

Visual Programming

MCA Second Year

**School of Distance Education
Bharathiar University, Coimbatore - 641 046**

Author: Anindita Hazra

Copyright © 2008, Bharathiar University
All Rights Reserved

Produced and Printed
by

EXCEL BOOKS PRIVATE LIMITED

A-45, Naraina, Phase-I,
New Delhi-110028

for

SCHOOL OF DISTANCE EDUCATION
Bharathiar University
Coimbatore-641046

CONTENTS

	Page No.
UNIT I	
Lesson 1 Introduction to VB.Net	7
Lesson 2 Control Customization	17
Lesson 3 Property Setting	39
UNIT II	
Lesson 4 VB.Net Variables	47
Lesson 5 Decision Making	70
Lesson 6 Functions	83
UNIT III	
Lesson 7 Array	119
Lesson 8 VB.Net - Programming	134
Lesson 9 File Handling	157
UNIT IV	
Lesson 10 Visual C++ Programming	179
Lesson 11 Object Properties	196
Lesson 12 Document/View Architecture	216
UNIT V	
Lesson 13 Data Handling in VC++	239
Lesson 14 Thread-based Multitasking	263
Lesson 15 Wizard	268
Model Question Paper	281

VISUAL PROGRAMMING

SYLLABUS

UNIT I

Introduction to VB.Net- Welcome to VB.Net-opening & closing windows toolbars -Existing project-Auto Hide- Customizing windows placing control on a form- Selecting & resizing control- Relocating control properties windows & setting properties of form & control (using properties window & using event procedure).

UNIT II

VB.Net variables-Data type constant -Building project -Displaying output -Operators -Conditional statement -If-then, Select-Case- Looping Do, For Next , Nested loops.

Import statement -Msgbox -Function-input Box()-Function-user defined &built functions-controls.

UNIT III

Array-Menus & dialog boxes, structures programming -Object oriented programming. Files classification -Handling files using function & classes- Directory class-File class- File processing.

UNIT IV

Visual C++: Programming: MFC & windows -MFC Fundamentals-MFS class Hierarchy-MFC Member &Global Functions-Various Object properties - CObject, CArchive, CWinApp, CWnd, CFile, CGD, Object, CExcept, CDialog, CString, CEdit, CList- Resources:Menus- Accelerators, Dialog, Icon, Bitmaps, versions-Message Maps-Document/View Architecture.VC++ (contd): connecting to data source-DAO-ODBC-Thread-Based Multitasking Visual C++ APPWIZARD and class wizard.

UNIT V

VC++ (contd): connecting to data source-DAO-ODBC-Thread-Based Multitasking Visual C++ APPWIZARD and class wizard.

UNIT I

LESSON

1

INTRODUCTION TO VB.NET

CONTENTS

- 1.0 Aims and Objectives
- 1.1 Introduction
- 1.2 Welcome to VB.Net
 - 1.2.1 Evolution of VB.Net
 - 1.2.2 Features of VB.Net
- 1.3 Opening and Closing Windows
- 1.4 Let us Sum up
- 1.5 Lesson End Activity
- 1.6 Keywords
- 1.7 Questions for Discussion
- 1.8 Suggested Readings

1.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- An overview VB.net
- Evolution of VB.net
- Features of VB.net
- Concept of opening and closing window

1.1 INTRODUCTION

Microsoft launched the .NET Framework and overhauled its most popular language: Visual Basic. With the introduction of the Framework, the language was reconstructed and given a new name: VB.NET. It's now a language that fully supports object-oriented programming. It is true that VB6 supported some notion of objects, but lack of support for inheritance made it unfit to be called an OOP language. For Microsoft, changing VB is more of a necessity than a choice, if VB is to become one of the .NET Framework languages and in order for VB programmers to be able to use the .NET Framework Class Library.

1.2 WELCOME TO VB.NET

The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet.

Services

NET Framework provides the following services:

- Tools for developing software applications,
- run-time environments for software application to execute,
- server infrastructure,
- value added intelligent software which helps developers to do less coding and work efficiently,

The .Net Framework will enable developers to develop applications for various devices and platforms like windows application web applications windows services and web services.

Objectives

The .NET Framework is designed to fulfill the following objectives:

- A consistent object-oriented programming environment, where object code can be stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- A code-execution environment that minimizes software deployment and versioning conflicts.
- A code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party. A code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- Developers can experience consistency across widely varying types of applications, such as Windows-based applications and Web-based applications.
- Build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

Understanding the .NET Framework Architecture

The .NET Framework has two components: the .NET Framework class library and the common language runtime.

The .NET Framework class library facilitates types (CTS) that are common to all .NET languages.

The common language runtime consists of (class loader) that load the IL code of a program into the runtime, which compiles the IL code into native code, and executes and manage the code to enforce security and type safety, and provide thread support.

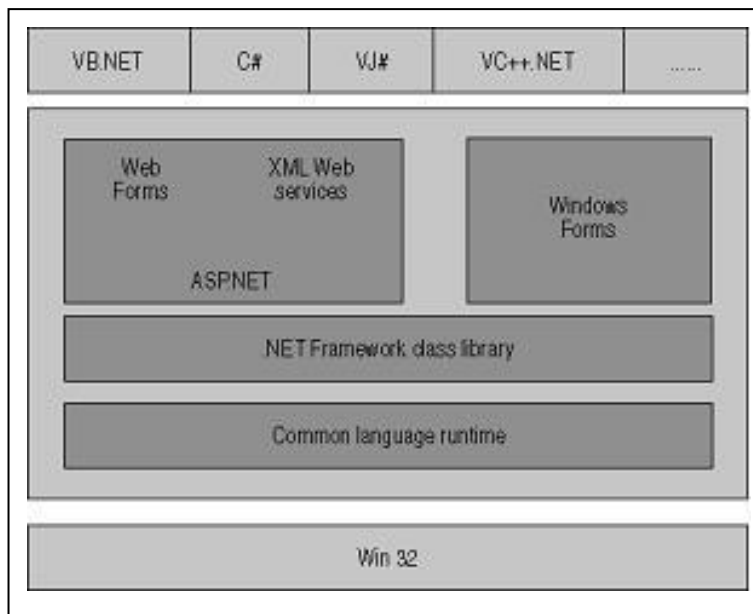


Figure 1.1

.NET Framework Architecture has languages at the top such as VB .NET C#, VJ#, VC++ .NET; developers can develop (using any of above languages) applications such as Windows Forms, Web Form, Windows Services and XML Web Services. Bottom two layers consist of .NET Framework class library and Common Language Runtime.

Visual Basic .NET is a major component of Microsoft Visual Studio .NET suite. The .NET version of Visual Basic is a new improved version with more features and additions.

1.2.1 Evolution of VB.Net

Visual Basic for Windows was debuted on March 20, 1991, at a show called “Windows World,” although its roots go back to a tool called Ruby that Alan Cooper developed in 1988.

The first two versions of Visual Basic for Windows were quite good for building prototypes and demo applications—but not much else. Both versions tied excellent IDEs with relatively easy languages to learn. The languages themselves had relatively small feature sets. When VB 3 was released with a way to access databases that required learning a new programming model, the first reaction of many people was, “Oh great, they’ve messed up VB!” With the benefit of hindsight, the database features added to VB3 were necessary for it to grow beyond the toy stage into a serious tool. With VB4 came a limited ability to create objects and hence a very limited form of object-oriented programming. With VB5 and VB6 came more features from object-oriented programming, and it now had the ability to build controls and to use interfaces. But the structure was getting pretty rickety since the object-oriented features were bolted on to a substructure that lacked support for it. For example, there was no way to guarantee that objects were created correctly in VB—you had to use a convention instead of the constructor approach used by practically every other object-oriented language. Ultimately the designers of VB saw that, if they were going to have a VB-ish tool for their new .NET platform, more changes were needed since, for example, the .NET Framework depends on having full object orientation.

VB.NET is the new version of Visual Basic. The Visual Studio.NET is an Integrated Development Environment that hosts VB.NET, C#, and C++.NET. Underlying all this is the .NET Framework and its core execution engine, the Common Language Runtime.

VB.NET lets VB developers build truly free-threaded components and applications for the first time. There are also things that VB.NET can do that you cannot do today in VB. For example, Web Applications are a new form of project. Gone is Visual InterDev with its interpreted VBScript code. Instead, you now build your ASP.NET pages with VB.NET (or C# or C++), and they are truly compiled for better performance.

VB.NET lets you create Windows services natively for the first time by providing a Windows Services project type. And yes, VB.NET lets VB developers build truly free-threaded components and applications for the first time.

1.2.2 Features of VB.Net

Visual Basic .NET provides the easiest, most productive language and tool for rapidly building Windows and Web applications. Visual Basic .NET comes with enhanced visual designers, increased application performance, and a powerful integrated development environment (IDE). It also supports creation of applications for wireless, Internet-enabled hand-held devices. The following are the features of Visual Basic .NET.

Powerful Windows-based Applications

Visual Basic .NET comes with features such as a powerful new forms designer, an in-place menu editor, and automatic control anchoring and docking. Visual Basic .NET delivers new productivity features for building more robust applications easily and quickly. With an improved integrated development environment (IDE) and a significantly reduced startup time, Visual Basic .NET offers fast, automatic formatting of code as you type, improved IntelliSense, an enhanced object browser and XML designer, and much more.

