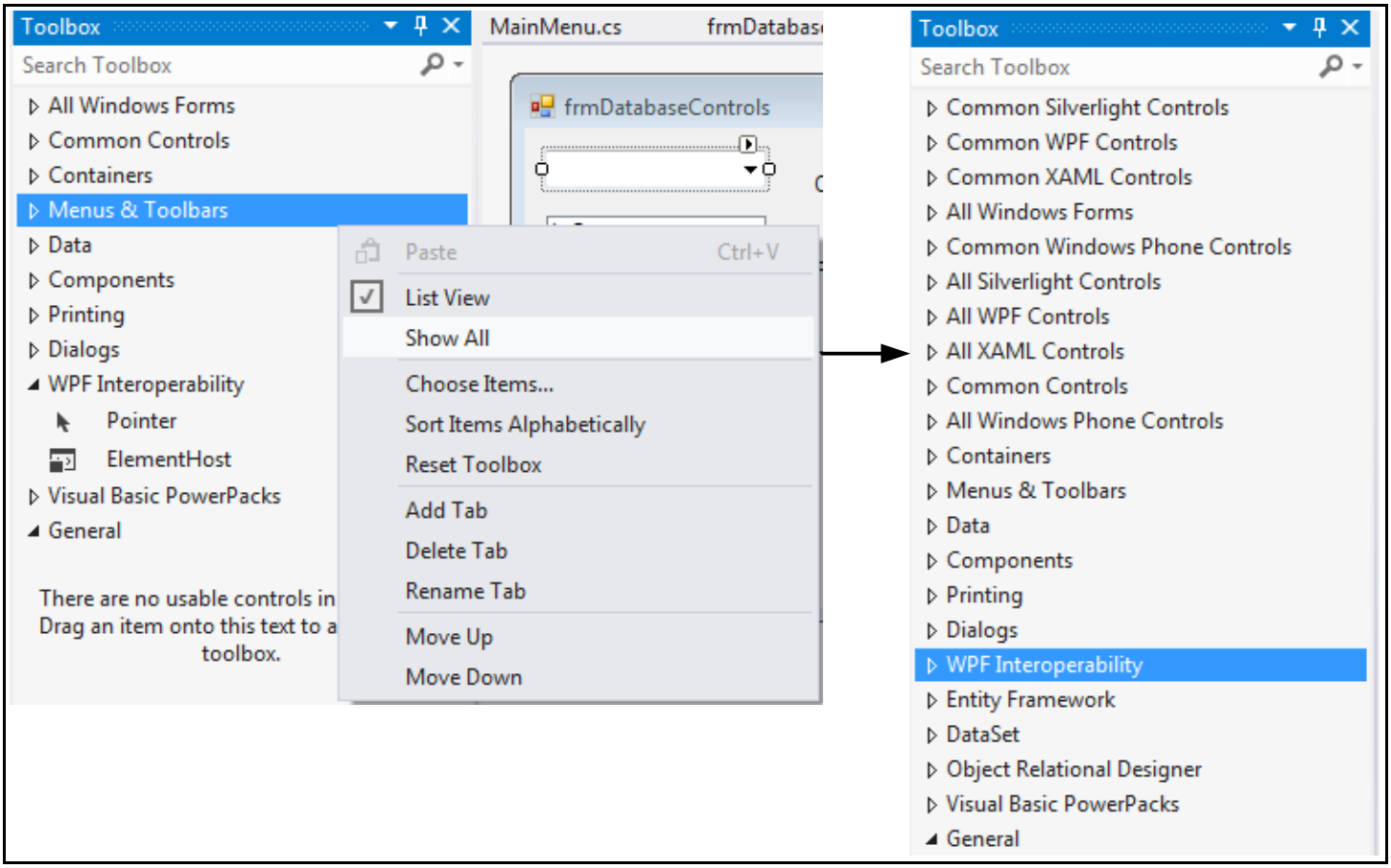


Figure 79: Show All Items in Toolbox

In the System.Windows.Forms namespace, all controls derive from the Control class. The raw Control class is not normally instantiated but rather forms the basis for further refining the user interface hierarchy.

The Control class implements the basic functionality of controls and, where appropriate, provides for members to be overridden. This approach promotes not only reusability but also standardization. This can be seen in two Control class properties, Name and Text. The Name property, the equivalent to a control ID in Win32, applies to all controls regardless of type and therefore is not declared "virtual," whereas a Text property implementation differs depending on the type and can be overridden. For example, Form.Text refers to the caption in the title bar, while TextBox.Text returns the user's keyboard input.

23.2 Common Properties and Methods

As mentioned in the previous chapter, all controls derive from the base control class. This class contains basic properties, methods, and events that every control inherits. Some of these are overridden to slightly change the functionality.

Table 39 shows the properties of the control base class from which all controls are derived.

Properties	Description
Anchor	Specifies how the control relocates and resizes whenever the form is resized.
AutoSize	If set to true, the control will automatically size itself to fit the contents.
BackColor	The background color of the control.
CausesValidation	Specifies whether the control will raise validation events.
Controls	A collection of child controls within this control.
Enabled	Tells whether the user can interact with the control. Set to false to disable the control.
ForeColor	The foreground color of the control (Also the font color of the text inside the control).
Height	The height of the control in pixels.
Location	The location of the control relative to its container.
MaximumSize	Specifies the maximum size of the control.
MinimumSize	Specifies the minimum size of the control.
Name	The name of the control. This is used to reference the control in code.
Parent	The parent of the control.
Right	The distance between the right edge of the control and the left edge of its container.
Size	The size of the control. Composed of Width and Height subproperties.
TabIndex	Specifies the number of the control in the tab order of its container.
TabStop	Specifies whether the control can be accessed by the tab key.
Tag	Used to assign special or useful values about the control.
Text	The text shown inside the control.
TextAlign	Specifies the alignment of text of the control.
Top	The distance between the top edge of the control and the top edge of its container.
Visible	Sets the visibility of the control.
Width	The width of the control in pixels.

Table 39: Selected Properties of Base Control Class

Table 40 shows the methods of the base control class.

Method	Description
Focus	Sets focus to the control.
Hide, Show	Hide and show the control by setting visible property.
PointToScreen	Computes the location of the client point p in screen coordinates.
PreProcessMessage	Called by the application's message loop to preprocess input messages before they are dispatched.
RectangleToClient, RectangleToScreen	Location of the screen rectangle in client coordinates or screen coordinates.
Refresh	Forces the control to invalidate and immediately repaint itself and any children.
SetClientSizeCore	Sets the height and width of the client area of the control.
Update	Forces the control to paint any currently invalid areas.

Table 40: Control Base Class Methods

Table 41 shows the events of the base control class.

Event	Description
Click, DoubleClick	Occur when the control is clicked or double-clicked.
Enter	Occurs when the control is entered.
GotFocus, LostFocus	Occur when the control receives or loses focus.
KeyDown, KeyPress, KeyUp	Occur when a key is down, pressed, or up, accordingly.
MouseDown, MouseEnter, MouseHover, MouseLeave, MosueMove, MouseUp, MouseWheel	Different mouse events that occur when the mouse button is down, the mouse enters the control, mouse is moved, mouse button is up, etc.
Move	Occurs when a control is moved.
Resize	Occurs when a control is resized.

Table 41: Control Base Class Events

23.3 Control Tab Order

As we have seen above, each control has an ordinal position within the tab order. This position is initially set in the chronological order the control was added to the form. This is a quite natural process when you develop forms that the order you added the controls onto the form should not be the final tab order. Once you are done developing the form, you should review the tab order for the form.

Each control has a tab index property that you can set individually. This is a quite tedious process if you have many controls and want to finally adjust the tab order. Luckily, there is an interface available for viewing the tab order for an entire form and adjust it for all the controls.

Select the form or any control, then navigate to the pull-down menu VIEW → Tab Order. Then each control on the form gets a number assigned which is the tab index as shown in Figure 80.

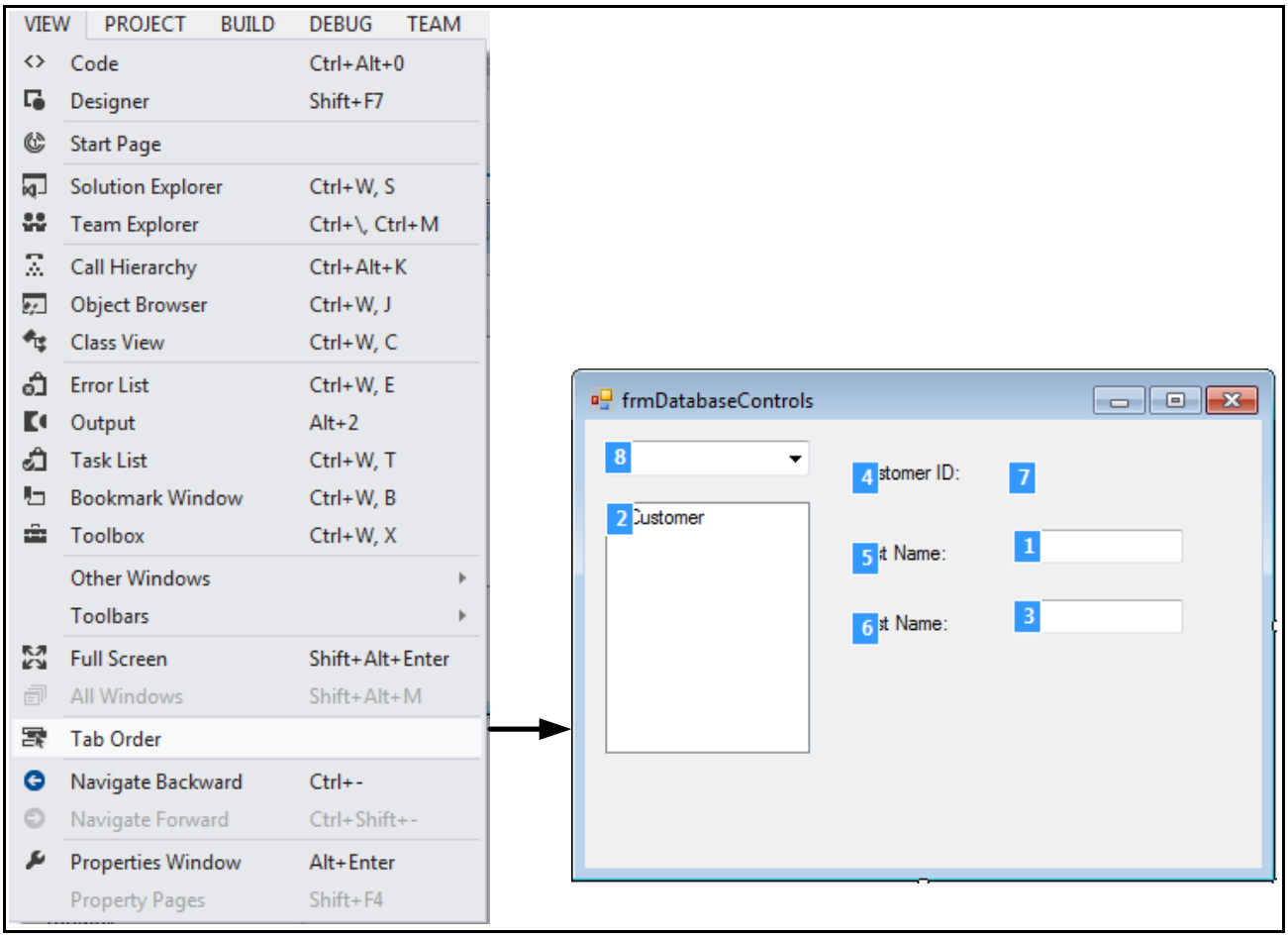


Figure 80: Controls Tab Order

To change the tab order, you simply click inside the blue boxes to increment the tab index by one (1). Clicking again will increment again. Repeating this process will increment the tab index until the total number of controls minus one (zero-based) is reached. After that it will start at zero(0) and then increment again.

