**Entry Validation**

Throughout your work with databases, you will find that viewing database information is an easy task with Visual C#. Things quickly become difficult, though, when you want to modify information in a database. And, things become very difficult when you allow your user to type information. That’s why, if at all possible, don’t allow your user to type things. Use point and click type controls whenever possible.

Checking input information from a user requires programming on your part. You must insure information being put in a database is correct. There are two steps to checking information from a user: **entry** validation and **input** validation. Entry validation is associated with text box controls and checks for proper keystrokes. Input validation is associated with several control types and checks to make sure entries and choices meet certain requirements. In this section, we address entry validation. Input validation is addressed in the next section of this chapter.

As mentioned, entry validation checks for proper keystrokes. For example, if a numerical entry is needed, only allow the pressing of number keys. If spaces are not allowed, don’t allow them. If an input must be in upper case letters, don’t allow lower case letters to be typed. Restricting keystrokes is referred to as **key** **trapping**.

**Key Trapping**

Key trapping is done in the **KeyPress** event method of a text box control. Such a method has the form (for a text box named **txtExample**):

**private void TxtExample\_KeyPress(object sender, KeyPressEventArgs e)**

**{**

**}**

What happens in this method is that every time a key is pressed in the corresponding text box, the **KeyPressEventArgs** class passes the key that has been pressed into the method via the **char** type **e.KeyChar** property. Recall the **char** type is used to represent a single character. We can thus examine this key. If it is an acceptable key, we set the **e.Handled** property to **False**. This tells Visual C# that this method has not been handled and the KeyPress should be allowed. If an unacceptable key is detected, we set **e.Handled** to **true**. This ‘tricks’ Visual C# into thinking the KeyPress event has already been handled and the pressed key is ignored.

We need some way of distinguishing what keys are pressed. The usual alphabetic, numeric and character keys are fairly simple to detect. To help detect non-readable keys, we can examine the key’s corresponding Unicode value. Two values we will use are:

**Definition Value**

Backspace 8

Carriage return (<Enter> key) 13

As an example, let’s say we have text box (**txtExample**) and we only want to enter numbers or a decimal point. There are several ways to build a key trapping routine. I suggest an if block that, based on different values of **e.KeyChar**, takes different steps. If e.KeyChar represents a number, a decimal point or a backspace key (always include backspace or the user won’t be able to edit the text box properly), we will allow the keypress (e.Handled = false). Otherwise, we will set e.Handled = true to ignore the keypress (we also add a beep). The code to do this is:

**if (e.KeyChar >= '0' && e.KeyChar <= '9')**

**{**

**// number values**

**e.Handled = false;**

**}**

**else if ((int) e.KeyChar == 8)**

**{**

**// backspace**

**e.Handled = false;**

**}**

**else if (e.KeyChar == '.')**

**{**

**// decimal point**

**e.Handled = false;**

**}**

**else**

**{**

**// any other character**

**e.Handled = true;**

**}**

Note the use of single quotes to signify **char** types, the same type as e.KeyChar.

**Example 5-5**

**Authors Table Input Form**

**(Entry Validation)**

In the Authors Table Input Form, the **Year Born** field can only be numeric data.

1. Load Example 5-4 completed earlier. We will modify this example to include entry validation.
2. Use this code to the **txtYearBorn KeyPress** event (make sure you select the proper event – don’t use the TextChanged event!):

**private void TxtYearBorn\_KeyPress(object sender, KeyPressEventArgs e)**

**{**

**if ((e.KeyChar >= '0' && e.KeyChar <= '9') || (int) e.KeyChar == 8)**

**{**

**//Acceptable keystrokes**

**e.Handled = false;**

**}**

**else**

**{**

**e.Handled = true;**

**Console.Beep();**

**}**

**}**

1. Save and run the application (saved in the **Example 5-5** folder in **VCSDB\Code\Class 5** folder). Click **Edit** to switch to Edit state**.** Click the **Year Born** text box. Try some typing. You should only be able to type numbers (or use the backspace key) in the Year Born entry box. You should hear a beep sound when you type an incorrect key.

**Example 5-5**

**Using SQL Server Databases**

Make the same changes to the SQL Server version of the project. The SQL Server version is saved in the **Example 5-5 SQL** folder in **VCSDB\Code\Class 5** folder.

**Input Validation**

In the example just studied, although the user can only input numeric data for the Year Born field, there is no guarantee the final input would be acceptable. What if the input year is past the current year? What if the year is 1492? A second step in validation is to check values in context. Do the input values make sense? Do the values meet established rules? This step is **input** **validation**.

Some common validation rules are:

1. Is this field required? If a field is required and no input is provided, this could cause problems.
2. Is the input within an established range? For example, if entering a day number for the month of April, is the value between 1 and 30?
3. Is the input the proper length? Social security numbers (including hyphens) require 11 characters. If 11 characters are not detected, the input is not a valid social security number. The C# **Length** property can be used here, as can a text box **MaxLength** property (to limit the length).
4. Is the input conditional? Some fields only need to filled in if other fields are filled in. For example, if a user clicks to ship to another address, you need to make sure that address exists.
5. Is the input a primary key? If so, and the user has the capability of entering a value, we must insure it is a unique value. Each primary key value in a table must be different.

The amount of input validation required is dependent on the particular field. Many times, there is none needed. You, as the programmer, need to examine each input field and answer the questions posed above: is the field required, must it be within a range, is it length restricted, is it conditional? Any Yes answers require C# code to do the validation. You will probably find additional questions as you develop your database skills.

Where does the validation code go? It really depends on how you implement database editing. We will discuss this topic in detail in Chapter 6. For our example we have been creating, we will write a general method named **ValidateData** that is called in the **Click** event of the **Save** button. The user clicks this button when done editing, making it a great place to check validity. If any validation rules are violated, we don’t allow the requested change(s).

We see entry and input validation require a bit of programming on our part. But, it is worth it. Field validation insures the integrity of the information we are putting in a database. We always need to maintain that integrity. And, one last time for emphasis (are you getting the idea this is important) – if you can eliminate user typing – do it!

**Example 5-6**

**Authors Table Input Form**

**(Input Validation)**

As mentioned, the **Year Born** must be validated. We will make sure that, if an input is attempted (we won’t require a year be input), the year has no more than four characters, is not greater than the current year and is greater than 150 years prior to the current year (by not hard coding a minimum year, the code automatically upgrades itself). We will also make sure the user enters an Author Name.

1. Load Example 5-5 completed earlier. We will modify this example to include input validation.
2. Set **MaxLength** property for **txtYearBorn** text box to **4**.
3. Add a method named **ValidateData** that returns a Boolean argument (if true, all validation rules were met). Add this code:

**private bool ValidateData()**

**{**

**string message = "";**

**int inputYear, currentYear;**

**bool allOK = true;**

**// Check for name**

**if (txtAuthorName.Text.Trim().Equals(""))**

**{**

**message = "You must enter an Author Name." + "\r\n";**

**txtAuthorName.Focus();**

**allOK = false;**

**}**

**// Check length and range on Year Born**

**if (!txtYearBorn.Text.Trim().Equals(""))**

**{**

**inputYear = Convert.ToInt32(txtYearBorn.Text);**

**currentYear = DateTime.Now.Year;**

**if (inputYear > currentYear || inputYear < currentYear - 150)**

**{**

**message += "Year born must be between " + (currentYear - 150).ToString() + " and " + currentYear.ToString();**

**txtYearBorn.Focus();**

**allOK = false;**

**}**

**}**

**if (!allOK)**

**{**

**MessageBox.Show(message, "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Information);**

**}**

**return (allOK);**

**}**

In this code, we first check to see if an **Author** **Name** is entered and then validate the **Year Born** field. If either validation rule is violated, the variable **allOK** is set to **false** and a message box displayed. If any of this code is unfamiliar, try Visual C# on-line help for assistance.

1. Modify the **btnSave Click** event to read (new lines are shaded):

**private void BtnSave\_Click(object sender, EventArgs e)**

**{**

**if (!ValidateData())**

**{**

**return;**

**}**

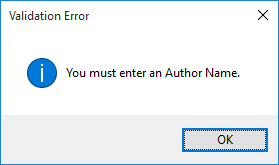
**MessageBox.Show("Record saved.", "Save", MessageBoxButtons.OK, MessageBoxIcon.Information);**

**SetState("View");**

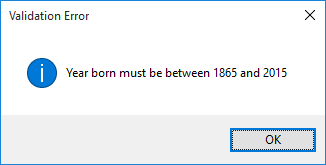
**}**

In the new line of code, if the **ValidateData** function returns a **false**, the data is not valid and we exit the method.

1. Save and run the application. Click **Edit** and blank out the **Author** **Name**. Click **Save**. A message box should appear:



Click **OK** and the focus is reset on the **Author** **Name** text box, helping the user. Type an invalid numeric value in the **Year Born** box. Click **Save**. A new message should be displayed:



If you attempt a year, you must either enter a valid value or click **Cancel**. Try a valid year and valid name – make sure they are accepted.

1. After typing a new Author name, to type a Year Born, you need to click in that text box. This clicking (especially when working with lots of text boxes) is cumbersome. A preferred method would be a programmatic shift of focus. Add this code at the top of the **txtAuthorName KeyPress** event:

**private void TxtAuthorName\_KeyPress(object sender, KeyPressEventArgs e)**

**{**

**if ((int) e.KeyChar == 13)**

**{**

**txtYearBorn.Focus();**

**}**

**}**

In this code, if the **<Enter>** key is pressed, the focus is shifted from the Author text box to the Year Born text box (if a valid name is input). This programmatic change of focus is used all the time in database interfaces. Users like to see the focus move when they press <Enter>. It is an additional step in maintaining proper application state. To shift from the Year Born box to the Author box, add the shaded code to the **txtYearBorn KeyPress** event:

**private void TxtYearBorn\_KeyPress(object sender, KeyPressEventArgs e)**

**{**

**if ((e.KeyChar >= '0' && e.KeyChar <= '9') || (int) e.KeyChar == 8)**

**{**

**//Acceptable keystrokes**

**e.Handled = false;**

**}**

**else if ((int)e.KeyChar == 13)**

**{**

**txtAuthorName.Focus();**

**}**

**else**

**{**

**e.Handled = true;**

**Console.Beep();**

**}**

**}**

1. Save (saved in the **Example 5-6** folder in **VCSDB\Code\Class 5** folder) and run the example again. Click **Edit**. Notice how the focus shifts between the two text boxes as you change the values and press <Enter>. Pressing <Tab> should also change the focus appropriately.

**Example 5-6**

**Using SQL Server Databases**

Make the same changes to the SQL Server version of the project. The SQL Server version is saved in the **Example 5-6 SQL** folder in **VCSDB\Code\Class 5** folder.

**Error Trapping and Handling**

Even with a well-designed, ‘user-proof’ interface, errors can still occur. This is especially true when working with databases. Occasionally, data cannot be written to, or deleted from, the database or invalid fields are encountered. Without any action on our part, these **run-time errors** might bring our application to an unceremonious end. If, however, we recognize an error has occurred and inform the user of the problem, we might be able to recover.

**Run-time errors** (referred to in Visual C# as **exceptions**)are “catchable.” That is, Visual C# recognizes an error has occurred and enables you to catch it and take corrective action (handle the error). As mentioned, if an error occurs and is not caught, your program will usually end in a rather unceremonious manner. Most run-time errors occur when your application is working with files, either trying to open, read, write or save a file. Other common run-time errors are divide by zero, overflow (exceeding a data type’s range) and improper data types.

Error trapping and handling must be implemented in every method in your application where you think it might be needed. Visual C# does not allow global error trapping. At a minimum, you should implement error trapping and handling in every method that writes to or reads from the database.

Visual C# uses a structured approach to catching and handling exceptions. The structure is referred to as a **try**/**catch**/**finally** block. And the annotated syntax for using this block is:

##### try

**{**

**// here is code you try where some kind of**

**// error may occur**

**}**

**catch (ExceptionType ex)**

**{**

**// if error described by exception of ExceptionType**

**// occurs, process this code**

**}**

**catch (Exception ex)**

**{**

**// if any other error occurs, process this code**

**}**

### finally

**{**

**// Execute this code whether error occurred or not**

**// this block is optional**

**}**

**// Execution continues here**

The above code works from the top, down. It ‘tries’ the code between **try** and the first **catch** statement. If no error is encountered, any code in the **finally** block will be executed and the program will continue after the right brace closing the **try/catch/finally** block. If an exception (error) occurs, the program will look to find, if any, the first **catch** statement (you can have multiple catch statements and must have at least one) that matches the exception that occurred. If one is found, the code in that respective block is executed (code to help clear up the error – the exception handling), then the code in the **finally** block, then program execution continues after the closing brace. If an error occurs that doesn’t match a particular exception, the code in the ‘generic’ **catch** block is executed, followed by the code in the **finally** block. And, program execution continues after the closing brace.

