SIT 221 – Event Driven Programming

EXPLORING THE VISUAL BASIC .NET TOOLBOX

Function Overloading

- Overloading lets a function vary its behavior based on its input arguments.
- Visual Basic .NET will have multiple functions with the same name, but with different argument lists.
- The different argument lists may have different numbers of arguments and different types of arguments.

What are the implications of overloading?

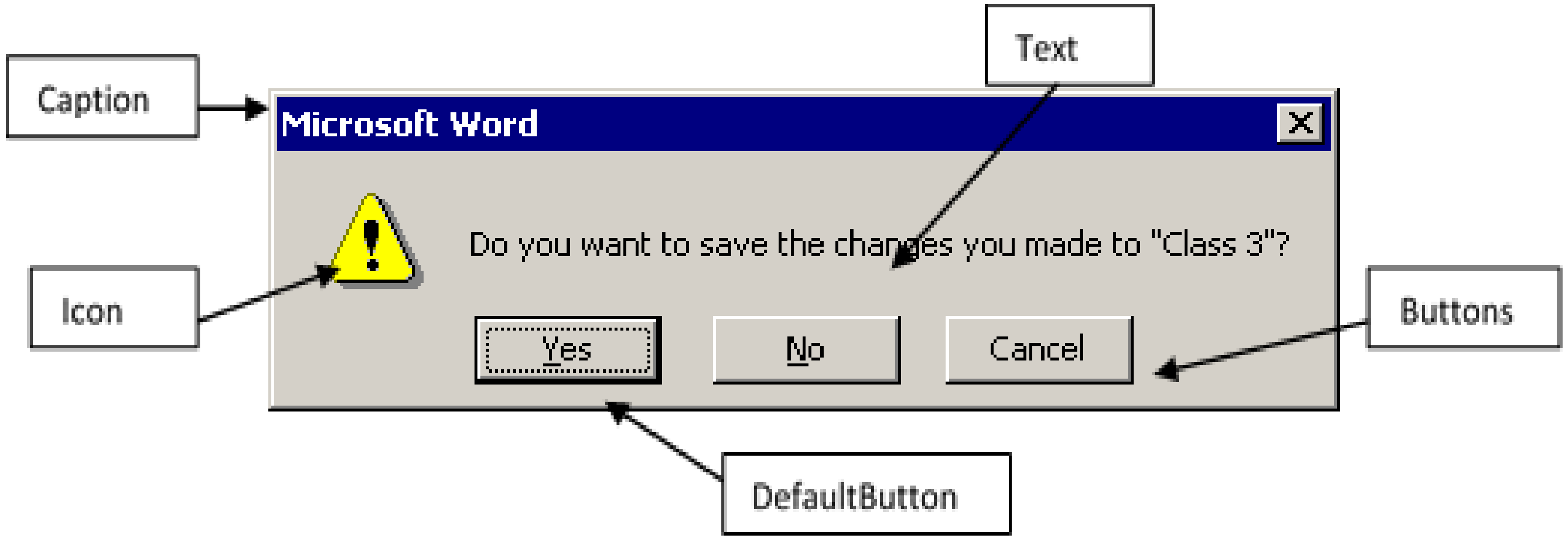
- What this means to us is that when using a Visual Basic .NET function, there will be several different ways to use that function.
- Another implication is that, when typing code for a function, the Intellisense feature of the Visual Basic .NET IDE will appear with a drop-down list of all implementations of a particular function. As you type in your implementation, Intellisense adjusts to the particular form of the function it “thinks” you are using.

Overloading is a powerful feature of Visual Basic .NET.

MessageBox Dialog

- One of the most often used functions in Visual Basic .NET is the **MessageBox** function.
- This function lets you display messages to your user and receive feedback for further information.
- It can be used to display error messages, describe potential problems or just to show the result of some computation.
- The **MessageBox** function is versatile, with the ability to display any message, an optional icon, and a selected set of buttons.
- The user responds by clicking a button in the message box

Example



Do you want to save the changes you made to "Class 3"?

To use the **MessageBox** function, 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 function 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 Basic .NET 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

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.

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**
DefaultButton1	First button in message box is default
DefaultButton2	Second button in message box is default
DefaultButton3	Third button in message box is default

To specify a default button, the syntax is:

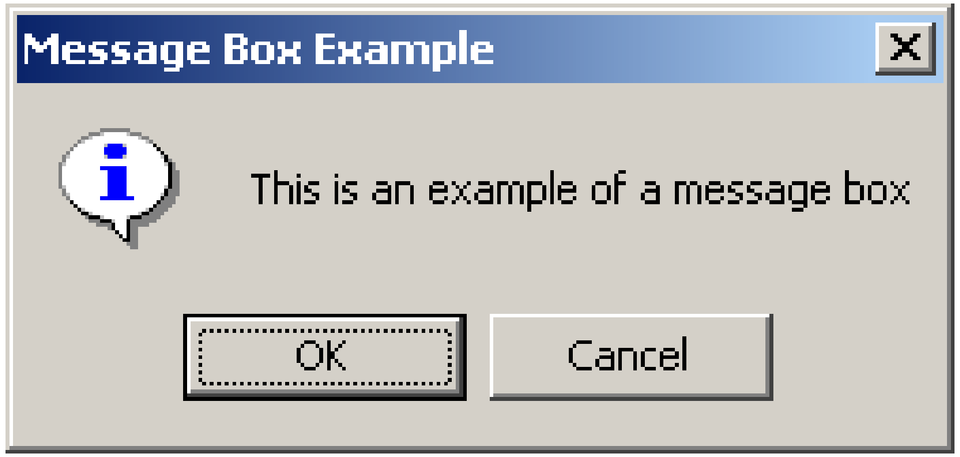
MessageBoxDefaultButton.Member

When you invoke the Show method of the MessageBox function, the function 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

Example 1: - Rectify the code

```
If MessageBox.Show

("This is an example of a message box", "Message Box Example", MessageBoxButtons.OKCancel,
messageBoxIcon.Information, MessageBoxDefaultButton 'everything is OK

Else

  'cancel was pressed

End If
```



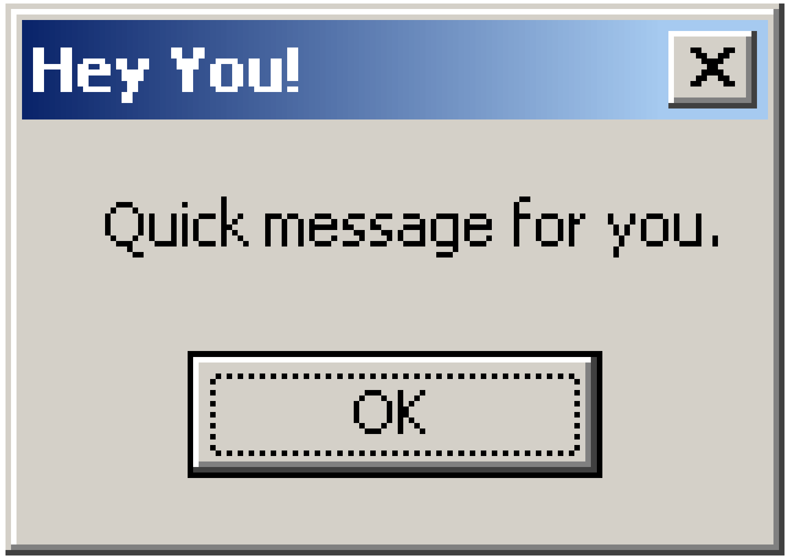
Of course, you would need to add code for the different tasks depending on whether OK or Cancel is clicked by the user.

Example 2:

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:

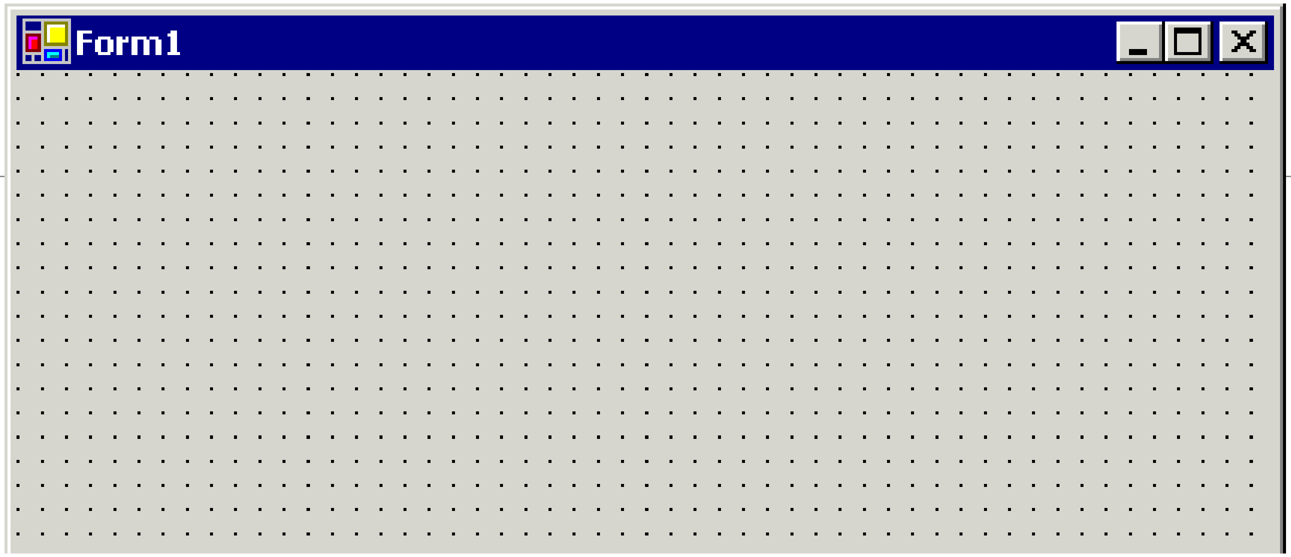
```
MessageBox.Show("Quick message for you.", "Hey You!")
```

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 Visual Basic .NET.

Form Object



The **Form** is the object where the user interface is drawn. It is central to the development of Visual Basic .NET applications.

The form is known as a **container** object, since it 'holds' other controls.

One implication of this distinction is that controls placed on the form will share **BackColor**, **ForeColor** and **Font** properties.

To change this, select the desired control (after it is placed on the form) and change the desired properties. Another feature of a container control is that when its Enabled property is False, all controls in the container are disabled.

Properties, Methods and Events for the form

Recall **properties** described the appearance and value of a control, **methods** are actions you can impose on controls and **events** occur when something is done to the control (usually by a user).

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 **Me**. This is a Visual Basic .NET keyword used to refer to a form. Hence, to close a form, use:

```
Me.Close()
```

Form Events:

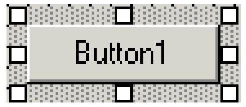
Activated	Occurs when the form is activated in code or by the user.
Click	Occurs when the form is clicked by the user.
Closing	Occurs when the form is closing.
DoubleClick	Occurs when the form is double clicked.
Load	Occurs before a form is displayed for the first time.
Paint	Occurs when the form is redrawn.

To access a form event in the Code window, select the **Base Class Events** object, then the event.

Button Control

In Toolbox:

On Form (Default Properties):



It is probably the most widely used Visual Basic .NET control. It is used to begin, interrupt, or end a particular process.

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 event for a button.

Button Events:

Click	Event triggered when button is selected either by clicking on it or by pressing the access key.
DblClick	Event triggered when user double-clicks on a label.

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

Set the **Name** property. Initialize **Text** to desired string.

Assign **Text** property (String type) in code where needed.

You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

TextBox Control

A **TextBox** control 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.

TextBox Properties:

Name	Gets or sets the name of the text box (three letter prefix for text box name is **txt**).
AutoSize	Gets or sets a value indicating whether the height of the text box automatically adjusts when the font assigned to the control is changed.
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 line	Gets or sets the character used to mask characters of a password in a single-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.
Tag	Stores a string expression.
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.
Copy	Copies selected text to clipboard.
Cut	Moves selected text to clipboard.
Focus	Places the cursor in a specified text box.
Paste	Replaces the current selection in the text box with the contents of the Clipboard.
SelectAll	Selects all text in text box.
Undo	Undoes the last edit operation in the text box.

TextBox Events:

Click	Occurs when the user clicks the text box.
Enter	Occurs when the control receives focus.
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 property value has changed.

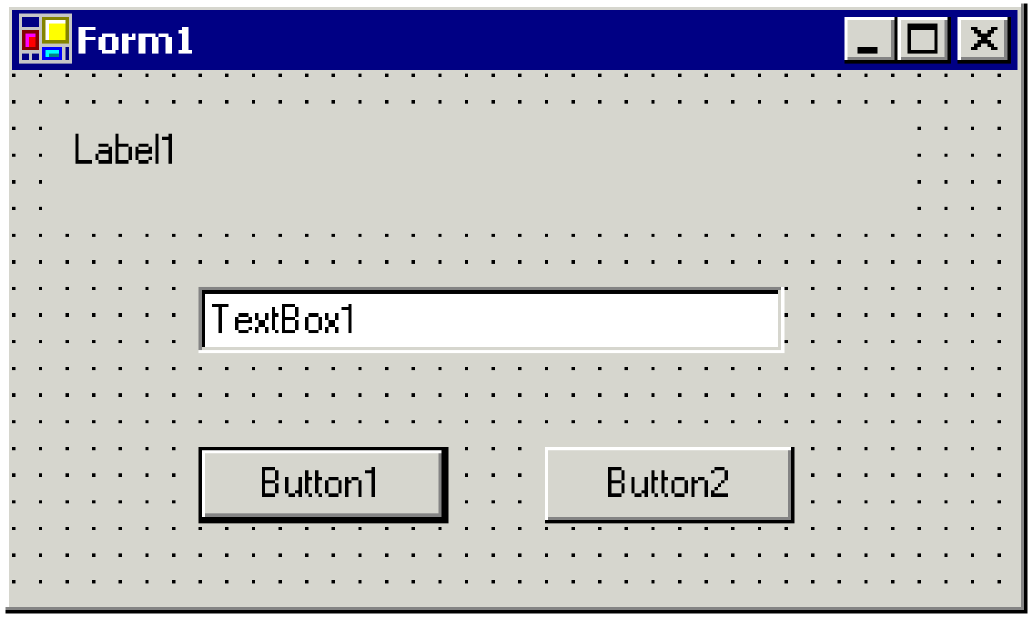
Typical use of **TextBox** control as input device:

- 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.

Example - Password Validation

Start a new project. The idea of this project is to ask the user to input a password. If correct, a message box appears to validate the user. If incorrect, other options are provided.

Place a two buttons, a label box, and a text box on your form so it looks something like this:



Set the properties of the form and each object.

Form1:

Name	frmPassword
AcceptButton	btnValid
CancelButton	btnExit
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Password Validation

Label1:

BorderStyle	Fixed3D
Font Size	10
Font Style	Bold
Text	Please Enter Your Password:
TextAlign	MiddleCenter

Text1:

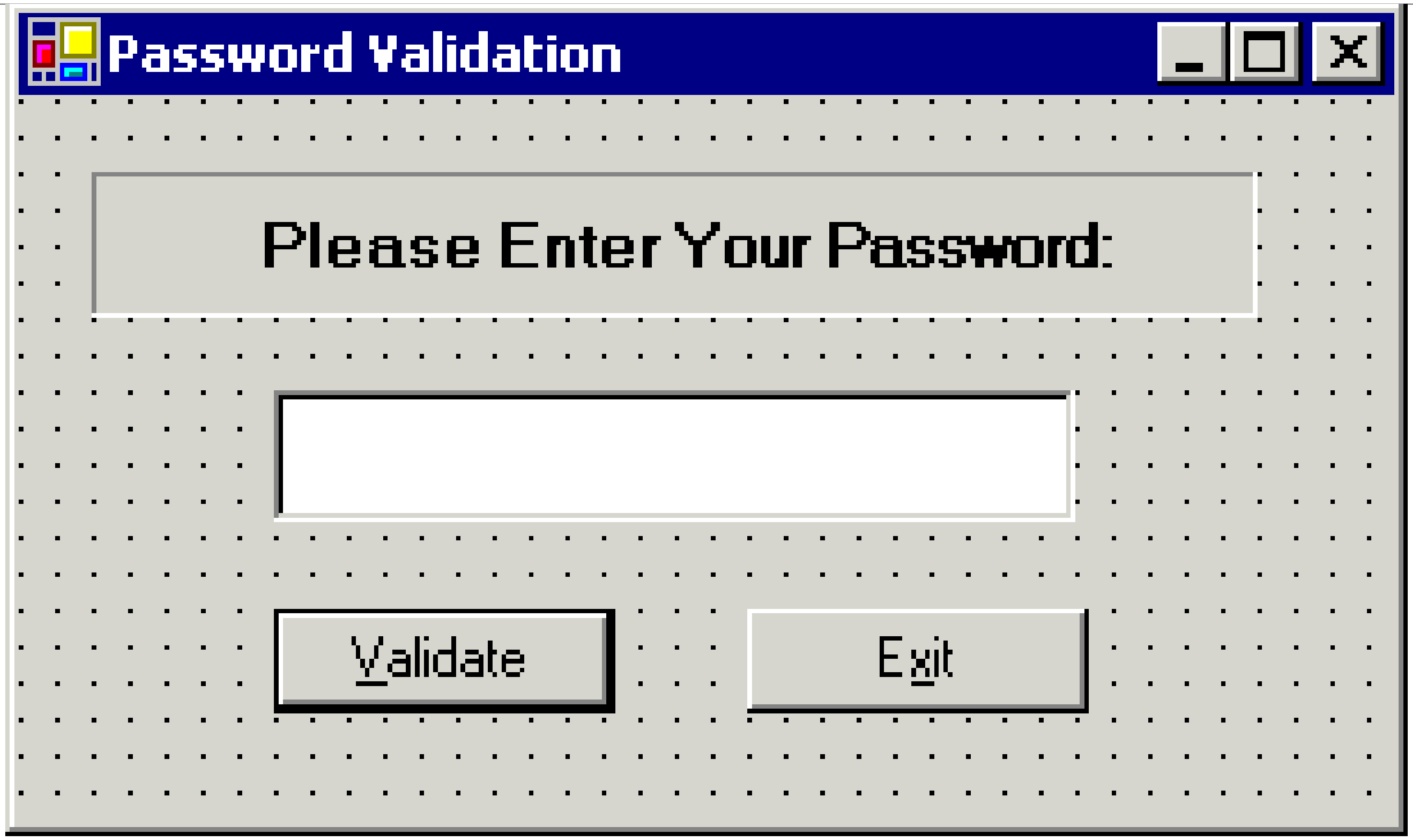
Name	txtPassword
Font Size	14
PasswordChar	*
Tag	[Whatever you choose as a password]
Text	[Blank]

Button1:

Name	btnValid
Text	&Validate

Button2:

Name	btnExit
Text	E&xit


Password Validation
Please Enter Your Password:
Validate
Exit

```
Private Sub btnValid_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnValid.Click
    'This procedure checks the input password
   Dim Response As DialogResult
    If txtPassword.Text = CStr(txtPassword.Tag) Then
      'If correct, display message box
      MessageBox.Show("You've passed security!", "Access Granted", MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
    Else
      'If incorrect, give option to try again
      Response = MessageBox.Show("Incorrect password", "Access Denied", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error)
      If Response = DialogResult.Retry Then
        txtPassword.SelectionStart = 0
        txtPassword.SelectionLength = Len(txtPassword.Text)
      Else
        Me.Close()
      End If
    End If
    txtPassword.Focus()
  End Sub
```

btnValid Click event

This code checks the input password to see if it matches the stored value. If so, it prints an acceptance message.

If incorrect, it displays a message box to that effect and asks the user if they want to try again. If Yes (Retry), another try is granted.

If No (Cancel), the program is ended. Notice the use of **SelectionLength** and **SelectionStart** to highlight an incorrect entry. This allows the user to type right over the incorrect response.

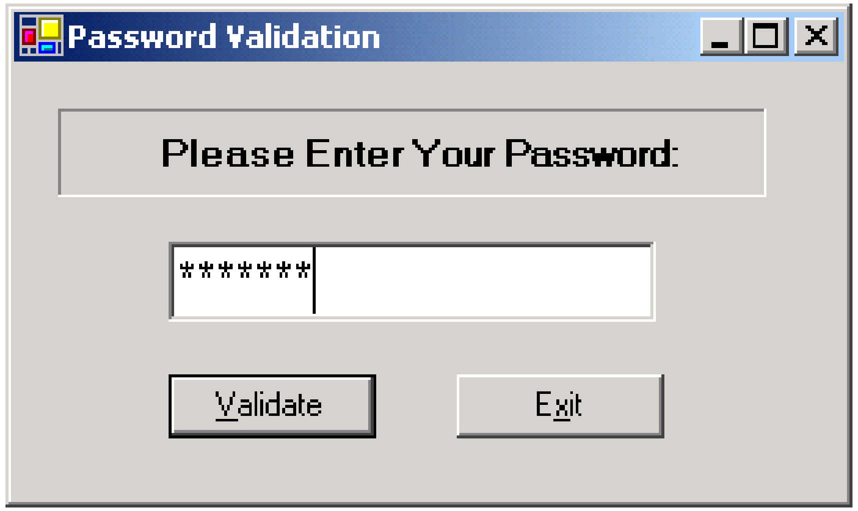
frmPassword_Load event

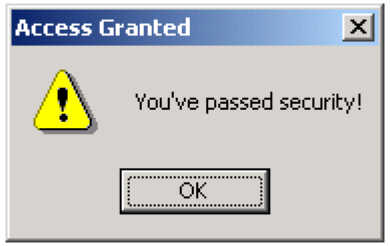
```
Private Sub frmPassword_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load

   'make sure focus starts in text box

   txtPassword.Focus()

  End Sub
```

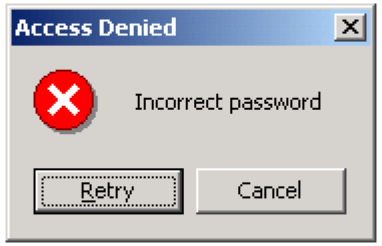
btnExit_ Click event

```
Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
btnExit.Click
    Me.Close()
  End Sub
```

Run the program







Define a constant, TRYMAX = 3, and modify the code to allow the user to have just TRYMAX attempts to get the correct password. After the final try, inform the user you are logging him/her off. You'll also need a variable that counts the number of tries (make it a **Static** variable).

CheckBox Control

The **CheckBox** control provides a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected.

Check boxes are used in all Windows applications (even the Visual Basic .NET IDE).

Examples of their use would be to turn options on and off in an application or to select from a 'shopping' list.

CheckBox Properties:

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

CheckedChange	Occurs when the value of the Checked 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 Basic .NET.

When a check box is clicked, if there is no check mark there (Checked = False), Visual Basic .NET will place a check there and change the Checked property to True. If clicked and a check mark is there (Checked = True), then a check mark will appear and the Checked property will be changed to False.

RadioButton Control

RadioButton controls provide the capability to make a “mutually exclusive” choice among a group of potential candidate choices. This simply means, radio buttons work as a group, only one of which can be selected. Radio buttons are seen in all Windows applications.

They are called radio buttons because they work like a tuner on a car radio – you can only listen to one station at a time! Examples for radio button groups would be twelve buttons for selection of a month in a year, a group of buttons to let you select a color or buttons to select the difficulty in a game.

A first point to consider is how do you define a ‘group’ of radio buttons that work together? Any radio buttons placed on the form will act as a group. That is, if any radio button on a form is ‘selected’, all other buttons will be automatically ‘unselected.’

What if you need to make two independent choices; that is, you need two independent groups of radio buttons? To do this requires one of two grouping controls in Visual Basic .NET: the **GroupBox** control or the **Panel** control. Radio buttons placed on either of these controls are independent from other radio buttons.

RadioButton Properties:

Name	Gets or sets the name of the radio button (three letter prefix for radio button name is **rdo**).
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 event for the button, simulating a click by a user.

RadioButton Events:

CheckedChange	Occurs when the value of the Checked 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 Basic .NET.

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

We've seen that both radio buttons and check boxes usually work as a group. Many times, there are logical groupings of controls. For example, you may have a scroll device setting the value of a displayed number. The **GroupBox** control provides a convenient way of grouping related controls in a Visual Basic .NET application. And, in the case of radio buttons, group boxes affect how such buttons operate.

To place controls in a group box, you first draw the GroupBox control on the form. Then, the associated controls must be placed in the group box. This allows you to move the group box and controls together. There are several ways to place controls in a group box:

Place controls directly in the group box using any of the usual methods.

Draw controls outside the group box and drag them in.

Copy and paste controls into the group box (prior to the paste operation, make sure the group box is selected).

To insure controls are properly place in a group box, try moving it and make sure the associated controls move with it. To remove a control from the group box, simple drag it out of the control.

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).

Panel Control

The **Panel** control is another Visual Basic .NET 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.

Controls are placed in a Panel control in the same manner they are placed in the GroupBox. And, radio buttons in the Panel control act as an independent group. Panel controls can also be used to display graphics (lines, curves, shapes, animations). We will look at those capabilities in later chapters.

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).

Handling Multiple Events in a Single Event Procedure

Now that we are grouping controls like check boxes and radio buttons, it would be nice if a single procedure could handle multiple events.

For example, if we have 4 radio buttons in a group, when one button is clicked, it would be preferable to have a single procedure where we decide which button was clicked, as opposed to having to monitor 4 separate event procedures.

Assume we have four radio buttons (**RadioButton1**, **RadioButton2**, **RadioButton3**, **RadioButton4**). The form for the **CheckedChanged** event procedure for **RadioButton1** is:

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton1.CheckedChanged
          .
          .
End Sub
```

The name assigned to this procedure by Visual Basic .NET (**RadioButton1_CheckedChanged**) is arbitrary. We could change it to anything we want and it would still handle the **CheckedChanged** event for **RadioButton1**. Why, you ask?

The important clause in the procedure structure is the **Handles** statement:

Handles RadioButton1.CheckedChanged

This clause assigns the **RadioButton1.CheckedChanged** event to this particular procedure. This clause cannot be changed. We can assign additional events to this procedure by appending other control events to the **Handles** clause. For example, if we want this procedure to also handle the **CheckedChanged** event for **RadioButton2**, we can change the clause to:

Handles RadioButton1.CheckedChanged, RadioButton2.CheckedChanged

Then if either of these two radio buttons is clicked (changing the Checked property), the procedure named **RadioButton1_CheckedChanged** will be invoked.

The code that does this is:

```
Private Sub MyButtons_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles RadioButton1.CheckedChanged, RadioButton2.CheckedChanged,
RadioButton3.CheckedChanged, RadioButton4.CheckedChanged

        .

        .

End Sub
```

In our example with a single procedure handling the CheckedChanged event for 4 radio buttons, how do we know which of the 4 buttons was clicked to enter the procedure? The **sender** argument of the event procedure provides the answer.

Each event procedure has a **sender** argument (the first argument of two) which identifies the control whose event caused the procedure to be invoked. With a little BASIC coding, we can identify the **Name** property (or any other needed property) of the sending control. For our radio button example, that code is:

```
Dim rdoExample As RadioButton
Dim ButtonName As String
'Determine which button was clicked
rdoExample = CType(sender, RadioButton)
ButtonName = rdoExample.Name
```

Control Arrays

When using controls that work in groups, like check boxes and radio buttons, it is sometimes desirable to have some way to quickly process every control in that group. A concept of use in this case is that of a **control array**.

Say we have 10 check boxes (chkBox01, chkBox02, chkBox03, chkBox04, chkBox05, chkBox06, chkBox07, chkBox08, chkBox09, chkBox10) on a form and we need to examine each check box’s Checked property. If that property is True, we need to process 30 lines of additional code. For one check box, that code would be:

```
If chkBox01.Checked Then
  [do these 30 lines of code]
End If
```

Here’s the solution. Define an array of 10 check box controls and assign the array values to existing controls:

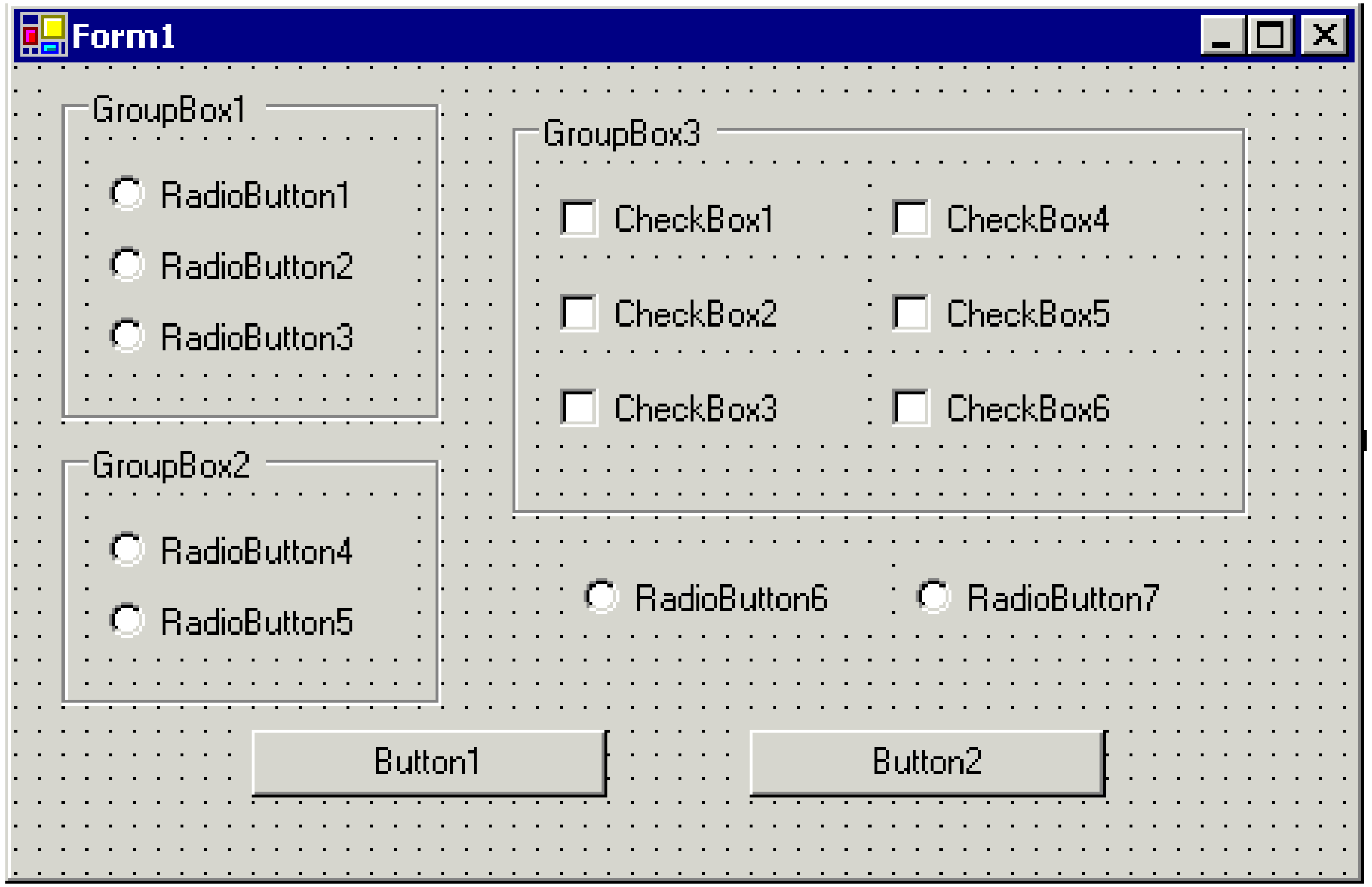
```
Dim MyCheck(10) As CheckBox
    MyCheck(1) = chkBox01
    MyCheck(2) = chkBox02
    MyCheck(3) = chkBox03
    MyCheck(4) = chkBox04
    MyCheck(5) = chkBox05
    MyCheck(6) = chkBox06
    MyCheck(7) = chkBox07
    MyCheck(8) = chkBox08
    MyCheck(9) = chkBox09
    MyCheck(10) = chkBox10
```

Example - Pizza Order

Start a new project. We'll build a form where a pizza order can be entered by simply clicking on check boxes and radio buttons.

Draw three group boxes. In the first, draw three radio buttons, in the second, draw two radio buttons, and in the third, draw six check boxes. Draw two radio buttons on the form.

Add two buttons. Make things look something like this.


Form1
GroupBox1
RadioButton1
RadioButton2
RadioButton3
GroupBox2
RadioButton4
RadioButton5
GroupBox3
CheckBox1
CheckBox4
CheckBox2
CheckBox5
CheckBox3
CheckBox6
RadioButton6
RadioButton7
Button1
Button2

Set the properties of the form and each control

Form1:

Name	frmPizza
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Pizza Order

GroupBox1:

Text	Size

GroupBox2:

Text	Crust Type

GroupBox3

Text	Toppings

RadioButton1:

Name	rdoSmall
Checked	True
Text	Small

RadioButton2:

Name	rdoMedium
Text	Medium

RadioButton3:

Name	rdoLarge
Text	Large

RadioButton4:

Name	rdoThin
Checked	True
Text	Thin Crust

RadioButton5:

Name	rdoThick
Text	Thick Crust

RadioButton6:

Name	rdoIn
Checked	True
Text	Eat In

RadioButton7:

Name	rdoOut
Text	Take Out

CheckBox1:

	Name	chkCheese
	Text	Extra Cheese

CheckBox2:

	Name	chkMushrooms
	Text	Mushrooms

CheckBox3:

	Name	chkOlives
	Text	Black Olives

CheckBox4:

	Name	chkOnions
	Text	Onions

CheckBox5:

	Name	chkPeppers
	Text	Green Peppers

CheckBox6:

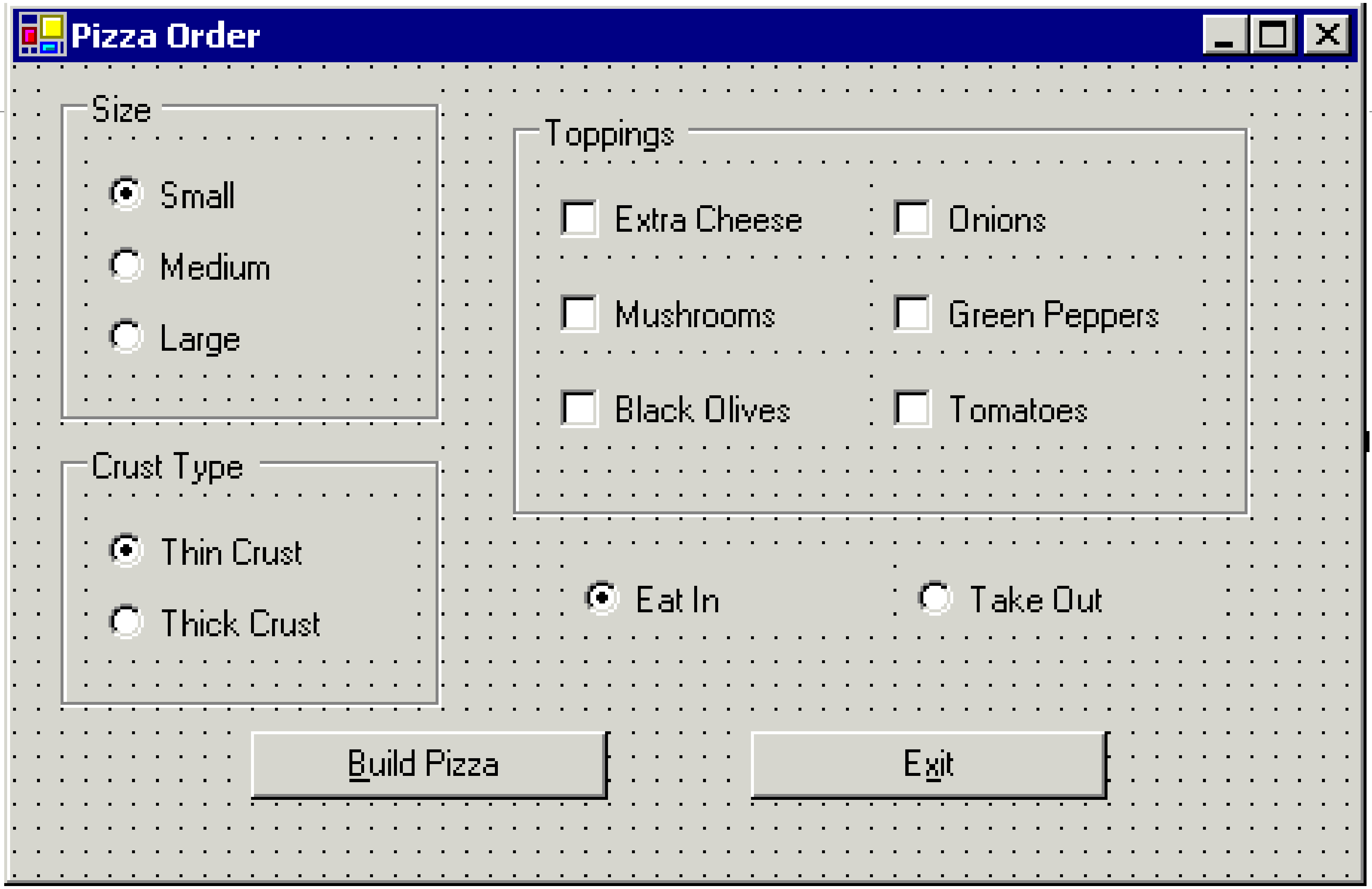
	Name	chkTomatoes
	Text	Tomatoes

Button1:

	Name	btnBuild
	Text	&Build Pizza

Button2:

	Name	btnExit
	Text	E&xit


Pizza Order
Size
Small
Medium
Large
Crust Type
Thin Crust
Thick Crust
Toppings
Extra Cheese
Onions
Mushrooms
Green Peppers
Black Olives
Tomatoes
Eat In
Take Out
Build Pizza
Exit

Form level scope variable declarations:

```
Dim PizzaSize As String
Dim PizzaCrust As String
Dim PizzaWhere As String
Dim Topping(6) As CheckBox
```

This makes the size, crust, and location variables global to the form. The array of check box controls will help us determine which toppings are selected.

Use this code in the **Form_Load** procedure

```
Private Sub frmPizza_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
    'Initialize parameters
    PizzaSize = rdoSmall.Text
    PizzaCrust = rdoThin.Text
    PizzaWhere = rdoIn.Text
    'Define an array of topping check boxes
    Topping(1) = chkCheese
    Topping(2) = chkMushrooms
    Topping(3) = chkOlives
    Topping(4) = chkOnions
    Topping(5) = chkPeppers
    Topping(6) = chkTomatoes
  End Sub
```

Single **CheckedChanged** events

```
Private Sub rdoSize_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles rdoSmall.CheckedChanged, rdoMedium.CheckedChanged,
rdoLarge.CheckedChanged

    Dim rdoSize As RadioButton

    'Determine which button was clicked and change size

    rdoSize = CType(sender, RadioButton)

    PizzaSize = rdoSize.Text

  End Sub
```

```
Private Sub rdoCrust_CheckedChanged(ByVal sender As Object, ByVal e As System.EventArgs)
Handles rdoThin.CheckedChanged, rdoThick.CheckedChanged

    Dim rdoCrust As RadioButton

    'Determine which button was clicked and change crust

    rdoCrust = CType(sender, RadioButton)

    PizzaCrust = rdoCrust.Text

  End Sub
```

```
Private Sub rdoWhere_CheckedChanged(ByVal sender As Object, ByVal e As System.EventArgs)
Handles rdoIn.CheckedChanged, rdoOut.CheckedChanged

    Dim rdoWhere As RadioButton

    'Determine which button was clicked and change where

    rdoWhere = CType(sender, RadioButton)

    PizzaWhere = rdoWhere.Text

  End Sub
```

In each of these routines, when an radio button is clicked (changing the Checked property), the value of the corresponding button’s Text is loaded into the respective variable.

btnBuild_Click event

```
Private Sub btnBuild_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnBuild.Click
  'This procedure builds a message box that displays your pizza type
  Dim Message As String
  Dim I As Integer
  Message = PizzaWhere + ControlChars.Cr
  Message += PizzaSize + " Pizza" + ControlChars.Cr
  Message += PizzaCrust + ControlChars.Cr
  'Check each topping using the array we set up
  For I = 1 To 6
    If Topping(I).Checked Then
      Message += Topping(I).Text + ControlChars.Cr
    End If
  Next I
  MessageBox.Show(Message, "Your Pizza", MessageBoxButtons.OK)
End Sub
```

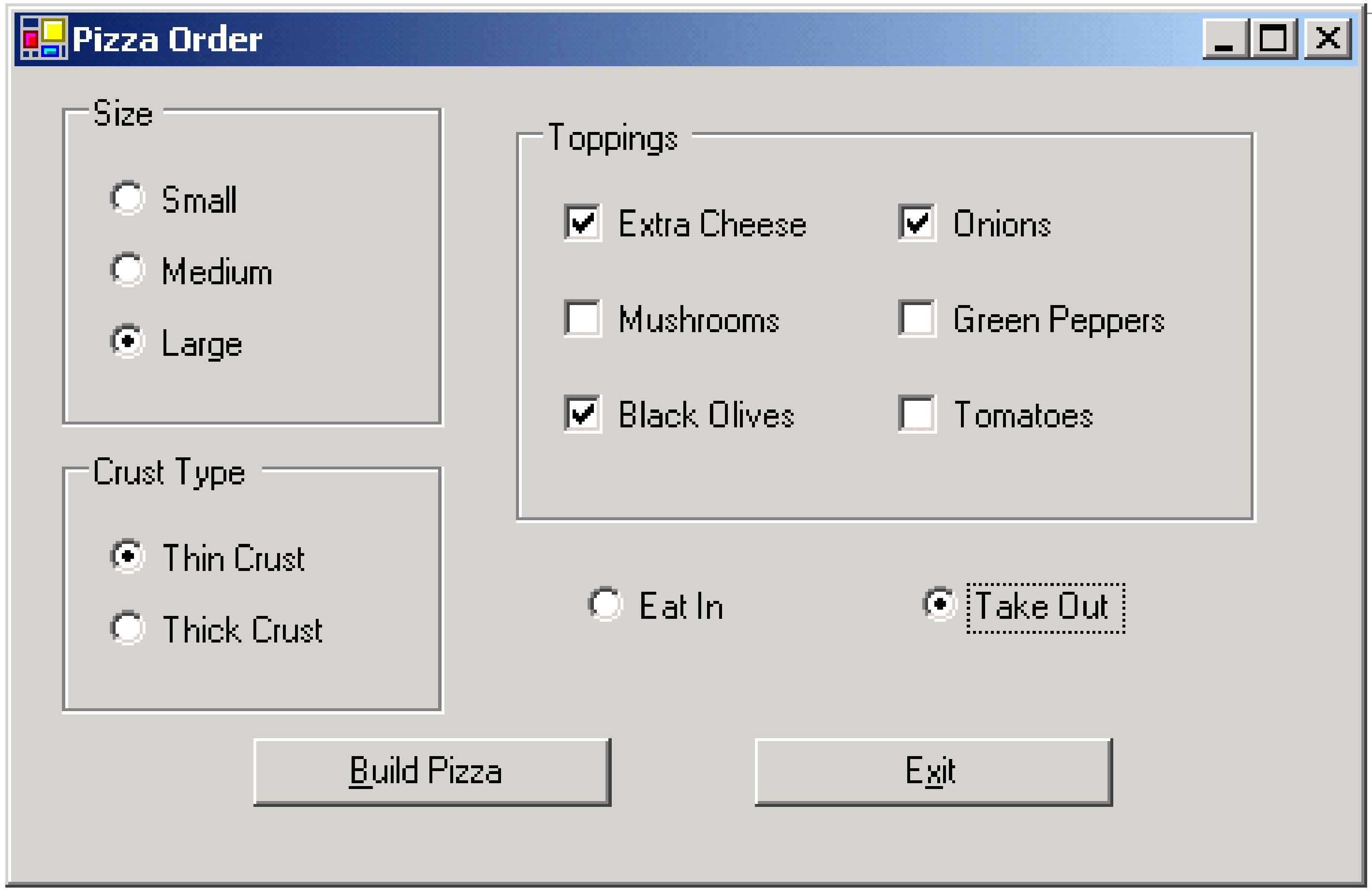
This code forms the first part of a message for a message box by concatenating the pizza size, crust type, and eating location (recall **ControlChars.Cr** is a constant representing a ‘carriage return’ that puts each piece of ordering information on a separate line).

Next, the code cycles through the six topping check boxes (defined by our **Topping** array) and adds any checked information to the message. The code then displays the pizza order in a message box.

btnExit_Click event

```
Private Sub btnExit_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnExit.Click
    Me.Close()
  End Sub
```

Get the application working


Pizza Order

Size

- ( ) Small
- ( ) Medium
- (●) Large

Crust Type

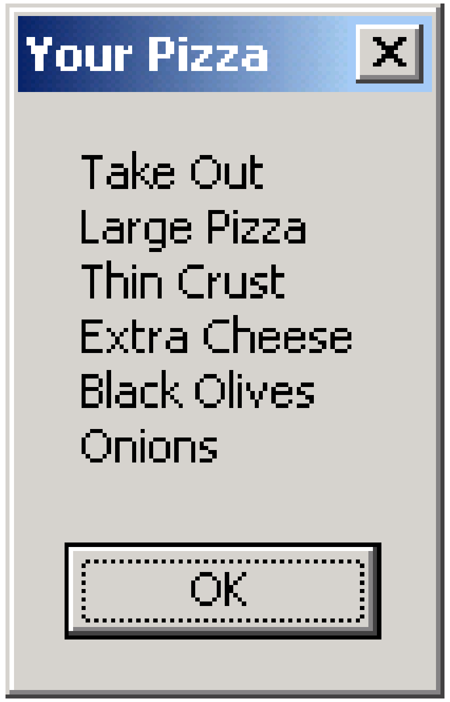
- (●) Thin Crust
- ( ) Thick Crust

Toppings

- [✓] Extra Cheese
- [✓] Onions
- [ ] Mushrooms
- [ ] Green Peppers
- [✓] Black Olives
- [ ] Tomatoes

( ) Eat In [✓ (●) Take Out]

Then, when I click **Build Pizza**


Your Pizza

```
Take Out
Large Pizza
Thin Crust
Extra Cheese
Black Olives
Onions
```

Exercise:

1. Add a new program button that resets the order form to the initial default values. You'll have to reinitialize the three global variables, reset all check boxes to unchecked, and reset all three radio button groups to their default values.
2. Modify the code so that if no toppings are selected, the message "Cheese Only" appears on the order form. You'll need to figure out a way to see if no check boxes were checked.

ListBox Control

Check boxes are useful controls for selecting items from a list. But, what if your list has 100 items? Do you want 100 check boxes? No, but fortunately, there is a tool that solves this problem.

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.

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.
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	Selected Items object of the list box.
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.

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 property has changed.

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 option 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.

ComboBox Control

The **ListBox** control is equivalent to a group of check boxes (allowing multiple selections in a long list of items).

The equivalent control for a long list of radio buttons is the **ComboBox** control. The ComboBox allows the selection of a single item from a list. And, in some cases, the user can type in an alternate response.

ComboBox Properties:

ComboBox properties are nearly identical to those of the ListBox, with the deletion of the **SelectionMode** property and the addition of a **DropDownStyle** property.

Name	Gets or sets the name of the combo box (three letter prefix for combo box name is **cbo**).
BackColor	Get or sets the combo box background color.
DropDownStyle	Specifies one of three combo box styles.
Font	Gets or sets font name, style, size.
ForeColor	Gets or sets color of text.
Items	Gets the Items object of the combo box.
MaxDropDownItems	Maximum number of items to show in dropdown portion.
SelectedIndex	Gets or sets the zero-based index of the currently selected item in list box portion.
SelectedItem	Gets or sets the currently selected item in the list box portion.
SelectedText	Gets or sets the text that is selected in the editable portion of combo box.
Sorted	Gets or sets a value indicating whether the items in list box portion are sorted alphabetically.
Text	String value displayed in combo box.

ComboBox Events:

KeyPress	Occurs when a key is pressed while the combo box has focus.
SelectedIndexChanged	Occurs when the SelectedIndex property has changed.

The **Items** object for the ComboBox control is identical to that of the ListBox control. You add and remove items in the same manner and values are read with the same properties.

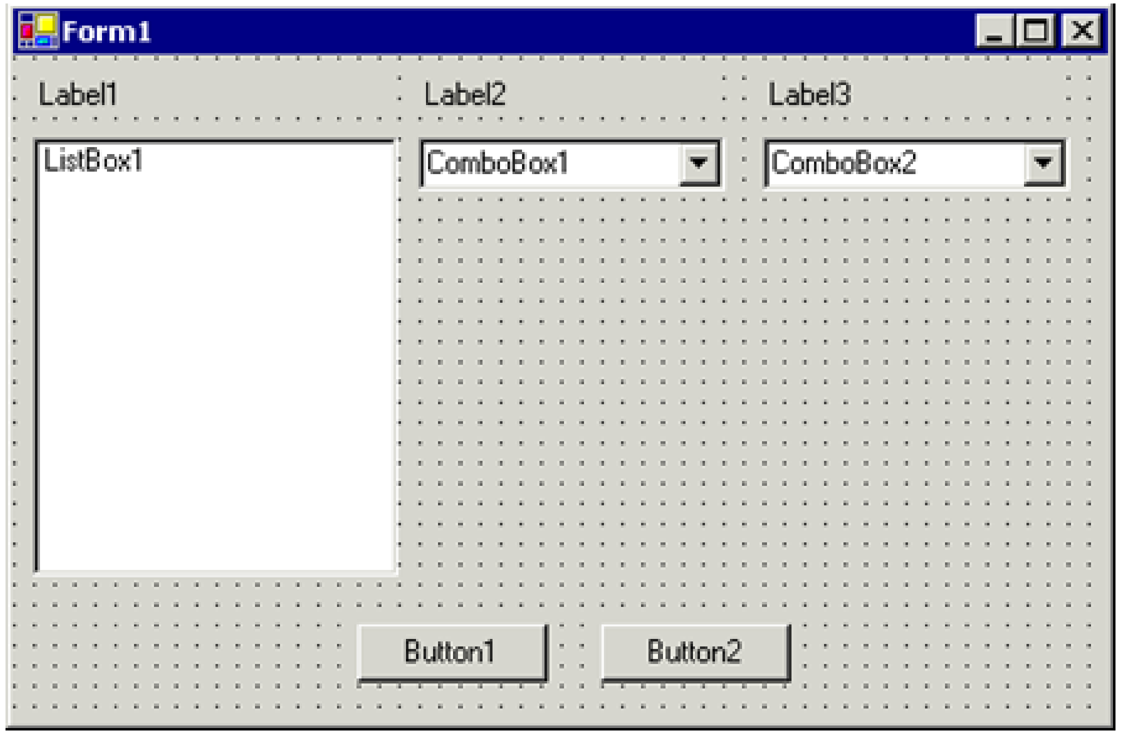
The **DropDownStyle** property has three different values. The values and their description are:

Value	**Description**
DropDown	Text portion is editable; drop-down list portion.
DropDownList	Text portion is not editable; drop-down list portion.
Simple	The text portion is editable. The list portion is always visible. With this value, you'll want to resize the control to set the list box portion height.

Example - Flight Planner

Start a new project. In this example, you select a destination city, a seat location, and a meal preference for airline passengers.

Place a list box, two combo boxes, three labels and two buttons on the form. The form should appear similar to this:



Set the form and object properties:

Form1:

Name	frmFlight
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Flight Planner

ListBox1:

Name	lstCities
Sorted	True

ComboBox1:

Name	cboSeat
DropDownStyle	DropdownList

ComboBox2:

Name	cboMeal
DropDownStyle	Simple
Text	[Blank]

(You'll want to resize this control after setting properties).

Label1:

Text	Destination City

Label2:

Text	Seat Location

Label3:

Text	Meal Preference

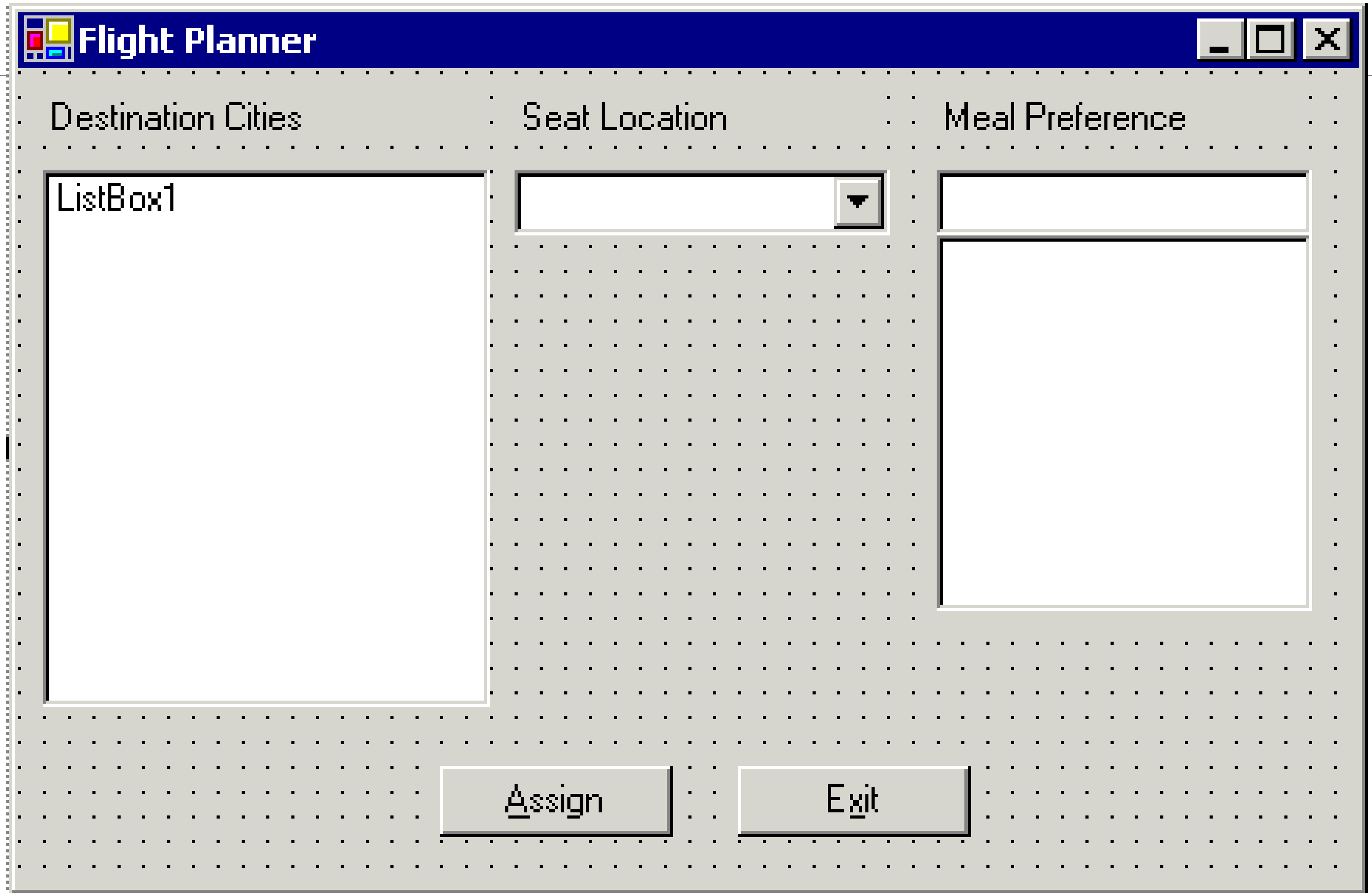
Button1:

Name	btnAssign
Text	&Assign

Button2:

Name	btnExit
Text	E&xit

Now, the form should look like this:



Use this code in the **Form_Load** procedure

```
Private Sub frmPlanner_Load(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MyBase.Load

   'Add city names to list box

   lstCities.Items.Clear()

   lstCities.Items.Add("San Diego")

   lstCities.Items.Add("Los Angeles")

   lstCities.Items.Add("Orange County")

   lstCities.Items.Add("Ontario")

   lstCities.Items.Add("Bakersfield")

   lstCities.Items.Add("Oakland")

   lstCities.Items.Add("Sacramento")

   lstCities.Items.Add("San Jose")
```

```
   lstCities.Items.Add("San Francisco")

   lstCities.Items.Add("Eureka")

   lstCities.Items.Add("Eugene")

   lstCities.Items.Add("Portland")

   lstCities.Items.Add("Spokane")

   lstCities.Items.Add("Seattle")

   lstCities.SelectedIndex = 0
```

```
'Add seat types to first combo box
   cboSeat.Items.Add("Aisle")
   cboSeat.Items.Add("Middle")
   cboSeat.Items.Add("Window")
   cboSeat.SelectedIndex = 0
```

```
'Add meal types to second combo box
   cboMeal.Items.Add("Chicken")
   cboMeal.Items.Add("Mystery Meat")
   cboMeal.Items.Add("Kosher")
   cboMeal.Items.Add("Vegetarian")
   cboMeal.Items.Add("Fruit Plate")
   cboMeal.Text = "No Preference"
 End Sub
```

This code simply initializes the list box and the list box portions of the two combo boxes.

Use this code in the **btnAssign_Click** event

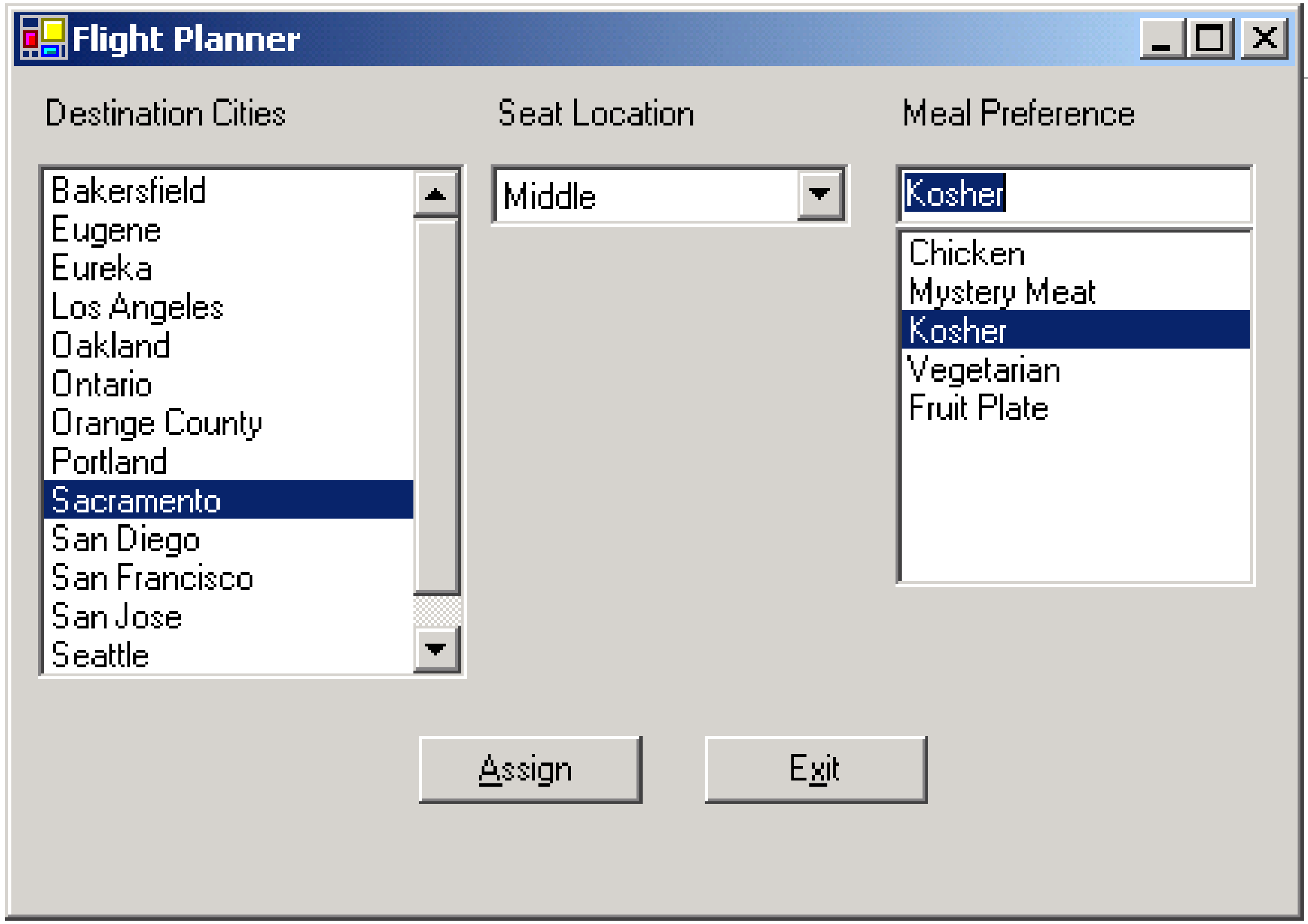
```
Private Sub btnAssign_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnAssign.Click

    'Build message box that gives your assignment

    Dim Message As String

    Message = "Destination: " + lstCities.Text + ControlChars.Cr

    Message += "Seat Location: " + cboSeat.Text + ControlChars.Cr

    Message += "Meal: " + cboMeal.Text + ControlChars.Cr

    MessageBox.Show(Message, "Your Assignment", MessageBoxButtons.OK,
MessageBoxIcon.Information)

  End Sub
```

When the **Assign** button is clicked, this code forms a message box message by concatenating the selected city (from the list box **lstCities**), seat choice (from **cboSeat**), and the meal preference (from **cboMeal**).

Use this code in the **btnExit_Click** event

```
Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
btnExit.Click

    Me.Close()

  End Sub
```

Run the application. Here's my screen with choices I made:



Flight Planner

Destination Cities

- Bakersfield
- Eugene
- Eureka
- Los Angeles
- Oakland
- Ontario
- Orange County
- Portland
- Sacramento
- San Diego
- San Francisco
- San Jose
- Seattle

Seat Location

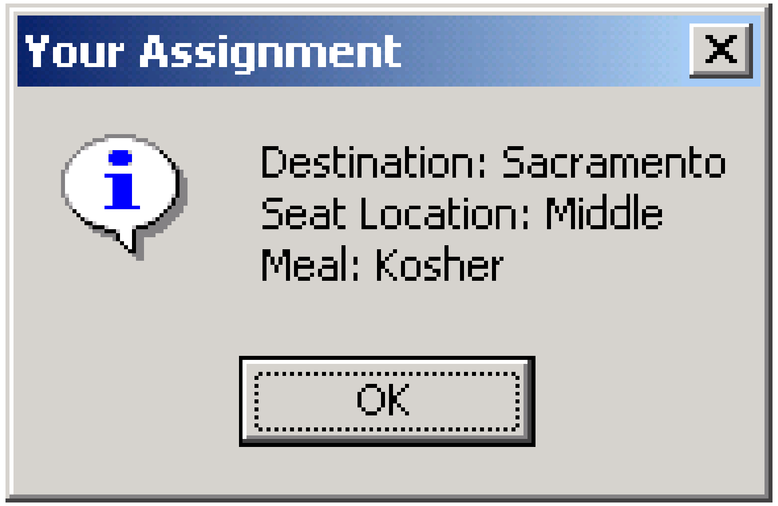
Middle

Meal Preference

Kosher

- Chicken
- Mystery Meat
- Kosher
- Vegetarian
- Fruit Plate

And, after clicking **Assign**, I see:



Your Assignment

Destination: Sacramento
Seat Location: Middle
Meal: Kosher

After completing this class, you should understand:

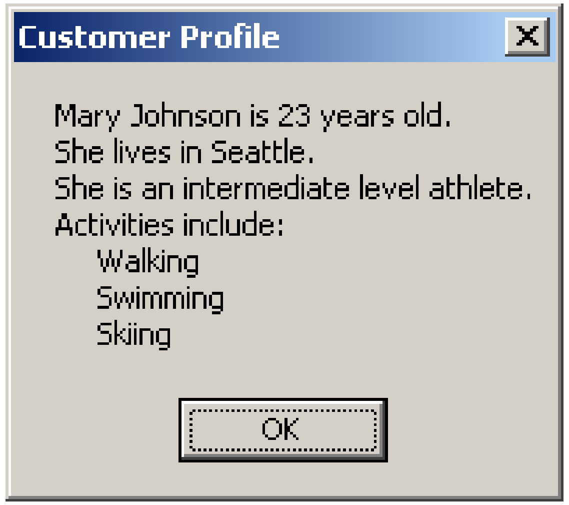
- How to use the MessageBox dialog assigning messages, icons and buttons
- Useful properties, events, and methods for the form, button, text box, label, check box, and radio button controls
- Where the above listed controls can and should be used
- How GroupBox and Panel controls are used to group controls, particularly radio buttons
- How several events can be handled by a single event procedure
- The concept of ‘control arrays’ and how to use them
- How to use list box and combo box controls

Exercise - Customer Database Input Screen

Customer Database Input Screen

A new sports store wants you to develop an input screen for its customer database. The required input information is:

1. Name
2. Age
3. City of Residence
4. Sex (Male or Female)
5. Activities (Running, Walking, Biking, Swimming, Skiing and/or In-Line Skating)
6. Athletic Level (Extreme, Advanced, Intermediate, or Beginner)



```
Mary Johnson is 23 years old.
She lives in Seattle.
She is an intermediate level athlete.
Activities include:
    Walking
    Swimming
    Skiing
```

Set up the screen so that only the Name and Age (use text boxes) and, perhaps, City (use a combo box) need to be typed; all other inputs should be set with check boxes and radio buttons. When a screen of information is complete, display the summarized profile in a message box. This profile message box should resemble this: