**5**

**Visual C# Interface Design**

**Review and Preview**

At this point in the course, we can use Visual C# to connect to a database and SQL statements allow us to obtain any view of the database information we desire. But, that’s all we can do - view the data.

We now want to know how to allow a user to interact with the data - obtain alternate views, modify it, add to it, delete it. To do this, we need a well-designed user interface.

In this chapter, we look at some design considerations for the Visual C# front-end. We examine the toolbox controls and Visual C# coding techniques needed to build a useful interface and application. Several examples illustrate use of the tools and techniques.

## Visual C# Standard Controls

The first step in building a Visual C# interface is to ‘draw’ the application on a form. We place the required controls on the form, set properties, and write C# code for the needed event and general methods. As the interface designer, you need to decide which controls best meet your needs regarding efficiency, applicability, and minimization of error possibilities.

In this section, we briefly look at the standard Visual C# controls. We examine how they might be used in a database ‘front-end’ and present some of the important properties, events, and methods associated with these controls. This information is provided as a quick review of what is available in the Visual C# toolbox - a “one-stop” reference to standard controls and how they are used with databases. A later look at more advanced controls will complete the reference.

**Form Control**

The **Form** is where the user interface is drawn. It is central to the development of Visual C# applications, whether for databases or other uses.

Form **Properties:**

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

**AcceptButton** Gets or sets the button on the form that is clicked when the user presses the <Enter> key.

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

**CancelButton** Gets or sets the button control that is clicked when the user presses the <Esc> key.

**ControlBox** Gets or sets a value indicating whether a control box is displayed in the caption bar of the form.

**Enabled** If False, all controls on form are disabled.

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

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

**FormBorderStyle** Sets the form border to be fixed or sizeable.

**Height** Height of form in pixels.

**Help** Gets or sets a value indicating whether a Help button should be displayed in the caption box of the form.

**Icon** Gets or sets the icon for the form.

**Left** Distance from left of screen to left edge of form, in pixels.

**MaximizeButton** Gets or sets a value indicating whether the maximize button is displayed in the caption bar of the form.

**MinimizeButton** Gets or sets a value indicating whether the minimize button is displayed in the caption bar of the form.

**StartPosition** Gets or sets the starting position of the form when the application is running.

**Text** Gets or sets the form window title.

**Top** Distance from top of screen to top edge of form, in pixels.

**Width** Width of form in pixels.

Form **Methods**:

**Close** Closes the form.

**Focus** Sets focus to the form.

**Hide** Hides the form.

**Refresh** Forces the form to immediately repaint itself.

**Show** Makes the form display by setting the Visible property to True.

The normal syntax for invoking a method is to type the control name, a dot, then the method name. For form methods, the name to use is **this**. This is a Visual C# keyword used to refer to a form. Hence, to close a form, use:

**this.Close();**

Form **Events**:

**Activated** Occurs when the form is activated in code or by the user.

**Click** Occurs when the form is clicked by the user.

**FormClosing** Occurs when the form is closing.

**DoubleClick** Occurs when the form is double clicked.

**Load** Occurs before a form is displayed for the first time.

**Paint** Occurs when the form is redrawn.

Typical use of **Form** object (for each control in this, and following chapters, we will provide information for how that control is typically used):

* Set the **Name** and **Text** properties
* Set the **StartPosition** property (in this course, this property will almost always be set to **CenterScreen**)
* Set the **FormBorderStyle** to some value. In this course, we will mostly use **FixedSingle** forms. You can have resizable forms in Visual C# (and there are useful properties that help with this task), but we will not use resizable forms in this course.
* Write any needed initialization code in the form’s **Load** event. To access this event in the Code window, double-click the form.

**Button Control**



The **button** is probably the most widely used control. It is used to begin, interrupt, or end a particular process. With **databases**, it is used to **navigate** among records, **add** records, and **delete** records.

Button **Properties**:

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

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

**Enabled** If False, button is visible, but cannot accept clicks.

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

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

**Image** Gets or sets the image that is displayed on a button control.

**Text** Gets or sets string displayed on button.

**TextAlign** Gets or sets the alignment of the text on the button control.

Button **Methods**:

**Focus** Sets focus to the button.

**PerformClick** Generates a [Click](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformscontrolclassclicktopic.htm) event for a button.

Button **Events**:

**Click** Event triggered when button is selected either by clicking on it or by pressing the access key.

Typical use of **Button** control:

* Set the **Name** and **Text** property.
* Write code in the button’s **Click** event.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

**Label Control**



A **label** is a control you use to display text. The text in a label can be changed at run-time in response to events. It is widely used in **database** applications for **information** **display**.

Label **Properties**:

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

**AutoSize** Gets or sets a value indicating whether the label is automatically resized to display its entire contents.

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

**BorderStyle** Gets or sets the border style for the label.

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

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

**Text** Gets or sets string displayed on label.

**TextAlign** Gets or sets the alignment of text in the label.

Note, by default, the label control has no resizing handles. To resize the label, set AutoSize to False.

Label **Methods:**

**Refresh** Forces an update of the label control contents.

Label **Events**:

**Click** Event triggered when user clicks on a label.

**DblClick** Event triggered when user double-clicks on a label.

Typical use of **Label** control for static, unchanging display:

* Set the **Name** (though not really necessary for static display) and **Text** property.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

Typical use of **Label** control for changing display:

* Set the **Name** property. Initialize **Text** to desired string.
* Set **AutoSize** to **False**, resize control and select desired value for **TextAlign**.
* Assign **Text** property (String type) in code where needed.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

**TextBox Control**



A **text box** is used to display information entered at design time, by a user at run-time, or assigned within code. The displayed text may be edited. This is the tool used in **database** applications for **editing** fields.

TextBox **Properties**:

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

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

**BorderStyle** Gets or sets the border style for the text box.

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

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

**HideSelection** Gets or sets a value indicating whether the selected text in the text box control remains highlighted when the control loses focus.

**Lines** Gets or sets the lines of text in a text box control.

**MaxLength** Gets or sets the maximum number of characters the user can type into the text box control.

**MultiLine** Gets or sets a value indicating whether this is a multiline text box control.

**PasswordChar** Gets or sets the character used to mask characters of a password in a single-line TextBox control.

**ReadOnly** Gets or sets a value indicating whether text in the text box is read-only.

**ScrollBars** Gets or sets which scroll bars should appear in a multiline TextBox control.

**SelectedText** Gets or sets a value indicating the currently selected text in the control.

**SelectionLength** Gets or sets the number of characters selected in the text box.

**SelectionStart** Gets or sets the starting point of text selected in the text box.

**Text** Gets or sets the current text in the text box.

**TextAlign** Gets or sets the alignment of text in the text box.

**TextLength** Gets length of text in text box.

TextBox **Methods**:

**AppendText** Appends text to the current text of text box.

**Clear** Clears all text in text box.

**Focus** Places the cursor in a specified text box.

**SelectAll** Selects all text in text box.

**Undo** Undoes the last edit operation in the text box.

TextBox **Events**:

**KeyDown** Occurs when a key is pressed down while the control has focus.

**KeyPress** Occurs when a key is pressed while the control has focus – used for key trapping.

**Leave** Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.

**TextChanged** Occurs when the [Text](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformscontrolclasstexttopic.htm) property value has changed.

Typical use of **TextBox** control as display control:

* Set the **Name** property. Initialize **Text** property to desired string.
* Set **ReadOnly** property to **True**.
* If displaying more than one line, set **MultiLine** property to **True**.
* Assign **Text** property in code where needed.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

Typical use of **TextBox** control as input device:

* Set the **Name** property. Initialize **Text** property to desired string.
* If it is possible to input multiple lines, set **MultiLine** property to **True**.
* In code, give **Focus** to control when needed. Provide key trapping code in **KeyPress** event. Read **Text** property when **Leave** event occurs.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

**CheckBox Control**



**Check boxes** provide a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected. With **databases**, check boxes are used for many kinds of **choices**.

CheckBox **Properties**:

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

**AutoSize** Gets or sets a value indicating whether the check box is automatically resized to display its entire contents.

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

**Checked** Gets or sets a value indicating whether the check box is in the checked state.

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

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

**Text** Gets or sets string displayed next to check box.

**TextAlign** Gets or sets the alignment of text of the check box.

CheckBox **Methods**:

**Focus** Moves focus to this check box.

CheckBox **Events**:

**CheckedChanged** Occurs when the value of the [Checked](F:\\Richland Files\\ITSE 2438\\VCSDB\\Notes\\frlrfsystemwindowsformscheckboxclasscheckedtopic.htm) property changes, whether in code or when a check box is clicked.

**Click** Triggered when a check box is clicked. **Checked** property is automatically changed by Visual C#.

Typical use of **CheckBox** control:

* Set the **Name** and **Text** property. Initialize the **Checked** property.
* Monitor **Click** or **CheckChanged** event to determine when button is clicked. At any time, read **Checked** property to determine check box state.
* You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

**RadioButton Control**



**Radio buttons** provide the capability to make a mutually exclusive choice among a group of potential candidate choices. Hence, radio buttons work as a group, only one of which can have a True (or selected) value. Radio buttons on a form work as an independent group as do groups of radio buttons within panels and group boxes. Radio buttons are not data bound controls, yet they can still be used for a variety of **options** in database interfaces.

RadioButton **Properties**:

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

**AutoSize** Gets or sets a value indicating whether the radio button is automatically resized to display its entire contents.

**BackColor** Get or sets the radio button background color.

**Checked** Gets or sets a value indicating whether the radio button is checked.

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

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

**TextAlign** Gets or sets the alignment of text of the radio button.

RadioButton **Methods**:

**Focus** Moves focus to this radio button.

**PerformClick** Generates a [Click](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformscontrolclassclicktopic.htm) event for the button, simulating a click by a user.

RadioButton **Events**:

**CheckedChanged** Occurs when the value of the [Checked](file:///F:\Richland%20Files\ITSE%202438\VCSDB\Notes\frlrfsystemwindowsformscheckboxclasscheckedtopic.htm) property changes, whether in code or when a radio button is clicked.

**Click** Triggered when a button is clicked. **Checked** property is automatically changed by Visual C#.

Typical use of **RadioButton** control:

* Establish a group of radio buttons.
* For each button in the group, set the **Name** (give each button a similar name to identify them with the group) and **Text** property. You might also change the **Font**, **BackColor** and **Forecolor** properties.
* Initialize the **Checked** property of one button to **True**.
* Monitor the **Click** or **CheckChanged** event of each radio button in the group to determine when a button is clicked. The ‘last clicked’ button in the group will always have a **Checked** property of **True**.

**GroupBox Control**



**Group boxes** provide a way of grouping related controls on a form. Radio buttons within a group box act independently of other radio buttons in an application.

GroupBox **Properties**:

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

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

**Enabled** Gets or sets a value indicating whether the group box is enabled. If False, all controls in the group box are disabled.

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

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

**Text** Gets or sets string displayed in title region of group box.

**Visible** If False, hides the group box (and all its controls).

The GroupBox control has some methods and events, but these are rarely used. We are more concerned with the methods and events associated with the controls in the group box.

Typical use of **GroupBox** control:

* Set **Name** and **Text** property (perhaps changing **Font**, **BackColor** and **ForeColor** properties).
* Place desired controls in group box. Monitor events of controls in group box using usual techniques.

**Panel Control**



The **Panel** control is another Visual C# grouping control. It is nearly identical to the **GroupBox** control in behavior. The panel control lacks a Text property (titling information), but has optional scrolling capabilities. Radio buttons in the panel control act as an independent group. Panel controls can also be used to display graphics (lines, curves, shapes, animations).

Panel **Properties**:

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

**AutoScroll** Gets or sets a value indicating whether the panel will allow the user to scroll to any controls placed outside of its visible boundaries.

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

**BorderStyle** Get or set the panel border style.

**Enabled** Gets or sets a value indicating whether the panel is enabled. If False, all controls in the panel are disabled.

**Visible** If False, hides the panel (and all its controls).

Like the GroupBox control, the Panel control has some methods and events, but these are rarely used (we will see a few Panel events in later graphics chapters). We usually only are concerned with the methods and events associated with the controls in the panel.

Typical use of **Panel** control:

* Set **Name** property.
* Place desired controls in panel control.
* Monitor events of controls in panel using usual techniques.

**PictureBox Control**



The **picture box** allows you to place graphics information on a form. In a **database**, picture boxes are used to store **graphic** data.

PictureBox **Properties**:

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

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

**BorderStyle** Indicates the border style for the picture box.

**Height** Height of picture box in pixels.

**Image** Establishes the graphics file to display in the picture box (jpeg, gif, bmp files).

**Left** Distance from left edge of form to left edge of picture box, in pixels.

**SizeMode** Indicates how the image is displayed.

**Top** Distance bottom of form title bar area to top edge of picture box, in pixels.

**Width** Width of picture box in pixels.

PictureBox **Events**:

**Click** Triggered when a picture box is clicked.

Typical use of **PictureBox** control for displaying images:

* Set the **Name** and **SizeMode** property (most often, **StretchImage**).
* Set **Image** property, either in design mode or at run-time, remembering icon files are not automatically displayed.

**Example 5-2**

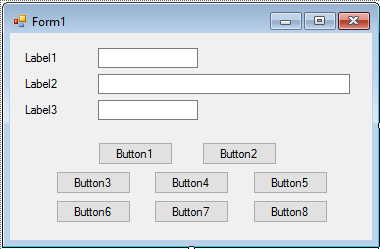
**Authors Table Input Form**

In Chapter 6, we will build a complete database management system for the books database. Each table in the database will require some kind of input form. In this chapter, we build such a form for the **Authors** table. Even though it is a very simple table (only three fields: **Au\_ID**, **Author**, **Year\_Born**), it provides an excellent basis to illustrate many of the steps of proper interface design. The SQL statement needed by the command object to retrieve the fields (sorted by **Author**) is:

**SELECT \* FROM Authors ORDER BY Author**

We need an input form that allows a user to edit an existing record, delete an existing record or add a new record. The form should also allow navigation from one record to another. The steps to follow:

1. Start a new application. We need three label controls and three text boxes to display the fields. We need two buttons to move from one record to the next. We need five buttons to control editing features and one button to allow us to stop editing. Place these controls on a form. The layout should resemble:



1. Set these properties for the form and controls:

**Form1**:

Name frmAuthors

FormBorderStyle FixedSingle

StartPosition CenterScreen

Text Authors

**label1**:

Text Author ID

**textBox1**:

Name txtAuthorID

BackColor White

ReadOnly True

**label2**:

Text Author Name

**textBox2**:

Name txtAuthorName

BackColor White

ReadOnly True

**label3**:

Text Year Born

**textBox3**:

Name txtYearBorn

BackColor White

ReadOnly True

**button1**:

Name btnPrevious

Text <= Previous

**button2**:

Name btnNext

Text Next =>

**button3**:

Name btnEdit

Text &Edit

**button4**:

Name btnSave

Text &Save

**button5**:

Name btnCancel

Text &Cancel

**button6**:

Name btnAddNew

Text &Add New

**button7**:

Name btnDelete

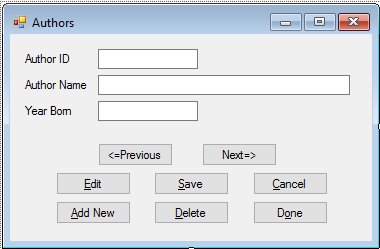
Text &Delete

**button8**:

Name btnDone

Text Do&ne

Note, we ‘lock’ (**ReadOnly** = True) all the text boxes. We will unlock them when we (as the programmer) decide the user can change a value (remember, we are in control). At this point, the form should appear as:



1. We will add features to this input application as we progress through the chapter. At this point, we add code to form the data table and allow us to navigate through the **Authors** table records. Add this line at the top of the code window:

**using System.Data.OleDb;**

1. Form level declarations to create data objects:

**OleDbConnection booksConnection;**

**OleDbCommand authorsCommand;**

**OleDbDataAdapter authorsAdapter;**

**DataTable authorsTable;**

**CurrencyManager authorsManager;**

1. Add this code the **frmAuthors Load** method:

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

**{**

**// 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];**

**}**

This code creates the needed data objects to open the database and form the **Authors** table (includes all fields ordered by Author). It then binds the controls to the currency manager object. This code is identical to code seen in the previous two chapters.

1. Add this code to the **frmAuthors Closing** event method to close the database connection:

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

**{**

**// close the connection**

**booksConnection.Close();**

**// dispose of the objects**

**booksConnection.Dispose();**

**authorsCommand.Dispose();**

**authorsAdapter.Dispose();**

**authorsTable.Dispose();**

**}**

1. Code for the two button **Click** events to allow navigation:

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

**{**

**authorsManager.Position--;**

**}**

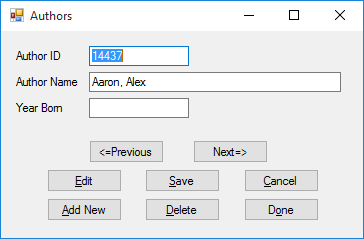
**private void BtnNext\_Click(object sender, EventArgs e)**

**{**

**authorsManager.Position++;**

**}**

1. Save the application (saved in the **Example 5-2** folder in **VCSDB\Code\Class 5** folder). Run it. Here is the first record:



Navigate among the records. Note you cannot edit anything. The text boxes are read-only. As we progress through this chapter (and the next), we will continue to add features to this example until it is complete.

**Example 5-2**

**Using SQL Server Databases**

Differences in the SQL Server version of the example:

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 **frmAuthors 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 **frmAuthors Load**:

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

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

The SQL Server version is saved in the **Example 5-2 SQL** folder in **VCSDB\Code\Class 5** folder.

**MessageBox Object**

Many times, in a database application, you will want to impart some information to your user. That information may be a courtesy message (“New record written”) or a question requiring feedback (“Do you really want to delete this record?”). Visual C# (and Windows) provides an excellent medium for providing such information – the **MessageBox** object.

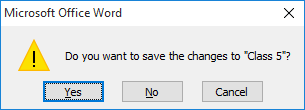
You've seen message boxes if you've ever used a Windows application. Think of all the examples you've seen. For example, message boxes are used to ask you if you wish to save a file before exiting and to warn you if a disk drive is not ready. For example, if while writing these notes in Microsoft Word, I attempt to exit, I see this message box:

Text

Caption

Buttons

Icon



DefaultButton

In this message box, the different parts that you control have been labeled. You will see how you can format a message box any way you desire.

To use the **MessageBox** object, you decide what the **Text** of the message should be, what **Caption** you desire, what **Icon** and **Buttons** are appropriate, and which **DefaultButton** you want. To display the message box in code, you use the MessageBox **Show** method.

The MessageBox is **overloaded** with several ways to implement the **Show** method. Some of the more common ways are:

**MessageBox.Show(Text);**

**MessageBox.Show(Text, Caption);**

**MessageBox.Show(Text, Caption, Buttons);**

**MessageBox.Show(Text, Caption, Buttons, Icon);**

**MessageBox.Show(Text, Caption, Buttons, Icon, DefaultButton);**

In these implementations, if **DefaultButton** is omitted, the first button is default. If **Icon** is omitted, no icon is displayed. If **Buttons** is omitted, an ‘OK’ button is displayed. And, if **Caption** is omitted, no caption is displayed.

You decide what you want for the message box **Text** and **Caption** information (string data types). The other arguments are defined by Visual C# predefined constants. The **Buttons** constants are defined by the **MessageBoxButtons** constants:

**Member Description**

AbortRetryIgnore Displays Abort, Retry and Ignore buttons

OK Displays an OK button

OKCancel Displays OK and Cancel buttons

RetryCancel Displays Retry and Cancel buttons

YesNo Displays Yes and No buttons

YesNoCancel Displays Yes, No and Cancel buttons

The syntax for specifying a choice of buttons is the usual dot-notation:

**MessageBoxButtons.Member**

So, to display an OK and Cancel button, the constant is:

MessageBoxButtons.OKCancel

You don’t have to remember this, however. When typing the code, the Intellisense feature will provide a drop-down list of button choices when you reach that argument! This will happen for all the arguments in the MessageBox object.

The displayed Icon is established by the **MessageBoxIcon** constants:

**Member Description**

IconAsterisk Displays an information icon

IconInformation Displays an information icon

IconError Displays an error icon (white X in red circle)

IconHand Displays an error icon

IconNone Display no icon

IconStop Displays an error icon

IconExclamation Displays an exclamation point icon

IconWarning Displays an exclamation point icon

IconQuestion Displays a question mark icon

To specify an icon, the syntax is:

### MessageBoxIcon.Member

Note there are eight different members of the **MessageBoxIcon** constants, but only four icons (information, error, exclamation, question) available. This is because the current Windows operating system only offers four icons. Future implementations may offer more.

When a message box is displayed, one of the displayed buttons will have focus or be the default button. If the user presses <Enter>, this button is selected. You specify which button is default using the **MessageBoxDefaultButton** constants:

**Member Description**

Button1 First button in message box is default

Button2 Second button in message box is default

Button3 Third button in message box is default

To specify a default button, the syntax is:

### MessageBoxDefaultButton.Member

The specified default button is relative to the displayed buttons, left to right. So, if you have Yes, No and Cancel buttons displayed and the second button is selected as default, the No button will have focus (be default). Always try to make the default response the “least damaging,” if the user just blindly accepts it.

When you invoke the **Show** method of the MessageBox object, the method returns a value from the **DialogResult** constants. The available members are:

**Member Description**

Abort The Abort button was selected

Cancel The Cancel button was selected

Ignore The Ignore button was selected

No The No button was selected

OK The OK button was selected

Retry The Retry button was selected

Yes The Yes button was selected

MessageBox **Example**:

This little code snippet (the first line is very long):

**if (MessageBox.Show("This is an example of a message box", "Message Box Example", MessageBoxButtons.OKCancel, MessageBoxIcon.Information, MessageBoxDefaultButton.Button1) == DialogResult.OK)**

**{**

**// everything is OK**

**}**

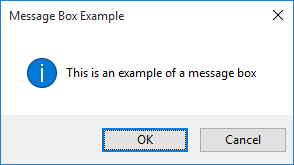
**else**

**{**

**// cancel was pressed**

**}**

displays this message box:



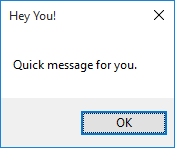
Of course, you would need to add code for the different tasks depending on whether OK or Cancel is clicked by the user.

Another MessageBox **Example**:

Many times, you just want to display a quick message to the user with no need for feedback (just an OK button). This code does the job:

**MessageBox.Show("Quick message for you.", "Hey You!");**

The resulting message box:



Notice there is no icon and the OK button (default if no button specified) is shown. Also, notice in the code, there is no need to read the returned value – we know what it is! You will find a lot of uses for this simple form of the message box (with perhaps some kind of icon) as you progress in this course.

Message boxes should be used whenever your application needs to inform the user of action or requires user feedback to continue. It is probably better to have too many message boxes, than too few. You always want to make sure your application is performing as it should and the more information you have, the better.

**Example 5-3**

**Authors Table Input Form (Message Box)**

There are two places where we could use message boxes in the Authors Table example. A simple box after saving an update to let the user know the save occurred and one related to deleting records.

1. Load Example 5-2 completed earlier. We will modify this example to include message boxes.
2. Use this code in the **btnSave Click** event:

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

**{**

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

**}**

Obviously, there will be more code in this event as we continue with this example. This code just implements the message box.

1. Use this code in the **btnDelete Click** event:

**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)**

**{**

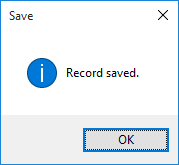
**return;**

**}**

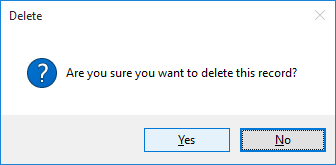
**}**

Note we exit the method if the user selects **No**. And, notice the **No** button is default – this makes the user think a bit before hitting **Enter**. Like above, there will be more code in this method as we proceed.

1. Save the application (saved in the **Example 5-3** folder in **VCSDB\Code\Class 5** folder) and run it. Click the **Save** button and you will see:



Click **OK**, then try clicking the **Delete** button to see:



**Example 5-3**

**Using SQL Server Databases**

Using the SQL Server version of the project, make the same changes to **btnSave Click**, and **btnDelete Click.** The SQL Server version is saved in the **Example 5-3 SQL** folder in **VCSDB\Code\Class 5** folder.

**Application State**

When presenting a Visual C# database interface to a user, it should be obvious, to the user, what needs to be done. Options should be intuitive and the possibility of mistakes minimized, if not completely eliminated. To maintain this obvious quality, you should always be aware of what **state** your application is in.

**Application** **state** implies knowing just what is currently being done within the interface. Are you adding a record, editing a record, deleting a record, or perhaps leaving the application? Once you know the state the application is in, you adjust the interface so that options needed for that particular state are available to the user. You also need to know when and how to transition from one state to another.

What options are adjusted to reflect application state? A primary option is a control’s **Enabled** property. By setting **Enabled** to **False**, you disable a control, making it unavailable to the user. So, if the user is not able to save a current record, the button that does the save should have an Enabled property of False. A more drastic disabling of a control is setting its Visible property to False. In this case, there is no misunderstanding about application state. As the application moves from one state to another, you need to determine which controls should be enabled and which should be disabled.

For **text** **box** controls, a property of importance is the **ReadOnly** property. If a value in a text box is not to be edited, set ReadOnly to True. When editing is allowed (the state changes), toggle the ReadOnly property to False. For text boxes that are always read-only (used for display, not editing purposes), use color (red is good) to indicate they are not accessible. When editing in a text box, use the **Focus** method to place the cursor in the box, making it the active control (giving it focus) and saving the user a mouse click. The **Focus** method can also be used to programmatically move the user from one text box to the next in a desired order.

Another mechanism for moving from one control to another in a prescribed order is the **TabIndex** property, in conjunction with **TabStop**. If **TabStop** is **True**, TabIndex defines the order controls become active (only one control can be active at a time) as the <**Tab**> key is pressed (the order is reversed when <**Shift**>-<**Tab**> is pressed). When controls are placed on a form at design time, they are assigned a TabIndex value with TabStop = true;. If you don’t want a control to be made active with <Tab>, you need to reset its TabStop property to False. If the assigned order is not acceptable, reset the TabIndex properties for the desired controls, starting with a low number and increasing that value with each control added to the <Tab> sequence. A primary application for <Tab> sequencing is moving from one text box to the next in a detailed input form.

If the concepts of control focus and tab movements are new or unfamiliar, try this. Start a new application in Visual C#. Add three buttons (**Button1**, **Button2**, **Button3**), then three text boxes (**Text1**, **Text2**, **Text3**). Run the application. The first button (Button1) should have focus (a little outline box is around the Text). If you press <Enter> at this point, this button is ‘clicked.’ Press the <Tab> key and the focus moves to the second button. Press <Tab> twice. The focus should now be in the first text box (the cursor is in the box). Keep pressing <Tab> and watch the focus move from one control to the other, always in the same order. Pressing <Shift>-<Tab> reverses the order. Now, for each button, set the **TabStop** property to **False** (removing them from the tab sequence). Re-run the application and you should note the focus only shifts among the text boxes. Try resetting the **TabIndex** properties of the text boxes to change the shift direction. Always use the idea of focus in your applications to indicate to the user what control is active.

All of this application state talk may sound complicated, but it really isn’t. Again, it’s all just a matter of common sense. After you design your interface, sit back and step through your application in the Visual C# environment, exercising every option available. With each option, ask yourself what the user needs to see. Implement the necessary logic to make sure this is all the user sees. Make sure moves from one state to another are apparent and correct. Try lots of things until you are comfortable with the finished product. The Visual C# environment makes performing such tasks quite easy.

**Example 5-4**

**Authors Table Input Form**

**(Application State)**

The Authors Table Input Form can operate in one of three states: **View** state, **Add** state or **Edit** state. In **View** state, the user can navigate from record to record, switch to Edit state, add and/or delete records, or exit the application.

In both **Add** and **Edit** states, no navigation should be possible, data can be changed, and the user should have access to the **Save** and **Cancel** functions. Each of these states can be implemented using button **Enabled** properties and text box **ReadOnly** properties. We use **TabIndex** (and **TabOrder**) to control shift of focus in the text box controls. We will use a general method to allow switching from one state to another.

1. Open Example 5-3 in the Visual C# environment. We will modify this example to include state considerations.
2. Remove the buttons from tab sequencing by setting all (eight buttons) of their **TabStop** properties to **False**. Also set **TabStop** to **False** for the **txtAuthorID** text box (we will not edit this value - we’ll explain why later). Set **TabIndex** for **txtAuthorName** to **1** and **TabIndex** for **txtYearBorn** to **2**.
3. Add a general method named **SetState** with string argument **appState**. Add this code to the **SetState** method:

**private void SetState(string appState)**

**{**

**switch (appState)**

**{**

**case "View":**

**txtAuthorID.BackColor = Color.White;**

**txtAuthorID.ForeColor = Color.Black;**

**txtAuthorName.ReadOnly = true;**

**txtYearBorn.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;**

**txtAuthorName.Focus();**

**break;**

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

**txtAuthorID.BackColor = Color.Red;**

**txtAuthorID.ForeColor = Color.White;**

**txtAuthorName.ReadOnly = false;**

**txtYearBorn.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;**

**txtAuthorName.Focus();**

**break;**

**}**

**}**

This code sets the application in View, Add or Edit state. Note which buttons are available and which are not. Notice the **Author** **ID** box is red in Add and Edit state to indicate it cannot be changed. Notice that the Add and Edit states are the same (for now) and are just a ‘toggling’ of the View state – this will occur quite often – a great place for ‘cut and paste’ coding. We now need to modify the application code to use this method to move from state to state.

1. We want to be in the **View** state when the application is initialized. Add these two lines at the bottom of the **frmAuthors\_Load** event:

**this.Show();**

**SetState("View");**

1. When the **Add** **New** button is clicked, we want to switch to **Add** state. Add this line of code at the top of the **btnAddNew\_Click** event method:

**SetState("Add");**

1. When the **Edit** button is clicked, we switch to **Edit** state. Add this line of code at the top of the **btnEdit\_Click** event method:

**SetState("Edit");**

1. Following a **Cancel** or **Save** operation (in **Add** or **Edit** state), we want to return to **View** state. Place this line at the end of the **btnCancel\_Click** and **btnSave\_Click** event methods:

**SetState("View");**

The **Delete** button does not need any change of state code – it only works in **View** state and stays in that state following a delete.

1. We’re almost done. This is a small change, but an important one that gives your application a professional touch. Notice that if you click the **Previous** button and the pointer is at the first record, nothing changes. Similarly, at the end of the data table, if you click **Next**, nothing happens. This lack of change might confuse the user. To give the user some feedback that they’ve reached a limit, I like to provide some audible feedback. Make the shaded changes to the **btnPrevious** and **btnNext** **Click** event methods to play a beep when the user bumps into a limit:

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

**{**

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

**{**

**Console.Beep();**

**}**

**authorsManager.Position--;**

**}**

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

**{**

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

**{**

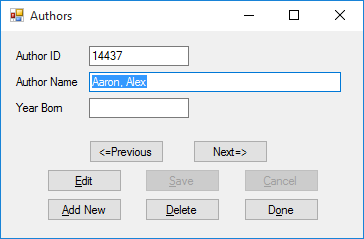
**Console.Beep();**

**}**

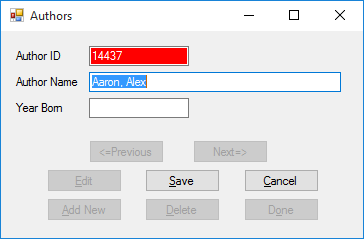
**authorsManager.Position++;**

**}**

1. Save and run the application (saved in the **Example 5-4** folder in **VCSDB\Code\Class 5** folder). The initial (**View**) state is:



Notice how the various buttons change state as different functions are accessed on the interface form. In **Add** and **Edit** state (the ID box is red), check the tab order of the two text boxes (a very short tab order!):



In each state, it is obvious to the user what functions are available and when they are available. Do you hear the beep when you try to move past a limit at the end or beginning of the data table?