This structure can be used to trap and handle any **Type** of exception defined in the Visual C# **Exception** class. There are hundreds of possible exceptions related to data access, input and output functions, graphics functions, data types and numerical computations. Here is a list of example exception types (their names are descriptive of the corresponding error condition):

**ArgumentException ArgumentNullException**

**ArgumentOutOfRangeException ArithmeticException**

**ArrayTypeMismatchException DivideByZeroException**

**DllNotFoundException Exception**

**FormatException IndexOutOfRangeException**

**DirectoryNotFoundException EndOfStreamException**

**FileNotFoundException IOException**

##### OutOfMemoryException OverflowException

Let’s take a closer look at the **catch** block. When you define a catch block, you define the exception type you want to catch. For example, if want to catch a divide by zero condition, an **DivideByZeroException**, we use:

##### catch (DivideByZeroException ex)

**{**

**// Code to execute if divide by zero occurs**

**}**

If in the **try** block, a divide by zero occurs, the code following this **catch** statement will be executed. You would probably put a message box here to tell the user what happened and provide him or her with options of how to fix the problem. To help with the messaging capability, the optional variable you define as the exception (**ex**, in this case) has a **Message** property you can use. The message is retrieved using **ex.Message**.

A **try** block may be exited using the **break** statement. Be aware any code in the **finally** block will still be executed even if break is encountered. Once the finally code is executed, program execution continues after the brace closing the try block.

**Example** of **try** block to catch a “file not found” error:

**try**

**{**

**// Code to open file**

**}**

**catch (FileNotFoundException ex)**

**{**

**// message box describing the error**

**MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**}**

**finally**

**{**

**//Code to close file (even if error occurred)**

**}**

**Example** of **try** block to catch a “formatting” error (happens when trying to convert an empty text string to a numeric value):

**try**

**{**

**// Code to format text string**

**}**

**catch (FormatException ex)**

**{**

**// write code that just sets numeric value to 0.0**

**}**

**finally**

**{**

**//Code to close file (even if error occurred)**

**}**

**Example** of a **generic** error catching routine:

**try**

**{**

**// Code to try**

**}**

**catch (Exception ex)**

**{**

**// message box describing the error**

**MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**}**

**finally**

**{**

**//Code to finish the block**

**}**

We’ve only taken a brief look at the structured run-time error handling capabilities of Visual C#. It is difficult to be more specific without knowing just what an application’s purpose is. You need to know what type of errors you are looking for and what corrective actions should be taken if these errors are encountered. As you build and run your own applications, you will encounter run-time errors. These errors may be due to errors in your code. If so, fix them. But, they may also be errors that arise due to some invalid inputs from your user, because a file does not meet certain specifications or because a disk drive is not ready. You need to use error handling to keep such errors from shutting down your application, leaving your user in a frustrated state.

**Example 5-7**

**Authors Table Input Form (Error Trapping)**

As mentioned, error trapping and handling should be included within every method where database information is read or written. It should also be included in methods where database files are being opened or saved.

1. Load Example 5-6 completed earlier. We will modify this example to include error trapping handling
2. Modify the **frmAuthors Load** method to incorporate error handling. Changes are shaded:

**private void FrmAuthors\_Load(object sender, EventArgs e)**

**{**

**try**

**{**

**// connect to books database**

**booksConnection = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source = c:\\VCSDB\\Working\\BooksDB.accdb");**

**booksConnection.Open();**

**// establish command object**

**authorsCommand = new OleDbCommand("Select \* from Authors ORDER BY Author", booksConnection);**

**// establish data adapter/data table**

**authorsAdapter = new OleDbDataAdapter();**

**authorsAdapter.SelectCommand = authorsCommand;**

**authorsTable = new DataTable();**

**authorsAdapter.Fill(authorsTable);**

**// bind controls to data table**

**txtAuthorID.DataBindings.Add("Text", authorsTable, "Au\_ID");**

**txtAuthorName.DataBindings.Add("Text", authorsTable, "Author");**

**txtYearBorn.DataBindings.Add("Text", authorsTable, "Year\_Born");**

**// establish currency manager**

**authorsManager = (CurrencyManager)this.BindingContext[authorsTable];**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show(ex.Message, "Error establishing Authors table.", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**return;**

**}**

**this.Show();**

**SetState("View");**

**}**

1. Modify the **btnAddNew Click**, **btnSave Click**, and **btnDelete Click** event methods to allow error trapping and handling. Use the generic code developed in this section, taking advantage of ‘cut and paste’ editing. The changes are shaded (more code will be added later):

**private void BtnAddNew\_Click(object sender, EventArgs e)**

**{**

**try**

**{**

**SetState("Add");**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show("Error adding record.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**}**

**}**

**private void BtnSave\_Click(object sender, EventArgs e)**

**{**

**if (!ValidateData())**

**{**

**return;**

**}**

**try**

**{**

**MessageBox.Show("Record saved.", "Save", MessageBoxButtons.OK, MessageBoxIcon.Information);**

**SetState("View");**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show("Error saving record.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**}**

**}**

**private void BtnDelete\_Click(object sender, EventArgs e)**

**{**

**DialogResult response;**

**response = MessageBox.Show("Are you sure you want to delete this record?", "Delete", MessageBoxButtons.YesNo, MessageBoxIcon.Question, MessageBoxDefaultButton.Button2);**

**if (response == DialogResult.No)**

**{**

**return;**

**}**

**try**

**{**

**}**

**catch (Exception ex)**

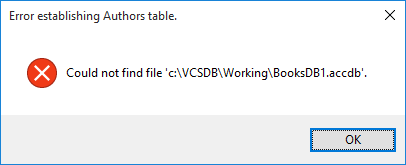
**{**

**MessageBox.Show("Error deleting record.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**}**

**}**

1. Save the application (saved in the **Example 5-7** folder in **VCSDB\Code\Class 5** folder). In the connection string, change the database name to **BooksDB1.accdb**. Run the application. The error trapping should display this message telling us it can’t find the database:



If error trapping were not in place, the user would have no idea what caused the program to stop. Change the database name back to the correct value (**BooksDB.accdb**).

**Example 5-7**

**Using SQL Server Databases**

Using the SQL Server version of the project, make the same changes to **frmAuthors Load**, **btnAddNew Click**, **btnSave Click**, and **btnDelete Click** event methods. The SQL Server version is saved in the **Example 5-7 SQL** folder in **VCSDB\Code\Class 5** folder.

**On-Line Help Systems**

So, at this point, we know how to build a powerful, intuitive interface, insure valid inputs, and handle any run-time errors that might occur. Even with all this work, there still may be times when the user is stumped as to what to do next. Instinct tells the user to press the **<F1>** function key. Long ago, someone in the old DOS world decided this would be the magic “Help Me!” key. Users expect help when pressing <F1> (I’m sure you rely on it a lot when using Visual C#). If nothing appears after pressing <F1>, user frustration sets in – not a good thing.

All applications written for other than your personal use should include some form of an on-line help system. It doesn’t have to be elegant, but it should be there. Adding a **help** **file** to your Visual C# application will give it real polish, as well as making it easier to use. In this section, we will show you how to build a very basic on-line help system for your database applications. This system will simply have a list of help topics the user can choose from.

We create what is known as an **HTML** **help** system. HTML stands for **hypertext** **markup** **language** and is the ‘programming’ language of choice for generating web pages. This language will be used to generate and display the topics displayed in the help system. Fortunately, we won’t need to learn much (if any) HTML. Building an HTML help system involves several files and several steps. In diagram form, we have:

###### Content Files

###### Navigation Files

HTML topic files (.HTM)

Index (.HHK)

Table of Contents (.HHC)

Graphics files (.GIF, .JPG)

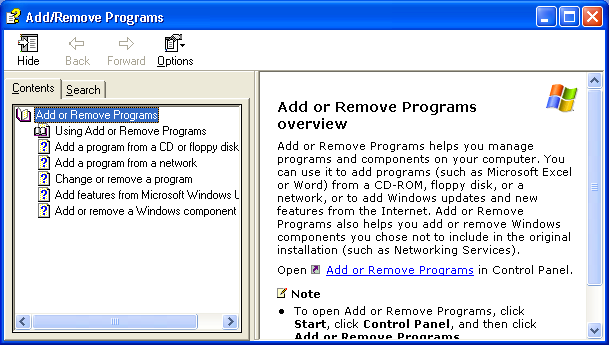
###### Compiled HTML Help File (.CHM)

###### Compiler

Project File (.HHP)

We need to create topic files (.**HTM** files) for each topic in our help system. (We could also add graphics.) These topics are organized by a Table of Contents file (.**HHC**) and Index file (.**HHK**). The Project File (.**HHP**) specifies the components of a help project. All of these files are ‘compiled’ to create the finished help file (.**CHM**). This file is the file that can be opened for viewing the finished help system.

The developed help system is similar to help systems used by all Windows applications. As an example, here is a help system (.CHM file) that explains how to add or remove programs from your computer:



The left frame is a hierarchical structure (**Contents**) of clickable topics. The right frame displays the currently selected topic information. Other tabs in the left frame allow a user to browse an **Index** (none shown here) and **Search** the help file. The file also features several navigation features and print options. The key point here is that this help system is familiar to your user. No new instruction is needed in how to use on-line help.

We will build an HTML help system similar to the one displayed above, but with minimal features. Learning how to build a full-featured help system would be a course in itself. In this chapter, we will learn how to create text-only topics, add a contents file, create a project file and see how to compile the entire package into a useful (if simple) help system.

**Creating a Help File**

We could create a help system using only text editors if we knew the required structure for the various files. We won’t take that approach. The on-line help system will be built using the **Microsoft** **HTML** **Help** **Workshop**. This is a free product from Microsoft that greatly simplifies the building of a help system. The workshop lets you build and organize all the files needed for building the help system. Once built, simple clicks allow compiling and viewing of the help system.

So, obviously, you need to have the workshop installed on your computer. The **HTML** **Help** **Workshop** can be downloaded from various Microsoft web sites. To find a download link, go to Microsoft’s web site (<http://www.microsoft.com>). Search on “**HTML** **Help**” – the search results should display a topic **HTML** **Downloads**. Select that link and you will be led to a place where you can do the download. Once downloaded, install the workshop as directed.

Creating a complete help file is a major task and sometimes takes as much time as creating the application itself! Because of this, we will only skim over the steps involved, generate a simple example, and provide guidance for further reference. There are five major steps involved in building your own help file:

1. Create your application and develop a hierarchical list of help system topics. This list would consist of headings and topics under specific headings.
2. Create the **topic** **files** (**HTM** extensions). Please make sure you spell and grammar check your topic files. If there are mistakes in your help system, your user base may not have much confidence in the care you showed in developing your application.
3. Create a **Table** **of** **Contents** (**HHC** extension).
4. Create the **Help Project File** (**HHP** extension).
5. Compile the help file to create your finished help system (**CHM** extension).

Step 1 is application-dependent. Here, we’ll look at how to use the HTML Help Workshop to complete the last four steps.

**Starting the HTML Help Workshop**

We will demonstrate the use of the **HTML** **Help** **Workshop** to build a very basic help system. The help file will have two headings. Each heading will have three sub-topics:

Heading 1

Topic 1

Topic 2

Topic 3

Heading 2

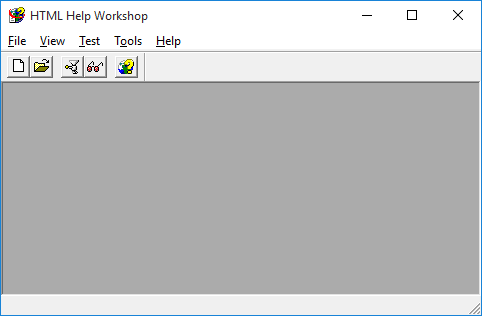
Topic 1

Topic 2

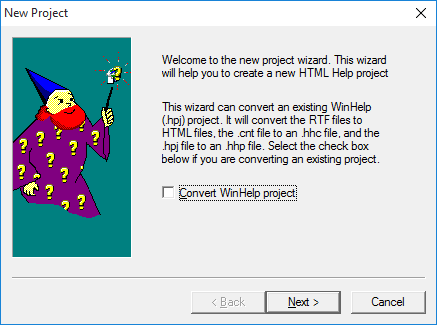
Topic 3

Though simple, the steps followed here can be used to build an adequate help system. All of the files created while building this help system can be found in the **VCSDB\Code\Class 5\Sample Help** folder.

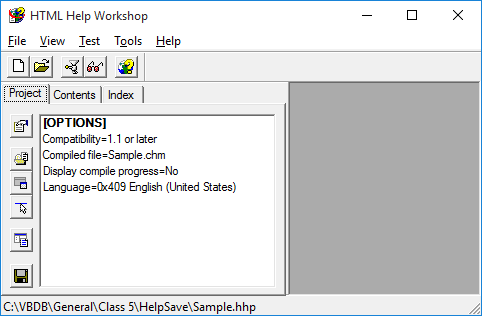
If properly installed, there will be an entry for the help workshop on your computer’s **Programs** menu. Click **Start**, then **Programs**. Select **HTML** **Help** **Workshop**, then **HTML** **Help** **Workshop** again. This dialog box should appear:



We want to start a new project. Select **New** under the **File** menu. In the selection box that appears, choose **Project** and click **OK**. A cute little **New** **Project** **Wizard** appears:



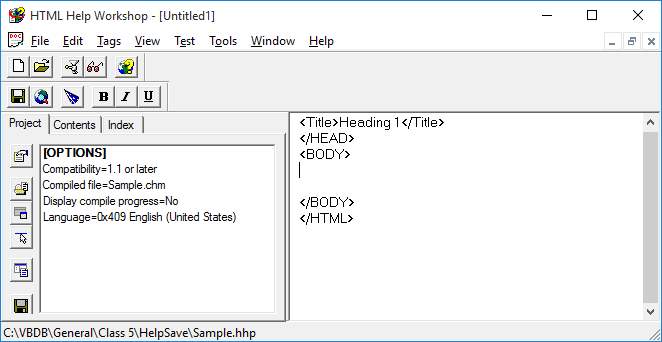
All we need to tell the wizard at this point is the name of our project file. Click **Next**. On the next screen, find (or create) the folder to hold your files (again, I used **VCSDB\Code\Class 5\Sample Help**) and use the project name **Sample**. Click **Next** two times (make no further selections), then **Finish**. The file **Sample.hhp** is created and you will see:



**Creating Topic Files**

At this point, we are ready to create our topic files. These are the files your user can view for help on topics listed in the contents region of the help system. We will have eight such files in our example (one for each of the two headings and one for each of the two sets of three topics).

Each file is individually created and saved as an HTM file. To create the first file (for Heading 1), choose **New** under the **File** menu. Select **HTML** **File** and click **OK**. Enter a name for the file (**Heading 1**) and click **OK**. A topic file HTML framework will appear:



The window on the right is where you type your topic information. The file has some HTML code there already. If you’ve never seen HTML before, don’t panic. We will make it easy. We are only concerned with what goes between the **<BODY>** and **</BODY>** ‘tags’. These tags mark the beginning and end of the text displayed when a user selects this particular heading topic.

Most HTML tags work in pairs. The first tag says start something, then the second tag with the slash preface **</>** says stop something. Hence, **<BODY>** says the body of the text starts here. The **</BODY>** tag says the body stops here. It’s really pretty easy to understand HTML.

It would help to know just a little more HTML to make your text have a nice appearance. To change the font, use the **FONT** tag:

**<FONT FACE=”**FontName**” SIZE=”**FontSize**”>**

where **FontName** is the name of the desired font and **FontSize** the desired size. Notice this is very similar to the **Font** constructor in Visual Basic. When you are done with one font and want to specify a new one, you must use a **</FONT>** tag before specifying the new font. To bold text, use the **<STRONG>** and **</STRONG>** tags. To delineate a paragraph in HTML, use the **<P>** and **</P>** tags. To cause a line break, use **<BR>.** There is no corresponding </BR> tag.

So, using our minimal HTML knowledge (if you know more, use it), we can create our first topic file. The HTML I used to create the first topic (**Heading1**) is:

**<BODY>**

**<STRONG>**

**This is Heading 1**

**</STRONG>**

**<P>**

**This is where I explain what the subtopics available under this heading are.**

**</P>**

**</BODY>**

This HTML will create this finished topic:

#### This is Heading 1

This is where I explain what the subtopics available under this heading are.

When done typing this first topic, choose **Close** **File** under the **File** menu. Select a file name (I used **Heading1.HTM**) to use and save the topic file. Of course, at any time, you can reopen, modify and resave any topic file.

You repeat the above process for every topic in your help system. That is, create a new file, type your topic and save it. You will have an HTM file for every topic in your help system. For our example, create seven more HTM files using whatever text and formatting you desire. The files I created are saved as**: Heading1.HTM, Topic11.HTM, Topic12.HTM, Topic13.HTM, Heading2.HTM, Topic21.HTM, Topic22.HTM, Topic23.HTM.**

Creating HTML topic files using the Help Workshop is a bit tedious. You need to use HTML tags and don’t really know what your topic file will look like until you’ve completed the help system. Using a **WYSIWYG** (what you see is what you get) editor is a better choice. Such editors allow you to create HTML files without knowing any HTML. You just type the file in a normal word processing-type environment, then save it in HTML format. There are several WYSIWYG HTML editors available. Check Internet download sites for options. Also, most word processors offer an option to save a document as an HTML file. I always use a WYSIWYG editor for topic files. I simply save each topic file in the same folder as my help system files, just as if I was using the built-in editor.

Next, we create a **Table of Contents** file. But, before leaving your topic files, make sure they are as complete and accurate as possible. And, again, please check for misspellings – nothing scares a user more than a poorly prepared help file. They quickly draw the conclusion that if the help system is not built with care, the application must also be sloppily built.

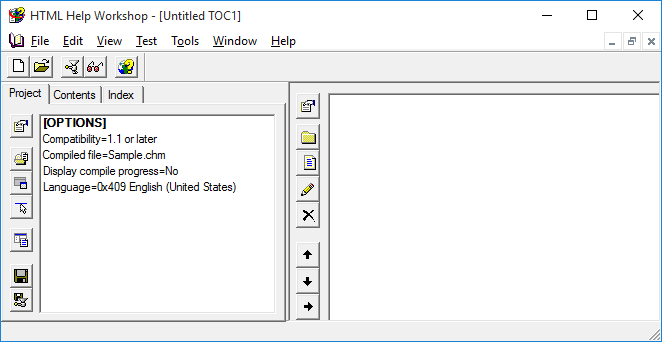
**Creating Table of Contents File**

The **Table of Contents** file specifies the hierarchy of topics to display when the help system’s **Contents** tab is selected. In the **HTML** **Help** **Workshop**, choose the **New** option under the **File** menu. Choose **Table of Contents**, then click **OK**. The following window appears:

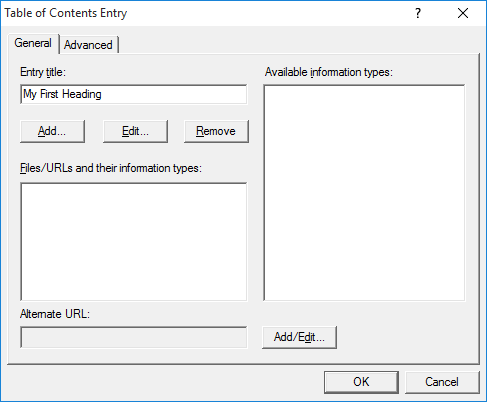
Insert a page

Insert a heading

Contents properties

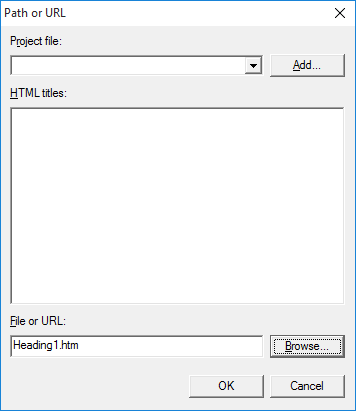


We want to add two headings with three topics under each. To insert a heading, in the right frame, click the toolbar button with a folder (**Insert a** **heading**). This window appears:



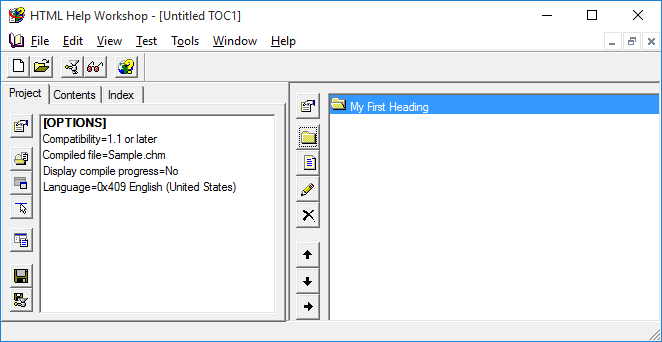
Type a title for the entry in **Entry** **title** (this is what will appear in the **Contents** – I used **My First Heading**).

You also need to link this topic to its topic file (HTM file). To do this, click **Add** and this appears:



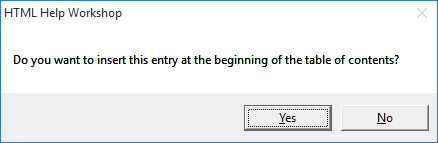
Click the Browse button and ‘point’ to the corresponding topic file (**Heading1.HTM** in this case). Click **OK** to close this window.

Click **OK** to close the Table of Contents entry window and you’ll now see:



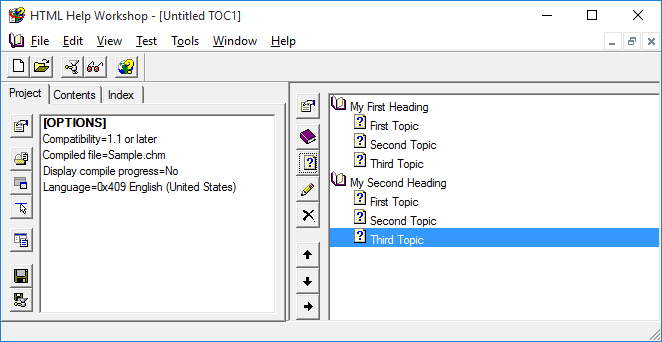
You’ve created your first entry in the Table of Contents. Notice the icon next to the heading is an ‘open folder.’ To change this to the more familiar ‘open book,’ click the top toolbar button (**Contents** **properties**). In the window that appears, remove the check mark next to ‘**Use Folders Instead of Books**,’ and click **OK**.

Now, we need to enter our first topic under this first heading. Click the toolbar button (**Insert a page**) under the heading button. This dialog will appear:



Answer **No** – we want the entry after the heading topic. At this point, you follow the same steps followed for the heading: enter a title and add a link to the topic file.

Add Table of Contents entries for all topic files in our little example. Use whatever titling information you choose. When you enter the second heading, it will be listed under the third topic in the first heading. To move it to the left (making it a major heading), right-click the heading and choose **Move** **Left**, the left arrow button on the toolbar). When done, I have this:



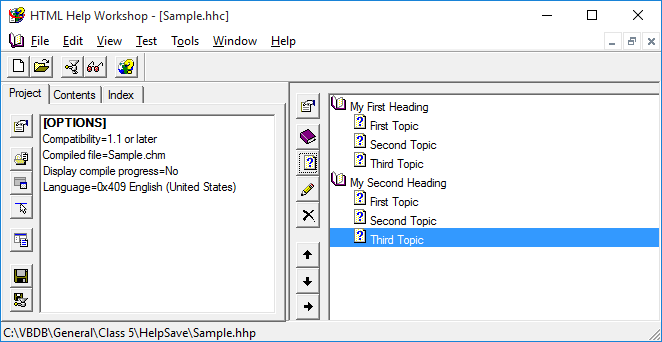
Save the contents file. Choose **Close File** under the **File** menu and assign a name of **Sample.HHC** to this contents file.

**Compiling the Help File**

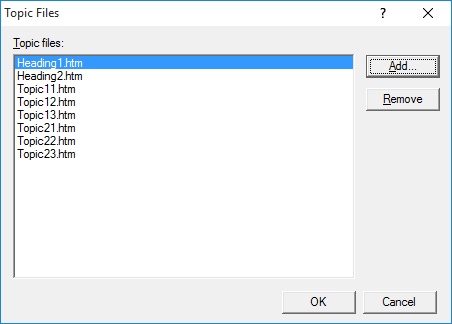
We’re almost ready to compile and create our help system. Before doing this, we need to add entries to the **Project** file. The project file at this point appears as:

Add/Remove topic files

Change project options

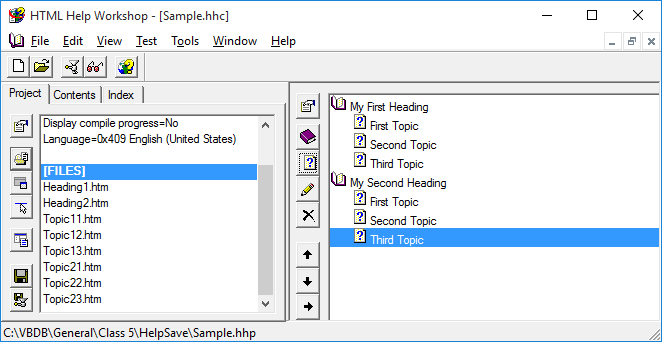


We first need to add our topic files. To do this, choose the **Add/remove topic files** toolbar button. In the window that appears, click **Add**, then select all topics files. You should see:

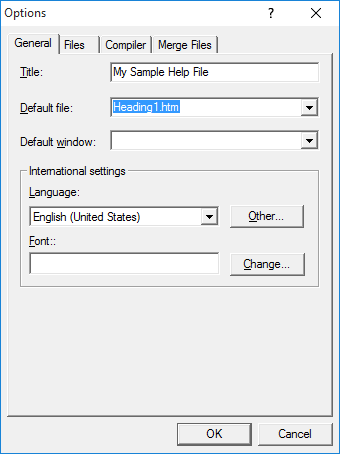


Click **OK.**

Now, the project file has the topic files added:



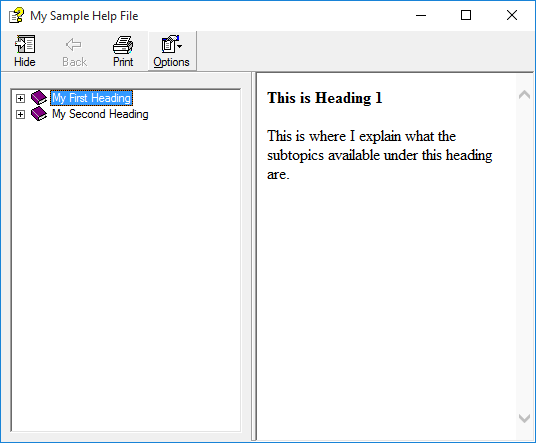
Now, we specify the **Table of Contents** file and set a few other properties. Click the **Change project options** toolbar button. Click the **General** tab and type a title for your help system (I used **My Sample Help File**) and specify the default file (**Heading1.htm**):



Click on the **Files** tab and select **Sample.hhc** as your contents file. Click **OK** to complete specification of your project file. At this point, save all files by choosing **Save** **Project** under the **File** menu.

We can now compile our project into the finished product – a complete HTML help system. To do this, click the **Compile HTML file** button (resembles a meat grinder) on the workshop toolbar. **Browse** so your project file (**Sample.hhp**) is selected. Choose **Compile** in the resulting window and things start ‘grinding.’ If no errors are reported, you should now have a **CHM** file in your directory. If errors did occur, you need to fix any reported problems.

At long last, we can view our finished product. Click on the **View** **compiled** **file** button (a pair of sunglasses) on the workshop toolbar. **Browse** so your help file (**Sample.chm**) is selected. Choose **View**, and this will appear (I’ve expanded the headings to show the topics):



Click to see the various topics and headings displayed.

After all this work, you still only have a simple help file, nothing that rivals those seen in most applications. But, it is a very adequate help system. To improve your help system, you need to add more features. Investigate the HTML Help Workshop for information on tasks such as adding an index file, using context-sensitive help, adding search capabilities and adding graphics to the help system.

**HelpProvider Control**



Once we have a completed HTML help system, we need to connect our Visual C# application to the help file. You need to decide how you want your application to interact with the help system. We will demonstrate a simple approach. We will have the help system appear when the user presses **<F1>** or clicks some control related to obtaining help (menu item, button control). The Visual C# **HelpProvider** control provides this connection.

HelpProvider **Properties**:

**Name** Gets or sets the name of the help provider control (three letter prefix for label name is **hlp**).

**HelpNamespace** Complete path to compiled help file (**CHM** file)

The **HelpNamespace** property is usually established at run-time. The help file is often installed in the application directory (**Bin\Debug** folder). If this is the case, we can use the **Application.StartupPath** parameter to establish **HelpNamespace**. You also must include the help file in any deployment package you build for your application.

To have the help file appear when a user presses **<F1>,** we set the **HelpNavigator** property of the application form to **TableofContents**. With this setting, the help file will appear displaying the **Table of Contents**, set to the default form.

To have the help file appear in a **Click** event, we use the **ShowHelp** method of the **Help** object. The Visual C# **Help** object allows us to display HTML help files. To replicate the **<F1>** functionality above, we use the syntax:

Help.ShowHelp(this, HelpProvider.HelpNamespace);

This line of code will display the specified help file.

Typical use of **HelpProvider** control:

* Set the **Name** property.
* Set **HelpNameSpace** property in code (file is usually in **Bin** folder of application).
* Set **HelpNavigator** property for form to **TableofContents**.
* Write code for events meant to display the help file (use **Help.ShowHelp**).

The steps above provide minimal, but sufficient, access to an HTML help system. If you need more functionality (context-sensitive help, help on individual controls, pop-up help, adding help to dialog boxes), consult the Visual C# documentation on the **Help** **Provider** control.

**Example 5-8**

**Authors Table Input Form (On-Line Help)**

We will build a simple help system for our Authors Table Input Form and attach it to our application. Refer back to the notes to complete each step listed here.

1. Prepare a single topic file (saved as **authors.htm** in the **VCSDB\Code\Class 5\Example 5-8\HelpFile** folder):

**Authors Input Form**

Available options for managing Authors database table:

**Add New Record**

Click the **Add** **New** button to add a record to the Authors database. Type in the Author Name (required), then Year Born (optional). Click **Save** to save the new record; click **Cancel** to ignore the new record.

**Edit Record**

Click the **Edit** button to edit the displayed record. Make any needed changes. The Author Name is required and the Year Born is optional. Click **Save** to save the changes; click **Cancel** to ignore the changes.

**Delete Record**

Click the **Delete** button to delete the displayed record.

**Exit Program**

Click the **Done** button to quit the application.

1. Using HTML Help Workshop, build a help project file, adding the topic file (no contents file) (**authors.hhp** in the **VCSDB\Code\Class 5\Example 5-8\HelpFile** folder).
2. Using HTML Help Workshop, compile and build the project file (**authors.chm** in the **VCSDB\Code\Class 5\Example 5-8\HelpFile** folder).
3. Load Example 5-7 completed earlier. We will modify this example to include our help system. Copy **authors.chm** to the application’s **Bin\Debug** folder (you may have to create the folder first). Add a help provider control to the project – name it **hlpAuthors**. Set the **HelpNavigator** property of **frmAuthors** to **TableofContents**.
4. Add the shaded code near the top of the **frmAuthors Load** method. This code points to the help file in the project’s application folder:

**private void FrmAuthors\_Load(object sender, EventArgs e)**

**{**

**try**

**{**

**// point to help file**

**hlpAuthors.HelpNamespace = Application.StartupPath + "\\authors.chm";**

**// connect to books database**

**booksConnection = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source = c:\\VCSDB\\Working\\BooksDB.accdb");**

**booksConnection.Open();**

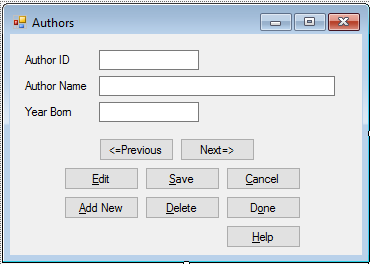
**// establish command object**

**.**

**.**

**}**

1. Add a button to the form. Assign a **Text** of **&Help** and a **Name** of **btnHelp**. The form now looks like this:



1. Use this code in the **btnHelp Click** event to display the help file:

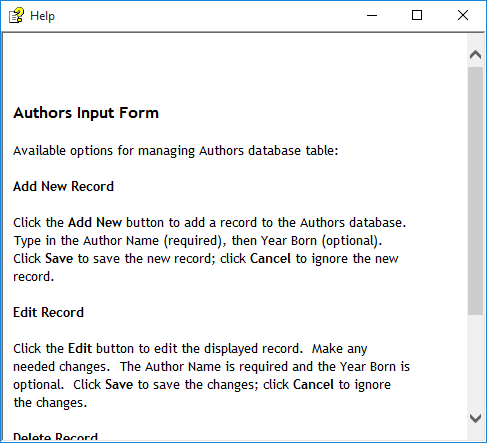
**private void BtnHelp\_Click(object sender, EventArgs e)**

**{**

**Help.ShowHelp(this, hlpAuthors.HelpNamespace);**

**}**

1. Save (saved in the **Example 5-8** folder in **VCSDB\Code\Class 5** folder) and run the application. Press <**F1**> or click **Help**. With either, you should see the help file:



**Example 5-8**

**Using SQL Server Databases**

Make the same changes to the SQL Server version of the project. The SQL Server version is saved in the **Example 5-8 SQL** folder in **VCSDB\Code\Class 5** folder.

**Application Testing**

Our discussion of Visual C# interface design has, for now, come to an end. And, we have a fairly complete interface for the books database Authors table. We, obviously, still need the remainder of the code that goes behind the buttons. We’ll do that in the next chapter.

Once you have completed an application, you need to test it to make sure it performs as expected. If you are careful in building your application, no big surprises should appear in this final testing. In fact, the Visual C# environment helps achieve this goal. The event-driven nature of Visual C# makes it easy to build an application in stages, testing and debugging each stage as it is built. In other words, you don’t have to have a complete application before testing can begin. We have done this with the Authors table example.

The event-driven nature of Visual C# also makes it easy to modify an application. We will see in Chapter 6, as we modify the books database example, that we have made some omissions and errors in our design. But these omissions and errors will be easily corrected using the Visual C# environment. These corrections will give you additional insight into application building and testing process.

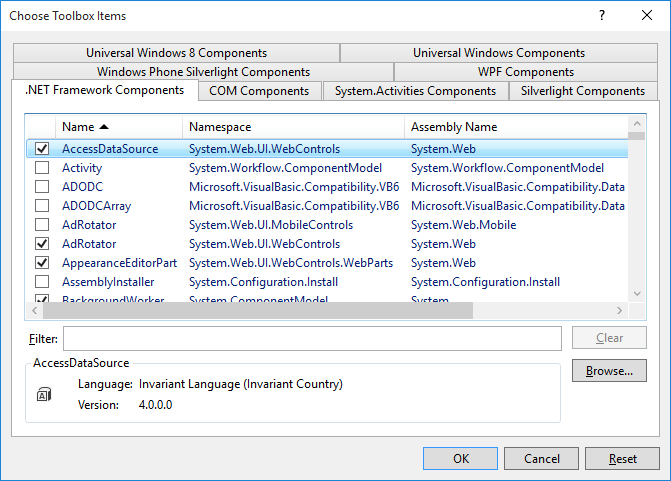
Let others (particularly potential users) try your application and see if its use is as obvious as you planned it to be. Are the inputs and outputs of the project appropriate? Is application state clear? Implement and retest any necessary changes, based on user feedback. And, keep track of all feedback after you ‘release’ your application. This information can be used in future updates of your product.

Before leaving this chapter, let’s look at some other Visual C# controls that you might like to use in your database interface arsenal.

**Other Controls**

In addition to the standard Visual C# controls discussed earlier in this chapter, there are many **other** **controls** that can be used to build a database interface. If not in the toolbox, these controls will have to be added to the Visual C# toolbox before they can be used.

To load a control, make sure the toolbox is visible in the development environment. Choose **Choose Toolbox Items** from the Visual C# **Tools** menu. Select **.NET Framework Components** in the resulting dialog:



To add a control or controls, select the check box next to the desired selection(s). When done, choose **OK** and the selected control(s) will now appear in the toolbox. The Visual C# on-line help system can provide details for usage.

Here, we look at several other controls (some data bound, some not) and how they can be used with a Visual C# interface.

**MaskedTextBox Control**



The **masked text box control** is a data bound control used to prompt users for data input using a mask pattern. It works like a text box, but the mask allows you to specify exactly the desired input format. In a **database**, this control could be used to prompt for a date, a time, number, or currency value. Or it could be used to prompt for something that follows a pattern, like a phone number or social security number. Use of this control can eliminate many of the entry validation problems mentioned earlier in the chapter. If needed, to load this control into the toolbox, select **MaskedTextBox** from the **.NET Framework Components** dialog box.

Masked Text Box **Properties**:

**Mask** Determines the type of information that is input into the control. It uses characters to define the type of input. Check on-line help for mask formatting.

**PromptChar** Character used for missing information.

**Text** Contains data entered into the control (including all prompt characters of the input mask). This is the property bound to the database.

**TextMaskFormat** Used to indicate if the **Text** property includes literal and prompts used.

Masked Text Box **Events**:

**Leave** Event called when the user leaves the control.

**LostFocus** Event called when control loses focus.

**MaskInputRejected** Event called when the data being entered by the user does not match the input mask.

Typical use of **MaskedTextBox** control:

* Set the **Name** property. Initialize **Text** property to desired string. Set **Mask** property.
* In code, give **Focus** to control when needed. Read **Text** property when **Leave** event occurs.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

This control features built-in input validation to lessen your tasks as a programmer. We will use the masked edit control in some of our example applications in later chapters.

**NumericUpDown Control**



The **NumericUpDown** control is used to obtain a numeric input. It looks like a text box control with two small arrows. Clicking the arrows changes the displayed value, which ranges from a specified minimum to a specified maximum. The user can even type in a value, if desired. Such controls are useful for supplying a date in a month or are used as volume controls in some Windows multimedia applications.

NumericUpDown **Properties**:

**Name** Gets or sets the name of the numeric updown (three letter prefix for numeric updown name is **nud**).

**BackColor** Get or sets the numeric updown background color.

**BorderStyle** Gets or sets the border style for the updown control.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text or graphics.

**Increment** Gets or sets the value to increment or decrement the updown control when the up or down buttons are clicked.

**Maximum** Gets or sets the maximum value for the updown control.

**Minimum** Gets or sets the minimum value for the updown control.

**ReadOnly** Gets or sets a value indicating whether the text may be changed by the use of the up or down buttons only.

**TextAlign** Gets or sets the alignment of text in the numeric updown.

**Value** Gets or sets the value assigned to the updown control.

NumericUpDown **Methods**:

**DownButton** Decrements the value of the updown control.

**UpButton** Increments the value of the updown control.

NumericUpDown **Events**:

**Leave** Occurs when the updown control loses focus.

**ValueChanged** Occurs when the [Value](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformsnumericupdownclassvaluetopic.htm) property has been changed in some way.

The Value property can be changed by clicking either of the arrows or, optionally by typing a value. If using the arrows, the value will always lie between Minimum and Maximum. If the user can type in a value, you have no control over what value is typed. However, once the control loses focus, the typed value will be compared to Minimum and Maximum and any adjustments made. Hence, if you allow typed values, only check the Value property in the Leave event.

Typical use of **NumericUpDown** control:

* Set the **Name, Minimum** and **Maximum** properties. Initialize **Value** property. Decide on value for **ReadOnly**.
* Monitor **ValueChanged** (or **Leave**) event for changes in Value.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

The NumericUpDown control is a ‘point-and-click’ type control that can be used in place of a user’s typed input. We will use the NumericUpDown control in some of our example applications in later chapters.

**TabControl Control**



The **TabControl** control provides an easy way to present several dialogs or screens of information on a single form. This is the same interface seen in many commercial Windows applications. The tab control provides a group of tabs, each of which acts as a container (works just like a group box or panel) for other controls. In particular, groups of radio buttons within a tab ‘page’ operate as an independent group. Only one tab can be active at a time. Using this control is easy. Just build each tab container as a separate group: add controls, set properties, and write code like you do for any application. Navigation from one tab to the next is simple: just click on the corresponding tab.

TabControl **Properties**:

**Name** Gets or sets the name of the tab control (three letter prefix for control name is **tab**).

**BackColor** Get or sets the tab control background color.

**BorderStyle** Gets or sets the border style for the tab control.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text or graphics.

**ItemSize** Size structure determining tab size.

**SelectedIndex** Gets or sets the currently displayed tab index.

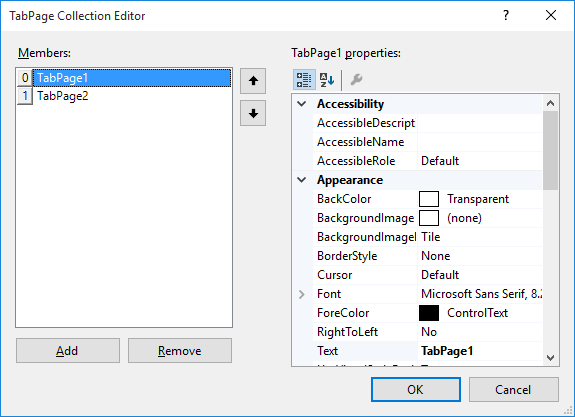
**SizeMode** Determines how tabs are sized.

**TabPages** Collection describing each tab page.

TabControl **Events**:

**SelectedIndexChanged** Occurs when the **SelectedIndex** property changes.

The most important property for the tab control is **TabPages**. It is used to design each tab (known as a **TabPage**). Choosing the **TabPages** property in the Properties window and clicking the ellipsis that appears will display the **TabPage** **Collection** **Editor**. With this editor, you can add, delete, insert and move tab pages. To add a tab page, click the **Add** button. A name and index will be assigned to a tab. There are two tabs added initially so the editor appears like this:



Add as many tab pages as you like. The tab page ‘array’ is zero-based; hence, if you have N tabs, the first is index 0, the last index N – 1. You can change any property you desire in the **Properties** area.

TabPage **Properties**:

**Name** Gets or sets the name of the tab page (three letter prefix for control name is **tab**).

**BackColor** Get or sets the tab page background color.

**BorderStyle** Gets or sets the border style for the tab page.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text or graphics.

**Text** Titling information appearing on tab.

When done, click **OK** to leave the TabPage Collection Editor.

The next step is to add controls to each ‘page’ of the tab control. This is straightforward. Simply display the desired tab page by clicking on the tab. Then place controls on the tab page, treating the page like a group box or panel control. Make sure your controls become ‘attached’ to the tab page. You can still place controls on the form that are not associated with any tab. As the programmer, you need to know which tab is active (**SelectedIndex** property). And, you need to keep track of which controls are available with each tab page.

Typical use of **TabControl** control:

* Set the **Name** property and size appropriately.
* Establish each tab page using the **TabPage** **Collection** **Editor**.
* Add controls to tabs and form.
* Write code for the various events associated with controls on the tab control and form.

The tab control is becoming a very popular control in Windows applications. It allows you to put a lot of ‘input power’ into a single form - minimizing the need for multi-form applications. We will use the tab control in a Weather Monitor example in Chapter 10.

**ToolStrip (Toolbar) Control**



Almost all Windows applications these days use toolbars. A toolbar provides quick access to the most frequently used menu commands in an application. In a database application, it could be used to add, delete, or edit records. It could be used to access database reports or obtain different database views. The **ToolStrip** control (also referred to as the **Toolbar** control) is a mini-application in itself. It provides everything you need to design and implement a toolbar into your application. Possible uses for this control include: provide a consistent interface between applications with matching toolbars, place commonly used functions in an easily-accessed space and provide an intuitive, graphical interface for your application.

ToolStrip **Properties**:

**Name** Gets or sets the name of the toolstrip (toolbar) control (three letter prefix for label name is **tlb**).

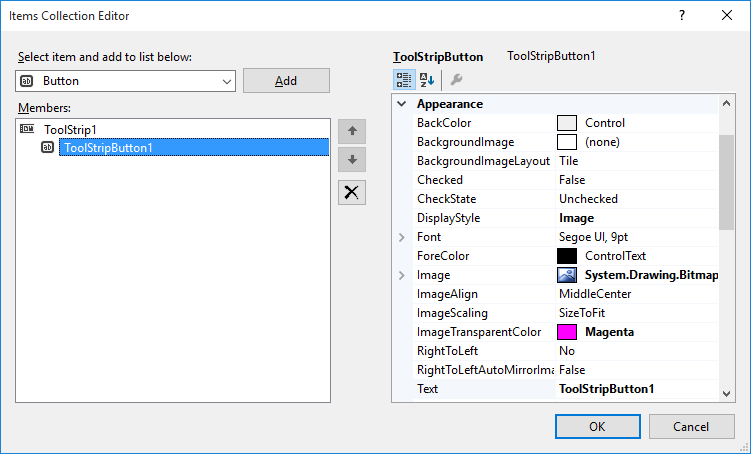
**BackColor** Background color of toolstrip.

**Items** Gets the collection of controls assigned to the toolstrip control.

**LayoutStyle** Establishes whether toolbar is vertical or horizontal.

**Dock** Establishes location of toolbar on form.

The primary property of concern is the **Items** collection. This establishes each item in the toolbar. Choosing the **Items** property in the Properties window and clicking the ellipsis that appears will display the **Items** **Collection** **Editor**. With this editor, you can add, delete, insert and move items. We will look at adding just two types of items: **ToolStripButton** and **ToolStripSeparator** (used to separate tool bar buttons). To add a button, make sure **ToolStripButton** appears in the drop-down box and click the **Add** button. A name will be assigned to a button. After adding one button, the editor will look like this:



Add as many buttons as you like. You can change any property you desire in the **Properties** area.

ToolStripButton **Properties**:

**Name** Gets or sets the name of the button (three letter prefix for control name is **tlb**).

**DisplayStyle** Sets whether image, text or both are displayed on button.

**Image** Image to display on button.

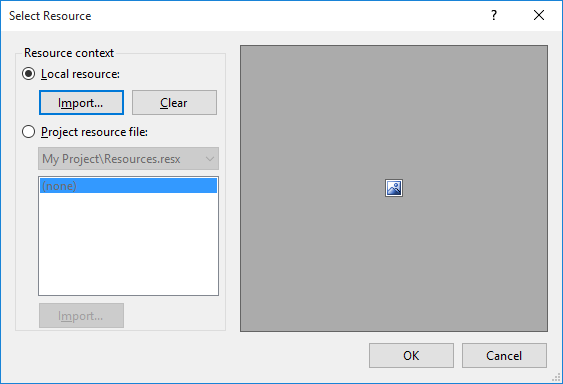
**Text** Caption information on the button, often blank.

**TextImageRelation** Where text appears relative to image.

**ToolTipText** Text to display in button tool tip.

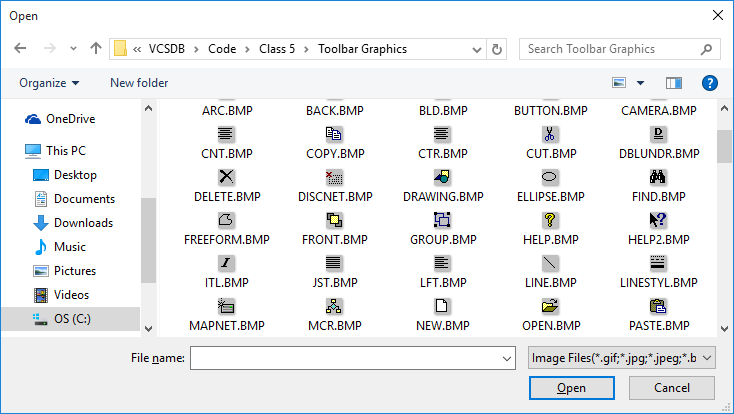
To add a separator, make sure **ToolStripSeparator** appears in drop-down box and click **Add**. When done editing buttons, click **OK** to leave the Items Collection Editor.

Setting the **Image** property requires a few steps (a process similar to that used for the picture box control). First, click the ellipsis next to the **Image** property in the property window. This **Select Resource** window will appear:



The images will be a local resource, so select the **Local resource** radio button and click the **Import** button.

An **Open** window will display graphics files (if you want to see an **ico** file, you must change **Files of type** to **All Files**). In the **VCSDB\Code\Class 5** folder is a folder named **Toolbar Graphics**. In this folder, there are many bitmap files for toolbar use:



Select the desired file and click **Open**. Once an image is selected, click **OK** in the **Select** **Resource** window. It will be assigned to the **Image** property

After setting up the toolbar, you need to write code for the **Click** event for each toolbar button. This event is the same **Click** event we encounter for button controls.

Typical use of **ToolStrip** control:

* Set the **Name** property and desired location.
* Decide on image, text, and tooltip text for each button.
* Establish each button/separator using the **Items** **Collection** **Editor**.
* Write code for the each toolbar button’s **Click** event.

The toolbar is a very powerful and professional tool. And, it’s easy to implement and use. Try to use it whenever it fits the design of your interface.

**ListBox Control**



A **ListBox** control displays a list of items (with as many items as you like) from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added. Both single item and multiple item selections are supported. For database applications, you can display multiple rows of data (a given field) in the same control (see the **DataSource** and **DisplayMember** properties).

ListBox **Properties**:

**Name** Gets or sets the name of the list box (three letter prefix for list box name is **lst**).

**BackColor** Get or sets the list box background color.

**DataSource** Data table to bind control to.

**DisplayMember** Field from data source to display.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text.

**Items** Gets the Items object of the list box.

**SelectedIndex** Gets or sets the zero-based index of the currently selected item in a list box.

**SelectedIndices** Zero-based array of indices of all currently selected items in the list box.

**SelectedItem** Gets or sets the currently selected item in the list box.

**SelectedItems** SelectedItems object of the list box.

**SelectedValue** The value provided by/to the ValueMember for data binding.

**SelectionMode** Gets or sets the method in which items are selected in list box (allows single or multiple selections).

**Sorted** Gets or sets a value indicating whether the items in list box are sorted alphabetically.

**Text** Text of currently selected item in list box.

**TopIndex** Gets or sets the index of the first visible item in list box.

**ValueMember** The data source field corresponding to same record shown by DisplayMember.

ListBox **Methods**:

**ClearSelected** Unselects all items in the list box.

**FindString** Finds the first item in the list box that starts with the specified string.

**GetSelected** Returns a value indicating whether the specified item is selected.

**SetSelected** Selects or clears the selection for the specified item in a list box.

ListBox **Events**:

**SelectedIndexChanged** Occurs when the [SelectedIndex](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformslistboxclassselectedindextopic.htm) property has changed.

One use for the data bound list control is to fill the list (**DisplayMember**) from a database table (**DataSource**), then allow selections. This allows us to list all values of a particular field in a database table. The selections can be used by any control on a form, whether it is data bound or not. For the **BooksDB.accdb** database **Authors** table we’ve been using, if we display the **Author** field, a list box will show:



More complex data binding (using the **SelectedValue** and **ValueMember** properties) is possible. We will look at this complex binding as we develop a management system for the books database in later chapters (see Example 6-9).

Some further discussion is need to use the list box **Items** object, **SelectedItems** object and **SelectionMode** property. The Items object has its own properties to specify the items in the list box. It also has its own methods for adding and deleting items in the list box. The **Items** object is a zero-based array of the items in the list and **Count** (a property of **Items**) is the number of items in the list. Hence, the first item in a list box named **lstExample** is:

**lstExample.Items[0]**

The last item in the list is:

**lstExample.Items[lstExample.Items.Count – 1]**

The minus one is needed because of the zero-based array.

To add an item to a list box, use the **Add** method, to delete an item, use the **Remove** or **RemoveAt** method and to clear a list box use the **Clear** method. For our example list box, the respective commands are:

Add Item: **lstExample.Items.Add(**ItemToAdd**)**

Delete Item: **lstExample.Items.Remove(**ItemToRemove**)**

**lstExample.Items.RemoveAt(**IndexofItemToRemove**)**

Clear list box: **lstExample.Items.Clear**

List boxes normally list string data types, though other types are possible. Note, when removing items, that indices for subsequent items in the list change following a removal.

In a similar fashion, the **SelectedItems** object has its own properties to specify the currently selected items in the list box Of particular use is **Count** which tells you how many items are selected. This value, in conjunction with the SelectedIndices array, identifies the set of selected items.

The **SelectionMode** property specifies whether you want single item selection or multiple selections. When the property is **SelectionMode.One**, you can select only one item (works like a group of radio buttons). When the SelectionMode property is set to **SelectionMode.MultiExtended**, pressing <Shift> and clicking the mouse or pressing <Shift>and one of the arrow keys extends the selection from the previously selected item to the current item. Pressing <Ctrl>and clicking the mouse selects or deselects an item in the list. When the property is set to **SelectionMode.MultiSimple**, a mouse click or pressing the spacebar selects or deselects an item in the list.

Typical use of **ListBox** control:

* Set **Name** property, **SelectionMode** property and populate **Items** object (usually in **Form\_Load** method).
* If using with database, set **DataSource** property to desired data table and **DisplayMember** property to corresponding data table field.
* Monitor **SelectedIndexChanged** event for individual selections.
* Use **SelectedIndex** and **SelectIndices** properties to determine selected items.

**ComboBox Control**



The **combo box control** is nearly identical to the list box, hence we won’t look at a separate set of properties or another example. A primary difference between the two controls is the way data is displayed – the combo control has a list box portion and a text box portion that displays the selected item. And, with the combo control, the user is (optionally) given the opportunity to type in a choice not in the list box.

As mentioned, data display is different with the combo control. Display is established by the **DropDownStyle** property:

#### Style Description

DropDown Drop-down list box, user can change selection

Simple Displayed list box, user can change selection

DropDownList Drop-down list box, user cannot change selection

When using the **Simple** style, make sure you sufficiently size the control (so the list box portion appears) when it is placed on the form.

Typical use of **ComboBox** control:

* Set **Name** property, **DropDownStyle** property and populate **Items** object (usually in **Form\_Load** method).
* If using with database, set **DataSource** property to desired data table and **DisplayMember** property to corresponding data table field.
* Monitor **SelectedIndexChanged** event for individual selections.
* Use **SelectedIndex** or **Text** properties to determine selected item.

When should you use the combo control instead of the list box control? The combo control is an excellent data entry control. Its advantage over the list box is that it provides experienced users the ability to type in values they know are correct, speeding up the data entry process. The list box control does not allow any typing. It is also a good control when you are short on form space. Using the **DropDownList** style replicates the functionality of the list box control without needing space for the list box.

**DataGridView Control**



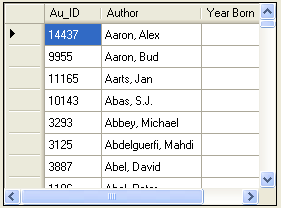
The **data grid view control** tool is one of the most useful data bound controls. It can display an entire database table. The table can then be edited as desired. Recall we used this control in our SQL Tester program in Chapter 4.

The data bound grid control is in a class by itself, when considering its capabilities. It is essentially a separate, highly functional program. It has one primary property:

**DataSource** Name of the data table to display.

The data grid view control is a collection of **DataColumn** objects, corresponding to fields in the table, and **DataRow** objects, corresponding to records. Cells can be accessed and edited via mouse operations or programmatically.

For the books database, if we display the **Authors** table (**DataSource** property) we’ve been using in a data grid view, we will see:



Typical use of **DataGridView** control:

* Set **Name** property.
* Set **DataSource** property to desired data table.
* Add any desired editing features.

You are encouraged to further study the data grid view control (properties, events, methods) as you progress in your database studies. We will use it in applications studied in later chapters.

**MonthCalendar Control**



The **MonthCalendar** control allows a user to select a date. It is a very easy to use interface – just point and click. This control is useful for ordering information, making reservations or choosing the current date. It can be used to select a single date or a range of dates.

MonthCalendar **Properties**:

**Name** Gets or sets the name of the month calendar (three letter prefix for label name is **cal**).

**BackColor** Get or sets the month calendar background color.

**CalendarDimensions** Gets or sets the number of columns and rows of months displayed.

**FirstDayOfWeek** Gets or sets the first day of the week as displayed in the month calendar.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text or graphics.

**MaxDate** Gets or sets the maximum allowable date.

**MaxSelectionCount** The maximum number of days that can be selected in a month calendar control.

**MinDate** Gets or sets the minimum allowable date.

**SelectionEnd** Gets or sets the end date of the selected range of dates.

**SelectionRange** Retrieves the selected range of dates for a month calendar control.

**SelectionStart** Gets or sets the start date of the selected range of dates.

**ShowToday** Gets or sets a value indicating whether the date represent by the [TodayDate](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformsmonthcalendarclasstodaydatetopic.htm) property is shown at the bottom of the control.

**ShowTodayCircle** Gets or sets a value indicating whether today's date is circled.

**TodayDate** Gets or sets the value that is used by MonthCalendar as today's date.

MonthCalendar **Methods:**

**SetDate** Sets date as the current selected date.

MonthCalendar **Events**:

**DateChanged** Occurs when the date in the MonthCalendar changes.

**DateSelected** Occurs when a date is selected.

Typical use of **MonthCalendar** control:

* Set the **Name** property. Set **MaxSelectionCount** (set to 1 if just picking a single date).
* Monitor **DateChanged** and/or **DateSelected** events to determine date value(s). Values are between **SelectionStart** and **SelectionEnd** properties.

**DateTimePicker Control**



The **DateTimePicker** control works like the MonthCalendar control with a different interface and formatting options. It allows the user to select a single date. The selected date appears in a combo box. The calendar portion is available as a ‘drop down.’ This control can also be used to select a time; we won’t look at that option.

DateTimePicker **Properties**:

**Name** Gets or sets the name of the date/time picker control (three letter prefix for label name is **dtp**).

**BackColor** Get or sets the control background color.

**Font** Gets or sets font name, style, size.

**ForeColor** Gets or sets color of text or graphics.

**Format** Gets or sets the format of the date displayed in the control.

**MaxDate** Gets or sets the maximum allowable date.

**MinDate** Gets or sets the minimum allowable date.

**Value** Gets or sets the date value assigned to the control.

DateTimePicker **Events**:

**ValueChanged** Occurs when the **Value** property changes.

Typical use of **DateTimePicker** control:

* Set the **Name** and Format properties.
* When needed, read **Value** property for selected date.

**OpenFileDialog Control**



In all examples studied in this course, the database name has been assumed to be known at design time (before running the application). There will be times when this is not true. For example, say a schoolteacher uses a database application to keep track of grades. There will database files for each class. When the teacher starts the application, he or she needs to specify which particular database file is being accessed and that file needs to be opened at run-time. Not only do we need the capability to open a user specified file, but we also need to be able to save database files with user specified names.

What we need from the user, whether opening or saving files is a complete path to the filename of interest. We could provide a text box and ask the user to type the path, but that’s only asking for trouble. We would have to validate existence of drives, directories, and files! Fortunately, we can use the Windows standard interface for working with files. Visual C# provides this interface through the **common dialog controls**. These controls display the same interface you see when opening or saving a file in any Windows application. Such an interface is familiar to any Windows user and gives your application a professional look. And, some context-sensitive help is available while the interface is displayed. Let’s look first at the **open** **file** **dialog** **control**.

OpenFileDialog **Properties**:

**Name** Gets or sets the name of the open file dialog (I usually name this control **dlgOpen**).

**AddExtension** Gets or sets a value indicating whether the dialog box automatically adds an extension to a file name if the user omits the extension.

**CheckFileExists** Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a file name that does not exist.

**CheckPathExists** Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a path that does not exist.

**DefaultExt** Gets or sets the default file extension.

**FileName** Gets or sets a string containing the file name selected in the file dialog box.

**Filter** Gets or sets the current file name filter string, which determines the choices that appear in "Files of type" box.

**FilterIndex** Gets or sets the index of the filter currently selected in the file dialog box.

**InitialDirectory** Gets or sets the initial directory displayed by the file dialog box.

**Title** Gets or sets the file dialog box title.

OpenFileDialog **Methods**:

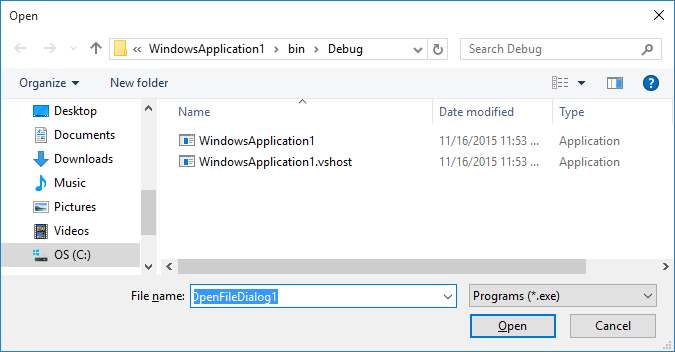
**ShowDialog** Displays the dialog box. Returned value indicates which button was clicked by user (**OK** or **Cancel**).

To use the **OpenFileDialog** control, we add it to our application the same as any control. Since the OpenFileDialog control has no immediate user interface (you control when it appears), the control does not appear on the form at design time. Such Visual C# appear in a ‘tray’ below the form in the IDE Design window. Once added, we set a few properties. Then, we write code to make the dialog box appear when desired. The user then makes selections and closes the dialog box. At this point, we use the provided information for our tasks.

The **ShowDialog** method is used to display the **OpenFileDialog** control. For a control named **dlgOpen**, the appropriate code is:

**dlgOpen.ShowDialog();**

And the displayed dialog box is similar to this:



The user selects a file using the dialog control (or types a name in the **File** **name** box). The file type is selected form the **Files** **of** **type** box (values here set with the **Filter** property). Once selected, the **Open** button is clicked. **Cancel** can be clicked to cancel the open operation. The ShowDialog method returns the clicked button. It returns **DialogResult.OK** if Open is clicked and returns **DialogResult.Cancel** if Cancel is clicked. The nice thing about this control is that it can validate the file name before it is returned to the application. The **FileName** property contains the complete path to the selected file.

Typical use of **OpenFileDialog** control:

* Set the **Name, Filter**, and **Title** properties**.**
* Use **ShowDialog** method to display dialog box.
* Read **FileName** property to determine selected file

**SaveFileDialog Control**



To obtain a file name for saving we use the **SaveFileDialog** control. This control insures that any path selected for saving a file exists and that if an existing file is selected, the user has agreed to overwriting that file.

SaveFileDialog **Properties**:

**Name** Gets or sets the name of the save file dialog (I usually name this control **dlgSave**).

**AddExtension** Indicates whether the dialog box automatically adds an extension to a file name if the user omits the extension.

**CheckFileExists** Indicates whether the whether the dialog box displays a warning if the user specifies a file name that does not exist. Useful if you want the user to save to an existing file.

**CheckPathExists** Indicates whether the dialog box displays a warning if the user specifies a path that does not exist.

**CreatePrompt** Indicates whether the dialog box prompts the user for permission to create a file if the user specifies a file that does not exist.

**DefaultExt** Gets or sets the default file extension.

**FileName** Gets or sets a string containing the file name selected in the file dialog box.

**Filter** Gets or sets the current file name filter string, which determines the choices that appear in "Files of type" box.

**FilterIndex** Gets or sets the index of the filter currently selected in the file dialog box.

**InitialDirectory** Gets or sets the initial directory displayed by the file dialog box.

**OverwritePrompt** Indicates whether the dialog box displays a warning if the user specifies a file name that already exists. Default value is True.

**Title** Gets or sets the file dialog box title.

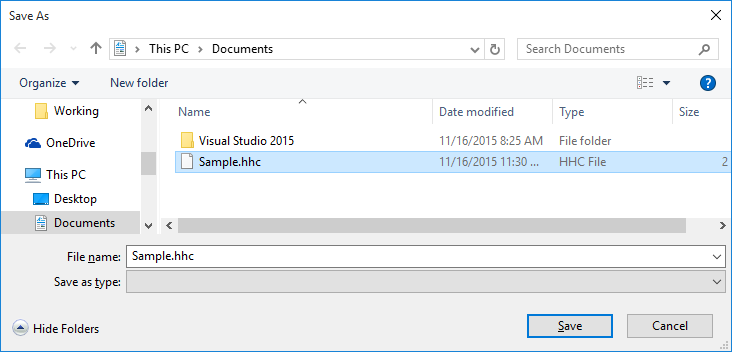
SaveFileDialog **Methods**:

**ShowDialog** Displays the dialog box. Returned value indicates which button was clicked by user (**OK** or **Cancel**).

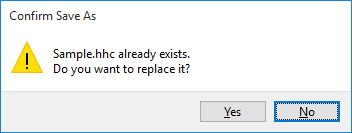
The **SaveFileDialog** control will appear in the tray area of the design window. The **ShowDialog** method is used to display the **SaveFileDialog** control. For a control named **dlgSave**, the appropriate code is:

**dlgSave.ShowDialog();**

And the displayed dialog box is similar to this:



The user types a name in the File name box (or selects a file using the dialog control). The file type is selected form the **Files** **of** **type** box (values here set with the **Filter** property). Once selected, the **Save** button is clicked. **Cancel** can be clicked to cancel the save operation. If the user selects an existing file and clicks **Save**, the following dialog will appear:



This is the aforementioned protection against inadvertently overwriting an existing file.

The ShowDialog method returns the clicked button. It returns **DialogResult.OK** if Save is clicked and returns **DialogResult.Cancel** if Cancel is clicked. The **FileName** property contains the complete path to the selected file.

Typical use of **SaveFileDialog** control:

* Set the **Name, DefaultExt, Filter**, and **Title** properties**.**
* Use **ShowDialog** method to display dialog box.
* Read **FileName** property to determine selected file

**Summary**

There is wealth of material covered here. You now have a complete reference to the Visual C# toolbox and how those tools can be used for proper interface design. The Visual C# interface is very important and we wanted to make sure you have many tools at your disposal. This will make your (and your user’s) task much easier.

Even with all this work, our interface is not complete. We still need code that allows us to edit, add, and delete records from a database. We need to know how to validate and save changes properly. We need to know how to ‘undo’ unwanted changes. We need to know how to properly exit an application. These topics, and more, are covered in the next chapter where we learn to design the total database management system.

**Example 5-9**

**Publishers Table Input Form**

In this chapter, we built the framework for an interface that allows us to maintain the **Authors** table in the books database (**BooksDB.accdb**). This framework will be modified in the next chapter and implemented as part of a complete database management system. This database management system will also need interfaces to maintain the **Publishers** and **Titles** tables. The Titles table interface is a little tricky, in that it uses foreign keys to reference information in other tables. We will develop this interface in the next chapter. As an exercise here, we will begin the interface to maintain the **Publishers** table.

We will follow the same steps used in this chapter to build the Authors table input form:

* Build interface
* Add message box(es)
* Code application state
* Perform entry validation
* Perform input validation
* Add error trapping and handling
* Add on-line help system
* Application testing

Rather than starting from scratch, however, we will follow a ‘tried and true’ programming method – adapting an existing application to a new use. The Publishers table interface will essentially be the same as the Authors table interface. It will just have more (and different) input fields. Adapting an existing application saves us programmers a lot of time. You do have to make sure the modification implements the needs of the new application while at the same time eliminates vestiges of the old application. This exercise illustrates the modification steps followed and crosschecks required. An important step: **Save** your work often. You want to make sure your changes are always there.

**Build Interface**

1. **Make a copy of the Example 5-8 project folder** (the last version of the **Authors** table input form). Rename the newly copied folder to something else (I used **Example 5-9**). We now have a copy of the **Authors** table input form project to modify to a **Publishers** table input form. Open the copied project in Visual C#. The Publishers table has ten (10) fields that must be input:

**PubID**

**Name**

**Company\_Name**

**Address**

**City**

**State**

**Zip**

**Telephone**

**Fax**

**Comments**

The SQL statement needed by the command object to retrieve these fields (sorted by the **Name** field) is:

**SELECT \* FROM Publishers ORDER BY Name**

We need a label and text box for each fields. Resize the form so it is much taller (tall enough to hold ten labels and text boxes). Move the buttons to the bottom of the resized form. Don’t worry where things are right now – they can always be resized and/or moved later.

1. Change these properties on the existing form, labels and text box controls as:

**frmAuthors** (current name):

Name frmPublishers

Text Publishers

**Label1** (current name):

Text Publisher ID

**txtAuthorID** (current name):

Name txtPubID

**Label2** (current name):

Text Name

**txtAuthorName** (current name):

Name txtPubName

**Label3** (current name):

Text Company Name

**txtYearBorn** (current name):

Name txtCompanyName

MaxLength 32767

1. Add seven additional label and text box controls. Set these properties:

**Label4**:

Text Address

**TextBox1**:

Name txtPubAddress

BackColor White

ReadOnly True

TabIndex 3

**Label5**:

Text City

**TextBox2**:

Name txtPubCity

BackColor White

ReadOnly True

TabIndex 4

**Label6**:

Text State

**TextBox3**:

Name txtPubState

BackColor White

ReadOnly True

TabIndex 5

**Label7**:

Text Zip

**TextBox4**:

Name txtPubZip

BackColor White

ReadOnly True

TabIndex 6

**Label8**:

Text Telephone

**TextBox5**:

Name txtPubTelephone

BackColor White

ReadOnly True

TabIndex 7

**Label9**:

Text FAX

**TextBox6**:

Name txtPubFAX

BackColor White

ReadOnly True

TabIndex 8

**Label10**:

Text Comments

**TextBox7**:

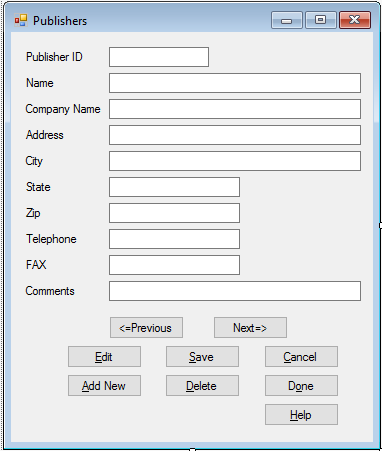
Name txtPubComments

BackColor White

ReadOnly True

TabIndex 9

At this point, my modified form looks like this:



The interface looks good. Let’s eliminate the vestiges (old code) from the application and add any needed new code.

1. We rename the data objects to reflect the **Publishers** table. Make the shaded changes to the Form level declarations:

**OleDbConnection booksConnection;**

**OleDbCommand publishersCommand;**

**OleDbDataAdapter publishersAdapter;**

**DataTable publishersTable;**

**CurrencyManager publishersManager;**

1. Make the shaded changes to the Form **Load** event method (you will have to reassign this method to the **Load** event):

**private void FrmPublishers\_Load(object sender, EventArgs e)**

**{**

**try**

**{**

**// point to help file**

**hlpAuthors.HelpNamespace = Application.StartupPath + "\\authors.chm";**

**// connect to books database**

**booksConnection = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source = c:\\VCSDB\\Working\\BooksDB.accdb");**

**booksConnection.Open();**

**// establish command object**

**publishersCommand = new OleDbCommand("SELECT \* FROM Publishers ORDER BY Name", booksConnection);**

**// establish data adapter/data table**

**publishersAdapter = new OleDbDataAdapter();**

**publishersAdapter.SelectCommand = publishersCommand;**

**publishersTable = new DataTable();**

**publishersAdapter.Fill(publishersTable);**

**// bind controls to data table**

**txtPubID.DataBindings.Add("Text", publishersTable, "PubID");**

**txtPubName.DataBindings.Add("Text", publishersTable, "Name");**

**txtCompanyName.DataBindings.Add("Text", publishersTable, "Company\_Name");**

**txtPubAddress.DataBindings.Add("Text", publishersTable, "Address");**

**txtPubCity.DataBindings.Add("Text", publishersTable, "City");**

**txtPubState.DataBindings.Add("Text", publishersTable, "State");**

**txtPubZip.DataBindings.Add("Text", publishersTable, "Zip");**

**txtPubTelephone.DataBindings.Add("Text", publishersTable, "Telephone");**

**txtPubFAX.DataBindings.Add("Text", publishersTable, "FAX");**

**txtPubComments.DataBindings.Add("Text", publishersTable, "Comments");**

**// establish currency manager**

**publishersManager = (CurrencyManager)this.BindingContext[publishersTable];**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show(ex.Message, "Error establishing Publishers table.", MessageBoxButtons.OK, MessageBoxIcon.Error);**

**return;**

**}**

**this.Show();**

**SetState("View");**

**}**

These changes reflect the new data object naming, the new SQL string and the proper data binding for the text box controls.

1. Make the shaded name change in the **FormClosing** method (you will have to reassign this method to the **FormClosing** event):

**private void FrmPublishers\_FormClosing(object sender, FormClosingEventArgs e)**

**{**

**// close the connection**

**booksConnection.Close();**

**// dispose of the objects**

**booksConnection.Dispose();**

**publishersCommand.Dispose();**

**publishersAdapter.Dispose();**

**publishersTable.Dispose();**

**}**

1. Make the shaded changes to the **btnPrevious** and **btnNext** **Click** event methods to reflect new name for currency manager:

**private void BtnPrevious\_Click(object sender, EventArgs e)**

**{**

**if (publishersManager.Position == 0)**

**{**

**Console.Beep();**

**}**

**publishersManager.Position--;**

**}**

**private void BtnNext\_Click(object sender, EventArgs e)**

**{**

**if (publishersManager.Position == publishersManager.Count - 1)**

**{**

**Console.Beep();**

**}**

**publishersManager.Position++;**

**}**

**Add Message Box(es)**

In its current state, all the message boxes within our code, except one, are generic in nature. These generic message boxes can be left as is. The one exception is the message box we added to inform the user if they typed an invalid date for the old **Year Born** field. This message box will be deleted in the next step.

**Code Application State**

1. In this step, we modify the code to reflect proper application state. We will eliminate all old code, so when we are done the application will run without errors. The biggest changes are in the **SetState** method. The modification locks and unlocks the text boxes (using the ReadOnly property), depending on state. The method is (new code is shaded):

**private void SetState(string appState)**

**{**

**switch (appState)**

**{**

**case "View":**

**txtPubID.BackColor = Color.White;**

**txtPubID.ForeColor = Color.Black;**

**txtPubName.ReadOnly = true;**

**txtCompanyName.ReadOnly = true;**

**txtPubAddress.ReadOnly = true;**

**txtPubCity.ReadOnly = true;**

**txtPubState.ReadOnly = true;**

**txtPubZip.ReadOnly = true;**

**txtPubTelephone.ReadOnly = true;**

**txtPubFAX.ReadOnly = true;**

**txtPubComments.ReadOnly = true;**

**btnPrevious.Enabled = true;**

**btnNext.Enabled = true;**

**btnAddNew.Enabled = true;**

**btnSave.Enabled = false;**

**btnCancel.Enabled = false;**

**btnEdit.Enabled = true;**

**btnDelete.Enabled = true;**

**btnDone.Enabled = true;**

**txtPubName.Focus();**

**break;**

**default: // Add or Edit if not View**

**txtPubID.BackColor = Color.Red;**

**txtPubID.ForeColor = Color.White;**

**txtPubName.ReadOnly = false;**

**txtCompanyName.ReadOnly = false;**

**txtPubAddress.ReadOnly = false;**

**txtPubCity.ReadOnly = false;**

**txtPubState.ReadOnly = false;**

**txtPubZip.ReadOnly = false;**

**txtPubTelephone.ReadOnly = false;**

**txtPubFAX.ReadOnly = false;**

**txtPubComments.ReadOnly = false;**

**btnPrevious.Enabled = false;**

**btnNext.Enabled = false;**

**btnAddNew.Enabled = false;**

**btnSave.Enabled = true;**

**btnCancel.Enabled = true;**

**btnEdit.Enabled = false;**

**btnDelete.Enabled = false;**

**btnDone.Enabled = false;**

**txtPubName.Focus();**

**break;**

**}**

**}**

1. Eliminate the **txtAuthorName KeyPress** and **txtYearBorn KeyPress** event methods since these controls no longer exist. Add this **txtInput KeyPress** event method (called by all nine editable controls). This implements the code to programmatically move from text box to text box using the <Enter> key (as an alternate to using <Tab>):

**private void TxtInput\_KeyPress(object sender, KeyPressEventArgs e)**

**{**

**TextBox whichBox = (TextBox) sender;**

**if ((int) e.KeyChar == 13)**

**{**

**switch (whichBox.Name)**

**{**

**case "txtPubName":**

**txtCompanyName.Focus();**

**break;**

**case "txtCompanyName":**

**txtPubAddress.Focus();**

**break;**

**case "txtPubAddress":**

**txtPubCity.Focus();**

**break;**

**case "txtPubCity":**

**txtPubState.Focus();**

**break;**

**case "txtPubState":**

**txtPubZip.Focus();**

**break;**

**case "txtPubZip":**

**txtPubTelephone.Focus();**

**break;**

**case "txtPubTelephone":**

**txtPubFAX.Focus();**

**break;**

**case "txtPubFAX":**

**txtPubComments.Focus();**

**break;**

**case "txtPubComments":**

**txtPubName.Focus();**

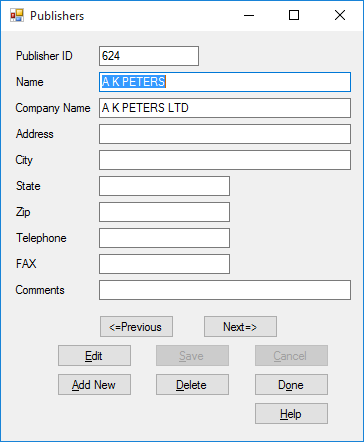
**break;**

**}**

**}**

**}**

1. Save and run the application. You should now be able to move from record to record and use the other buttons to switch from state to state (don’t click **Save** or **Help** yet). Here’s what I see for the first record (not all fields will have values):



### Perform Entry Validation

We need to eliminate any old entry validations done and add required new ones. The only field that appears to need entry validation is **Zip** (it only uses numbers and hyphens, for 9 digit zips). We won’t add any validation, though. Why? Perhaps, in the future, the post office will develop a zip code with letters. We want to be ready for this possibility. And, other countries have a wide variety of zip formats. Since we are doing nothing but displaying this value, validation is not that important. If we were doing math with a value or using it in some other function, validation would take on greater importance.

The old validation we need to eliminate is in the **KeyPress** event method for the **txtYearBorn** control. That method has already been eliminated.

### Perform Input Validation

Again, we need to eliminate any old input validations done and add required new ones. All of the inputs here are generic in nature and don’t need much validation. We will just insure a publisher **Name** field is entered.

1. Modify the **ValidateData** method to read (just eliminate the **Year Born** validation and modify the old **Author** **Name** validation a bit with the shaded changes):

**private bool ValidateData()**

**{**

**string message = "";**

**bool allOK = true;**

**// Check for name**

**if (txtPubName.Text.Trim().Equals(""))**

**{**

**message = "You must enter an Publisher Name." + "\r\n";**

**txtPubName.Focus();**

**allOK = false;**

**}**

**if (!allOK)**

**{**

**MessageBox.Show(message, "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Information);**

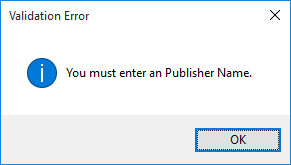
**}**

**return (allOK);**

**}**

You may be asking – isn’t the **PubID** field important enough to be validated? Well, yes, but being a primary key, it is treated differently. We will see how to handle this in Chapter 6.

1. Save the application and run it. Click **Edit**. Blank out the **Publisher** **Name** field and click **Save**. This message box should appear:



Stop the application.

**Add Error Trapping and Handling**

The error trapping and handling code in the old application still applies to the new application, hence no change is needed here. This is often the case in modifying existing applications. For other applications, you may need to modify existing error trapping schemes or add new ones.

### Add On-Line Help System

Use the HTML Help Workshop to develop a help system named **Publishers.chm**.

1. Prepare a single HTML topic file (**Publishers.htm**). The topic I used is:

**Publishers Input Form**

Available options for managing Publishers database table:

**Add New Record**

Click the **Add** **New** button to add a record to the Publishers database. Type in the requested fields. The Publisher Name is a required field. Click **Save** to save the new record; click **Cancel** to ignore the new record.

**Edit Record**

Click the **Edit** button to edit the displayed record. Make any needed changes. The Publisher Name is a required field. Click **Save** to save the changes; click **Cancel** to ignore the changes.

**Delete Record**

Click the **Delete** button to delete the displayed record.

#### Exit Program

Click the **Done** button to quit the application.

1. In the HTML Help Workshop, prepare a project file (**Publishers.hhp**). Compile the help file (**Publishers.chm**). All the help files are saved in the **VCSDB\Code\Class 5\Example 5-9\HelpFile** folder. Copy **Publishers.chm** to our application’s **Bin\Debug** folder.
2. Go back to your application in Visual C#. Change the **Name** of the help provider control to **hlpPublishers**. Make the shaded change near the top of the **frmPublishers Load** method:

**private void FrmPublishers\_Load(object sender, EventArgs e)**

**{**

**try**

**{**

**// point to help file**

**hlpPublishers.HelpNamespace = Application.StartupPath + "\\Publishers.chm";**

**// connect to books database**

**booksConnection = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source = c:\\VCSDB\\Working\\BooksDB.accdb");**

**booksConnection.Open();**

**.**

**.**

**.**

**}**

And make the shaded change to the **btnHelp Click** method:

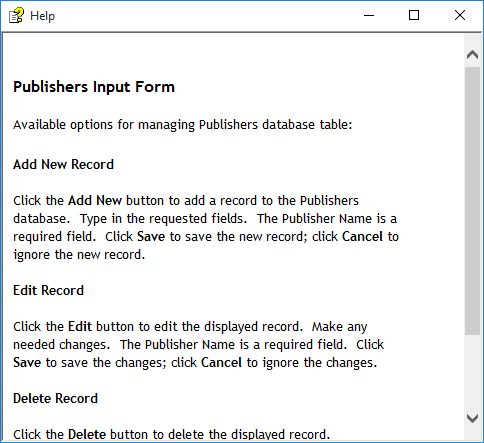
**private void BtnHelp\_Click(object sender, EventArgs e)**

**{**

**Help.ShowHelp(this, hlpPublishers.HelpNamespace);**

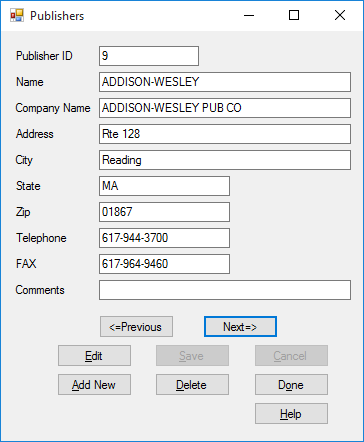
**}**

1. Save the application. Run it. Make sure both the <**F1**> key and **Help** button bring up the help system properly:



**Application Testing**

If you did all the above steps carefully, the application should be running properly. If not, make the changes required to get it running. Here’s one of the first records I found with most of the fields:



As with the Authors form, we still need code to add the database management functions. This is addressed in the next chapter. The final version of this example is saved in the **Example 5-9** folder in **VCSDB\Code\Class 5** folder.

**Example 5-9**

**Using SQL Server Databases**

1. The SQL Server version of the books database is **SQLBooksDB.mdf**. Copy **SQLBooksDB.mdf** to your working directory
2. Use this using statement:

**using System.Data.SqlClient;**

1. In declarations, use these objects:

**SqlConnection booksConnection;**

**SqlCommand authorsCommand;**

**SqlDataAdapter authorsAdapter;**

1. In **frmPublishers Load** method, use this connection object:

**booksConnection = new SqlConnection("Data Source=.\\SQLEXPRESS; AttachDbFilename=c:\\VCSDB\\Working\\SQLBooksDB.mdf; Integrated Security=True; Connect Timeout=30; User Instance=True");**

1. In **frmPublishers Load** method:

Change all instances of **OleDbCommand** to **SqlCommand**

Change all instances of **OleDbDataAdapter** to **SqlDataAdapter**

The SQL Server version is saved in the **Example 5-9 SQL** folder in **VCSDB\Code\Class 5** folder.