24. Introduction to Specific Controls

In this chapter we are going to discuss the most common controls in more detail.

24.1 Label and Textbox

A label control is mostly used in conjunction with other data controls displaying metadata about the data controls. For example, a text box used to collect a person's name should have a label in front reading "Name:" (without quotes). Other uses for labels are to display dynamic data that is evaluated and processed in event handlers and/or methods and subsequently displayed on a form. Furthermore, labels are used to display headers for forms or form instructions and help information.

The most important property is the Text property, which is displayed in the label on the form.

The TextBox control has a few more properties in addition to the general ones covered in the previous chapter.

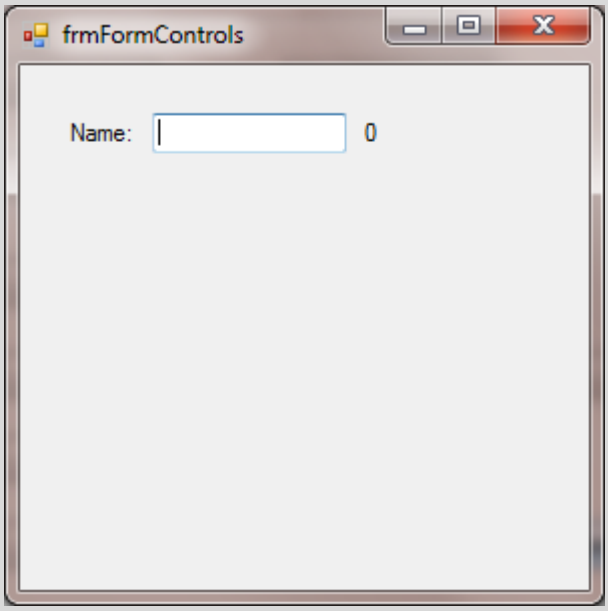
A TextBox control can be set up to function as a multi-line control. In that mode, you can set scrollbars (horizontal and vertical). The scrollbars property is only in effect for MultiLine controls. When MultiLine is set to true, then vertical scrollbars are displayed. When WordWrap is set to False, then horizontal scrollbars are displayed when the scrollbars property is set accordingly.

The default event for the textbox control is the TextChanged event. This event is fired whenever a change of the Text property occurs. We will explore this event in the next example.

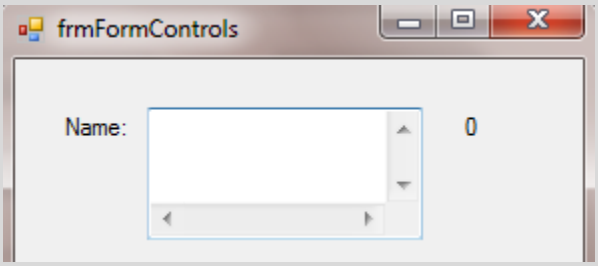
The label control does have events, however, this control cannot be edited, so certain events cannot be caused by the user directly. Only if you change the text property of a label programmatically the TextChanged of the label control is raised.

Example 9-3: Using Label and TextBox controls

1. Add a new windows form to the project in the PL folder and name it frmFormControls. Change the namespace to CourseExamples (remove the PL), you may also need to do this in the designer.cs file.
2. Add a button on the MainMenu form and name it btnFormControls. Set its Text property to "Form Controls". Add the usual two lines of code to open the form frmFormControls.
3. Add two labels and a textbox control on the form frmFormControls as shown.



4. Name the left label lblName, the textbox txtName, and the right label control lblCount.
5. Now select the textbox control and set its MultiLine property to True. You notice that now all the handle bars appear to size the control. Size the height of the control to approximately two lines.
6. Now set the scrollbar property to Both. You notice at this point that there is only a vertical scrollbar.
7. Now set the WordWrap property to False, and you should see now both scrollbars as shown below:



8. Now we are going to use an event of the text box control, the TextChanged event. Select the textbox control, and in the property sheet click on the lightning bolt button. Double-click inside the TextChanged event and create the following event handler:

```
private void txtName_TextChanged(object sender, EventArgs e)
{
    lblCount.Text = txtName.Text.Length.ToString();
}
```

9. Save the project and run it. You notice that while you enter and modify data in the text box control, the label on the right side displays the number of character. This is due to the event handler we code above. This event handler gets the length of the text property every time there is a change of the text property and displays it in the label control.

Another variation of the textbox control is the maskedTextBox control. This is control behaves likes a textbox, but has an input mask feature. An input mask is a tool to validate data input into a control.

 **Note:** The maskedTextBox control is very similar to the Microsoft Access Input Mask property of a textbox control.

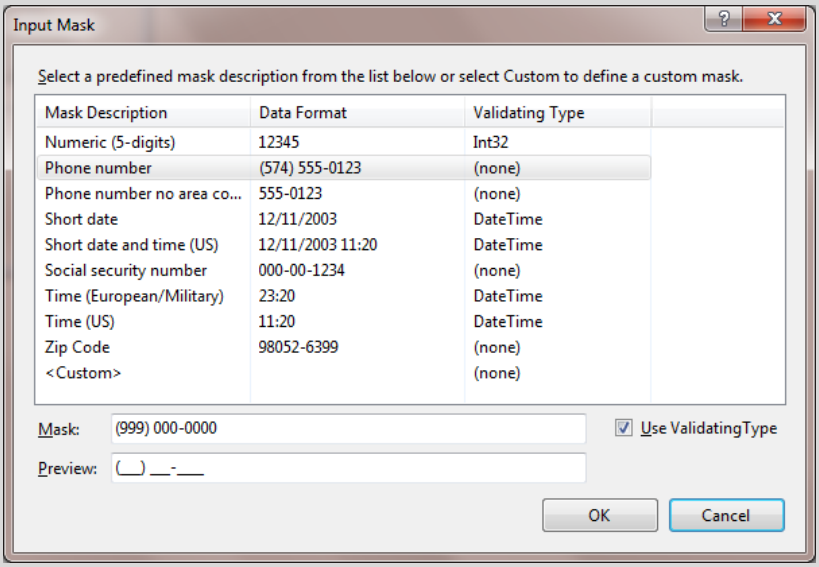
The input mask feature is most useful for fixed-width data having common formats, such as phone or social security numbers. You define the mask and what kinds of characters are allowed in each character position (alpha, numeric characters). Furthermore, you define the literal characters that make up the format surrounding the actual data, such as spaces, parentheses, hyphen, etc.

Property	Description
PromptChar	The character used as a placeholder (default is the underscore).
AllowPromptAsInput	Whether the placeholder character is allowed as input. Ideally, you want to set up your input mask in such a way that the placeholder character is not allowed as input.
CutCopyMaskFormat	Include Literals, ExcludePromptAndLiterals, IncludePrompt, IncludePromptAndLiterals: Specifies when selecting and copying data from this control, whether literals and prompt characters are pasted into the target environment.
HidePromptOnLeave	True/False: Whether placeholder characters are displayed or not when control does not have focus (leaving the control).
Mask	The actual input mask, click on the build button to the right to access the Input Mask Builder (very similar to Microsoft Access).
PasswordChar	The character that is displayed when typing characters, this effectively hides the input.
RejectInputOnFirstFailure	When entering several characters in a single operation (usually copy and paste) and RejectInputOnFirstFailure is set to True, then processing stops at the first invalid character. If set to false, the invalid character is reject but parsing continues.
TextMaskFormat	Include Literals, ExcludePromptAndLiterals, IncludePrompt, IncludePromptAndLiterals: Whether Text property includes or excludes placeholder characters and literals.
UseSystemPasswordChar	Gets or sets a value indicating whether the operating system-supplied password character should be used.

Table 42: MaskedTextBox Control Properties

Example 9-4: Using MaskedTextBox control

1. Add a label and masked textbox control below the previous label and textbox control.
2. Name the masked textbox mtbPhone. Click inside the Mask property, then click on the ellipsis on the right side to launch Input Mask Dialog box.
3. Select the Phone number mask and then click on OK.
4. Double-click on the masked textbox control to create the default MaskInputRejected event.
5. Back to the designer, create another event by selecting the masked textbox control, then in the property sheet click on the lightning bolt, then double-click inside the Validated event to create this event. Write the following code for both event methods.




```
private void mtbPhone_MaskInputRejected(object sender, MaskInputRejectedEventArgs e)
{
    MessageBox.Show("Only numeric characters are allowed.");
}

private void mtbPhone_Validated(object sender, EventArgs e)
{
    if (!mtbPhone.MaskCompleted)
    {
        MessageBox.Show("Enter a valid phone number!");
    }
}
```

6. The MaskInputRejected event is fired everything a single character violates the defined mask, in this case when any alphabetic character is typed.
7. The Validated event is fired when focus is leaving the control. The MaskCompleted property shows whether all the required characters have been entered.
8. Save and run the project. Enter any alphabetic character into the masked textbox control. Also enter an incomplete phone number and tab or click outside the field.

Another textbox control is the richtTextBox control. This control can be extended to perform basic editing tasks of the text within the control. However, you must write code to create this functionality.

24.2 Combobox and Listbox

The combobox and listbox controls are controls that can display data from either a data source or a preset list of data in code, such as an array. You then select a value, and the selected value can be used on your form or stored back into a database. Combobox and Listbox controls handle data either as a simple list (one column) or in a two-column format.

If you use a simple list, then each value is assigned an index in the items collection. The first value is assigned to 0, the second to one, and so on. There is no value member, only a display member.

If you use a two-column list, then the first column is the so-called Value column and the second column is the Display column. The value column is not shown but is the value that is actually stored in the Text property when a value is selected. Table 43 shows the most common properties of the list and combo box controls.

Property	**Description**
DisplayMember	The display value in the control
ValueMember	The stored value in the control
SelectedIndex	The index of the selected item. Items are numbered from 0. If no item is selected, this property is set to -1.
SelectedItem	The object of the selected item.
SelectedValue	The currently selected value
Text	The value that is stored in the control. For a simple list, it is the selected item. For a two-column list, it is the value member (the hidden column).
Sorted	If set to true, the items in the list are sorted alphabetically in ascending order.
Items	Provides access to the collection of items.
DropDownStyle	This property applies only to Combobox controls: Values: DropDownList, DropDown If set to DropDown, the user can either enter text in the text box at the top of the combobox control or select a value from the list. If set to DropDownList, the user can only select a value from the list.
SelectionMode	This property applies only to ListBox controls: Values: One, MultiSimple, MultiExtended If set to one, the user can only select one item from the list. If set to MultiSimple, the user simply can click multiple items to select those items.

Table 43: Selected Properties for Combo/List Box

The most common event for combobox and listbox controls is the SelectedIndexChanged event, which is the default event. This event is triggered when the user selects an item from the list.

When working with combo or list box controls, the items collection is an important object. You can programmatically add, change, or delete values from a list or combo box control using the items collection. Table 44 shows the most common members of the items collection.

Indexer	**Description**
[index]	Gets or sets the item at the specified index in the list.
Property	**Description**
Count	Gets the number of items in the list.
Method	**Description**
Add(*object*)	Adds the specified item to the list.
Insert(*index,object*)	Inserts an item into the list at the specified index.
Remove(*object*)	Removes the specified item from the list.
RemoveAt(*index*)	Removes an item at the specified index from the list.
Clear()	Removes all items from the list.

Table 44: Items Collection Members

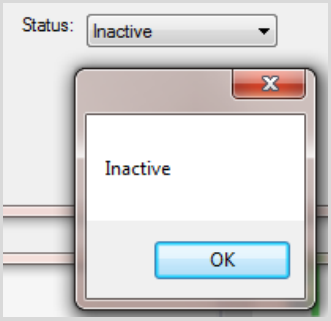
Example 9-5: Using List and Combo Box control (Simple List)

1. Add a ComboBox control and a label control below the phone number control.
2. Name the combobox control cmbStatus. Set the Text property of the label to "Status:".
3. Double-click on an empty area of the form to create the load event for the form. Add the following code:

```
private void frmFormControls_Load(object sender, EventArgs e)
{
    cmbStatus.DropDownStyle = ComboBoxStyle.DropDown;
    cmbStatus.Items.Clear();
    cmbStatus.Items.Add("Active");
    cmbStatus.Items.Add("Inactive");
}
```

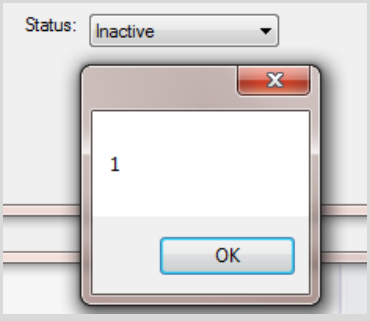
4. Save the project and run the form. Test the combobox. You should be able to enter a value in the textbox portion of the combobox or select a value from the list.
5. Back in the load event of the form, change the ComboBoxStyle property to DropDownList.
6. Now run the form and verify that you cannot enter a value in the textbox of the combobox.
7. Now create an event for the SelectedIndexChanged event for the combobox. Type the following line:

```
private void cmbStatus_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show(cmbStatus.SelectedItem.ToString());
}
```



8. Run the form, and test the event handler by selecting different values.
9. Now change the line of code to:

```
private void cmbStatus_SelectedIndexChanged(object sender, EventArgs e)
{
    //MessageBox.Show(cmbStatus.SelectedItem.ToString());
    MessageBox.Show(cmbStatus.SelectedIndex.ToString());
}
```



10. Run the form and verify the message box, you should see now a 0 for the first item and a 1 for the second item.

The next example will use a two-column list. This is most commonly used in database application where the data is drawn from a table with two columns, one is the key value and the other one is the descriptive or translated value. For example, for a table containing the US states, the first column contains CA and the second one California.

When binding data from a database, you would use a datatable object of the dataset. You have to specify which column is ValueMember and which column is the DisplayMember.

Example 9-6: Using List and Combo Box control (Two-column list)

1. Comment out the current code in the form's load event.
2. Create a simple, public class named CustomerStatus as shown below.

```
public class CustomerStatus
{
    public CustomerStatus(int intID, string strStatus)
    {
        ID = intID;
        Status = strStatus;
    }
    public int ID { get; set; }
    public string Status { get; set; }
}
```

3. In the form's load event, write the following code.

```
private void frmFormControls_Load(object sender, EventArgs e)
{
    cmbStatus.DropDownStyle = ComboBoxStyle.DropDownList;
    //cmbStatus.Items.Clear();
    //cmbStatus.Items.Add("Active");
    //cmbStatus.Items.Add("Inactive");
    CustomerStatus[] Status = new CustomerStatus[]
    {new CustomerStatus(1, "Active"), new CustomerStatus(2, "Inactive" )};
    cmbStatus.DataSource = Status;
    cmbStatus.ValueMember = "ID";
    cmbStatus.DisplayMember = "Status";
}
```

4. Note that we are creating an array of type CustomerStatus and adding two pairs of data.
5. In the SelectedIndex event, we need to change now the way how to retrieve the selected items. Since we are using a two column data set, we cannot use the items collection anymore. Now we have to retrieve the SelectedValue property (ValueMember), remember the DisplayMember is not stored in the control. But the DisplayMember is automatically assigned into the Text property of the combo box.

```
private void cmbStatus_SelectedIndexChanged(object sender, EventArgs e)
{
    if (this.ContainsFocus)
    {
        MessageBox.Show(cmbStatus.SelectedValue.ToString() + ": " + cmbStatus.Text);
        MessageBox.Show(cmbStatus.SelectedIndex.ToString());
    }
}
```

6. Note that we are only displaying the data when the combo box actually has the focus and not when the form is loading (which also fires the SelectedIndexChanged event).
7. Save and run the project.

24.3 DateTimePicker and MonthCalendar

The DateTimePicker control is a powerful, yet easy to use control to select date and time values. Furthermore, using code you can customize this control to a great extent. This control is displayed like a drop-down list. However, when you click on the drop-down arrow, an interactive calendar is displayed as shown in Figure 81.

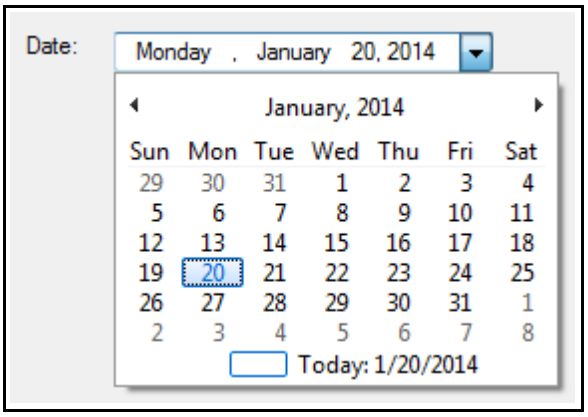


Figure 81: DateTimePicker Control

You can customize this control by using the property sheet. Further customization is possible using C# code. One common functionality is to limit the range of dates that the user can select from this control based on the current date. This control has a MinDate and MaxDate property that allows you to specify dynamically a date range.

Warning: You cannot select a null date in this control. You have to write quite some amount of code to implement that functionality.

The default event is the ValueChanged event. This event is fired every time a value is changed in the control.

The most common properties of the DateTimePicker control are shown in Table 46.

Property	Description
Checked	The ShowCheckBox property should be set to false to work with this property. When this value is true, the selected date can be changed/updated. When this is false, the selected date cannot be changed.
CustomFormat	Accepts a custom format string the will format the date displayed.
Format	Determines the format of the date displayed by the control.
MaxDate	Gets or sets the maximum date and time that can be selected in the control.
MinDate	Gets or sets the minimum date and time that can be selected in the control.
ShowCheckBox	If true, shows a text box to the left part of the DateTimePicker control. When checked, the selected date can be changed. When unchecked, the selected date cannot be changed.
ShowUpDown	When set to true, shows an up-down button in place of the drop down button. You cannot access the calendar when this is shown. Instead, you need to select a date component and use the up-down arrows to adjust the date.
Value	The currently selected date.

Table 45: Selected Properties of DateTimePicker Control

Example 9-7: Using DateTimePicker control

1. Add a label and DateTimePicker control below the combobox control.
2. Name the DateTimePicker control dtpDate and set the Text property of the label to "Date:".
3. Add the following code to the form's load event to restrict the dates that a user can choose from to today's date and 60 days from today's date.

```
//DateTimePicker min and max date
dtpDate.MinDate = DateTime.Today;
dtpDate.MaxDate = DateTime.Today.AddDays(60);
```

4. Double-click on the control to create the default ValueChanged event. Write the following code.

```
private void dtpDate_ValueChanged(object sender, EventArgs e)
{
    MessageBox.Show("Selected Date: " + dtpDate.Value.ToShortDateString());
}
```

5. Save and run the project.

The MonthCalendar control resembles a calendar and shows a month and all its dates. The MonthCalendar control allows you to choose a month and a date. To choose a month, click the left and right arrows to move the month to the next or previous month.

The MonthCalendar control also allows you to select a date range using a Start date and End date. You can limit this control to one date by setting the MaxSelectionCount to 1.

Clicking the month header, will show all the months of the current year. Clicking the header even further will show all the years of the current decade, and clicking it once more will show all the year of the current century as shown in Figure 82.

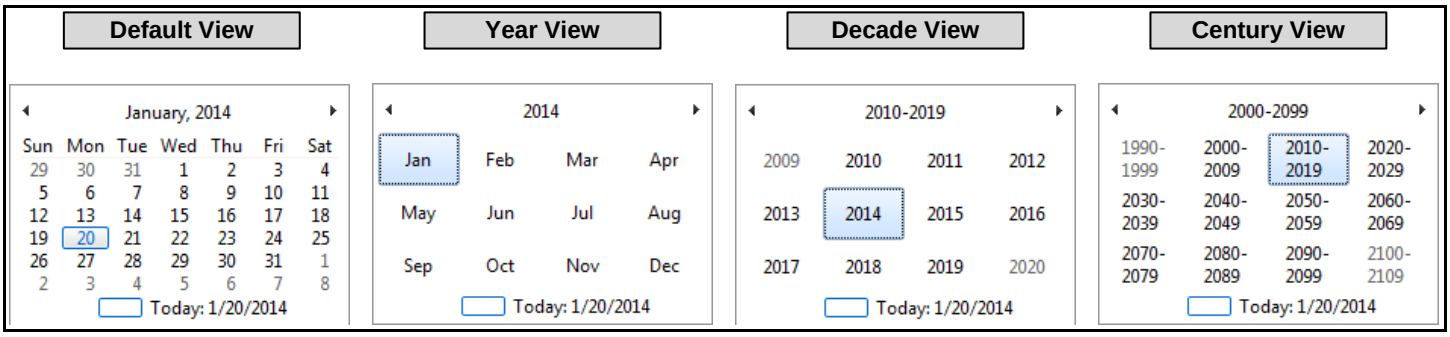


Figure 82: MonthCalendar Control

You can customize the MonthCalendar control using the different properties as shown in Table 46.

Property	Description
FirstDayOfWeek	Specifies what day the calendar will consider as the first day of the week.
MaxDate	Specifies the maximum allowable date.
MaxSelectionCount	Specifies the maximum dates that the user can simultaneously select.
MinDate	Specifies the minimum allowable date.
SelectionEnd	If the user selects a range of date, this property indicates the last date in the range of dates.
SelectionRange	If the user selects a range of date, this property contains all the dates whithin the range of dates.
SelectionStart	If the user selects a range of date, this property indicates the first date in the range of dates.
ShowToday	Specifies whether to show the date today at the bottom of the control.
TodayDate	The date used by the MonthCalendar as today's date.

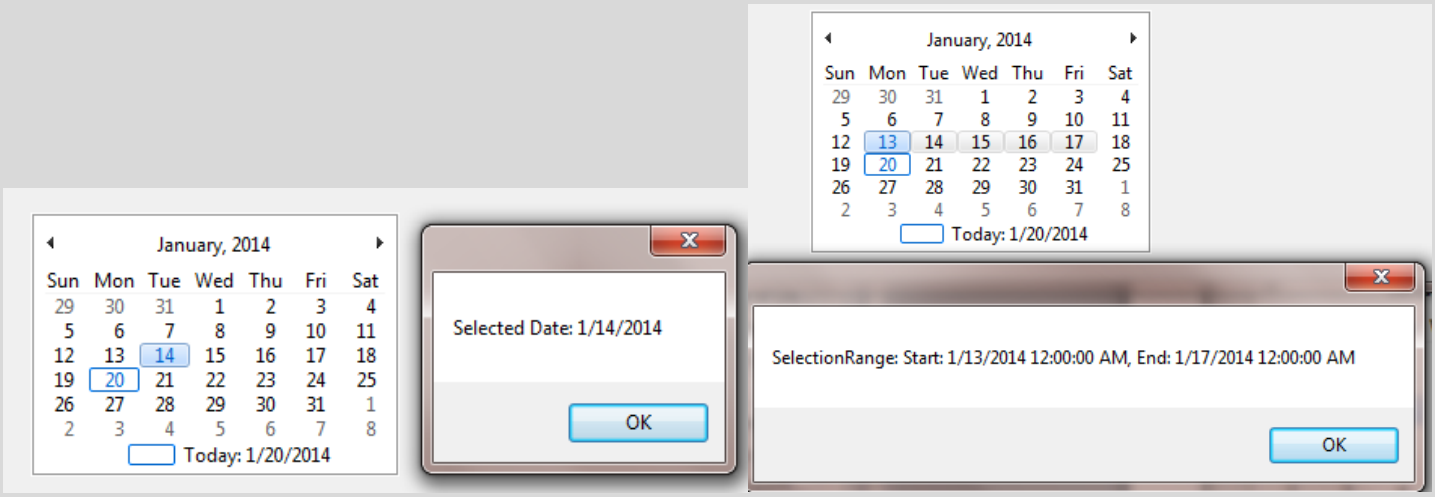
Table 46: Selected Properties of MonthCalendar

Example 9-8: Using MonthCalendar control

1. Add a MonthCalendar control below the DateTimePicker control and named in mcDate.
2. In the property sheet, click on the lightning bolt and select the event DateSelected. Double-click inside this property to create the event.
3. Write the following code.

```
private void mcDate_DateSelected(object sender, DateRangeEventArgs e)
{
    if (mcDate.SelectionRange.Start == mcDate.SelectionRange.End)
    {
        MessageBox.Show("Selected Date: " + mcDate.SelectionRange.Start.ToShortDateString());
    }
    else
    {
        MessageBox.Show(mcDate.SelectionRange.ToString());
    }
}
```

4. The event method checks whether a single date was selected by comparing the start and end date of the selected date range. If they are the same, then only one date was selected. Otherwise, display the date range.
5. Save and run the project. Test selecting a single date and a date range by clicking the mouse and drag over multiple dates.



24.4 RadioButton, CheckBox, and GroupBox

RadioButton and CheckBox controls are Boolean controls, they are either checked (true) or not checked (false). The main difference is that RadioButtons are mutually exclusive and check boxes operate independently.

When you place multiple RadioButton controls on a form they are all together mutually exclusive, meaning only one of the RadioButton controls can be checked at all times. If you need different groups of RadioButton controls on a form, then use the GroupBox control to limit the mutual exclusive behavior to the RadioButtons contained in the group.

The GroupBox control can be used for other controls as well, even if you do not wish to implement the mutual exclusive behavior. It adds a nice design aspect to the form and allows you to move all controls in the group by simply moving the GroupBox control.

Note: The GroupBox control can be found under All Windows Forms section in the toolbox.

Example 9-9: Using RadioButton and GroupBox control

1. Add two RadioButton controls on the top right side of the form frmFormControls.
2. Set their Text property to "Radio 1" and "Radio 2", respectively.
3. Save the project and run the form. Click both Radio button controls and you notice that when you select one the other one gets deselected.
4. In Design view, add two more RadioButton controls below the previous ones. Set the text property to Radio 3 and Radio 4.
5. Save the project and run the form. You now notice that all four RadioButton controls behave in a mutual exclusive manner.
6. Back in design view, add a GroupBox control onto the form. Drag the first two RadioButton controls into this group. Set the Text property of the GroupBox control to Group 1.
7. Save the project and run the form. Now you notice that the Group 1 radio buttons act mutually exclusive, but independent from the other two radio buttons. Now you have two groups of radio buttons.

The default event of CheckBox and RadioButton controls is the CheckedChanged event. CheckBox and RadioButton controls are not only used to bind it to data, but for user interface functionality. For example, there may be a group of radio buttons where you select the payment method. Based on the selection, you would hide certain controls and show other controls on your form that apply to the payment method selected.

Warning: The CheckedChanged event is also triggered when a RadioButton control is deselected by clicking on another RadioButton control.

The radio button has a CheckChanged and Click event. The CheckChanged event is triggered when the state of the button is changed. For example if the radio button turns from off to on, then the CheckChanged event will execute. The Click event is sent when the radio button is clicked. By default, clicking a radio button will change its state, which will therefore trigger the CheckChanged event.

The difference of the Click event is that you can actually change this default behavior of the radio button. There is a property for the radio button control called AutoCheck which if set to false, clicking the radio button will not change its state, but will sent a Click event and through code, you can manually set the radio button's Checked property to true.

The default event for the radio button is the CheckChanged event, therefore, double clicking a radio button while in the VS/VCE forms designer will generate an event handler for the said event. The following example demonstrates the use of the radio button.

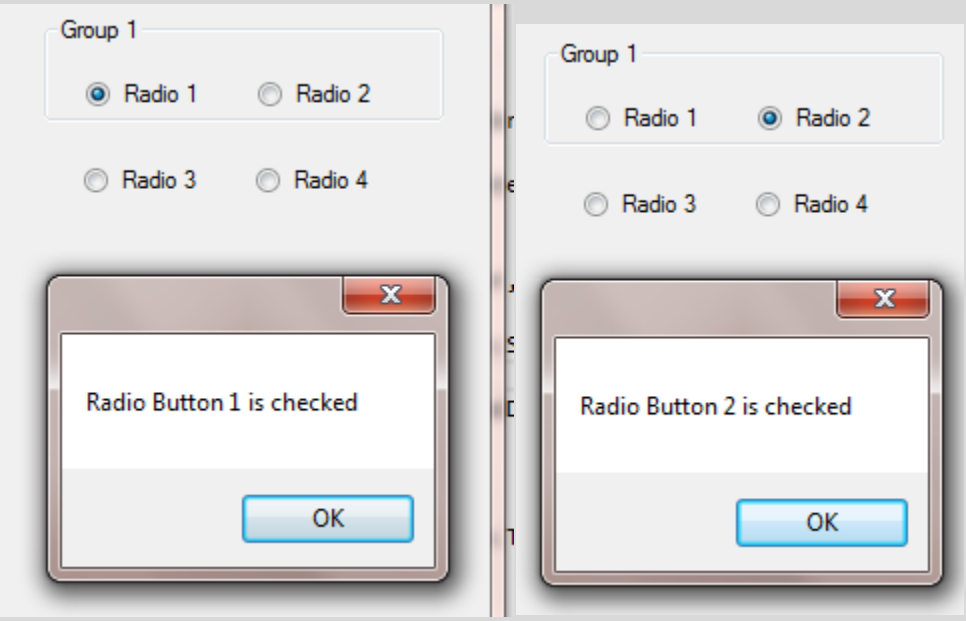
The most important property is the Checked property. This is a Boolean, and you can test in code whether a CheckBox or RadioButton control is selected (that means checked).

Example 9-10: Using CheckedChanged Event

1. Write the following generic event method.

```
private void RadioButtonChecked(object o, EventArgs e)
{
    if (radioButton1.Checked)
    {
        MessageBox.Show("Radio Button 1 is checked");
    }
    else if (radioButton2.Checked)
    {
        MessageBox.Show("Radio Button 2 is checked");
    }
}
```

2. In form designer, select the Radio Button 1, navigate to the property sheet, click on the lightning bolt, and in the CheckedChanged property, select the event method RadioButtonChecked.
3. Repeat step 2 for the Radio Button 2.
4. Save and run the project. Test both radio buttons and checked both of them.



24.5 DataGridView

With the DataGridView control, you can display and edit tabular data from many different kinds of data sources.

The DataGridView control makes it easy to define the basic appearance of cells and the display formatting of cell values. The cell is the fundamental unit of interaction for the DataGridView. All cells derive from the DataGridViewCell base class. Each cell within the DataGridView control can have its own style, such as text format, background color, foreground color, and font. Typically, however, multiple cells will share particular style characteristics. The data type for the cell's Value property by default is of type Object.

This control is complex and allows for many customizations. In the next example we will demonstrate just a few features.

Example 9-11: Using DataGridView control

1. Add a DataGridView control below the radio buttons and name it dgvCustomer.
2. Write the following method name SetCustomerGridView and call it in the form frmFormControls open event.

```
private void SetCustomerGridView()
{
    //Creating Customer Data Table
    DataTable dt = new DataTable();
    dt.Columns.Add("ID");
    dt.Columns.Add("FName");
    dt.Columns.Add("LName");
    dt.Rows.Add(1,"Bob","Fuller");
    dt.Rows.Add(2,"Mary","Lee");
    dt.Rows.Add(3,"Lisa","Evans");
    dt.Rows.Add(4,"Ruth","Muller");
    dt.Rows.Add(5, "Peter", "Pollick");
    //Binding data to grid view control
    dgvCustomer.DataSource = dt;
    //Changing some properties
    dgvCustomer.Columns[0].Width= 30;
    dgvCustomer.Columns[1].Width = 75;
    dgvCustomer.Columns[2].Width = 75;
    dgvCustomer.AlternatingRowsDefaultCellStyle.BackColor = System.Drawing.Color.Olive;
    dgvCustomer.AllowUserToAddRows = false;
}
```

```
//DateTimePicker min and max date
dtpDate.MinDate = DateTime.Today;
dtpDate.MaxDate = DateTime.Today.AddDays(60);
SetCustomerGridView();
```

3. Navigate to the event property SelectionChanged and double-click inside to create the event.
4. Save and run the project.

```
private void dgvCustomer_SelectionChanged(object sender, EventArgs e)
{
    if (this.ContainsFocus)
    {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < dgvCustomer.Columns.Count; i++)
        {
            sb.Append(dgvCustomer.CurrentRow.Cells[i].Value.ToString() + "/ ");
        }
        MessageBox.Show("Current Row: " + sb.ToString());
    }
}
```

24.6 MessageBox Class

Although we have already used the message box extensively, we have not explored the basic functionality.

The System.Windows.Forms.MessageBox is a static class that is used to show message boxes for prompting, confirmation and warning users. To show a message box, simply call the Show method of the MessageBox class. The simplest version of the Show method is the one that accepts a string message as an argument, this is the one we have been using so far.

However, this method is overloaded (21 overloaded methods) and can accept quite a few more parameters. We will focus here on the most important ones as it relates to this course.

MessageBox.Show(*Prompt*, *Title*, Buttons, Icons, DefaultButton)

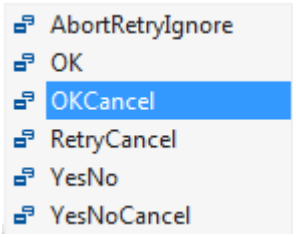
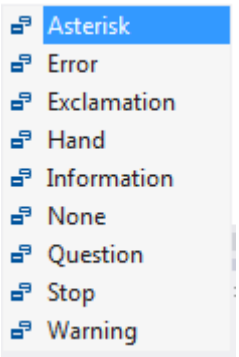
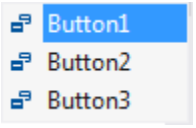
Parameter	**Description**
Prompt	The actual text inside the message dialog box that is presented to the user.
Title	The title of the message dialog box
Buttons	Accepts values from the enumeration MessageBoxButton The combination of buttons to place on a message dialog box. Note that any combination besides the OK allows the user to read which button the user pressed. 
Icons	Accepts values from enumeration MessageBoxIcon Displays an icon to the left of the prompt indicating the category of the message, such as Information, Warning, or Critical (using Stop or Error). 
DefaultButton	Accepts values from the enumeration MessageBoxDefaultButton 

Table 47: MessageBox Parameters

The Show() method returns a value from the System.Windows.Forms.DialogResult enumeration. This is useful to determine what button you pressed in the message box. For example, if you click the "Yes" button in the message box, then the Show() method will return the value DialogResult.Yes.

There are seven different dialog result value based on the allowed buttons in a message dialog box:

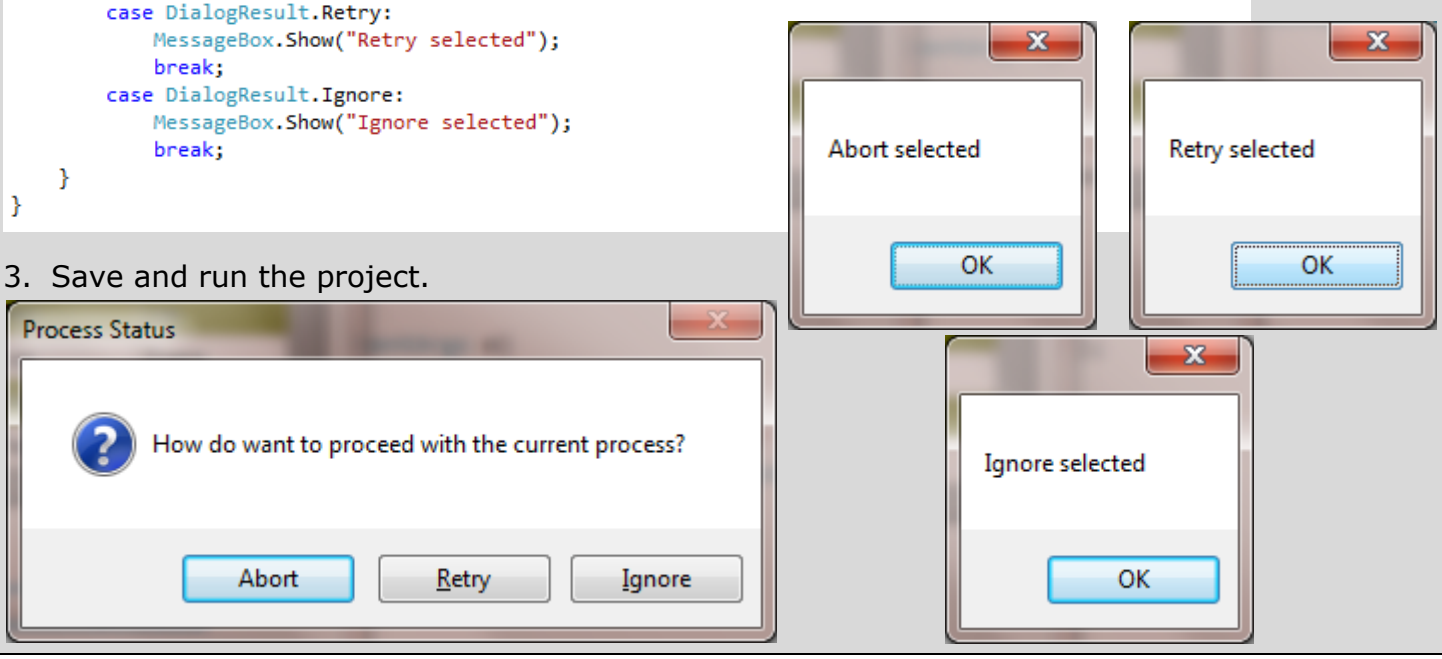
- Yes
- No
- Abort
- Retry
- Cancel
- Ignore
- OK

Example 9-12: Using MessageBox Class

1. Add a button below the DataGridView control and name it btnMessageBox. Set its Text property to "Message Box".
2. Double-click on the button to create the default click event. Write the following method.

```
private void btnMessageBox_Click(object sender, EventArgs e)
{
    DialogResult result;
    result = MessageBox.Show("How do want to proceed with the current process?", "Process Status",
        MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Question, MessageBoxDefaultButton.Button1);
    switch (result)
    {
        case DialogResult.Abort:
            MessageBox.Show("Abort selected");
            break;
        case DialogResult.Retry:
            MessageBox.Show("Retry selected");
            break;
        case DialogResult.Ignore:
            MessageBox.Show("Ignore selected");
            break;
    }
}
```

3. Save and run the project.



CLASS 10

25. Advanced Topics

This chapter presents some selected, advanced topics. Students are also invited to suggest any topics that they want to learn more about it. Contact the instructor well in advance to allow for sufficient time to prepare the topic.

25.1 LINQ

LINQ stands for Language Integrated Query. LINQ is kind of like SQL, that is it allows you to query data. You can use database data with LINQ, but also other data sources such as in-memory data structures, for instance arrays and generic lists. This way, you can use the same language to query a variety of data sources.

Overview of LINQ

As mentioned earlier, LINQ is similar to SQL, yet somewhat different. If you know SQL, then LINQ is fairly easy to understand. It uses similar keywords such as SELECT and FROM, but the order is different. One of the main issues in SQL is that the order of keywords in the SELECT statement is different from the order of how these keywords and their associated operations are processed.

```
SQL: SELECT c.FName, c.LName FROM customer c;
LINQ: FROM c IN customer SELECT new {c.Fname,c.LName};
```

Table 48 displays the most common keywords of LINQ.

Keyword	Description
from	Source of data for the query
where	Condition for data retrieval from data source
orderby	Sort order of data in query
select	Content of the returned elements
join	Combines data from multiple sources

Table 48: LINQ Keywords

LINQ is implemented in a three-stage process as shown below:

1. Get the data source (array, database table, etc.)
2. Define the query using LINQ keywords
3. Execute (or process the query) to return the results

The advantages of LINQ are:

- Familiar syntax for writing queries.
- Compile-time checking for syntax errors and type safety.
- Improved debugger support.
- IntelliSense support.
- Ability to work directly with XML elements instead of creating a container XML document, as required with W3C DOM.
- In-memory XML document modification that is powerful, yet simpler to use than XPath or XQuery.
- Powerful filtering, ordering, and grouping capabilities.
- Consistent model for working with data across various kinds of data sources and formats.

Using LINQ with Arrays

We will first take a look at how LINQ works with an array as a data source. As you know, an array can be multi-dimensional, for simplicity, we will only consider one-dimensional arrays here.

The next example shows exactly the steps how to create a LINQ query using an array as a data source.

Example 10-1: Using LINQ with an Array

1. Add a new windows form to the project in the PL folder and name it frmAdvancedTopics. Change the namespace to CourseExamples (remove the PL), you may also need to do this in the designer.cs file.
2. Add a button on the MainMenu form and name it btnAdvancedTopics. Set its Text property to "Advanced Topics". Add the usual two lines of code to open the form frmAdvancedTopics.
3. Place a button on form frmAdvancedTopics and name it btnLINQArray. Set its Text property to "LINQ Array". Double-click the button to create the default click event.
4. Save and run the project.

```
private void btnLINQArray_Click(object sender, EventArgs e)
{
    //Variables
    string strNumbers = string.Empty;
    string strSortedNumbers = string.Empty;
    //Generate array of numbers
    int[] aintNumbers = new int[] { 231, 34, 856, 238, 35, 85, 385, 594 };
    strNumbers = string.Join(", ",aintNumbers);
    MessageBox.Show(strNumbers);

    //Define LINQ query, perform sorting
    var nums = from num in aintNumbers orderby num descending select num;
    //Convert LINQ query back to an array
    int[] aintSortedNumbers = nums.ToArray();
    strSortedNumbers = string.Join(", ", aintSortedNumbers);
    MessageBox.Show(strSortedNumbers);

}
```

Using LINQ with Databases

Using LINQ with database requires you to set up a typed dataset since LINQ is required to know the column names and data types at compile time (early binding). In the examples so far, we have created untyped datasets that are resolved at run-time (late-binding).

You can either use the built-in interface to create an Object-Relational map between the database and the C# environment (Entity Framework) or create a mapping class.

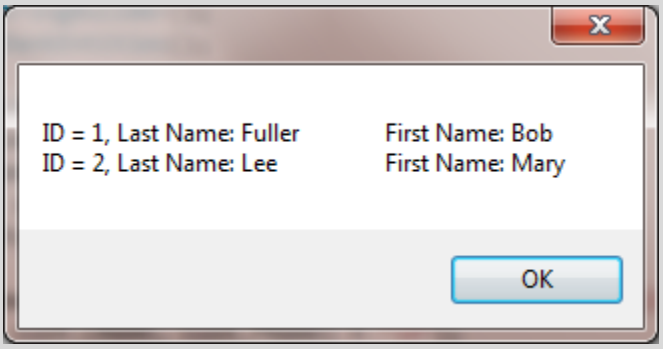
In the next example, we are using the existing EDM created earlier in this course.

Example 10-2: Using LINQ with Databases

1. Place a button on form frmAdvancedTopics and name it btnLINQDB. Set its Text property to "LINQ DB".
2. Double-click the button to create the default click event and write the following method.

```
private void btnLinqDB_Click(object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder();
    BankEntities bank = new BankEntities();
    //LINQ Query
    var custs = from c in bank.Customers
                where c.CustomerID < 3
                orderby c.LName
                select new { c.LName, c.FName, c.CustomerID };
    foreach (var cust in custs)
    {
        sb.Append(string.Format("ID = {0}, Last Name: {1} \t First Name: {2}",
            cust.CustomerID, cust.LName, cust.FName) + "\n");
    }
    MessageBox.Show(sb.ToString());
}
```

3. Save and run project.



25.2 Processing Text Files

To implement data interfaces in your application, you may need to import data from an outside source or export data from your application. To do that, you need to build files and process them accordingly. In this chapter we will discuss the basic classes to achieve that.

There are three elements when working with files. First of all, you need to have tools to access the file system of the underlying operating system. This is needed to specify paths and filenames, to search, create, and delete files, etc. The second piece is the FileStream object that gives you access to the file, whether for input or output. And finally the third element is StreamReader/StreamWriter class to read or write a stream of characters.

System.IO Class

The System.IO namespace contains many different classes. Table 49 below shows the classes used to work with drives and subdirectories.

Class	Description
Directory	Used to create, edit, delete, or get information on directories (folders).
File	Used to create, edit, delete, or get information on files.
Path	Used to get path information from a variety of platforms.

Table 49: System.IO Main Classes

Directory Class

You can use the methods of the Directory class to create or delete a directory or determine if a directory exists.

Table 50 shows a list of common methods of the Directory class.

Method	Description
Exists(*path*)	Returns a Boolean indicating whether a directory exists.
CreateDirectory(*path*)	Creates the directories in a specified path
Delete(*path*)	Deletes the directory at the specified path. The directory must be empty.
Delete(*path*,recursive)	Deletes the directory at the specified path. If true is specified for the recursive argument, any subdirectories and files in the directory are deleted. If false is specified, the directory must be empty.

Table 50: Common Methods of Directory Class

The File Class

You can use the methods of the File class to copy, delete, or move a file, or to determine if a file exists.

Table 51 shows a list of common methods of the File class.

Method	Description
Exists(*path*)	Returns a Boolean indicating whether a file exists.
Delete(*path*)	Deletes a file. The file does not have to exist.
Copy(*source, dest*)	Copies a file from a source path to destination path.
Move(*source,dest*)	Moves a file from a source path to destination path.

Table 51: Common Methods of the File Class

Example 10-3: Using the File Class

1. Use the form frmAdvancedTopics and add a button named btnCreateFile. Set its Text property to "Create File". Double-click on the button to create the default click event handler.
2. Write the following code into the event handler. Make sure to place the System.IO directive at the top of the class.

```
private void btnCreateFile_Click(object sender, EventArgs e)
{
    string strFilePath = @"c:\temp\WinApps\test.txt";
    if (File.Exists(strFilePath))
    {
        File.Delete(strFilePath);
        MessageBox.Show("File Deleted", "Delete", MessageBoxButtons.OK);
    }
    else
    {
        //Use Resource acquisition statement to ensure
        //that created file object is properly disposed of
        using (File.Create(strFilePath))
        {
            MessageBox.Show("File Created", "Create", MessageBoxButtons.OK);
        }
    }
}
```

3. Save the project and run the form.
4. Test the button by clicking it once. Verify that the file has been created.
5. Click the button again and verify that the file has been deleted

The FileStream Class

Now we are ready to process files. There are two kinds of files, text files and binary files. A text file contains data that is stored as text characters. A binary file contains other characters that are used and processed by the application the file was created.

In Figure 83 you see an example of a binary file, in this case an Access mdb file opened in a text editor.

To handle files, the .NET framework uses streams. A stream is a flow of data from one point to another. For example, an input stream can flow from an external file into memory. The FileStream class is used to connect a stream to a file.

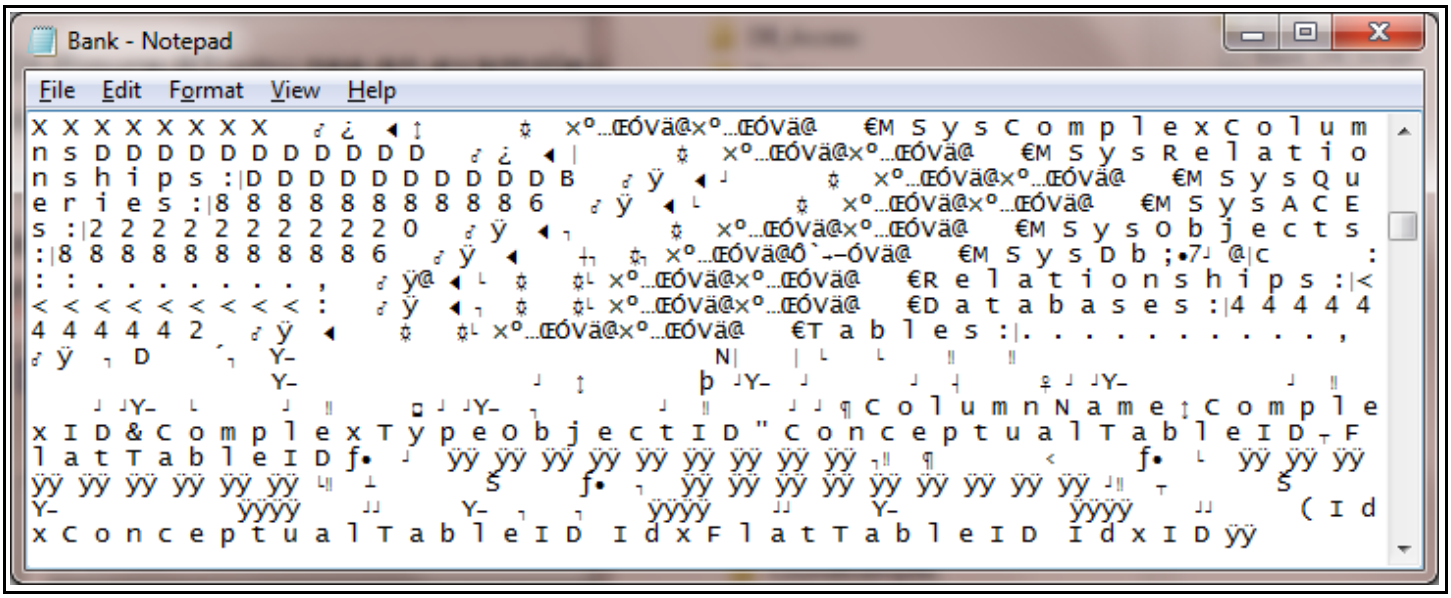


Figure 83: Sample Binary File

The syntax for creating a FileStream object is shown below:

```
FileStream fs = new FileStream(path, FileMode [, FileAccess [, FileShare] ] )
```

Table 52 shows the members in the FileMode enumeration:

Member	Description
Append	Opens the file if it exists and moves the pointer to the end of the file. If the file does not exist, it is created. This member can only be used with the Write file access.
Create	Creates a new file. If the file exists, it is overwritten.
CreateNew	Create a new file. If the file already exists, an exception is thrown.
Open	Opens an existing file. If the file does not exist, an exception is thrown.
Open Or Create	Opens a file if it exists, or creates a new file if it does not exist.
Truncate	Opens an existing file and truncates it so its size is zero bytes.

Table 52: FileMode Enumeration Members

Table 53 shows members in the FileAccess enumeration:

Member	Description
Read	Data can be read from the file, but not written to it.
ReadWrite	Date can be read and written to the file. This is the default
Write	Data can be written to the file but not read from it.

Table 53: FileAccess Enumeration Members

Table 54 shows members in the FileShare enumeration

Member	Description
None	The file cannot be opened by another application.
Read	Allows other applications to open the file for reading only. This is the default.
ReadWrite	Allows other applications to open the file for both reading and writing
Write	Allows other applications to open the file for writing only.

Table 54: FileShare Enumeration Members


Warning: Operating system level permissions may limit which file access and file share options you can use.

When working with files, there are many possible I/O errors. For example, the directory may not exist or the file may not exist. Depending on the options chosen for the FileStream object, an exception is thrown. Table 55 shows a list of common exception classes for file I/O operations.

Exception Class	Description
IOException	The base class for exceptions thrown during processing of a stream, file, or directory.
DirectoryNotFoundException	Occurs when directory cannot not be found.
FileNotFoundException	Occurs when file cannot be found
EndOfStreamException	Occurs when application attempts to read beyond the end of a stream.

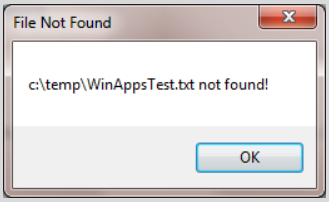
Table 55: IO Exception Classes

Example 10-4: Using the FileStream Class

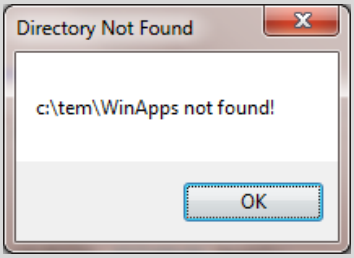
1. Add a new button on frmAdvancedTopics and name it btnFileStream. Set its Text property to "File Stream".
2. Double-click on the button btnFileStream to create the default click event handler.
3. Write the following code:

```
private void btnFileStream_Click(object sender, EventArgs e)
{
    string strDir = @"c:\temp\WinApps\";
    string strFile = "Test.txt";
    FileStream fs = null;
    try
    {
        fs = new FileStream(strDir + strFile, FileMode.Open,FileAccess.ReadWrite);
    }
    catch (FileNotFoundException)
    {
        MessageBox.Show(strDir + strFile + " not found!", "File Not Found", MessageBoxButtons.OK);
    }
    catch (DirectoryNotFoundException)
    {
        MessageBox.Show(strDir + " not found!", "Directory Not Found", MessageBoxButtons.OK);
    }
    catch (IOException ex)
    {
        MessageBox.Show(ex.Message, "IO Exception", MessageBoxButtons.OK);
    }
    finally
    {
        if (fs != null)
        {
            fs.Close();
            fs.Dispose();
        }
    }
}
```

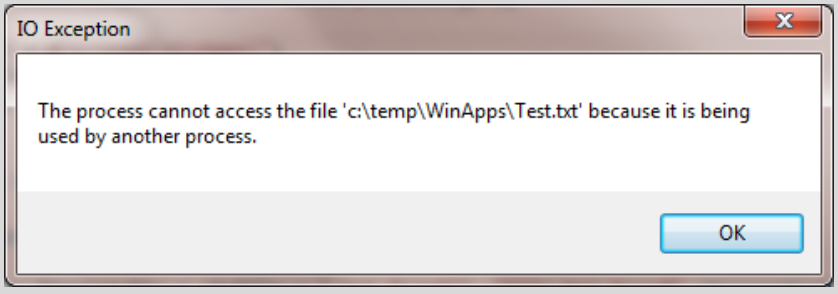
4. Save the project and run the form. Test the error handler functionality by using the Create File button of Example 10-3 to create and delete the file.
5. When the file is deleted, the following message box is displayed:



6. Modify the path in the event handler to "c:\tem\WinApps". The following message box is displayed.



7. Create the file again by clicking on the Create File button. Then Click on the File Stream button. The following message box is displayed:



The StreamReader/StreamWriter Classes

The StreamReader/Writer classes allow you now to finally process the information contained in the files. The basic syntax is shown below. These classes need a FileStream object as an input.

```
StreamWriter sw = new StreamWriter(filestream)
StreamReader sr = new StreamReader(filestream)
```

Table 56 and Table 57 show some common methods of the StreamWriter and the StreamReader classes.

StreamWriter Method	Description
Write(data)	Writes data to the output stream.
WriteLine(data)	Writes the data to the output stream and appends a line terminator (usually a carriage return and a line feed).
Close()	Closes the StreamWriter object and the associated FileStream object.

Table 56: StreamWriter Class Methods

StreamReader Method	Description
Peek()	Returns the next available character in the input stream without advancing to the next position. If no more characters are available, a -1 is returned.
Read()	Reads the next character from the input stream.
ReadLine()	Reads the next line of characters from the input stream and returns is as a string.
ReadToEnd()	Reads the data from the current position in the input stream to the end of the stream and returns it as a string. This is typically used to read the entire file.

Table 57: StreamReader Class Methods

When working with files, it is common to let the user browse the file system to select a file to read or write to. You can use the OpenFileDialog class to accomplish this task. The syntax is shown below:

```
OpenFileDialog ofd = new OpenFileDialog();
```

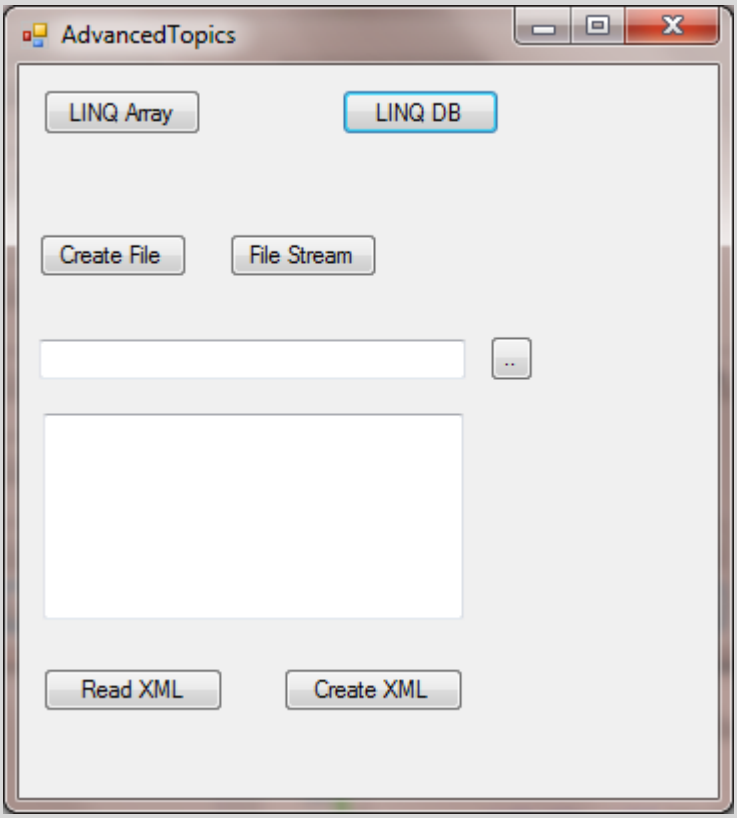
To display the file open dialog, you use the ShowDialog() method. Usually, you embed this method in an if statement and check whether the user click on the OK button to select a file or the Cancel button:

```
if (ofd.ShowDialog() == DialogResult.OK)
{
        statements
}
else
{
        MessageBox.Show("No File selected!", "Cancel", MessageBoxButtons.OK);
}
ofd.Dispose();
```

We will use the open file dialog box in the next example.

Example 10-5: Using the StreamReader Class

1. Add a button named btnOpenFileDialog to the form frmAdvancedTopics. Double-click on the button btnOpenFileDialog to create the default click event handler. Write the following code:




```
private void btnOpenFileDialog_Click(object sender, EventArgs e)
{
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Title = "Select a text file";
    ofd.Filter = "txt files(*.txt)|*.txt|All files(*.*)|*.*";
    ofd.InitialDirectory = @"c:\temp\Winapps\";
    if (ofd.ShowDialog() == DialogResult.OK)
    {
        txtFilePath.Text = ofd.FileName;
        using (FileStream fs = new FileStream(txtFilePath.Text.ToString(),FileMode.Open, FileAccess.Read))
        {
            StreamReader sr = new StreamReader(fs);
            txtFileContent.Text = sr.ReadToEnd();
        }
    }
    else
    {
        MessageBox.Show("No File selected!", "Cancel", MessageBoxButtons.OK);
    }
    ofd.Dispose();
}
```

2. Create a simple text file under c:\temp\WinApps and name it StreamReader.txt.
3. Save the project and run the form. Test the file open dialog box and selecting a file.
4. The content of the file should be displayed in the big textbox control whereas the filepath should be displayed in the upper textbox control.

25.3 Processing XML Files

Processing XML files is similar to processing text files. We will take a look at the two most important classes, the XmlReader and XmlWriter class. Both of these classes are contained in the System.Xml namespace.

Structure of XML Markup Language

An XML document contains data in a structured way. It contains tags, declarations, comments, similar to HTML. Whereas in HTML the tags are pre-defined, in XML you define your own tags.

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Customer Data-->
<Customers>
  <Customer ID="1">
    <FirstName>Bob</FirstName>
    <LastName>Fuller</LastName>
    <Gender>M</Gender>
  </Customer>
  <Customer ID="2">
    <FirstName>Mary</FirstName>
    <LastName>Lee</LastName>
    <Gender>F</Gender>
  </Customer>
</Customers>
```

Figure 84: Sample XML File

Each XML element has a start and an end tag. The end tag contains a slash before the name of the tag, but inside the angle brackets. Comments are specified in the same way as in HTML, using the following syntax:

<!—*comment* -->

The very first line in the XML document is the declaration, it signifies that this is an XML document and that it conforms to a certain version.

Between the start and the end tag you put the actual data. Besides content, elements can contain other elements, known as child elements. This allows you to structure your data in a parent-child relationship.

In the above figure, you see that the FirstName, LastName, and Gender elements are within the Customer element. These elements are child elements of the Customer element.

Besides child elements you can also structure your data as attributes. In the above figure, the ID is an attribute of the Customer element.

The XmlReader Class

You use the XmlReader class to read an xml file into your program. To create an object from this class, you use the Create method and supply a path and file name as a string. In addition to this class, you can also use the XmlReaderSettings class to specify certain settings when reading an xml file such as ignoring comments, for instance. The most commonly used properties and methods of both classes are shown in Table 58.

Property	**Description**
NodeType	Returns a member of the XmlNodeType enumeration that indicates the type of the current node.
Name	Gets the name of the current node (if node has a name).
Value	Gets the value of the current node (if node has a value).
EOF	Returns a true if the reader has reached the end of the stream.
Method	**Description**
Create(*path* [,*settings*])	A static method that creates an XmlReader object using the specified path and optionally the specified settings.
Read	Reads the next node.
ReadStartElement	Checks that the current node is a start tag with the specified name, and then advances to the next node.
ReadEndElement	Checks that the current node is an end tag, then advances to the next node.
ReadToDescendant(*name*)	If specified descendant is found, this method advances to it and returns a true. Otherwise, it advances to the end tag of the current element and returns a false.
ReadToNextSibling(name)	If the specified sibling element is found, this method advances to it and returns a true. Otherwise, it advances to the end tag of the current element and returns a false.
ReadElementContentAsString() **ReadElementContentAs**……	Reads the current element, returns the content as string, and moves past the element's end tag. Other ReadElementContentAs include Decimal and other data types.
Close()	Closes the XmlReader and releases any resources.

Table 58: Properties and Methods of XmlReader Class

Property	Description
IgnoreWhiteSpace	Gets or sets a Boolean indicating whether to ignore white space. If set to false, white space is treated as a node.
IgnoreComments	Gets or sets a Boolean indicating whether to ignore comments.

Table 59: Properties of XmlReaderSettings Class

Example 10-6: Using the XmlReader Class

1. First of all, we need to create a Customer class to store customer data that is processed in the XML files. Due to the Entity Framework model we already have instances of customer defined, so we are going to call this class cust (the filename is still Customer.cs).
2. Create a class file named Customer.cs and place it in the BAL folder. Write the following code.

```
public class Cust
{
    public Cust()
    {
    }
    public Cust(string id, string fName, string lName, string gender)
    {
        ID = id;
        FirstName = fName;
        LastName = lName;
        Gender = gender;
    }
    public string ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Gender { get; set; }
}
public class CustomerList
{
    private List<Cust> customers;
    public CustomerList()
    {
        customers = new List<Cust>();
    }
    public void Add(Cust cust)
    {
        customers.Add(cust);
    }
    public void Add(string id, string fName, string lName, string gender)
    {
        Cust c = new Cust(id, fName, lName, gender);
    }
}
```

Example 10-6 (continued): Using the XmlReader Class

3. Create the following static method inside the frmAdvancedTopics code file:

```
public static List<Cust> GetCustomer()
{
    //Create the Customer list
    List<Cust> customers = new List<Cust>();
    //Create the filepath
    string strFilePath = @"C:\Temp\Winapps\Customer_Read.xml";
    //Create the XML Reader Settings  object
    XmlReaderSettings xmlSettings = new XmlReaderSettings();
    xmlSettings.IgnoreWhitespace = true;
    xmlSettings.IgnoreComments = true;
    //Create the XML Reader object using resource acquisition statement
    using (XmlReader xmlCustomer = XmlReader.Create(strFilePath, xmlSettings))
    {
        //read past all nodes to the first customer node
        if (xmlCustomer.ReadToDescendant("Customer"))
        {
            do
            {
                Cust customer = new Cust();
                customer.ID = xmlCustomer["ID"];
                xmlCustomer.ReadStartElement("Customer");
                customer.FirstName = xmlCustomer.ReadElementContentAsString();
                customer.LastName = xmlCustomer.ReadElementContentAsString();
                customer.Gender = xmlCustomer.ReadElementContentAsString();
                customers.Add(customer);
            }
            while (xmlCustomer.ReadToNextSibling("Customer"));
            xmlCustomer.Close();
            MessageBox.Show("Finish Reading XML File!\n Customer Count: " + customers.Count,
                "Read XML", MessageBoxButtons.OK);
        }
    }
    return customers;
}
```

4. Add a button named btnReadXMl and set its text property to "Read XML". Double-click on the button btnReadXML to create the default click event. Simply call the above method in the event handler:

```
private void btnReadXML_Click(object sender, EventArgs e)
{
    lstCustomer = GetCustomer();
}
```

```
public partial class frmAdvancedTopics : Form
{
    List<Cust> lstCustomer;

    public frmAdvancedTopics()
    {
        InitializeComponent();
    }
```

5. Create the following object reference at the form's class level:
6. Make sure the Customer_Read.xml file is in the C:\Temp directory.
7. Save the project and run it. Test the Read XML functionality.

The XmlWriter Class

The XmlWriter class makes it easy to write an XML file. It takes care of generating the XML tags, so that you can focus on the document's structure and content. This class is very similar to the XmlReader class in terms of methods and properties, except of course, it is writing XML elements rather than reading.

Table 60 shows the most commonly used methods.

Method	Description
Create(*path* [, *setting*])	A static method that creates an XmlWriter object using the specified path and optional settings.
WriteStartDocument()	Writes an XML declaration at the beginning of a document.
WriteComment(*comment*)	Writes a comment to the XML document.
WriteStartElement(*element_name*)	Writes a start tag using the element name provided.
WriteAttributeString(*name, value*)	Adds an attribute value to the supplied element name.
WriteEndElement()	Writes an end tag for the current element.
WriteElementString(*name, content*)	Writes a complete element including a start, tag, content, and end tag.

Table 60: Methods of XmlWriter Class

Table 61 shows the most commonly used properties of the XmlWriterSettings class:

Property	Description
Indent	Gets or sets a Boolean indicating whether to indent elements.
IndentChars	Gets or sets the string to use when indenting.

Table 61: Properties of XmlWriterSettings Class

Example 10-7: Using the XmlWriter Class

1. Add another button named btnCreateXML and set its Text property to "Create XML".
2. Double-click on this button to create the default click event. Write the following code to call a static method that we will code afterwards.

```
private void btnCreateXML_Click(object sender, EventArgs e)
{
    SaveCustomer(lstCustomer);
}
```

3. Again, this SaveCustomer method depends on the Customer class that is included in this project. The lstCustomer is the object that stores the customer information that was read from the XML file in Example 10-6.
4. Now code the following static method:

```
public static void SaveCustomer(List<Cust> customers)
{
    //Create the filepath
    string strFilePath = @"C:\Temp\Winapps\Customer_Write.xml";
    //XmlWriterSettings object
    XmlWriterSettings xmlSettings = new XmlWriterSettings();
    xmlSettings.Indent = true;
    xmlSettings.IndentChars = ("    ");
    //Create XmlWriter object using resource acquisition statement
    using (XmlWriter xmlCustomer = XmlWriter.Create(strFilePath, xmlSettings))
    {
        //Write the start section of the document
        xmlCustomer.WriteStartDocument();
        xmlCustomer.WriteStartElement("Customers");
        foreach (Cust customer in customers)
        {
            xmlCustomer.WriteStartElement("Customer");
            xmlCustomer.WriteAttributeString("ID", customer.ID);
            xmlCustomer.WriteElementString("FirstName", customer.FirstName);
            xmlCustomer.WriteElementString("LastName", customer.LastName);
            xmlCustomer.WriteElementString("Gender", customer.Gender);
            xmlCustomer.WriteEndElement();
        }
        //Write end section of document
        xmlCustomer.WriteEndElement();
        //Close the XMLWriter object
    }
    MessageBox.Show("Finish Saving XML File!", "Save XML", MessageBoxButtons.OK);
}
```

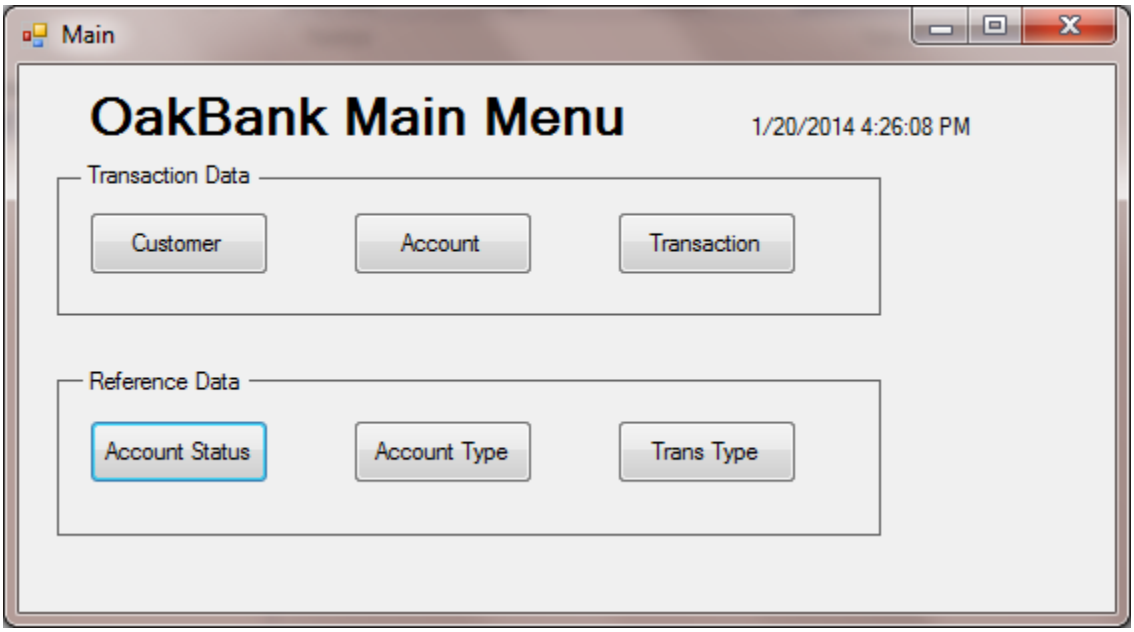
5. Save the project and run it. You need to first click on the Read XML button to load the lstCustomer object. Then click on the Create XML button to generate the xml file.
6. Now make sure that the Customer_Write.xml file was created. Open it in Notepad and verify that it contains all the information.

26. Sample Database Application

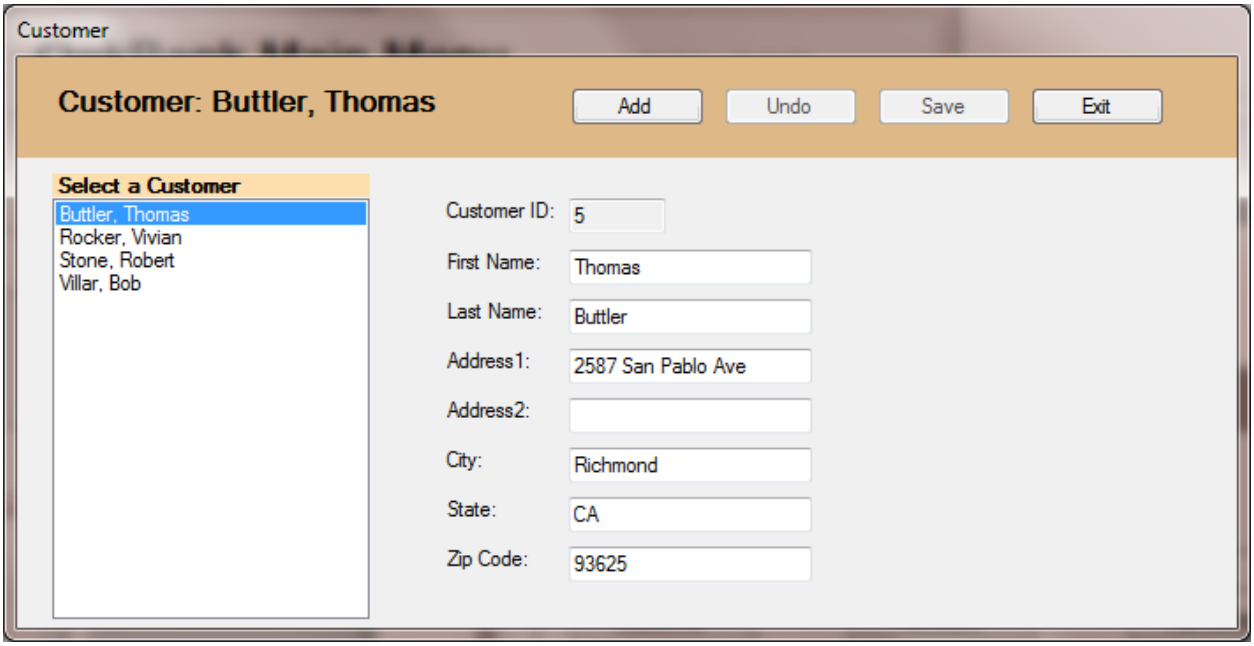
In this chapter, we will build together an application based on the class examples and homework assignments.

26.1 Main Screens

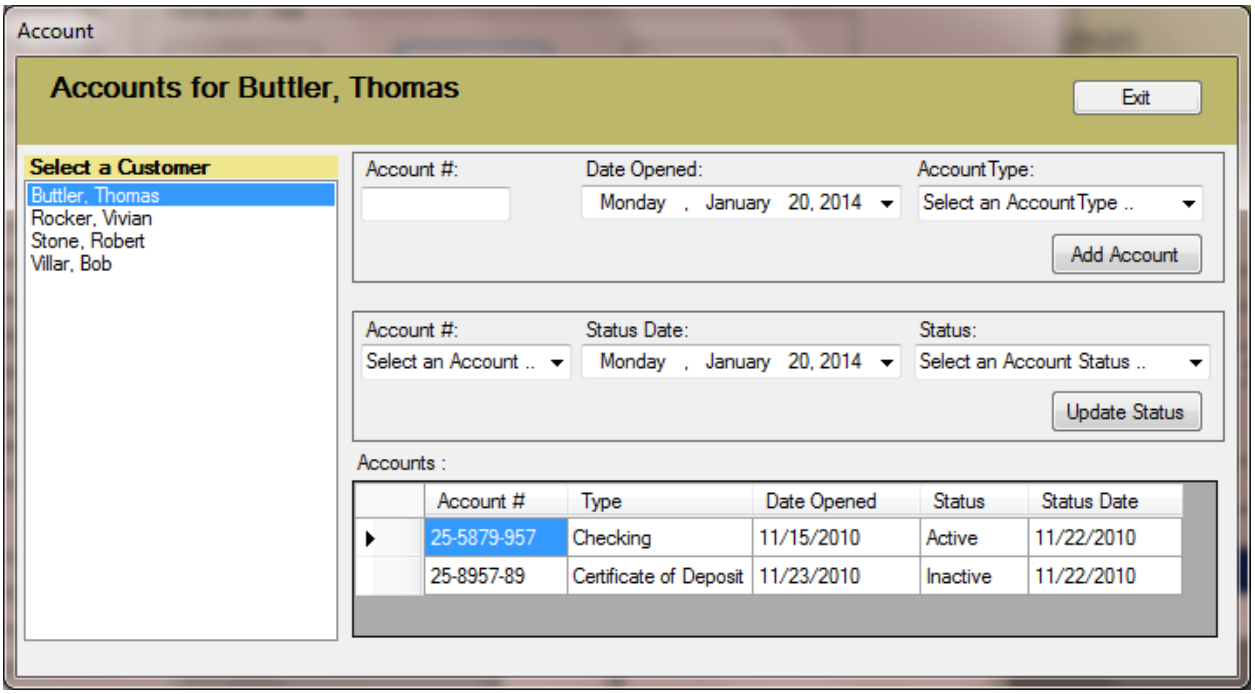
The OakBank Main Menu:



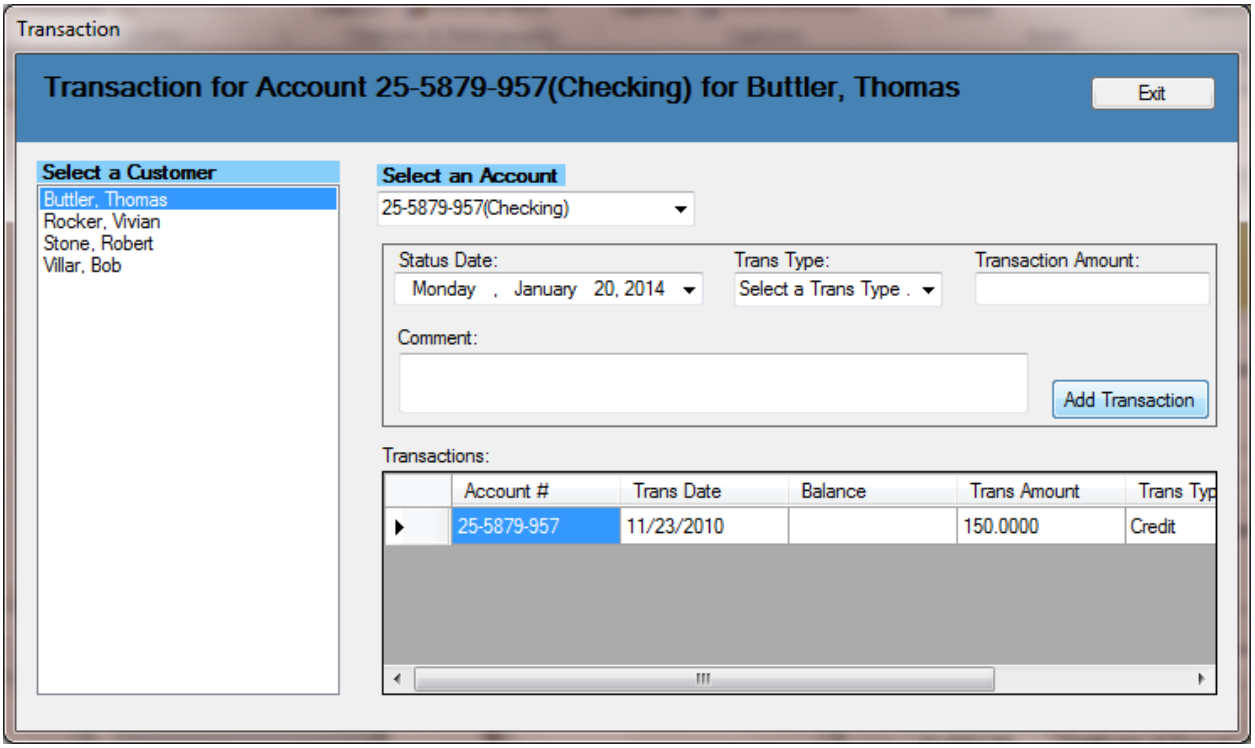
Customer Form:



Account Form:



Transaction Form:



Reference Data Forms: