

# Visual Basic .NET Fundamentals

part II

*Wiktor Tracz*

[wiktor.tracz@wl.sggw.pl](mailto:wiktor.tracz@wl.sggw.pl)

*Faculty of Forestry, Warsaw University of Life Sciences, Poland*

# Lectures

1. Working with controls: *use some of the more advanced standard controls, common dialog controls.*
  2. Manipulating files and directories: *displaying and changing directory, reading from and writing to a file.*
  3. Error handling: *how to handle errors in a way that avoid disruption of the program and users' work.*
  4. Enhancing the user interface – menus and toolbars: *giving an application a more professional look through the use of menus, toolbars, and status bars.*
  5. Accessing data stored in database.
- \* Principles of user interface design.

# Visual Basic naming conventions

*In Visual Basic application, the first character of that name must be an alphabetic character, a digit, or an underscore. Note, however, that names beginning with an underscore are not compliant with the Common Language Specification (CLS).*

The following suggestions apply to naming.

- Begin each separate word in a name with a capital letter, as in **FindLastRecord** and **RedrawMyForm**.
- Begin function and method names with a verb, as in **InitNameArray** or **CloseDialog**.
- Begin class, structure, module, and property names with a noun, as in **EmployeeName** or **CarAccessory**.
- Begin interface names with the prefix "I", followed by a noun or a noun phrase, like **IComponent**, or with an adjective describing the interface's behavior, like **IPersistable**. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.

# Visual Basic naming conventions

- Begin event handler names with a noun describing the type of event followed by the "EventHandler" suffix, as in "MouseEventHandler".
- In names of event argument classes, include the "EventArgs" suffix.
- If an event has a concept of "before" or "after," use a prefix in present or past tense, as in "ControlAdd" or "ControlAdded".
- For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" can lead to confusion.
- Avoid using names in an inner scope that are the same as names in an outer scope. Errors can result if the wrong variable is accessed. If a conflict occurs between a variable and the keyword of the same name, you must identify the keyword by preceding it with the appropriate type library. For example, if you have a variable called Date, you can use the intrinsic **Date** function only by calling System.DateTime.Date.

# Me, My, MyBase, and MyClass in VB

The apparent similarities between **Me**, **My**, **MyBase**, and **MyClass** in Visual Basic may confuse you if you are coming to the concepts for the first time. This page describes each of these entities in order to distinguish between them.

## Me

The **Me** keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. **Me** behaves like either an object variable or a structure variable referring to the current instance. Using **Me** is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure, or module.

## My

The **My** feature provides easy and intuitive access to a number of .NET Framework classes, enabling the Visual Basic user to interact with the computer, application, settings, resources, and so on.

## MyBase

The **MyBase** keyword behaves like an object variable referring to the base class of the current instance of a class. **MyBase** is commonly used to access base class members that are overridden or shadowed in a derived class. **MyBase.New** is used to explicitly call a base class constructor from a derived class constructor.

## MyClass

The **MyClass** keyword behaves like an object variable referring to the current instance of a class as originally implemented .






# Working with controls

- Overview of some standard controls.
- Using **ComboBox** and **ListBox** controls.
- Using **RadioButton** and **GroupBox** controls.
- Working with selected text.
- Common Dialog controls and **Timer** control.

# Kinds of controls





Kind	Description
Standard controls	These controls are contained in Visual Basic. Examples include the <b>Button</b> and <b>TextBox</b> controls. Also called intrinsic controls, standard controls are always available from the Toolbox and are the main focus of this course.
ActiveX controls	These controls are separate files with the .OCX extension. These controls can be added to the Toolbox.
Extended controls	Controls which can be derived from any existing Windows Forms control. All of the inherent functionality of a Windows Forms control can be retained, and then extend that functionality by adding custom properties, methods, or other features.

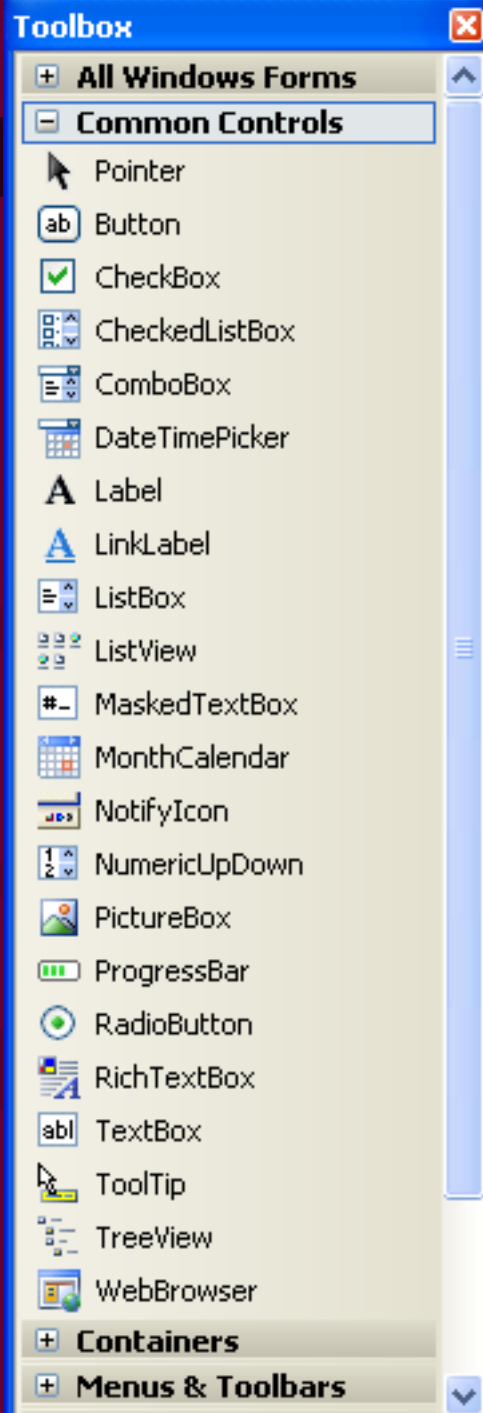
# Basic standard controls

Control	Description
Label 	The <b>Label</b> control most commonly <u>displays text</u> that the user cannot change.
TextBox 	The <b>TextBox</b> control <u>obtains information from the user</u> or displays information provided by the application.
Button 	The <b>Button</b> control <u>performs a task when the user clicks</u> the control.
CheckBox 	The <b>CheckBox</b> control <u>presents a single option or several different options</u> to the user, any or all of which may be selected. It can be turned on or off by the user.
RadioButton 	The <b>RadioButton</b> control <u>presents a group of options from which the user selects only one</u> . It can be turned on or off by the user.

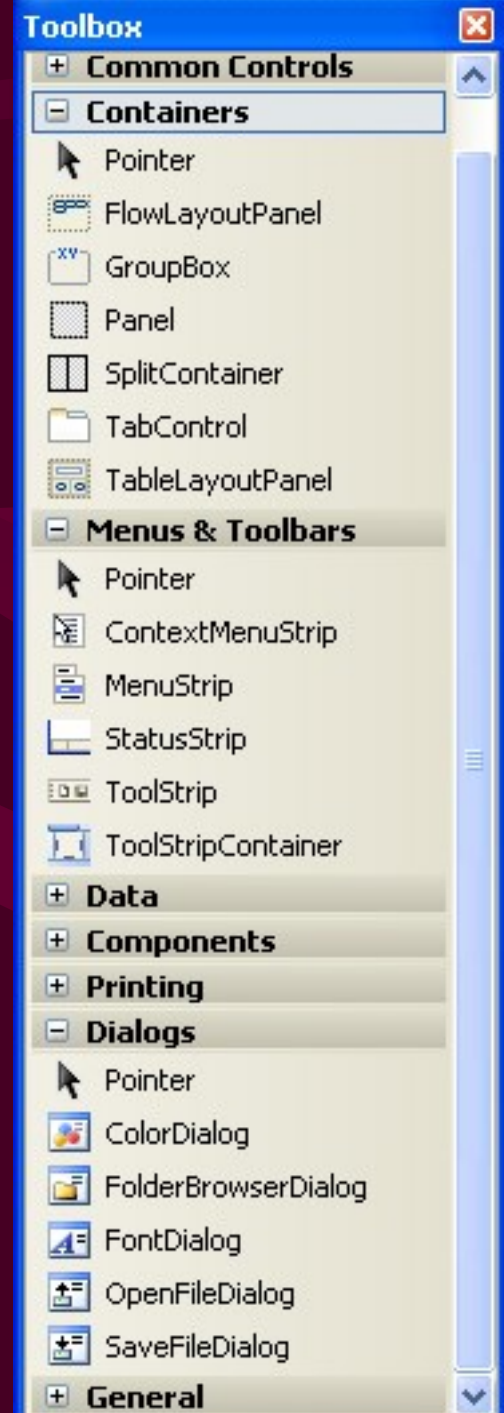


# Basic standard controls (cd.)

<b>GroupBox</b> 	The <b>GroupBox</b> control <u>groups together related controls</u> , either visually or functionally. A common use for the <b>Frame</b> control is to group several option buttons together on a form.
<b>PictureBox</b> 	The <b>PictureBox</b> control <u>displays a graphic or text</u> . A <b>PictureBox</b> control can also be used to display animated graphics.
<b>ComboBox</b> 	The <b>ComboBox</b> control <u>combines the features</u> of a <b>TextBox</b> control and a <b>ListBox</b> control. Users can enter information in the text box portion of the control or select an item from the list box portion. A <b>ComboBox</b> control can also be used to create a drop-down list box.
<b>ListBox</b> 	The <b>ListBox</b> control <u>displays a list</u> from which the user can select one or more items.



# Toolbox

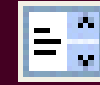


# ComboBox control



- **ComboBox** control is used to display data in a drop-down combo box.
- By default, the **ComboBox** control appears in two parts:
  - the top part is a text box that allows the user to type a list item,
  - the second part is a list box that displays a list of items from which the user can select one

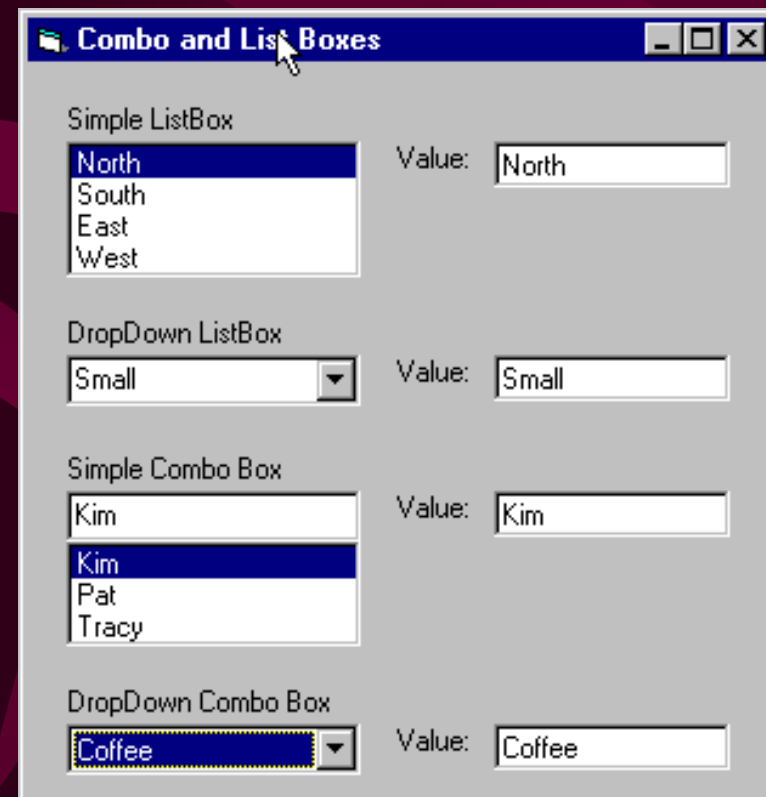
# ListBox control



- **ListBox** control displays a list from which the user can select one or more items.
- If the total number of items exceeds the number that can be displayed, a scroll bar is automatically added to the **ListBox** control.
- When the **MultiColumn** property is set to **false**, the list box displays items in a single column and a vertical scroll bar appears.

# Using ComboBox and ListBox controls

- List boxes and combo boxes present the user with choices.
- A **ComboBox** is appropriate when there is a list of suggested choices.
- A **ListBox** is appropriate when you want to limit input to what is on the list.
- A **ComboBox** contains an edit field, so choices not on the list can be typed in this field.
- Combo boxes save space on a form, because the full list is not displayed until the user clicks the down arrow, a combo box can easily fit in a small space where a list box would not fit.



# Adding items to a list

## Adding items at Design Time

To add items to a combo or list box at design time, set the Items property. Click  button on the right side of **Items** in the Property box, then add each item.

## Adding items at Run Time

To add items to a combo or list box at run time, use the Add method in code. The collection is referenced using the Items property:

```
myComboBox.Items.Add("North")
```

or

Insert the string or object at the desired point in the list with the Insert method:

```
myCheckedListBox.Items.Insert(0, "South")
```

# Working with selected items

- To access the text that has been selected in a **ComboBox** control, use the **SelectedItem** and **SelectedText** properties.
- **SelectedItem** gets or sets currently selected item in the **ComboBox**.
- **SelectedText** gets or sets the text that is selected in the editable portion of a **ComboBox**.

```
strMyText = myComboBox.SelectedItem
```

- If **DropDownStyle** property is set to **DropDownList**, the return value is an empty string ("").
- **Note** *These properties are available at run time only and are not displayed in the Properties window.*

# Assign an entire array to the **Items** collection

```
Dim myItemObject(9) As System.Object
Dim i As Integer
    For i = 0 To 9
        myItemObject(i) = "Item" & i
    Next i
myListBox.Items.AddRange(myItemObject)
```



# Removing items from a list

- Call the `Remove` or `RemoveAt` method to delete items.
- `Remove` has one argument that specifies the item to remove.
- `RemoveAt` removes the item with the specified index number.

```
' To remove item with index 0:  
myComboBox.Items.RemoveAt(0)
```

```
' To remove currently selected item:  
myComboBox.Items.Remove(myComboBox.SelectedItem)
```

```
' To remove "North" item:  
myComboBox.Items.Remove("North")
```

# Accessing items in combo box and list box

Query the **Items** collection using the index of the specific item:

```
Private Function GetItemText(i As Integer) As String
' Return the text of the item using the index:
    Return myComboBox.Items(i).ToString
End Function
```

# Data binding

**ComboBox** and **ListBox** can be bound to data to perform tasks such as browsing data in a database, entering new data, or editing existing data.

To **bind** a **ComboBox** or **ListBox** control:

1. Set the **DataSource** property to a data source object. Possible data sources bound to: data, a data table, a data view, a dataset, a data view manager, an array.
2. If binding to a table, set the **DisplayMember** property to the name of a column in the data source.
3. If binding to an **IList**, set the display member to a public property of the type in the list.

```
Private Sub BindComboBox()  
    myComboBox.DataSource = myDataSet.Tables("Suppliers")  
    myComboBox.DisplayMember = "ProductName"  
End Sub
```

# Sorting contents

- Windows Forms controls do not sort when they are data-bound.
- To display sorted data, use a data source that supports sorting and then have the data source sort it.
- Data sources that support sorting are data views, data view managers, and sorted arrays.
- To sort the list:  
    set the **Sorted** property to **True**

# RadioButton control



- **RadioButton** controls present a set of two or more mutually exclusive choices to the user.
- When a user selects a radio button, the other radio buttons in the same group cannot be selected as well.
- It is possible to choose one and only one choice from the set of choices.
- When a **RadioButton** control is clicked, its **Checked** property is set to **true** and the **Click** event handler is called.
- The **CheckedChanged** event is raised when the value of the **Checked** property changes.
- The text displayed within the control is set with the **Text** property, which can contain access key shortcuts.

# Using the **RadioButton** controls

```
Sub SubmitBtn_Click(Sender As Object, e As EventArgs)
    If myRadio1.Checked Then
        myLabel.Text = "You selected " & myRadio1.Text
    ElseIf myRadio2.Checked Then
        myLabel.Text = "You selected " & myRadio2.Text
    ElseIf myRadio3.Checked Then
        myLabel.Text = "You selected " & myRadio3.Text
    End If
End Sub
```

# CheckBox control



- **CheckBox** control indicates whether a particular condition is on or off.
- Check box controls are used in groups to display multiple choices from which the user can select one or more.
- In opposite to **RadioButton**, with the check box control any number of check boxes may be selected.
- The **Checked** property indicates whether the **CheckBox** is in the checked state (returns either **True** or **False**).
- **CheckState** property gets or sets the state of the **CheckBox**

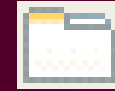
# GroupBox control



- **GroupBox** controls are used to provide an identifiable grouping for other controls.
- Group boxes are used to subdivide a form by function.
- There are three reasons to group controls:
  - to create a visual grouping of related form elements for a clear user interface,
  - to create programmatic grouping (of radio buttons, for example),
  - for moving the controls as a unit at design time.
- **GroupBox** control is similar to the **Panel** control; however, only the **GroupBox** control displays a caption, and only the **Panel** control can have scroll bars.
- Set the **Text** property of the group box to an appropriate caption.



# TabControl control



- Displays multiple tabs, like dividers in a notebook or labels in a set of folders in a filing cabinet.
- Tabs can contain pictures and other controls.
- Use the **TabControl** to produce the kind of multiple-page dialog box that appears many places in the Windows operating system.
- **TabControl** can be used to create property pages, which are used to set a group of related properties.

# Working with TabControl

- By default, a **TabControl** control contains two **TabPage** controls.
- Each individual tab is a **TabPage** object. When a tab is clicked, it raises the Click event for that **TabPage** object.
- The most important property of the **TabControl** is **TabPage**s, which contains the individual tabs.
- Set programmatically the **Enabled** property to enable or disable the **TabPage** control.
- Set the **Appearance** property to **Buttons** or **FlatButtons** value to display tabs as buttons.

# Using the InputBox function

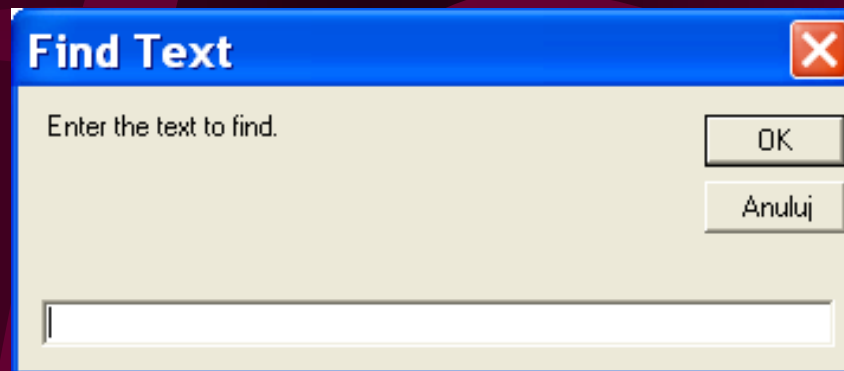
- Displays a prompt in a dialog box, waits for the user to input text or click a button, and then returns a string containing the contents of the text box.

```
Public Function InputBox( _  
    ByVal Prompt As String, _  
    Optional ByVal Title As String = "", _  
    Optional ByVal DefaultResponse As String = "", _  
    Optional ByVal Xpos As Integer = -1, _  
    Optional ByVal YPos As Integer = -1 _  
) As String
```

- Prompt* – required string expression displayed as the message in the dialog box.
- Title* – string expression displayed in the title bar of the dialog box.
- DefaultResponse* – string expression displayed in the text box as the default response if no other input is provided. If you omit *DefaultResponse*, the displayed text box is empty.
- Xpos*, *Ypos* – numeric expression that specifies, in twips, the position of upper left corner of InputBox window.

# InputBox example

```
InputBox("Enter the text to find.", "Find Text")
```



# Common dialog controls



**OpenFileDialog** – prompts the user to open a file



**SaveFileDialog** – prompts the user to select a location for saving a file



**FolderBrowserDialog** – prompts the user to select a folder

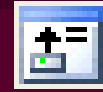


**FontDialog** – prompts the user to choose a font from among those installed on the local computer



**ColorDialog** – represents a common dialog box that displays available colors along with controls that enable the user to define custom colors

# OpenFileDialog control



- **OpenFileDialog** component is a pre-configured dialog box. It is the same **Open File** dialog box exposed by the Windows operating system.
- When using the **OpenFileDialog** component, you must write your own file-opening logic.
- Use the ShowDialog method to display the dialog at run time.
- Multiselect property enables opening multi-selected files.
- Filter property sets the current file name filter string, which determines the choices that appear in the "Files of type" box in the dialog box.
- Use CheckFileExists property or File.Exists method to determine whether the specified file exists.
- InitialDirectory property gets or sets the initial directory displayed by the file dialog box.

# Open files using the OpenFileDialog

- The dialog box returns the path and name of the file the user selected in the dialog box.
- Once the user has selected the file to be opened, there are two approaches to the mechanism of opening the file:
  - 1) create an instance of the **StreamReader** class
  - 2) use **OpenFile** method to open the selected file
- First technique can be used from the local machine, Intranet, and Internet zones.
- Second method is better suited for applications in the Intranet or Internet zones.
- A button's DialogResult property gets or sets a value that is returned to the parent form when the button is clicked.

# Open a file as a stream

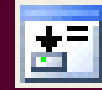
- Display the **Open File** dialog box and call a method to open the file selected by the user.
- Use the **ShowDialog** method to display the Open File dialog box, and use an instance of the **StreamReader** class to open the file.
- The next example uses the **Button** control's Click event handler to open an instance of the **OpenFileDialog** component.
- When a file is chosen and the user clicks **OK**, the file selected in the dialog box opens. In this case, the contents are displayed in a message box, just to show that the file stream has been read.



# Open a file as a stream – example

```
Private Sub myButton_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles myButton.Click  
    ' Pairs of filters, separated with "|".  
    ' Each pair consists of a description|file spec. Use a "|" between  
    pairs.  
    myOpenFileDialog.Filter = "Text files (*.txt)|*.txt|All files|*.*"  
    ' To be sure that the selected by user path exists.  
    myOpenFileDialog.CheckPathExists = True  
    myOpenFileDialog.Title = "Select a file to open"  
    If myOpenFileDialog.ShowDialog() = Windows.Forms.DialogResult.OK Then  
        Dim sr As New System.IO.StreamReader(myOpenFileDialog.FileName)  
        Dim myText As String  
        myText = sr.ReadToEnd  
        MessageBox.Show(myText) 'ReadToEnd method allows to read the stream  
        sr.Close()  
    End If  
End Sub
```

# SaveFileDialog control



- **SaveFileDialog** component is a pre-configured dialog box. It is the same as the standard **Save File** dialog box used by Windows.
- It is used as a simple solution for enabling users to save files instead of configuring your own dialog box.
- When using the **SaveFileDialog** component, you must write your own file-saving logic.
- ShowDialog method can be used to display the dialog box at run time.

# Saving a file using the **SaveFileDialog**

- The **SaveFileDialog** component allows users to browse the file system and select files to be saved.
- The dialog box returns the path and name of the file the user has selected in the dialog box. However, you must write the code to actually write the files to disk.
- To save a file: display the **Save File** dialog box and call a method to save the file selected by the user.
- A button's DialogResult property gets or sets a value that is returned to the parent form when the button is clicked.
- The OK value of the DialogResult property can be used to check whether **OK** button was clicked and then to get the name of the file.

# Example of saving a file

```
' If the user doesn't supply an extension, and if the AddExtension
  property
' is True, use this extension. The default is "".
mySaveFileDialog.DefaultExt = "txt"
' To be sure that the selected by user path exists.
mySaveFileDialog.CheckPathExists = True
mySaveFileDialog.Title = "Save a text file"
' Pairs of filters, separated with "|".
' Each pair consists of a description|file spec. Use a "|" between pairs.
mySaveFileDialog.Filter = "Text files (*.txt)|*.txt|All files|*.*"
' Show the Help button and Read-Only checkbox?
mySaveFileDialog.ShowHelp = True
If mySaveFileDialog.ShowDialog() = Windows.Forms.DialogResult.OK Then
    My.Computer.FileSystem.WriteAllText(mySaveFileDialog.FileName, _
    txbFileContents.Text, False)
End If
```

# FolderBrowserDialog control



- **FolderBrowserDialog** component is a modal dialog box that is used for browsing and selecting folders.
- The **FolderBrowserDialog** component is displayed at run time using the ShowDialog method.
- Set the **RootFolder** property to determine the top-most folder and any subfolders that will appear within the tree view of the dialog box.
- Once the dialog box has been shown, you can use the **SelectedPath** property to get the path of the folder that was selected.
- New folders can also be created from within the **FolderBrowserDialog** component.

# Choosing directory using **FolderBrowserDialog**

- Check the value of DialogResult property to see how the dialog box was closed. If an user accept choice (press OK. button) the OK value is returned.
- The value of the SelectedPath property contains the path of the folder that was selected.

```
Public Sub ChooseFolder()  
    If myFolderBrowserDialog.ShowDialog() = _  
        Windows.Forms.DialogResult.OK Then  
        myTextBox.Text = myFolderBrowserDialog.SelectedPath  
    End If  
End Sub
```

# FontDialog controls



- The control has a number of properties that configure font appearance. The properties that set the dialog-box selections are **Font** and **Color**.
- By default, the dialog box shows list boxes for *Font*, *Font style*, and *Size*; check boxes for effects like *Strikeout* and *Underline*; a drop-down list for *Script*; and a sample of how the font will appear.
- To display the font dialog box, call the **ShowDialog** method.

## Example 1

```
If myFontDialog.ShowDialog = Windows.Forms.DialogResult.OK Then
    myTextBox.Font = myFontDialog.Font
End If
```

## Example 2

```
If FontDialog1.ShowDialog <> DialogResult.Cancel Then
    textBox1.Font = FontDialog1.Font
    textBox1.ForeColor = FontDialog1.Color
End If
```

# ColorDialog control



- **ColorDialog** component is a pre-configured dialog box that allows the user to select a color from a palette and to add custom colors to that palette. Use it within a Windows-based application as a simple solution in configuration your own dialog box.
- The color selected in the dialog box is returned in the Color property.
- If the **AllowFullOpen** property is set to **False**, the "Define Custom Colors" button is disabled and the user is restricted to the predefined colors in the palette.
- If the **SolidColorOnly** property is set to **True**, the user cannot select dithered colors.
- To display the dialog box, you must call its **ShowDialog** method.

```
If myColorDialog.ShowDialog() = DialogResult.OK Then  
    myTextBox.ForeColor = myColorDialog.Color  
End If
```



# Timer control



- Provides a mechanism for executing a method at specified intervals. This component is designed for a Windows Forms environment.
- **Timer** is a component that raises an event at regular intervals.
- The length of the intervals is defined by the **Interval** property, whose value is in milliseconds.
- When the **Time** component is enabled, the **Tick** event is raised every interval. This is where you would add code to be executed.
- The key methods of the **Timer** component are **Start** and **Stop**, which turn the timer on and off.
- When the timer is switched off, it resets; there is no way to pause a **Timer** component. Setting the interval to **0** does not cause the timer to stop.

# Using Timer control

```
Private Sub InitializeTimer()  
    ' Set to 1 second (1000 ms) interval.  
    myTimer.Interval = 1000  
    ' Enable timer.  
    myTimer.Enabled = True  
End Sub  
  
Private Sub myTimer_Tick(ByVal Sender As Object, _  
    ByVal e As EventArgs) Handles myTimer.Tick  
    ' Set the caption to the current time.  
    lblShawTime.Text = DateTime.Now  
End Sub  
  
Private Sub myButton_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles myButton.Click  
    InitializeTimer()  
    If  
        myButton.Text = "Stop" Then  
        myButton.Text = "Start"  
        myTimer.Enabled = False  
    Else  
        myButton.Text = "Stop"  
        myTimer.Enabled = True  
    End If  
End Sub
```

# Manipulating files and directories

- Collecting information about a drive
- Creating and deleting a directory
- Renaming or moving a directory
- Creating a file
- Writing to a file
- Reading from a file
- Copying and deleting a file
- Renaming or moving a file

# Basic of File I/O and the file system

- The **System.IO** namespace contains the **File** and **Directory** classes, which provide the .NET Framework functionality that manipulates files and directories.
- Associated with these classes are the **FileInfo** and **DirectoryInfo** classes, which will be familiar to users of the **My** feature.
- To use these classes, you must fully qualify the names or import the appropriate namespaces by including the **Imports** statement(s) at the beginning of the affected code.
- The .NET Framework uses *streams* to support reading from and writing to files.
- **Stream** may be defined as a one-dimensional set of contiguous data, which has a beginning and an end, and where the cursor indicates the current position in the stream.
- The data contained in the stream may come from memory, a file, or a TCP/IP socket.
- Streams have fundamental operations that can be applied to them: reading, writing and seeking.

# File and Stream I/O

- The **System.IO** namespace contains types that allow synchronous and asynchronous reading and writing on data streams and files.
- A *file* is an ordered and named collection of a particular sequence of bytes having persistent storage. Therefore, with files, one thinks in terms of directory paths, disk storage, and file and directory names.
- A *streams*, in contrast, provide a way to write and read bytes to and from a backing store that can be one of several storage mediums. There are several kinds of streams other than file streams, for example: network, memory, and tape streams.
- The abstract base class **Stream** supports reading and writing bytes. **Stream** integrates asynchronous support. Its default implementations define synchronous reads and writes in terms of their corresponding asynchronous methods, and vice versa.
- All classes that represent streams inherit from the **Stream** class. The **Stream** class and its derived classes provide a generic view of data sources and repositories, isolating the programmer from the specific details of the operating system and underlying devices.
- The **My.Computer.FileSystem** object provides methods and properties for file I/O.

# Basic I/O classes for files, drives, and directories

Class	Description
System.IO. DriveInfo	Provides properties and methods to query drive information such as drive type, drive format, total size or total free space.
System.IO.Directory	Provides static methods for creating, moving, and enumerating through directories and subdirectories.
System.IO.File	Provides static methods for creating, copying, deleting, moving, and opening files, and aids in the creation of a <b>FileStream</b> .
System.IO.FileAccess	Defines constants for read, write, or read/write access to a file.
System.IO.FileAttributes	Provides attributes for files and directories such as Archive, Hidden, and ReadOnly.
System.IO.FileMode	Controls how a file is opened. This parameter is specified in many of the constructors for <b>FileStream</b> and <b>IsolatedStorageFileStream</b> , and for the <b>Open</b> methods of <b>File</b> and <b>FileInfo</b> .
System.IO.FileShare	Defines constants for controlling the type of access other file streams can have to the same file.
System.IO.Path	Provides methods and properties for processing directory strings.
System.Security.Permissions .FileIOPermission	Controls the access of files and folders by defining Read, Write, Append and PathDiscovery permissions.

# Classes used for Reading from and Writing to Streams

- **BinaryReader** and **BinaryWriter** read and write encoded strings and primitive data types from and to Streams.
- **StreamReader** reads characters from **Streams**, using **Encoding** to convert characters to and from bytes. **StreamReader** has a constructor that attempts to ascertain what the correct **Encoding** for a given **Stream** is, based on the presence of an **Encoding**-specific preamble, such as a byte order mark.
- **StreamWriter** writes characters to **Streams**, using **Encoding** to convert characters to bytes.
- **StringReader** reads characters from Strings. **StringReader** allows you to treat Strings with the same API, so your output can be either a Stream in any encoding or a String.
- **StringWriter** writes characters to Strings. **StringWriter** allows you to treat Strings with the same API, so your output can be either a **Stream** in any encoding or a String.
- **TextReader** is the abstract base class for **StreamReader** and **StringReader**. While the implementations of the abstract **Stream** class are designed for byte input and output, the implementations of **TextReader** are designed for Unicode character output.
- **TextWriter** is the abstract base class for **StreamWriter** and **StringWriter**. While the implementations of the abstract **Stream** class are designed for byte input and output, the implementations of **TextWriter** are designed for Unicode character input.
- **StreamReader** implements a **TextReader** that reads characters from a byte stream in a particular encoding.



# DriveInfo class

**DriveInfo** class provides instance methods for accessing information about a drive.

- Use **DriveInfo** class to: determine what drives are available and what type of drives they are, query to determine the capacity and available free space on the drive.
- IsReady property gets a value indicating whether a drive is ready.
- TotalSize property gets the total size of storage space on a drive.
- TotalFreeSpace property gets the total amount of free space available on a drive.
- RootDirectory property gets the root directory of a drive.
- Name property gets the name of a drive.
- DriveFormat property gets the name of the file system, such as NTFS or FAT32.
- GetDrives method retrieves the drive names of all logical drives on a computer.



# Using DriveInfo class

```
Dim myAllDrives() As DriveInfo = System.IO.DriveInfo.GetDrives()  
Dim myDrive As System.IO.DriveInfo  
' Index 2 in myAllDrives() array means drive D: because 0 is the drive A: and 1 is the drive C:  
myDrive = myAllDrives(2)  
If myDrive.IsReady = True Then  
    lblDriveName.Text = myDrive.Name  
    lblFileSystem.Text = myDrive.DriveFormat  
    lblTotalSize.Text = myDrive.TotalSize & " bytes"  
    lblTotalFreeSpace.Text = myDrive.TotalFreeSpace & " bytes"  
End If
```

# Directory class

**Directory** class exposes static methods for creating, moving, and enumerating through directories and subdirectories.

- Use the **Directory** class for typical operations such as copying, moving, renaming, creating, and deleting directories.
- Use the **Directory** class to get and set DateTime information related to the creation, access, and writing of a directory.
- Most **Directory** methods require the path to the directory that is manipulated. In members that accept a path, the path can refer to a file or just a directory.

# Creating and deleting a directory

- `Directory.CreateDirectory (Path)` – creates all directories and subdirectories as specified by path.
- `Directory.CreateDirectory (Path, DirectorySecurity)` – creates all the directories in the specified path, applying the specified Windows security.
- `Directory.Delete (Path)` – deletes an empty directory from a specified path.
- `Directory.Delete (Path, Boolean)` – deletes the specified directory and, if indicated, any subdirectories in the directory.

```
Dim path As String = "D:\MyDir"  
If System.IO.Directory.Exists(path) = False Then  
    System.IO.Directory.CreateDirectory(path)  
End If  
If System.IO.Directory.Exists(path) = True Then  
    System.IO.Directory.Delete(path)  
End If
```

# Renaming or moving a directory

Using `Directory.Move(sourceDirName, destDirName)` method you can rename or move directory. Note that the contents move with the directory.

## Renaming

```
Dim path As String = "D:\MyDir"  
Dim target As String = "D:\MyNewDir"  
' The contents move with the directory.  
Directory.Move(path, target)
```

## Moving

```
Dim path As String = "D:\MyDir"  
Dim target As String = "D:\NewFolder\MyDir"  
Directory.Move(path, target)
```

# Other Directory class's methods

- **GetDirectories** – gets the names of subdirectories in a specified directory
- **GetCurrentDirectory** – gets the current working directory of the application
- **SetCurrentDirectory** – sets the application's current working directory to the specified directory
- **GetCreationTime** – gets the creation date and time of a directory
- **GetDirectoryRoot** – returns the volume information, root information, or both for the specified path
- **SetCreationTime** – sets the creation date and time for the specified file or directory

# File class

**File** class provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of **FileStream** objects.

- Use the **File** class also to get and set file attributes or **DateTime** information related to the creation, access, and writing of a file.
- Many of the **File** methods return other I/O types when you create or open files. You can use these other types to further manipulate a file.
- All **File** methods require the path to the file that you are manipulating.
- By default, full read/write access to new files is granted to all users.

# File class's methods

- Exist method determines whether the specified file exists.
- Create method creates a file in the specified path.
- CreateText method creates or opens a file for writing UTF-8 encoded text.
- Copy method copies an existing file to a new file.
- Move method moves a specified file to a new location, providing the option to specify a new file name.
- OpenRead method opens an existing file for reading.
- OpenWrite method opens an existing file for writing.
- ReadAllText method opens a text file, reads all lines of the file into a string, and then closes the file.
- WriteAllText method creates a new file, write the contents to the file, and then closes the file. If the target file already exists, it is overwritten.
- Delete method deletes the specified file. An exception is not thrown if the specified file does not exist.

# Creating a file

```
Imports System.IO
Const FILE_NAME As String = "MyFile.txt"
Dim path As String = "D:\MyDir\"
' StreamWriter class implements a TextWriter for writing
' characters to a stream in a particular encoding.
Dim myStrWr As StreamWriter
If File.Exists(path & FILE_NAME) Then
    MsgBox("File already exists")
Else
    myStrWr = File.CreateText(path & FILE_NAME)
    myStrWr.Close()
End If
```



# Writing to a file

```
Imports System.IO
Const FILE_NAME As String = "MyFile.txt"
Dim path As String = "D:\MyDir\"
' Create an instance of StreamWriter to write text to a file.
Dim sw As StreamWriter = New StreamWriter(path & FILE_NAME)
' Add some text to the file.

If File.Exists(path & FILE_NAME) Then
    sw.Write("This is the ")
    sw.WriteLine("header for the file.")
    sw.WriteLine("-----")
    sw.Write("The date is: ")
    sw.WriteLine(DateTime.Now)
    sw.Close()
Else
    MsgBox("File does not exist")
End If
```

# Reading from a file

```
Imports System.IO
```

## 'EXAMPLE 1

```
Const FILE_NAME As String = "MyFile.txt"  
Dim path As String = "D:\MyDir\  
If File.Exists(path & FILE_NAME) Then  
    txbFileViewer.Text = File.ReadAllText(path & FILE_NAME)  
Else  
    MsgBox("File does not exist", MsgBoxStyle.Exclamation)  
End If
```

## 'EXAMPLE 2

```
Const FILE_NAME As String = "MyFile.txt"  
Dim path As String = "D:\MyDir\  
' Create an instance of StreamReader to read from a file.  
Dim sr As StreamReader = New StreamReader(path & FILE_NAME)  
txbFileViewer.Text = sr.ReadToEnd()  
sr.Close()
```

# Copying and deleting a file

- `File.Copy(sourceFileName, destFileName)` method copies an existing file to a new file. Overwriting a file of the same name is not allowed.
- `File.Copy(sourceFileName, destFileName, overwrite)` method copies an existing file to a new file. Overwriting a file of the same name is allowed.

## Copying

```
Dim path As String = "D:\MyTest.txt"  
Dim path2 As String = "D:\MyDir\MyTest.txt"  
File.Copy(path, path2)
```

## Deleting

```
Dim path As String = "D:\MyDir\MyTest.txt"  
File.Delete(path)
```

# Renaming or moving a file

- `File.Move(sourceFileName, destFileName)` method moves a specified file to a new location, providing the option to specify a new file name. Using this methods you can rename or move directory.

## Renaming

```
Dim sourceFile As String = "D:\MyDir\MyFile.txt"
Dim destFile As String = "D:\MyDir\MyNewFile.txt"
File.Move(sourceFile, destFile)
```

## Moving

```
Const FILE_NAME As String = "MyFile.txt"
Dim origin As String = "D:\MyDir\" + FILE_NAME
Dim target As String = "D:\MyDir\MyNewDir\"
If Directory.Exists(target) = False Then
    Directory.CreateDirectory(target)
    target = target + FILE_NAME
    File.Move(origin, target)
End If
```

# Exception handling

- Types of errors
- Structured exception handling
- Unstructured exception handling

**Exception handling** is the process of responding to the occurrence of *exceptions* – anomalous or exceptional events requiring special processing – often changing the normal flow of program execution.

# Types of errors

- In Visual Basic, errors fall into one of three categories:
  - 1) Syntax errors
  - 2) Run-time errors
  - 3) Logic errors (which generate *exceptions*)
- *Syntax errors* are those that appear while you write code. VB checks your code as you type it in the **Code Editor** window and alerts you if you make a mistake.
- *Run-time errors* are those that appear only after you compile and run your code. These involve code that may appear to be correct in that it has no syntax errors, but that will not execute.
- *Logic errors* are those that appear once the application is in use. They are most often unwanted or unexpected results in response to user actions. Logic errors are generally the hardest type to fix, since it is not always clear where they originate.

# Introduction to error handling

- By placing exception/error handling code in your application, you can handle most of the errors users may encounter and enable the application to continue running.
- Consider using exception handling in any method that uses operators that may generate an exception.
- If an exception occurs in a method that is not equipped to handle the exception, the exception is propagated back to the calling method, or the previous method. If the previous method also has no exception handler, the exception is propagated back to that method's caller, and so on. If it fails to find a handler for the exception, an error message is displayed and the application is terminated.
- Visual Basic supports both *structured* and *unstructured* exception handling.

# Structured exception handling

- **Structured** exception handling is simply use blocks of code test for specific circumstances and react accordingly. This allows your code to differentiate between different types of errors and react in accordance with circumstances.
- A single method can have multiple structured exception handling blocks, and the blocks can also be nested within each other.
- The **Try...Catch...Finally** statement is used specifically for structured exception handling.
- Structured exception handling is significantly more versatile, robust, and flexible than unstructured.



# Try...Catch...Finally statement

- The **Try** block contains the section of code to be monitored for exceptions.
- If an error occurs during execution of this section, VB examines each **Catch** statement until it finds one with a condition that matches that error.
- If an error is found, control transfers to the first line of code in the **Catch** block.
- If no matching **Catch** statement is found, an error is produced.
- The **Finally** block is always run, regardless of any actions occurring in preceding **Catch** blocks.
- Place cleanup code, such as that for closing files and releasing objects, in the **Finally** section.
- If you do not need to catch exceptions, but do need to clean up resources, consider using the **Using** statement rather than a **Finally** section.

# Structure of a Try...Catch...Finally statement

## **Try**

- ' Starts a structured exception handler.
- ' Place executable statements that may generate an exception in this block.

## **Catch [optional filters]**

- ' This code runs if the statements listed in the Try block fail and the filter on the Catch statement is true.

## **[Additional Catch blocks]**

## **Finally**

- ' This code always runs immediately before the Try statement exits.

## **End Try**

- ' Ends a structured exception handler.

# 1) Error filtering in the **Catch** block

- **Catch** blocks allow three options for specific error filtering.
- First, errors are filtered based on the *class of the exception* (the `Exception` exception in the example below).
- If a `Exception` error occurs, the code within the specified **Catch** block is executed

```
Try
  My.Computer.FileSystem.CopyFile("MyFile", "MyNewFile")
Catch e as Exception
  MsgBox("An error occurred")
Finally
  ' "Finally" block.
End Try
```

## 2) Error filtering in the **Catch** block

- Second error-filtering option, the **Catch** section can filter on any *conditional expression*.
- One common use of this type of **Catch** filter is to test for specific error numbers, as shown in the following code.
- When VB finds the matching error handler, it executes the code within that handler, and then passes control to the **Finally** block.

```
Try
    ' "Try" block.
Catch When ErrNum = 5 ' Type mismatch.
    ' "Catch" block.
Finally
    ' "Finally" block.
End Try
```

# Some of the exception classes

- 1) **Exception** class represents errors that occur during application execution.

This class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error.

- 2) **IOException** class – the exception that is thrown when an I/O error occurs – the base class for exceptions thrown while accessing information using streams, files and directories.
- 3) **SecurityException** exception is thrown when a security error is detected.
- 4) **ApplicationException** exception is thrown when a non-fatal application error occurs.

# Properties of the **Exception** class

- Message property gets a message that describes the current exception.
- Source property gets or sets the name of the application or the object that causes the error.
- StackTrace property gets a string representation of the frames on the call stack at the time the current exception was thrown.
- InnerException property gets the **Exception** instance that caused the current exception

# Display an exception's message

Use the **Message** property to display information about the current exception.

```
Try
    FileSystem.CopyFile("MyFile", "MyNewFile")
Catch ex As Exception
    MsgBox("An exception occurred:" & _
        ex.Message)
End Try
```

# Subclasses of the **IOException** class

- **DirectoryNotFoundException** exception is thrown when part of a file or directory cannot be found.
- **EndOfStreamException** exception is thrown when reading is attempted past the end of a stream.
- **FileNotFoundException** exception is thrown when an attempt to access a file that does not exist on disk fails.
- **FileLoadException** exception is thrown when a managed assembly is found but cannot be loaded.
- **PathTooLongException** exception is thrown when a pathname or filename is longer than the system-defined maximum length.



# Handling Exception and IOException exceptions

```
Try
```

```
    ' Add code for your I/O task here.
```

```
Catch dirNotFound As System.IO.DirectoryNotFoundException
```

```
    MsgBox("The directory is not found.")
```

```
Catch fileNotFound As System.IO.FileNotFoundException
```

```
    MsgBox("No such file in this directory.")
```

```
Catch pathTooLong As System.IO.PathTooLongException
```

```
    ' Code to handle PathTooLongException.
```

```
Catch ioEx As System.IO.IOException
```

```
    ' Code to handle IOException.
```

```
Catch security As System.Security.SecurityException
```

```
    ' Code to handle SecurityException.
```

```
Catch ex As Exception
```

```
    ' Rethrow exception if anything else has occurred.
```

```
    Throw ex
```

```
Finally
```

```
    ' Dispose of any resources you used or opened in the Try block.
```

```
End Try
```

# Structured exception handler example 1

```
Public Sub MyExample()  
    Dim x As Double = 6    ' Declare variables.  
    Dim y As Integer = 0  
    Try    ' Set up structured error handling.  
        x = x / y    ' Cause a "Divide by Zero" error.  
    Catch ex As Exception When y = 0    ' Catch the error.  
        MsgBox(ex.ToString)    ' Show friendly error message.  
    Finally  
        Beep()    ' This line is executed no matter what.  
    End Try  
End Sub
```

# Structured exception handler example 2

```
Imports System.IO
Dim Strings As New Collection
' Open the file.
Dim sr As StreamReader = File.OpenText(FileName)
Try
    While True
        ' Loop terminates with an EndOfStreamException error when end of stream is reached.
        Strings.Add(sr.ReadLine())
    End While
Catch eos As System.IO.EndOfStreamException
    ' No action is necessary; end of stream has been reached.
Catch IOExcep As System.IO.IOException
    ' Some kind of error occurred. Report error and clear collection.
    MsgBox(IOExcep.Message)
    Strings = Nothing
Finally
    sr.Close() ' Close the file.
End Try
```

# Unstructured exception handling

- The **On Error** statement is used specifically for unstructured exception handling.
- The **On Error** statement is placed at the beginning of a block of code. It then has "scope" over that block; it handles any errors occurring within the block.
- It is *impossible* to combine structured and unstructured exception handling in the same function.
- If you use an **On Error** statement, you cannot use a **Try...Catch** statement in the same function.

# On Error GoTo Line statement

- The **On Error GoTo Line** statement assumes that error-handling code starts at the line specified in the required *line* argument.
- If a run-time error occurs, control branches to the line label or line number specified in the argument, activating the error handler.
- The specified line must be in the same procedure as the **On Error GoTo Line** statement; otherwise, VB generates a compiler error.
- The **Resume** statement passes control back to the line of code where the error first occurred.

```
Sub TestSub
    On Error GoTo MyErrorHandler
    ' Code that may or may not contain errors.
    ' Place an Exit Sub statement immediately before the error-handling block.
Exit Sub

MyErrorHandler:
    ' Code that handles errors.
    Resume
End Sub
```

# On Error Resume Next statement

- The **On Error Resume Next** statement specifies that in the event of a run-time error, control passes to the statement immediately following the one in which the error occurred. At that point, execution continues.
- **On Error Resume Next** enables you to put error-handling routines where errors will occur, rather than transferring control to another location in the procedure.
- Another variation on the **Resume** statement is **Resume Line**, which is similar to **On Error GoTo Line**. **Resume Line** passes control to a line you specify in the *line* argument. You can use **Resume Line** only within an error handler.
- When debugging your code, you must disable the **On Error Resume Next** statement

# On Error GoTo statements

- The **On Error GoTo 0** statement disables any *error* handler in the current procedure.
- Without an **On Error GoTo 0** statement, an error handler is automatically disabled when a procedure is exited.
- The **On Error GoTo -1** statement disables any *exception* handlers in the current procedure.
- If an **On Error GoTo -1** statement is not included, the exception is automatically disabled when its procedure ends.

# Unstructured exception handler example 1

```
Sub ErrorTest ()
Dim x As Integer, y As Integer, z As Integer
    ' The exception handler is named "DivideByZero".
On Error GoTo DivideByZero
    ' The main part of the code, which might cause an error.
x = 2
y = 0
z = x \ y ' Divide two numbers and return an integer result (truncation).
    ' This line disables the exception handler.
On Error GoTo 0
    MsgBox (x & "/" & y & " = " & z)
    ' Exit the subroutine before the error-handling code.
    ' Failure to do so can create unexpected results.
Exit Sub
    ' This is the exception handler, which deals with the error.
DivideByZero:
    ' Include a friendly message to let the user know what is happening.
MsgBox("You have attempted to divide by zero!")
    ' Provide a solution to the error.
z = 2
    ' The Resume statement returns to the point at which the error first occurred,
    ' so the application can continue to run.
Resume
End Sub
```



# Unstructured exception handler example 2

```
Public Sub OnErrorDemo()  
    On Error GoTo ErrorHandler    ' Enable error-handling routine.  
    Dim x As Integer = 32  
    Dim y As Integer = 0  
    Dim z As Integer  
    z = x / y    ' Creates a divide by zero error  
    On Error GoTo 0    ' Turn off error trapping.  
    On Error Resume Next    ' Defer error trapping.  
    z = x / y    ' Creates a divide by zero error again  
    If Err.Number = 6 Then  
        MsgBox("There was an error attempting to divide by zero!")  
        Err.Clear() ' Clear Err object fields.  
    End If  
Exit Sub    ' Exit to avoid handler.  
ErrorHandler: ' Error-handling routine.  
    Select Case Err.Number    ' Evaluate error number.  
        Case 6    ' Divide by zero error  
            MsgBox("You attempted to divide by zero!")  
        Case Else  
            ' Insert code to handle other situations here...  
    End Select  
    Resume Next    ' Resume execution at same line that caused the error.  
End Sub
```

# Err object

In unstructured exception handling, to know which error occurred, the **Err** object may be used.

**Err** object contains information about run-time errors that just occurred.

- The properties of the **Err** object are set by the generator of an error – VB, an object, or the programmer.
- When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error and that you can use to handle the error.
- The **Err** object's properties are reset to zero or zero-length strings ("") after an **Exit Sub**, **Exit Function**, **Exit Property**, or **Resume Next** statement within an error-handling routine.
- Using any form of the **Resume** statement outside of an error-handling routine will not reset the **Err** object's properties. Use the **Clear** method to explicitly reset **Err**.
- The **Err** object is an intrinsic object with global scope. Therefore, you do not need to create an instance of it in your code.

# Properties and methods of the **Err** object

- **Number** – returns or sets a numeric value specifying an error.
- **Description** – returns or sets a descriptive string associated with an error.
- **Source** – returns or sets a **String** expression specifying the name of the object or application that originally generated the error.
- **Clear** method – clears all property settings of the **Err** object.
- **Raise** method – generates a run-time error; can be used instead of the **Error** statement.

The error message associated with **Err.Number** is contained in **Err.Description**.

# Using the Err object

```
Dim Msg As String
' If an error occurs, construct an error message.
On Error Resume Next      ' Defer error handling.
Err.Clear()
' Check for error, then show message.
If Err.Number <> 0 Then

    Msg = "Error # " & Str(Err.Number) & " was generated by "_
        & Err.Source & ControlChars.CrLf & Err.Description

    MsgBox(Msg, MsgBoxStyle.Information, "Error")
End If
```

# Using block

- The **Using** block is used to guarantee that the system disposes of a resource (resource such as a file handle) when your code exits the block.
- This is useful if you are using a system resource that consumes a large amount of memory, or that other components also want to use.
- The system disposes of the resource no matter how you exit the block, including the case of an unhandled exception.
- Use a **Using** block when you want to be sure to leave the resource available for other components after you have exited the **Using** block.

# Example of the Using block

```
Public Sub setbigbold(ByVal c As Control)
    Using nf As New System.Drawing.Font("Arial", _
        12.0F, System.Drawing.FontStyle.Bold)
        c.Font = nf
        c.Text = "This is 12-point Arial bold"
    End Using
End Sub
```

# Menus and toolbars controls

- Working with menus
- Pop-up menus
- Status bars
- Toolbars

# ToolStrip technology summary

- The **ToolStrip** and its associated classes provide a complete solution for creating toolbars, status bars, and menus.
- **ToolStrip** is the extensible base class for **MenuStrip**, **ContextMenuStrip**, and **StatusStrip** controls.
- Menus are created using a **MenuStrip** control, which provides a graphical menu designer. Menus can also be created programmatically.
- Context menus are created using **ContextMenuStrip** controls, which provides a graphical menu designer; they can also be created programmatically by creating a new instance of the **ContextMenuStrip** class.



# ToolStrip technology summary

## Rafting and Docking

- It is possible to raft, dock, or absolutely position **ToolStrip** controls. **ToolStrip** items are laid out by the **LayoutEngine** of the container.
- *Rafting* is the ability of toolbars to share horizontal or vertical space.
- *Docking* is the specifying of a control's simple location on the form's left, right, top, or bottom side.
- It is possible also to specify that a **ToolStrip** control be freely positioned on the form.

## Rendering

- To apply a style to a **ToolStrip** and all the **ToolStripItem** objects it contains set the **RenderMode** property to one of the **ToolStripRenderMode** values other than **Custom**.

# ToolStrip classes

Technology area	Class
Toolbar, Status, and Menu containers	ToolStrip MenuStrip ContextMenuStrip StatusStrip ToolStripDropDownMenu
ToolStrip items	ToolStripLabel ToolStripDropDownItem ToolStripMenuItem ToolStripButton ToolStripStatusLabel ToolStripSeparator ToolStripControlHost ToolStripComboBox ToolStripTextBox ToolStripProgressBar ToolStripDropDownButton ToolStripSplitButton
Location	ToolStripContainer ToolStripContentPanel ToolStripPanel
Presentation and rendering	ToolStripManager ToolStripRenderer ToolStripProfessionalRenderer ToolStripRenderMode ToolStripManagerRenderMode

# ToolStrip control using

- Create easily customized, commonly employed toolbars that support advanced user interface and layout features, such as docking, rafting, buttons with text and images, drop-down buttons and controls, overflow buttons, and run-time reordering of **ToolStrip** items.
- Support the typical appearance and behavior of the operating system.
- Handle events consistently for all containers and contained items, in the same way you handle events for other controls.
- Drag items from one **ToolStrip** to another or within a **ToolStrip**.
- Create drop-down controls and user interface type editors with advanced layouts in a **ToolStripDropDown**.

# Important ToolStrip members

Name	Description
Dock	Gets or sets which edge of the parent container a <b>ToolStrip</b> is docked to.
AllowItemReorder	Gets or sets a value indicating whether drag-and-drop and item reordering are handled privately by the <b>ToolStrip</b> class.
LayoutStyle	Gets or sets a value <u>indicating how the <b>ToolStrip</b> lays out its items.</u>
Overflow	Gets or sets whether a <b>ToolStripItem</b> is attached to the <b>ToolStrip</b> or <b>ToolStripOverflowButton</b> or can float between the two.
IsDropDown	Gets a value <u>indicating whether a <b>ToolStripItem</b> displays other items in a drop-down list</u> when the <b>ToolStripItem</b> is clicked.
OverflowButton	Gets the <b>ToolStripItem</b> that is the overflow button for a <b>ToolStrip</b> with overflow enabled.
Renderer	Gets or sets a <b>ToolStripRenderer</b> used to <u>customize the appearance and behavior (look and feel)</u> of a <b>ToolStrip</b> .
RenderMode	Gets or <u>sets the painting styles to be applied</u> to the <b>ToolStrip</b> .
RendererChanged	Raised when the <b>Renderer</b> property changes.

# Important **ToolStrip** companion classes

Name	Description
MenuStrip	Replaces and adds functionality to the MainMenu class.
StatusStrip	Replaces and adds functionality to the StatusBar class.
ContextMenuStrip	Replaces and adds functionality to the ContextMenu class.
<b>ToolStripItem</b>	Abstract base class that manages events and layout for all the elements that a <b>ToolStrip</b> , <b>ToolStripControlHost</b> , or <b>ToolStripDropDown</b> can contain.
ToolStripContainer	Provides a container with a panel on each side of the form in which controls can be arranged in various ways.
<b>ToolStripRenderer</b>	Handles the painting functionality for <b>ToolStrip</b> objects.

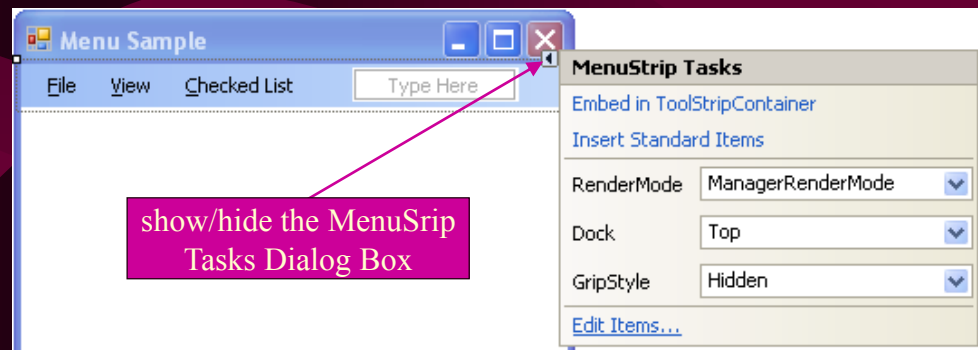
# MenuStrip control using

- Create easily customized, commonly employed menus that support advanced user interface and layout features, such as:
  - text and image ordering and alignment,
  - drag-and-drop operations, and
  - alternate modes of accessing menu commands.
- Support the typical appearance and behavior of the operating system.
- Handle events consistently for all containers and contained items, in the same way you handle events for other controls.
- Supports the multiple-document interface (MDI) and menu merging, tool tips, and overflow.
- Enhance the usability and readability of menus by adding access keys, shortcut keys, check marks, images, and separator bars.
- The **MenuStrip** control replaces and adds functionality to the **MainMenu** control; however, the **MainMenu** control is retained for backward compatibility and future use if you choose.

# MenuStrip control using

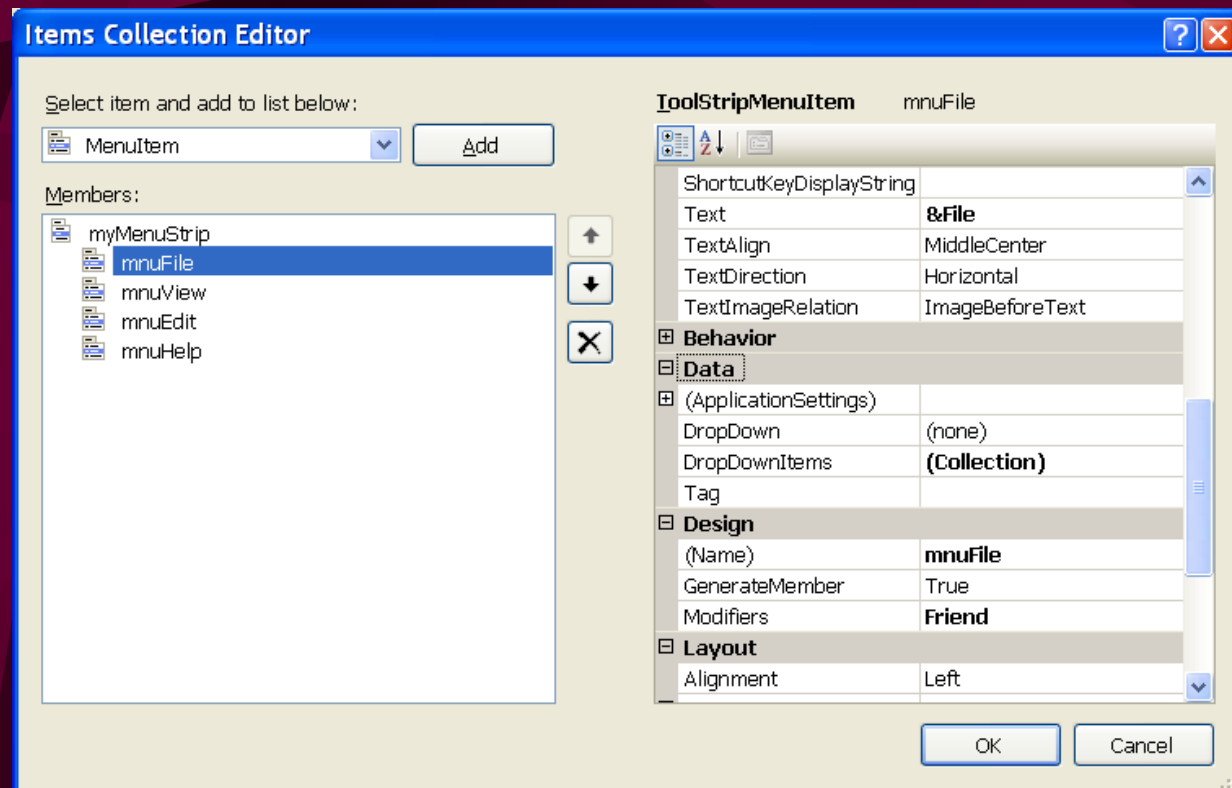
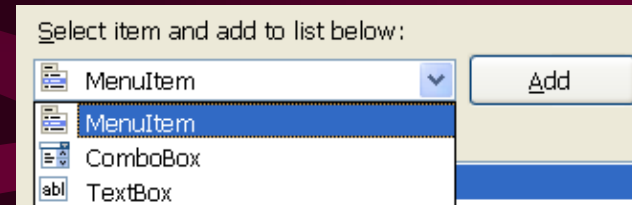
- The **MenuStrip** control represents the container for the menu structure of a form. You can add ToolStripMenuItem objects to the MenuStrip that represent the individual menu commands in the menu structure.
- Each **ToolStripMenuItem** can be a command for your application or a parent menu for other submenu items.
- **MenuStrip** is the container for **ToolStripMenuItem**, **ToolStripComboBox**, **ToolStripSeparator**, and **ToolStripTextBox** objects.
- Use the MenuStrip Tasks dialog box at design time to run common commands and edit items of the **MenuStrip** control.
- Add an item during run time:

```
Dim MenuStrip1 As New MenuStrip()  
Dim MyItem As New ToolStripMenuItem  
MenuStrip1.Items.Add(MyItem)
```



# Using Items Collection Editor

- The most convenient method of editing a menu.
- Three types of items can be added:





# MenuStrip control properties

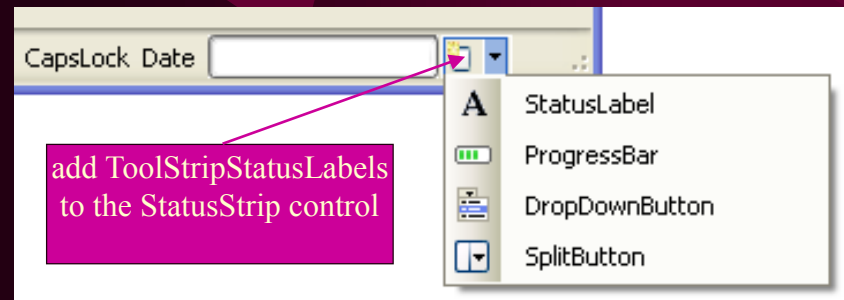
- **Items** – collection of items to display on the **ToolStrip**. This property retrieves all the items that have been added to, not just the items that are displayed.
- **Dock** – gets or sets which **ToolStrip** borders are docked to its parent control and determines how a **ToolStrip** is resized with its parent.
- **Locked** – determines if the control can be moved or resized

## MenuStrip items' properties

- **ShortcutKeys** – allows to define shortcut keys to a menu command. Shortcut keys are combinations of **Ctrl**, **Shift**, and **Alt** keys and a function or character key.

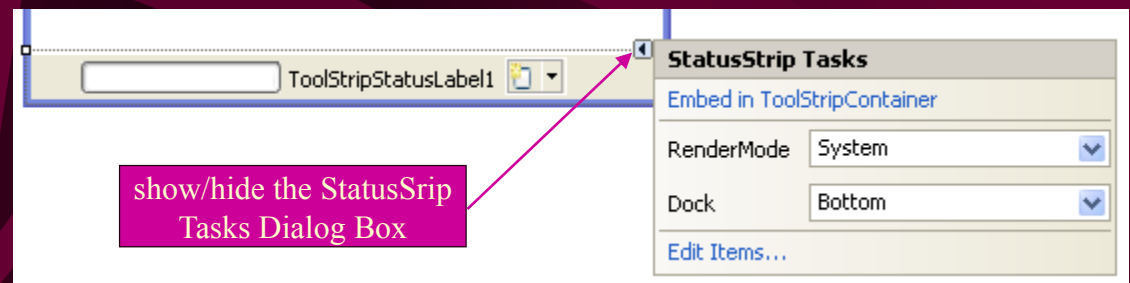
# StatusStrip control using

- A **StatusStrip** control displays information about an object being viewed on a form, the object's components, or contextual information that relates to that object's operation within your application.
- Typically, a **StatusStrip** control consists of **ToolStripStatusLabel** objects, each of which displays text, an icon, or both.
- The **StatusStrip** can also contains **ToolStripDropDownButton**, **ToolStripSplitButton**, and **ToolStripProgressBar** controls.



# StatusStrip control using

- The default **StatusStrip** has no panels.
- During run time: use the **ToolStripItemCollection.AddRange** method to add panels to a **StatusStrip**.
- At design time: use the StatusStrip Items Collection Editor to add, remove, or reorder items and modify properties.
- Use the StatusStrip Tasks Dialog Box at design time to run common commands and edit items of the **StatusStrip** control.



# ProgressBar properties and methods

## Properties

- **Minimum** – gets or sets the lower bound of the range that is defined for a progress bar
- **Maximum** – gets or sets the upper bound of the range that is defined for a progress bar
- **Value** – gets or sets the current value of the progress bar
- **Step** – gets or sets the amount by which to increment the current value of the progress bar when the **PerformStep** method is called

## Methods

- **Increment** – advances the current position of the progress bar by the specified amount
- **Invalidate** – invalidates some or all of the surface of the control and causes the control to be redrawn

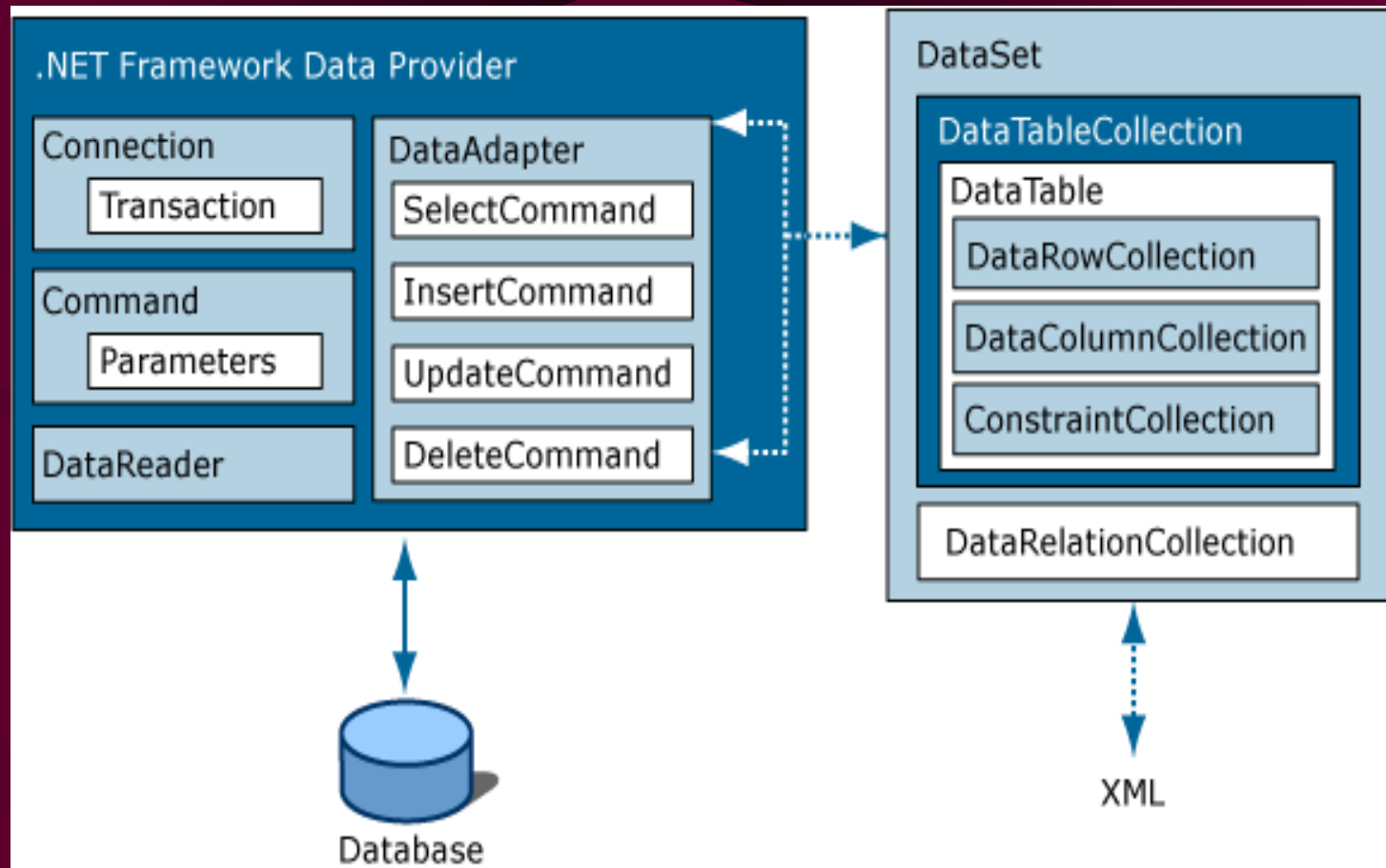
# ContextMenuStrip control using

- The **ContextMenuStrip** class represents shortcut menus that are displayed when the user clicks the right mouse button over a control or area of the form.
- Shortcut menus are typically used to combine different menu items from a **MenuStrip** of a form that are useful for the user given the context of the application. For example, you can use a shortcut menu assigned to a **TextBox** control to provide menu items for changing the font of the text, finding text within the control, or Clipboard features for copying and pasting text.
- You can also expose new **ToolStripMenuItem** objects in a shortcut menu that are not located within a **MenuStrip** to provide situation-specific commands that are inappropriate for the **MenuStrip** to display.
- Many visible controls, as well as the form itself, have a **Control.ContextMenuStrip** property that binds the **ContextMenuStrip** class to the control that displays the shortcut menu. More than one control can use a **ContextMenuStrip**.

# Accessing data stored in database

- ADO.NET model
- Data Source; connection to data source
- DataGridView; binding to data
- Command; SQL statement
- Fill in a control with data from database
- Presentation of the data selected from database

# ADO.NET architecture



# Objects of ADO.NET model

- **Connection** – for connection to and managing transactions against a database
- **Command** – for issuing SQL commands against a database
- **DataSet** – for storing, remoting and programming against flat data, XML data and relational data
- **DataAdapter** – for pushing data into a DataSet, and reconciling data against a database
- **DataReader** – for reading a forward-only (better performance) stream of data records from a data source



# DataGridView control

- Provides a visual interface to data
- Displays data in tabular format
- Allows editing for data
- Data edited in the DataGridView can then be persisted in the database

' fill DataGridView with data from DataSet

```
myDataGridView.DataSource = ds.Tables("Name")
```

' prepare sql statement to take data from ADRES field based on listbox setting

```
sql = "SELECT * FROM forest_stands WHERE ADRES = _  
" & lstbxSpecies.SelectedItem & "'"
```

# Add new Data Source

Good tutorial can be found here:

<http://www.homeandlearn.co.uk/NET/nets12p1.html>

## Other help

<https://www.connectionstrings.com/access-2000/>

<https://www.connectionstrings.com/using-jet-in-64-bit-environments/>

<https://support.microsoft.com/en-us/help/957570/the-microsoft-ole-db-provider-for-jet-and-the-microsoft-access-odbc-driver-are-available-in-32-bit-versions-only>

<https://techblog.aimms.com/2014/10/27/installing-32-bit-and-64-bit-microsoft-access-drivers-next-to-each-other/>

# Connection to database

There are two possibilities to connect to a database:

- SQL Server .NET Data Provider  
(System.Data.SqlClient) – to talk to Microsoft SQL Server
- OLE DB .NET Data Provider  
(System.Data.OleDb) – to talk to any OLE DB provider

# Connection coding

## `Access database

```
Dim connect As New OleDb.OleDbConnection
Dim dbProvider As String =
    "PROVIDER=Microsoft.Jet.OLEDB.4.0;"
    '"PROVIDER=Microsoft.ACE.OLEDB.12.0;" 'for ACE 12 driver
Dim dbSource As String = "Data Source=D:\VB\stands.mdb"
connect.ConnectionString = dbProvider & dbSource
connect.Open()
```

## ` lines of code

```
connect.Close()
```

## `SQL Server database

```
Dim connect As SqlClient.SqlConnection = New
    SqlConnection("server=(local)\SQLEXPRESS;Integrated
        Security=SSPI;database=stands")
```

# DataSet and DataAdapter coding

```
Dim connect As New OleDb.OleDbConnection
Dim ds As New DataSet
Dim da As OleDb.OleDbDataAdapter
Dim sql As String
'sql statement which will be sent to database
sql = "SELECT * FROM forest_stands"
da = New OleDb.OleDbDataAdapter(sql, connect)
'DataAdapter fill DataSet with records from the table
da.Fill(ds, "Stand description")
```

# Fill in a control with data

```
Dim connect As New OleDb.OleDbConnection
Dim dr As OleDb.OleDbDataReader
Dim command As OleDb.OleDbCommand
sql = "SELECT DISTINCT ADRES_FOR FROM forest_stands
      ORDER BY ADRES_FOR"
command = New OleDb.OleDbCommand(sql, connect)
dr = command.ExecuteReader()
'Read method - advances the OleDbDataReader to the next record.
While dr.Read()
    lstbxSpecies.Items.Add(dr(0))
End While
dr.Close()
```

# Principles of user interface design

- If an application has a well-designed user interface, a novice user can become productive quickly.
- The user interface should minimize user errors and provide visual cues to help a user discover how to complete a task.

# User control

- The **user initiates actions**, not the computer or software — the user plays an active, rather than reactive, role: use techniques to automate tasks, but implement them in a way that allows the user to choose or control the automation.
- Users must be able to **personalize** aspects of the interface. Software should reflect user settings for different system properties, such as color, fonts, or other options.
- Software should be as interactive and responsive as possible. Avoid modes whenever possible. A mode is a state that excludes general interaction or otherwise limits the user to specific interactions.



# Minimal modal interactions

- Software must give the user **maximum control** by minimizing modal interactions.
- Modal interaction is when a user must complete one task before continuing on with a different task.
- Modeless interactions allow the user to move freely between one task and another without fully completing one of them.
- Long processes should run in the background, keeping the foreground interactive.

# Personalization support

- Try to accommodate as many levels and styles as possible, because users of your application will have different working styles and different levels of experience working with the application.
- For example, do not hard-code colors in an applications. Program the application to use the system colors selected by the user.

# Direct user manipulation

- Use clear object metaphors so the user can directly manipulate objects. Metaphors support user recognition rather than recollection.
- Define "verbs" for the objects of interface that can be invoked when an object is selected. Add the actions to a pop-up menu and display the menu when the user right-clicks an object. The user selects an object, and then selects an action to perform on the object.
- Incorporate the concept of affordance into design. Affordance refers to the way an object communicates its use to the user. For example, use tooltips to assist the user in learning the purpose of controls.

# Consistency

- Consistency allows users to transfer existing knowledge to new tasks, learn new things more quickly, and focus more on their work without the need to spend time trying to remember the differences in interaction.
- Present common functions using a consistent set of commands and interfaces.
- Maintain a high level of consistency between the interaction and interface conventions provided by Windows – a software benefits from users' ability to apply interaction skills they have already learned.

# Simplicity

- An interface should be **simple** (not simplistic), **easy to learn**, and **easy to use**. It must also provide access to all functionality provided by an application.
- One way to support simplicity is to limit the presentation of information to a minimum. Display only information that is required to communicate adequately.
- Help users manage complexity by using progressive disclosure. Progressive disclosure involves careful organization of information so that it is shown only at the appropriate time.

## Key rules which help make an application simple to use:

- eliminate visual clutter.
- eliminate ambiguity.
- avoid jargon.
- use progressive disclosure.

# Forgiveness

- An effective interface allows **interactive discovery**. It provides only appropriate sets of choices and warns users about potential situations where they may damage the system or data.
- An effective design avoids situations that are likely to result in errors. It also accommodates potential user errors and makes it easy for the user to recover.

## To implement forgiveness in an application:

- minimize the opportunity for errors.
- give the user an escape route.
- handle errors gracefully.
- supply reasonable default values.

# Feedback

- Always provide feedback for a user's actions.
- Visual and audio signals can be presented to confirm that the software is responding to the user's input and to communicate details that distinguish the nature of the action.
- Effective feedback is timely, and is presented as close to the point of the user's interaction as possible.
- Nothing is more disconcerting than a "dead" screen that is unresponsive to input. A typical user will tolerate only a few seconds of an unresponsive interface.

# Visual design

- Effective visual design serves a greater purpose than decoration; it is an important tool for communication.
- Even the best product functionality can suffer if its visual presentation does not communicate effectively.



# User assistance

- User assistance is an important part of a product's design and can be supported in a variety of ways.
- Its content can be composed of contextual, procedural, explanatory, reference, or tutorial information. But user assistance should always be simple, efficient, and relevant so that a user can obtain it without becoming lost in the interface.
- A wizard is a special form of user assistance that automates a task through a dialog with the user. Wizards help the user accomplish tasks that can be complex and require experience.
- Wizards especially useful for complex or infrequent tasks that the user may have difficulty learning or doing.