Building Web-based Applications

With Visual Basic .NET we can create Web applications using the shared Web Forms Designer and the familiar "drag and drop" feature. You can double-click and write code to respond to events. Visual Basic .NET comes with an enhanced HTML Editor for working with complex Web pages. We can also use IntelliSense technology and tag completion, or choose the WYSIWYG editor for visual authoring of interactive Web applications.

Simplified Deployment

With Visual Basic .NET we can build applications more rapidly and deploy and maintain them with efficiency. Visual Basic .NET and .NET Framework makes "DLL Hell" a thing of the past. Side-by-side versioning enables multiple versions of the same component to live safely on the same machine so that applications can use a specific version of a component. XCOPY-deployment and Web auto-download of Windows-based applications combine the simplicity of Web page deployment and maintenance with the power of rich, responsive Windows-based applications.

Powerful, Flexible, Simplified Data Access

You can tackle any data access scenario easily with ADO.NET and ADO data access. The flexibility of ADO.NET enables data binding to any database, as well as classes, collections, and arrays, and provides true XML representation of data. Seamless access to ADO enables simple data access for connected data binding scenarios. Using ADO.NET, Visual Basic .NET can gain high-speed access to MS SQL Server, Oracle, DB2, Microsoft Access, and more.

Improved Coding

You can code faster and more effectively. A multitude of enhancements to the code editor, including enhanced IntelliSense, smart listing of code for greater readability and a background compiler for real-time notification of syntax errors transforms into a rapid application development (RAD) coding machine.

Direct Access to the Platform

Visual Basic developers can have full access to the capabilities available in .NET Framework. Developers can easily program system services including the event log, performance counters and file system. The new Windows Service project template enables to build real Microsoft Windows NT Services. Programming against Windows Services and creating new Windows Services is not available in Visual Basic .NET.

Full Object-Oriented Constructs

You can create reusable, enterprise-class code using full object-oriented constructs. Language features include full implementation inheritance, encapsulation, and polymorphism. Structured exception handling provides a global error handler and eliminates spaghetti code.

XML Web Services

XML Web services enable you to call components running on any platform using open Internet protocols. Working with XML Web services is easier where enhancements simplify the discovery and consumption of XML Web services that are located within any firewall. XML Web services can be built as easily as you would build any class in Visual Basic 6.0. The XML Web service project template builds all underlying Web service infrastructure.

Mobile Applications

Visual Basic .NET and the .NET Framework offer integrated support for developing mobile Web applications for more than 200 Internet-enabled mobile devices. These new features give developers a single, mobile Web interface and programming model to support a broad range of Web devices, including WML for WAP-enabled cellular phones, compact HTML (cHTML) for i-Mode phones, and HTML for Pocket PC, handheld devices, and pagers.

COM Interoperability

You can maintain your existing code without the need to recode. COM interoperability enables you to leverage your existing code assets and offers seamless bi-directional communication between Visual Basic 6.0 and Visual Basic .NET applications.

Reuse Existing Investments

You can reuse all your existing ActiveX Controls. Windows Forms in Visual Basic .NET provide a robust container for existing ActiveX controls. In addition, full support for existing ADO code and data binding enable a smooth transition to Visual Basic .NET.

Upgrade Wizard

You upgrade your code to receive all of the benefits of Visual Basic .NET. The Visual Basic .NET Upgrade Wizard, available in Visual Basic .NET, and higher, upgrades up to 95 percent of existing Visual Basic code and forms to Visual Basic .NET with new support for Web classes and User Controls.

Check Your Progress 1

Fill in the blanks:

1. The .NET Framework has two components the _____ and the _____.
2. VB.NET lets you create Windows services natively for the first time by providing a _____.
3. Full form of RAD is _____.

1.3 OPENING AND CLOSING WINDOWS

First, click Start, point to All programs, then point to Microsoft Visual Studio.NET, and finally click the program icon for Microsoft Visual Studio.NET. Relax; it may take a few seconds to open. If this is the first time you open the program, you will see the following window:

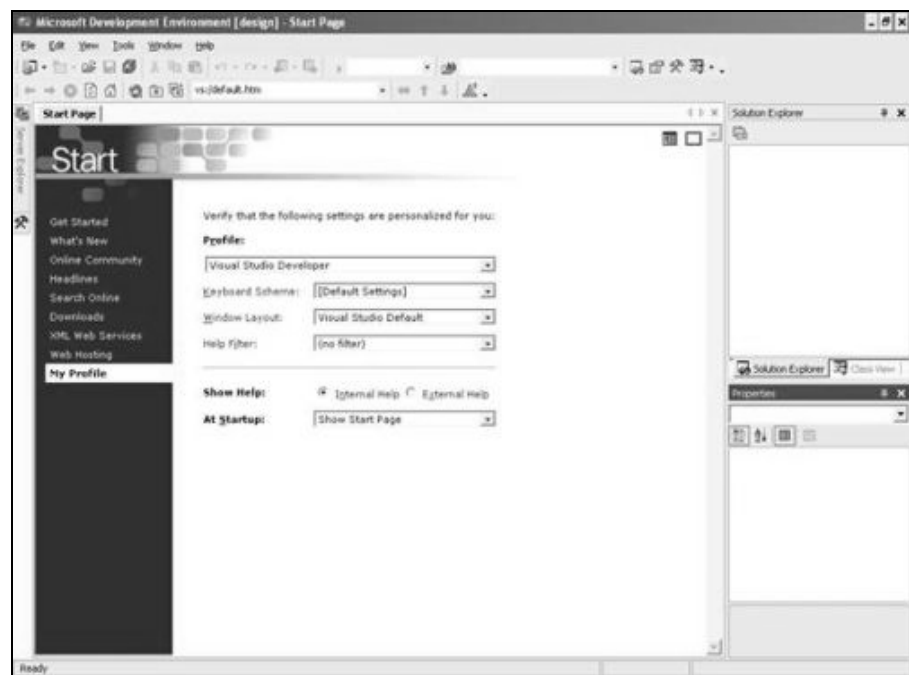


Figure 1.2: Opening screen of Microsoft Visual Studio.NET

Click Get Started, in the upper left. You should see the following:

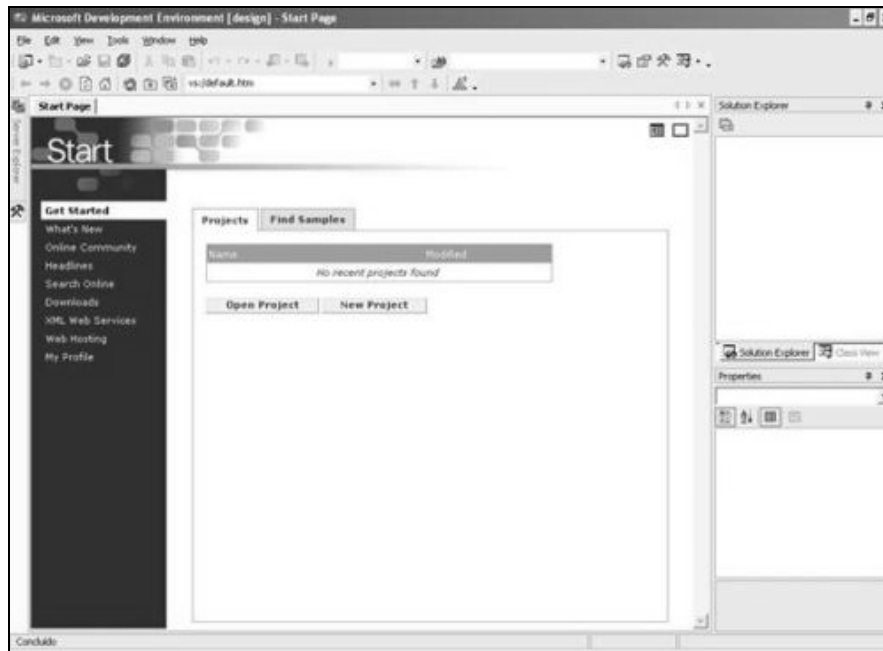


Figure 1.3: After clicking on get started

In this window, you can also see your recent projects. Of course, only if you have some. If you do, there will be a link you can click to open the project. However, only four links will be displayed here. If you have more than four recent projects, you can click Open Project and choose the project you want to open. You can also adjust this number to display up to 24 recent projects – not quite the thing for beginners. To change this number, click Tools, then Options and adjust the number accordingly. If you ever want to open your projects outside the development environment, go to this folder: C:\Documents and Settings\yourusername\My Files\Visual Studio Projects.

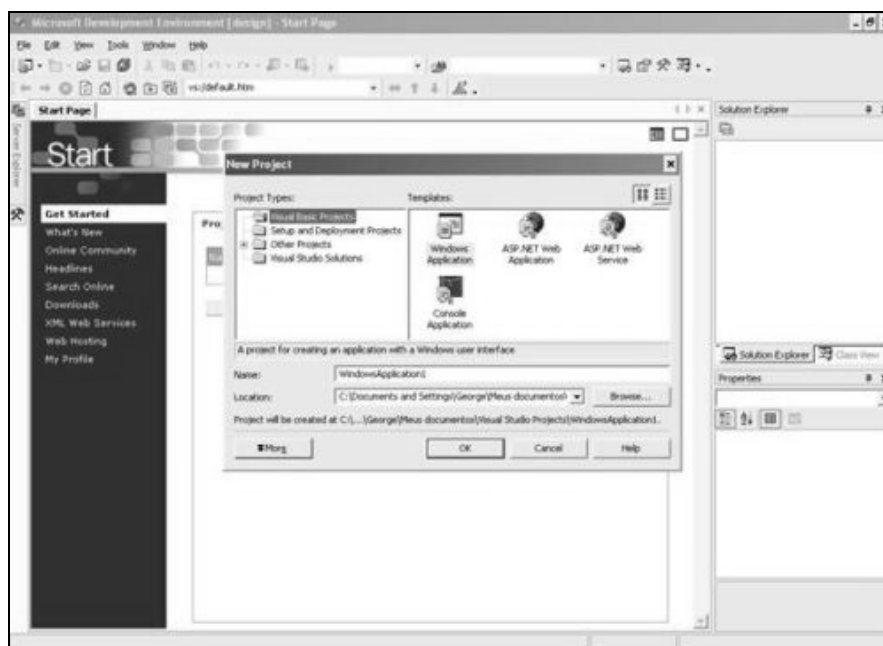



Figure 1.4: New project selection screen









To start a new project, click New Project (sounds obvious, doesn't it?). A new window will open. In the left pane, click Visual Basic Projects. In the right pane, click Windows Application. Double-click the name (WindowsApplication1) and type the name you would like to give your application. Click OK and wait until the project is created and opened. What you will see next should look like this:

After Microsoft Visual Studio has been opened, the screen you look at is called an Integrated Development Environment or IDE. The IDE is the set of tools you use to create a program.

The system icon  is used to identify the application that you are using. Almost every application has its own system icon. The system icon holds its own list of actions; for example, it can be used to move, minimize, maximize or close (when double-clicked) a window.

When you freshly start Visual Studio, the main section of the title bar displays the name of the application as Microsoft Developer Environment. Later on, if you start a project, the title bar would display the name of your project, followed by the name of the programming environment you selected.

The main section of the title bar is also used to move, minimize, maximize the top section of the IDE, or to close Visual Studio. On the right section of the title bar, there are three system buttons with the following roles:

Button		Role
		Minimizes the window
		Maximizes the window
		Restores the window
		Closes the window

Under the title bar, there is a range of words located on a gray bar. This is called the menu or main menu. Under the File menu Close, Save All, or Exit menu are exist.. For example, if you click Close, Microsoft Visual Studio will find out whether the current file had been saved already. If it has been, the file would be closed; otherwise, you would be asked whether you want to save it before closing it.

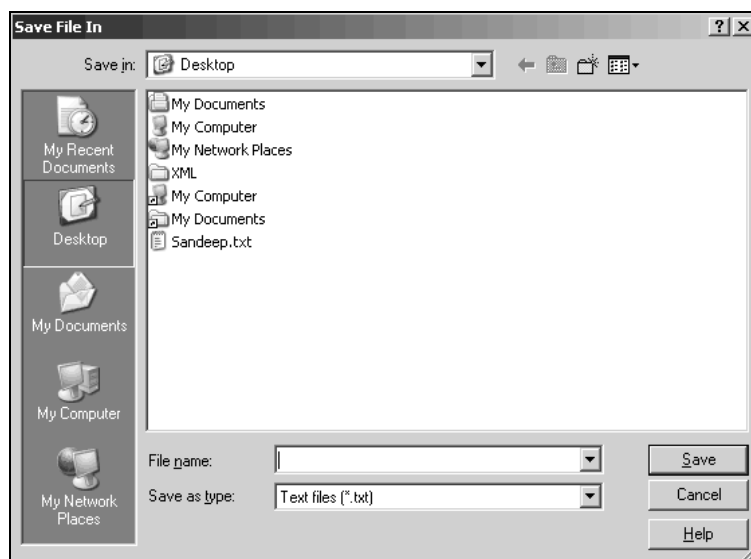


Figure 1.5: Save as screen

Check Your Progress 2

Fill in the blanks:

1. Visual Basic for Windows was debuted on _____.
2. The full form of IDE is _____.
3. Visual Basic .NET is a major component of _____.

1.4 LET US SUM UP

Visual Basic .NET is a major component of Microsoft Visual Studio .NET suite. Visual Basic .NET provides the easiest, most productive language and tool for rapidly building Windows and Web applications. Visual Basic .NET comes with enhanced visual designers, increased application performance, and a powerful integrated development environment (IDE). It also supports creation of applications for wireless, Internet-enabled hand-held devices. VB.NET. It's now a language that fully supports object-oriented programming.

1.5 LESSON END ACTIVITY

Describe the evolution of VB.net

1.6 KEYWORDS

Microsoft Visual Basic .NET: It is a programming environment used to create graphical user interface (GUI) applications for the Microsoft Windows family of operating systems.

Window: An instance of a program.

Integrated Development Environment or IDE: It the set of tools you use to create a program.

1.7 QUESTIONS FOR DISCUSSION

1. What is VB.net?
2. Describe the salient features of VB.NET.
3. Explain IDE.

Check Your Progress: Model Answers

CYP 1

1. .NET Framework class library, common language runtime
2. Windows Services project type
3. Rapid application development (RAD)

CYP 2

1. March 20, 1991
2. Integrated development environment
3. Microsoft Visual Studio .NET suite

1.8 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*, Pearson Edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic .Net*, 5th Edition, PHI

Kant, *Visual Basic.Net—A Beginners Guide*, TMCH.

LESSON

2

CONTROL CUSTOMIZATION

CONTENTS

- 2.0 Aims and Objectives
 - 2.1 Introduction
 - 2.2 Toolbars
 - 2.2.1 Adding a Toolbar
 - 2.2.2 Selecting the Images for the Buttons
 - 2.2.3 Adding the Buttons
 - 2.2.4 Writing the Button Code
 - 2.2.5 Other Toolbar Features
 - 2.3 Existing Project
 - 2.3.1 Open an Existing Project
 - 2.3.2 Save an Existing Project
 - 2.3.3 Import an Already Existing Form to a Project
 - 2.3.4 Add User Control to the Existing Project
 - 2.3.5 Inheriting a Form from an Existing Project
 - 2.4 Auto Hide
 - 2.5 Customizing Windows Placing Control on a Form
 - 2.5.1 Simplicity
 - 2.5.2 Positioning of Controls
 - 2.5.3 Consistency
 - 2.5.4 Aesthetics
 - 2.5.5 Shapes and Transparency
 - 2.6 Selecting and Resizing Control
 - 2.6.1 Single Control Selection
 - 2.6.2 Multiple Control Selection
 - 2.7 Relocating Control Properties of Windows
 - 2.7.1 Docking
 - 2.7.2 Anchoring as an Alternative Resizing Technique
 - 2.7.3 AutoScrolling Forms
 - 2.8 Let us Sum up
 - 2.9 Lesson End Activity
 - 2.10 Keywords
 - 2.11 Questions for Discussion
 - 2.12 Suggested Readings
-

2.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Concept of toolbar
- Different buttons on the toolbar
- Handling existing projects
- Idea of AutoHide
- How to customize the window

2.1 INTRODUCTION

The Toolbar Controls for Microsoft Office is a plug-in for Add-in Express that provides unique features for creating a commercial class user interface for your Microsoft Office add-ins. Using the Toolbar Controls you can easily create sophisticated user interfaces found in today's most recognizable commercial MS Office add-ins.

The Toolbar Controls for Microsoft Office adds one feature to Add-in Express. With the Toolbar Controls you can use any visual .NET controls, not only Office built-in controls, onto Outlook and Excel command bars. Now you can add any buttons, images, tree-views, grids, diagrams, edit and combo boxes, labels, reports to your command bars. It is what the Toolbar Controls is purposed for.

2.2 TOOLBARS

Almost every Windows application available has one—if not more—toolbar to enhance the user interface. Toolbars enable users to quickly get at an application's commonly used functions. Depending on the application, one or more toolbars could help with specific tasks, such as the Editor Toolbar in the Visual Basic IDE. Because toolbars are so widely used, most users now expect any desktop application they use to have them.

Adding toolbars to your application has become fairly easy with the Toolbar control supplied with Visual Basic. Creating a simple or complicated toolbar requires that you add the following controls to the form:

- The Toolbar control sets up the buttons of the actual toolbar displayed to users and handles user requests.
- The ImageList control contains the bitmaps used on the toolbar buttons.

To see how these controls interact to form an application toolbar, start a new project and name it Toolbar.

2.2.1 Adding a Toolbar

Here you see how to use the Toolbar control with the ImageList control. Adding a toolbar is as easy as adding the controls to the form and setting some of their properties. Of course, you still need to add the code to make anything actually happen in the application.

2.2.2 Selecting the Images for the Buttons

To create a toolbar, place an ImageList control on your form. To begin the process, add the images you want to use on the toolbar by using the Image Collection Editor

(see Figure 2.1). You can display this by clicking the Images button in the ImageList properties list.

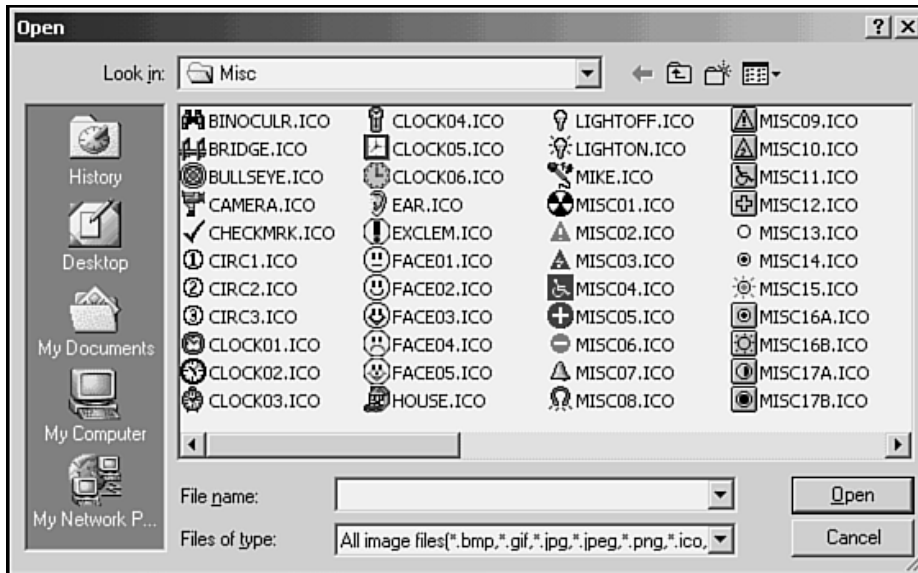


Figure 2.1: Using the Image Collection Editor to add images to the ImageList control

To add an image to the Image collection for the ImageList control, click the Add button. In the Open dialog box that appears, select the image you want to add. When you select the image and click Open, the image is added to the Members list as shown in Figure 2.2.

Find the New and Open bitmaps in /Program Files/ Microsoft Visual Studio.NET/Common7/Graphics/Bitmaps/TLBR_W95 directory and add them to the Image collection. When you're finished, click OK to close the Image Collection Editor.

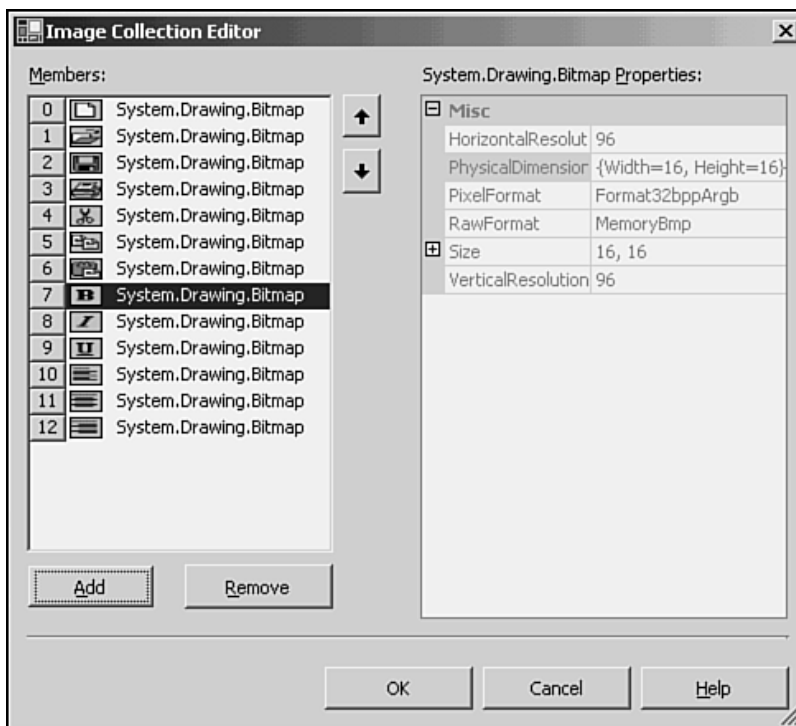


Figure 2.2: Displaying the images already included in the Image collection

To add the toolbar to your project, follow these steps:

1. Place a Toolbar control on the form. It doesn't really matter where you put it; the default Dock property places it at the top of the form.
2. Set the ImageList property to the ImageList control you've just added. This property identifies which ImageList control (if you have more than one on the form) the Toolbar control will use to provide the images for the buttons. (Now you can see why you had to set up the ImageList control first.)

2.2.3 Adding the Buttons

The real action starts when you create the buttons for the toolbar. You'll actually add the buttons by using the ToolBarButton Collection Editor (see Figure 2.3).

To add a button to the toolbar, perform the following steps; then if you want to reposition the button, use the up and down arrows to move it to the correct position on the toolbar.

1. Click the collection button for the Toolbar's Buttons property.
2. In the ToolBarButton Collection Editor, click Add to add a new button to the toolbar.
3. Select the image you want on the button by setting the ImageIndex property, using its associated drop-down list (see Figure 2.3).



Figure 2.3: The ToolBarButton Collection Editor

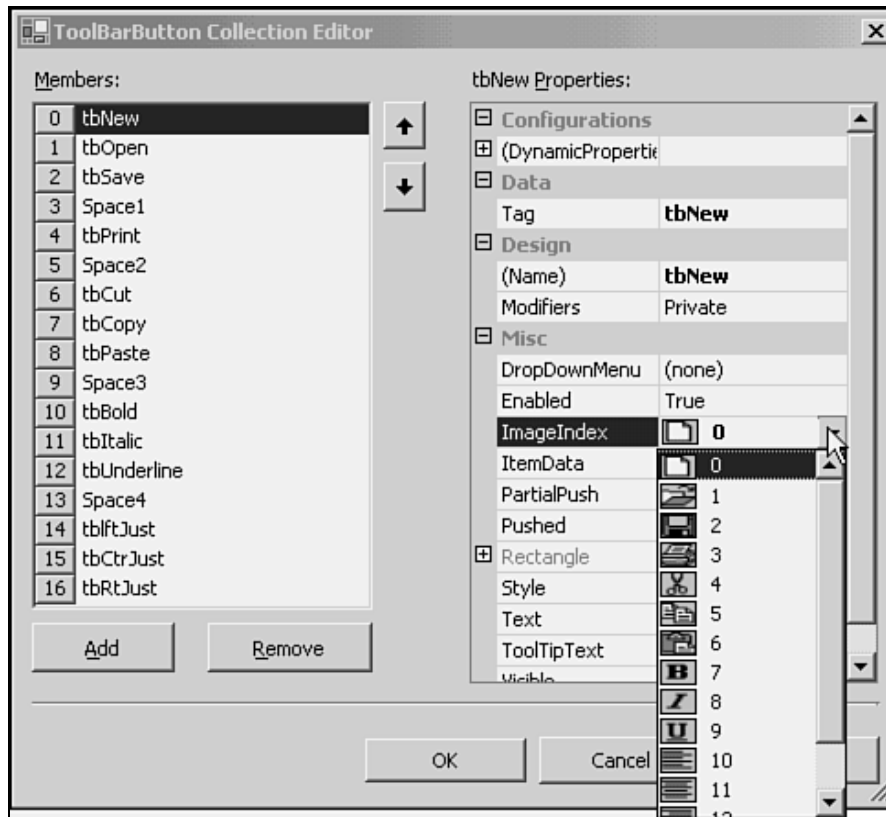


Figure 2.4: Selecting the image to use for a toolbar button

For each button you add, you will need to specify the following properties:

- The **ItemData** property specifies a string that you can use to identify the button in your code. The value of this property must be unique for each button. You should assign a string that's meaningful to you so you can easily remember it when you're writing your code.
- The **ImageIndex** property specifies the index of the image you want to appear on the button face. The index corresponds to the index of the picture in the **ImageList** control. A value of zero for the **ImageIndex** property will give you a button without an image.
- The **Style** property determines the type of button you're creating. The button type also determines how the button will behave in the toolbar. The following table lists the different **Style** property settings.

Setting Button Behavior with the **Style** Property

Setting	Description
PushButton (default)	Creates a standard push button
ToggleButton	Indicates that an option is on or off
Separator	Provides a space between other buttons
DropDownButton	Displays a drop-down menu list when the button is clicked

Also, you can set several optional properties for each button:

- Text displays text beneath the image on a button.
- ToolTipText appears when the mouse is placed on the button. (This text appears only if the ShowTips property of the toolbar is set to True.)
- Pushed sets or returns the current state of the button.
- If you set a button style property to DropDownButton, you must set the DropDownMenu property for that button in the ToolBarButton Collection Editor (see Figure 2.5).

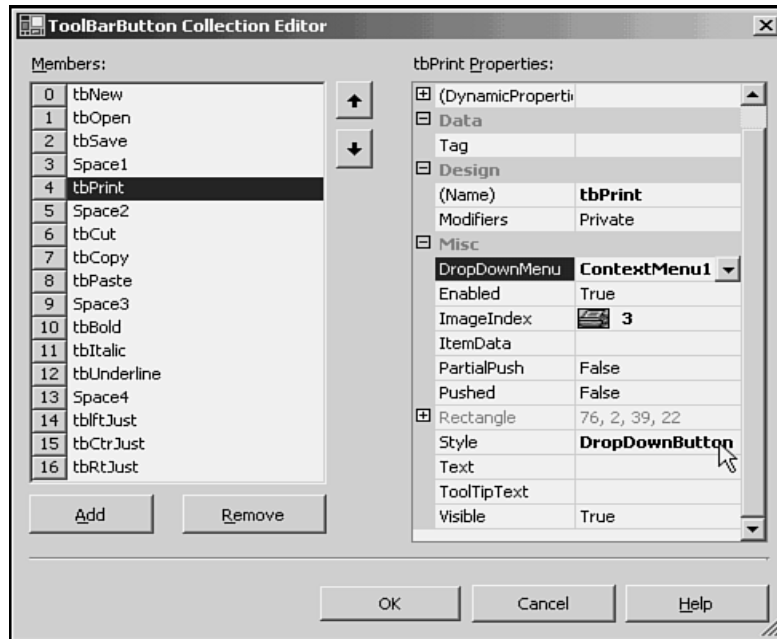


Figure 2.5: Setting a menu list on the ToolBarButton Collection Editor

This DropDownMenu feature of the Toolbar offers the functionality included in Visual Basic (see Figure 2.6) and many other Windows-based products.

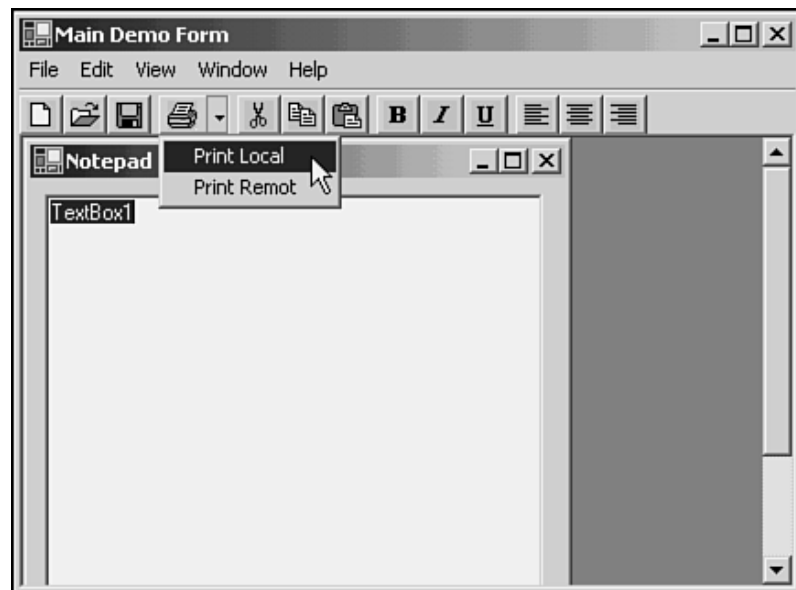


Figure 2.6: Using drop-down menus from a toolbar

After you add the buttons to your toolbar, click OK to close the Collection Editor and save the changes. Your form should look like the one in Figure 2.7.

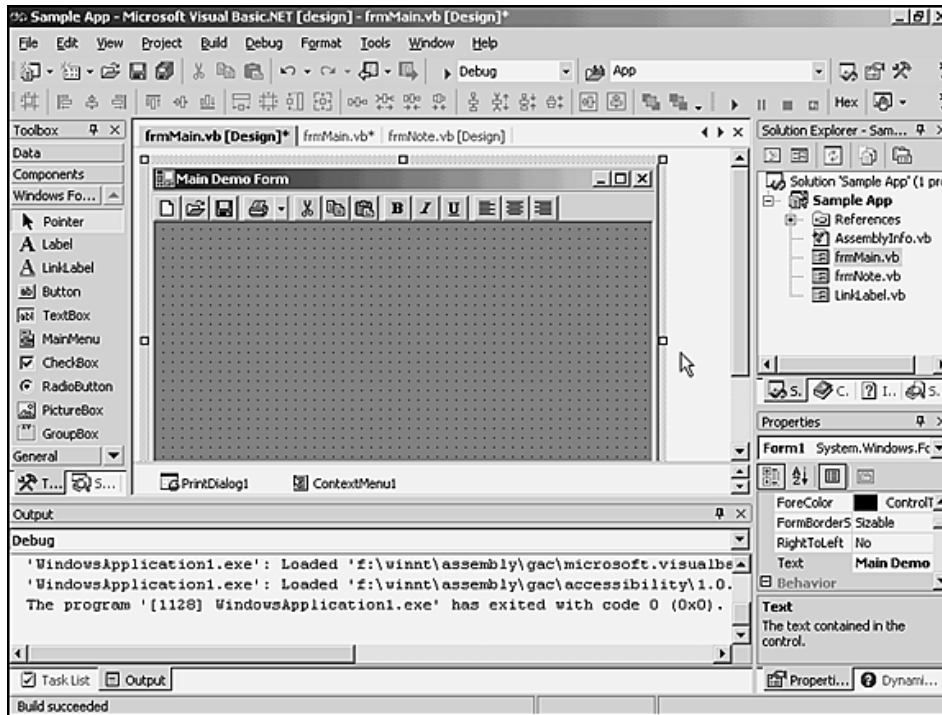


Figure 2.7: The final project form with the Toolbar and ImageList controls set

2.2.4 Writing the Button Code

You now have a toolbar on your form. If you execute the application, you can click the buttons and see them respond. However, until you add code to the toolbar's events, the buttons won't perform any functions because they don't have any events of their own.

ButtonClick is the toolbar event in which you'll place your button code. This event passes a set of event arguments in an object to the event procedure called e, which allows you to access all the button's properties. In your code, use the value of the ItemData property to determine which button was actually clicked. The following source code is typical for taking actions based on buttons clicked:

```
Private Sub ToolBar1_ButtonClick(ByVal sender As System.Object,
    ByVal e As System.Windows.Forms.ToolBarButtonClickEventArgs)
    Handles ToolBar1.ButtonClick
    Select Case e.Button.ItemData
        Case "New"
            `ToDo: Add `New' button code.
            MsgBox("Add `New' button code.")
        Case "Open"
            `ToDo: Add `Open' button code.
            MsgBox("Add `Open' button code.")
    End Select
End SubTIP
```

In an actual application, these Case statements should call the same code routine as the related menu options. This allows you to program the action once and then call it from both the menu and the toolbar. Doing this makes it easier to maintain your code because then any changes or corrections have to be made only once.

Notice that each Case statement in the Select statement will execute based on the clicked button. Now your toolbar is ready to have the remaining sections of your application code added.

2.2.5 Other Toolbar Features

One of the newest features to be added to Visual Basic is the ToolTip textbox feature. Almost every object and control in Visual Basic can display ToolTips. Using the toolbar, if you set the ShowTips property to True you can specify unique text for each toolbar button. Text placed in the ToolTipText property for a button is displayed during runtime when the mouse pointer remains on a button for a short period of time (see Figure 2.8).

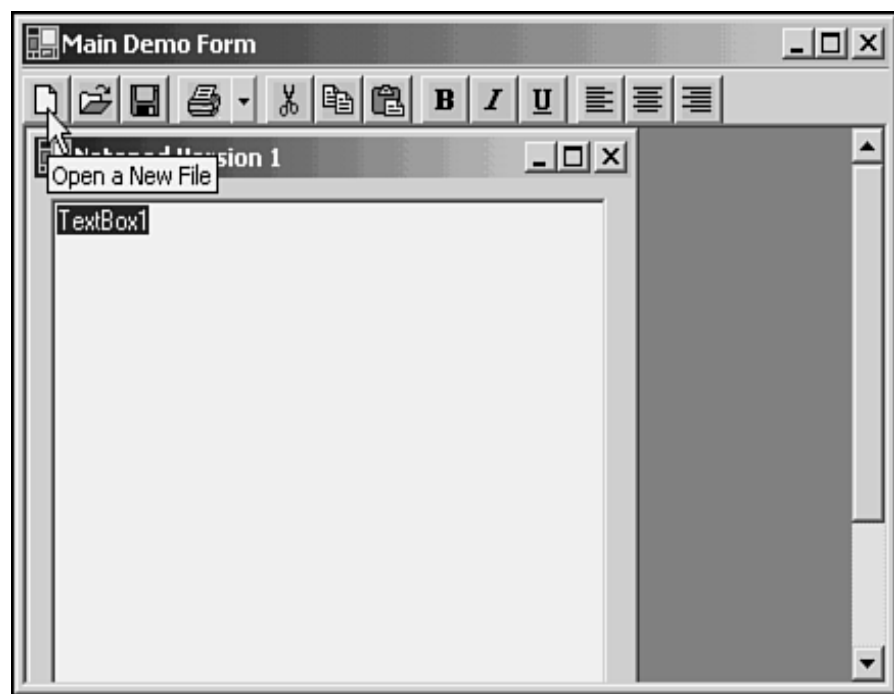


Figure 2.8: Use ToolTips to inform users what function each button performs

Check Your Progress 1

Fill in the blanks:

1. The ImageList control contains the _____ used on the toolbar buttons.
2. A value of _____ for the ImageIndex property will give you a button without an image.
3. The Style property determines the _____ of button you're creating.

2.3 EXISTING PROJECT

Different activities which you can perform with your existing project are described below:

2.3.1 Open an Existing Project

From the Start Page click 'Open Project'

OR From the menu, select File -> Open -> Project

Then, go to the directory containing your project, select your project name (the one with "Visual

Basic .NET project" file type

- If your project is listed on the Start Page, just click on the project name

OR

- Open the project's folder in Windows Explorer, and double click on file with extension .vbproj or

file description "Visual Basic .NET Project"

2.3.2 Save an Existing Project

You can save the existing project by selecting Select File -> Save all.

Select the specific folder WHERE you want to save the project. When you click Save, your form is saved with the name you gave it, and an extension ".frm"

Then, another window opens where you will save the project files. Do so using the same filename and in the same folder. VB will add the ".vbp" extension.

When you want to exit the project, you will be asked if you want to save changes. Say yes, because at this time VB will make a third file, the .vbw file

When working on the hard drive, follow these directions, too. Be sure you know where your files are, and that you have them all in the same folder. When you want to transfer something to a floppy to have at school, just copy the whole folder, making sure that all three files are in the folder.

Whatever you do, do not just drag files into a folder. The project file "remembers" where the form file was originally located. The .vbw file has recorded the path to the original forms and folders. Make sure the files are saved into the same folder. You can do "Save Project As" and "Save Form As" from the File menu to see where they are, and then cancel the "Save As" if they are OK. If the form file is not in the same folder as the project file, then use the "Save form As" command to put the form file into the right place.

2.3.3 Import an Already Existing Form to a Project

1. If you already have a project then open it, otherwise create a new project
2. Under Solution Explorer, right-click on your project name select Add-> Add existing item,
3. When the dialog box "Add Existing Item" show up, go to the directory containing your already existing form and select the form (select file .vb)
4. Click 'Open'

2.3.4 Add User Control to the Existing Project

You can add user control to the existing project by selecting Project->Add User Control. The image below displays the new project dialogue to add a User Control project.

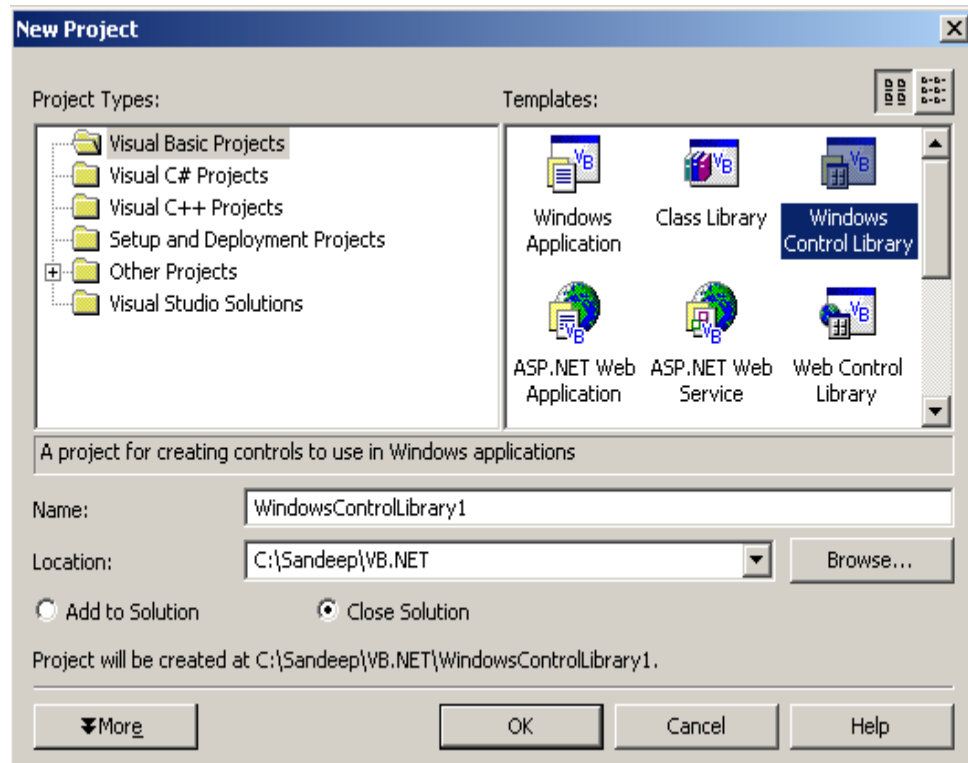


Figure 2.9

2.3.5 Inheriting a Form from an Existing Project

You can also inherit a form from the existing project. To inherit a form from other project, navigate to the project containing the form you want using the browse button in the Inheritance Picker dialog, click the name of the DLL file containing the form and click Open. This returns to the inheritance Picker dialog box where the selected project is now listed. Choose the appropriate form and click OK. A new inherited form is added to your project.

2.4 AUTO HIDE

Once the initial VB.NET screen appears, you'll find the controls stored inside of the toolbox on the left side of the screen. These are the objects you can place on a form. Users interact with these objects. You can run your mouse over each object and a tool tip will appear informing you of the object's purpose.

The direction of the push pin on the toolbox's title bar dictates whether or not auto hide is in effect. If the push pin is vertical, auto hide is turned off. This means that the toolbox is always visible. If you click on the push pin, it will be displayed vertically. At this point, auto hide is activated and moving your cursor off the toolbox will cause it to collapse. It will only expand when your cursor is over the toolbox icon that appears docked against the left corner of the window. Other windows also have autohide capabilities activated and deactivated by clicking the window's respective push pin.

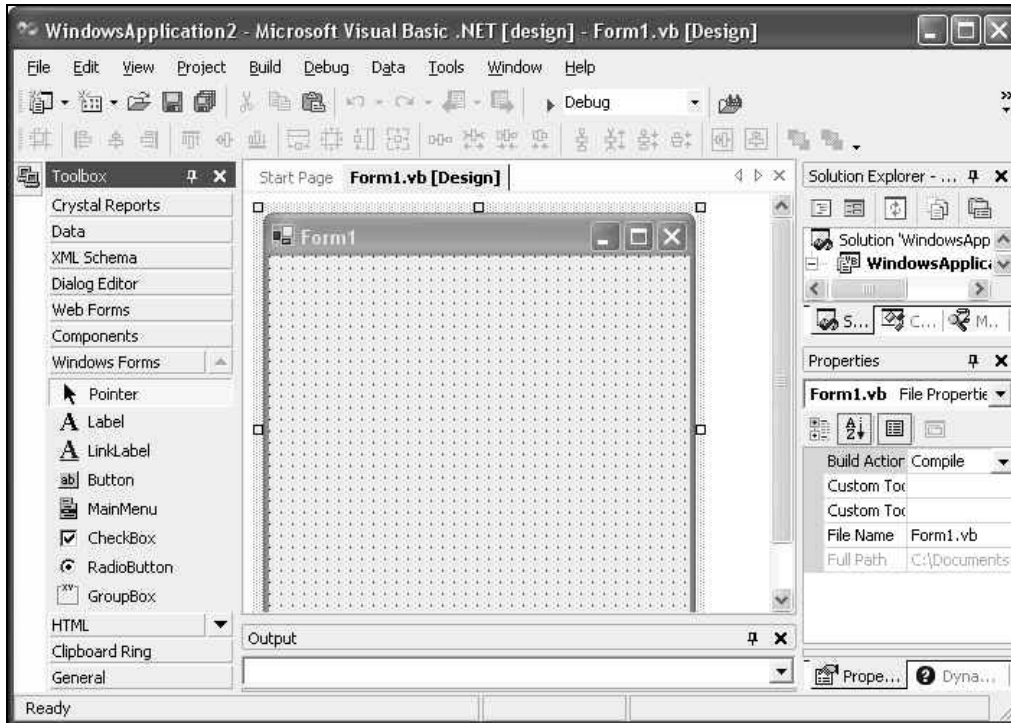


Figure 2.10: A window showing the push pin

2.5 CUSTOMIZING WINDOWS PLACING CONTROL ON A FORM

Customizing window and placing control on a form is a very important factor when we design our application (project). A form provides the interface for end users to interact with the application. End users are called target audience. Designing a good user interface which is easy to use and understand is crucial for a successful application. If you already know the target audience for whom you are developing the application then designing becomes simple as you will already be familiar with their corporate colors and likes. Target audience, for example, can be the employee's of a firm for whom you will design the application. A well designed user interface makes it easy and simple for the target audience to understand and use. On the other hand, a poorly designed user interface will be hard to understand and use and can lead to distraction and frustration.

Good user interface is possible if it is designed keeping in mind the following four principles:

1. Simplicity
2. Positioning of Controls
3. Consistency
4. Aesthetics

The image below is an example of a good user interface design.

New Account Registration

First Name

Last Name

Date of Birth

Sex ☐ Male ☐ Female

Address 1

Address 2

State

Figure 2.11: A well designed user interface

The above said four principles explained:

2.5.1 Simplicity

Simplicity is a key factor when designing a user interface. If a user interface looks crowded with controls then learning and using that application will be hard. Simplicity means, the user interface should allow the user to complete all the required tasks by the program quickly and easily. Also, program flow and execution should be kept on mind while designing. Try to avoid use of flashy and unnecessary images that distract the user. The simpler the user interface, the more friendly and easy it will be.

2.5.2 Positioning of Controls

Positioning of controls should reflect their importance. Say, for example, if you are designing an application that has a data-entry form with textboxes, buttons, radio buttons, etc. The controls should be positioned in such a way that they are easy to located and matches the program flow. Like, a submit button should be placed at the bottom of the form so that when the user enters all the data he can click it straight away. The image above is a perfect example of positioning of controls.

2.5.3 Consistency

The user interface should have a consistent look through out the application. The key to consistency lies during the design process. Before developing an application, we need to plan and decide a consistent visual scheme for the application that will be followed throughout. Using of particular fonts for special purposes, using of colors for headings, use of images, etc are all part of consistency.

2.5.4 Aesthetics

An application should project an inviting and pleasant user interface. The following should be considered for that.

Color : Use of color is one way to make the user interface attractive to the user. The color which you select for text and back-ground should be appealing. Care should be taken to avoid gaudy colors that are disturbing to the eye, for example, black text on a red back-ground.

Fonts : The fonts which you use for text should also be selected with care. Simple, easy-to-read fonts like Verdana, Times New Roman should be used. Try to avoid bold, strikeout text in most parts of the application. Use of bold, italics and other formatting should be limited to important text or headings.

Images : Images add visual interest to the application. Simple, plain images should be used wherever appropriate. Avoid using flashing images and images that are not necessary but are used only for show off.

2.5.5 Shapes and Transparency

NET Framework provides tools that allow us to create forms and controls with different levels of opacity. Apart from using traditional shapes like rectangles, etc, these tools also allow us to draw our own shapes which can provide some very powerful visual effects. User drawn shapes should be used only if the application requires it and care should be taken that the shapes which are drawn do not disturb the eye.

The image below is an example of a poorly designed user interface.

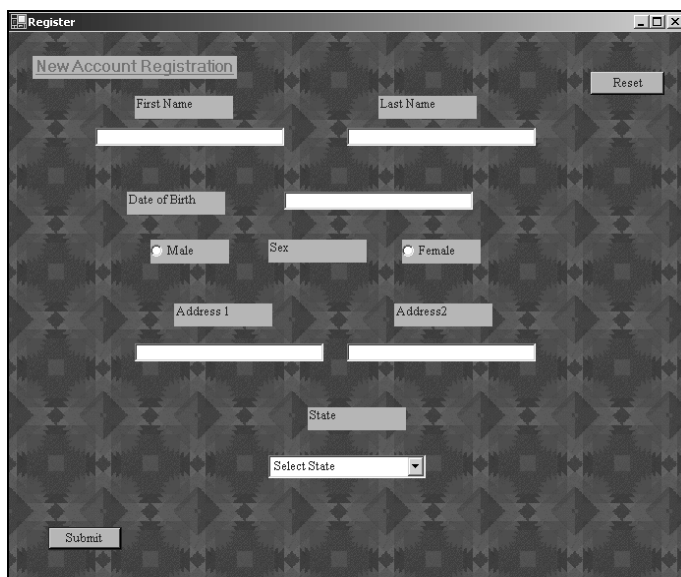


Figure 2.12: A poorly designed user interface

2.6 SELECTING AND RESIZING CONTROL

If you visually add a control to a form (at design time), in order to perform any type of configuration on the control, you must first select it. Sometimes you will need to select a group of controls.

2.6.1 Single Control Selection

To select a control, if you know its name, you can click the arrow of the combo box in the top section of the Properties window and select it.

To select a control on the form, you can simply click it. A control that is selected indicates this by displaying 8 small squares, also called handles, around it. Between these handles, the control is surrounded by dotted rectangles. In the following picture, the selected rectangle displays 8 small squares around its shape:

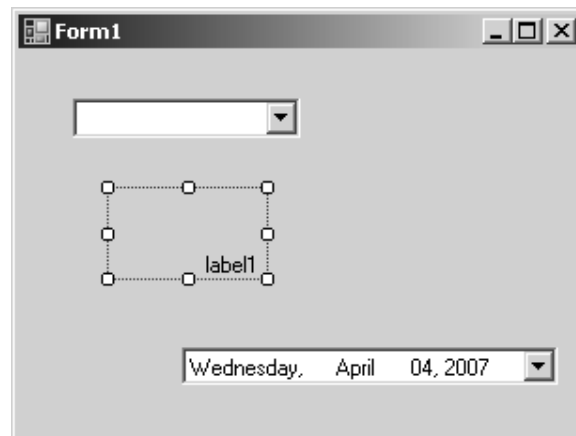


Figure 2.13: A single control selection

2.6.2 Multiple Control Selection

To select more than one control on the form, click the first. Press and hold either Shift or Ctrl, then click each of the desired controls on the form. If you click a control that should not be selected, click it again. After selecting the group of controls, release either Shift or Ctrl that you were holding. When a group of controls is selected, the last selected control displays 8 handles too but its handles are white while the others are black. In the following picture, a form contains four controls, three controls are selected:

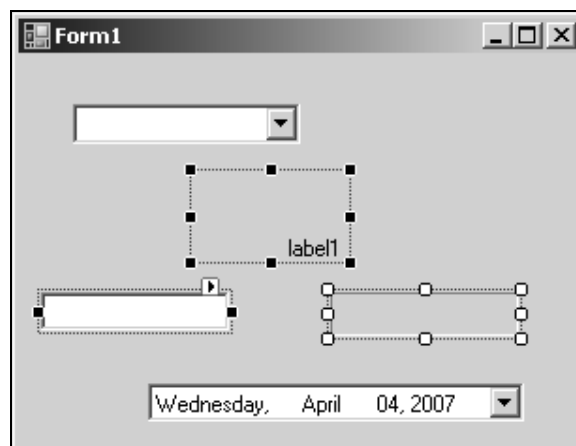


Figure 2.14: Multiple control selection

Another technique you can use to select various controls consists of clicking on an unoccupied area on the form, holding the mouse down, drawing a fake rectangle, and releasing the mouse:

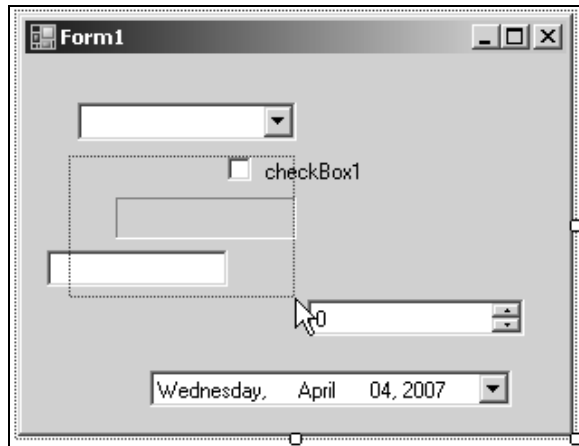


Figure 2.15: More than one control selection

Every control touched by the fake rectangle or included in it would be selected:

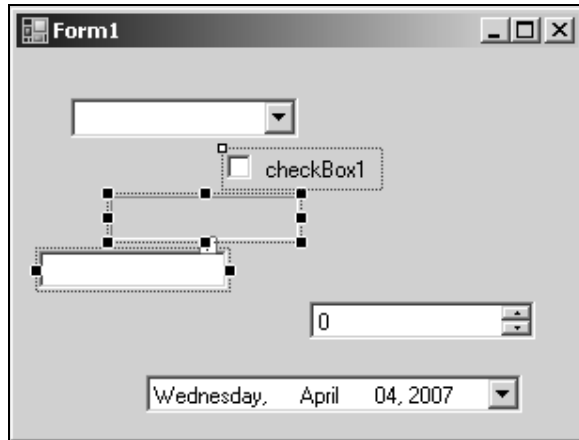


Figure 2.16: Multiple control selection

When various controls have been selected, you can resize them by click and drag process.

Of course, you can set the control's properties by coding also. You can set these properties before or after adding it to the form, but after the control is instantiated. A control has properties such as Left, Top, Width and Height for its size and position. A Button control also has the Text property that will appear as the text for the button object.

The code stated below brings a form to life and adds two controls, a Button and a TextBox, to the control and defines their size.

Adding controls to a form.

```
Imports System.Windows.Forms
```

```
Public Class Form1
```

```
    Inherits Form
```

```
    ' Control declaration: a Button and a TextBox
```

```
    Private Button1 As Button
```

```
Private TextBox1 As TextBox
    Public Sub New()
        InitializeComponent()
    End Sub
Private Sub InitializeComponent()

    Me.Text = "Developer Form"
    Me.Width = 400
    Me.Height = 300

    Button1 = New Button()
    TextBox1 = New TextBox()

    Button1.Left = 200
    Button1.Top = 200
    Button1.Width = 100
    Button1.Height = 40
    Button1.TabIndex = 0
    Button1.Text = "Click Me"

    TextBox1.Left = 200
    TextBox1.Top = 30
    TextBox1.Width = 150
    TextBox1.Height = 40

    Me.Controls.Add(Button1)
    Me.Controls.Add(TextBox1)
End Sub
End Class
```

Check Your Progress 2

Fill in the blanks:

1. Simplicity is a key factor when _____ a user interface.
2. A control that is selected indicates this by displaying 8 small squares, also called_____, around it.

2.7 RELOCATING CONTROL PROPERTIES OF WINDOWS

Docking and anchoring are designed to relocate your controls when the form is resized.

2.7.1 Docking

A docked item is attached to one of the edges of its container (such as a form). In addition to docking to one of the four sides, controls also support a fifth (sixth if you count None for no docking) docking setting, Fill. If you set the Dock property to Fill, that control becomes attached to all four sides of the container, adjusting itself automatically as the form is resized. You cannot adjust any size or position settings for a control that has been docked to fill the container.

Consider a form that contains only a single ListBox control. If you set the ListBox's Dock property to Left (see Figure 2.17), the height of the control will be fixed to the height of the form's client area (causing the list box to fill the form from top to bottom), while the position of the control will be locked to the left side of the form. The only thing you can change about the size of the list box at this point is its width, controlling how far out from the left side it extends. Docking to the right is essentially the same; the list box becomes attached to the right side of the form, but you can still change its width as desired. Docking to the top or bottom will cause the width to be fixed to fill the width of the form, but the height of the control can still be modified.

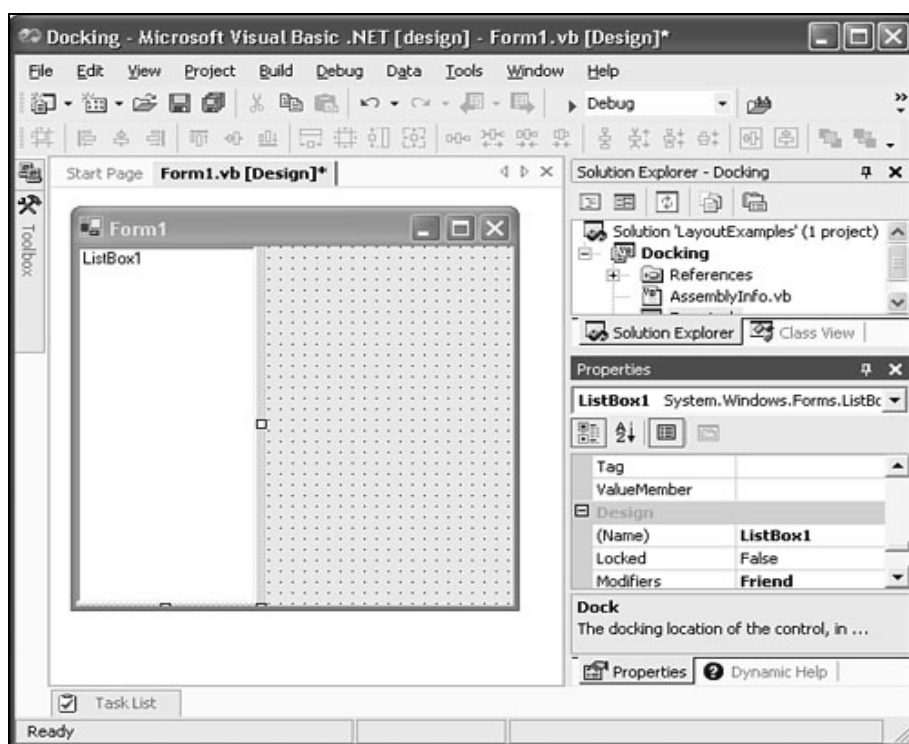


Figure 2.17: Docking a control

Remember that you are docking the control to its container, which in this example is the Form, but could be a container control such as a GroupBox or Panel.

The container (form, panel, or other type of container control) has a DockPadding property, which allows it to specify a certain amount of padding between it and docked controls. The padding values can be specified individually for the four sides of the container, or an overall padding value that applies to all of the sides at once. If a container has specified a DockPadding value of 10, for example, a control docked to the left will be positioned 10 pixels away from the left edge. The DockPadding setting is great for creating a more visually pleasing user interface as it results in a border

around the form while still enabling the automatic resizing of docked controls (see Figure 2.18).

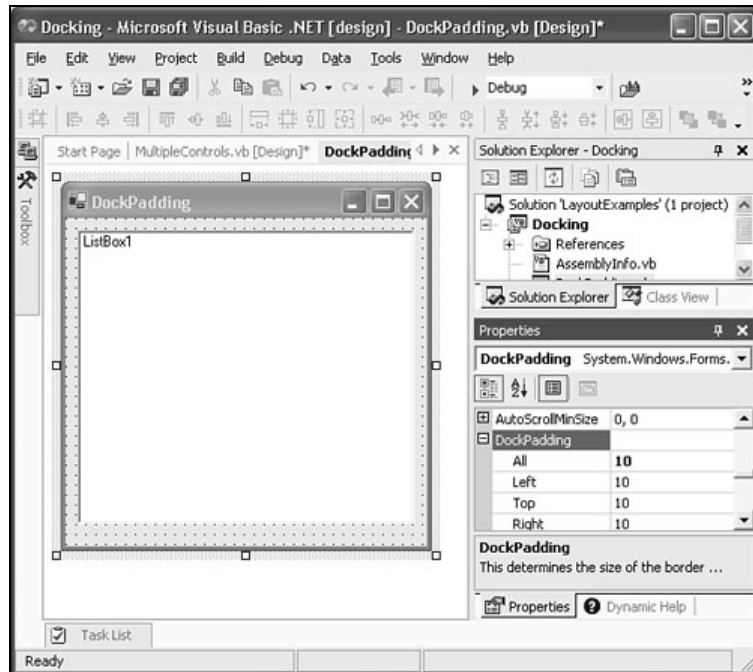


Figure 2.18: DockPadding

Docking gets a little more complicated when multiple docked controls are involved. If you dock more than one control to the same edge, the second control will dock alongside the first instead of directly to the container. Going back to the example with the ListBox on a form, you can try multiple docked controls to see what happens. If you docked the ListBox to the bottom and then added a new DataGridView to the form, setting its Dock property also to Bottom, you would have produced an interface similar to Figure 18, where the ListBox appears below the DataGridView.

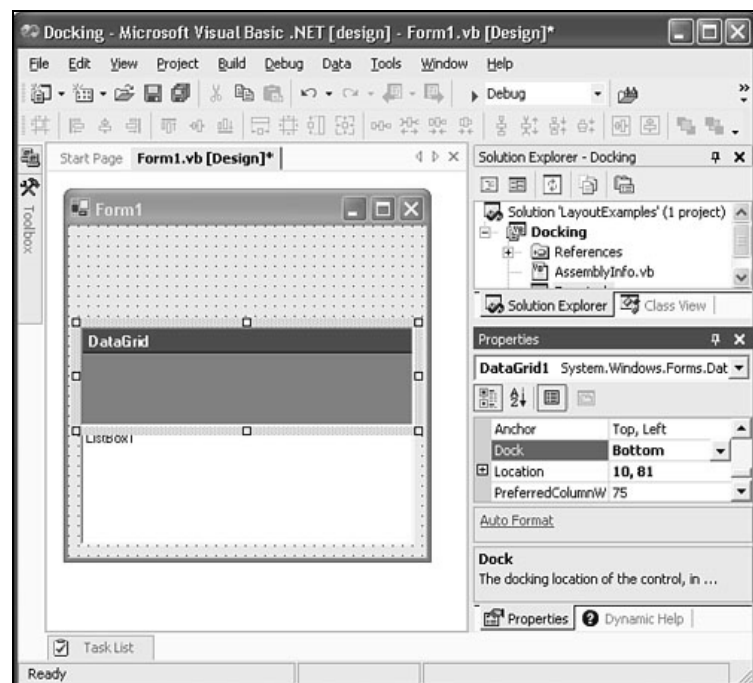


Figure 2.19: Multiple controls docked to the same side will stack instead of overlap

Both of the controls are docked to the form and will resize as the form is resized. If you have controls docked to one or more sides of your container, and then you set another control's Dock property to Fill, the control set to Fill will be automatically sized to use all of the remaining area of the container. If you have multiple controls docked on a form, you might want to use the Splitter control. Splitter is a special Windows Forms control that, when docked between two other controls, allows you to resize the two controls at runtime. Using the Splitter control and a few other key controls, you can create a standard Explorer view form in a matter of minutes.

To add a splitter to your form, you need to be careful of the order in which you add your controls. Try adding a ListBox to an empty form, and docking it to the left. Then add a splitter control, and dock it to the left as well (it is by default). Finally, add a DataGrid control, dock it to the left as well, or set its dock property to Fill, and you will have a working example of using a splitter!

2.7.2 Anchoring as an Alternative Resizing Technique

After docking, this has to be the coolest layout feature. Anchoring is a little simpler than docking, but it can be a powerful tool. Using a graphical property editor (see Figure 2.20), you can set the Anchor property for a control to any combination of Top, Left, Bottom, and/or Right.

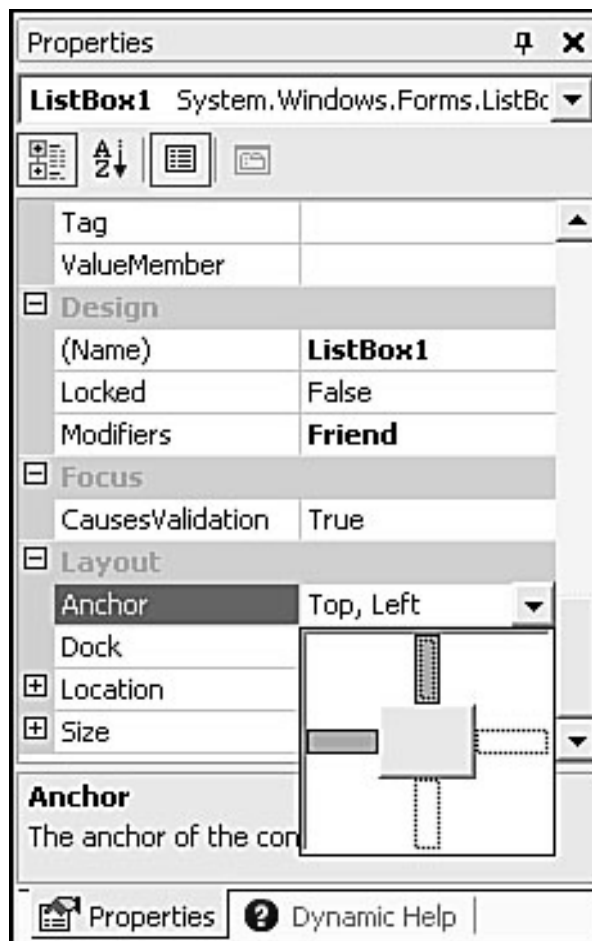


Figure 2.20: The property editor

To anchor a control to a specific side means that the distance between the control and that side of its container becomes fixed. Therefore, if you anchor a control to a specific side and then resize the form, the control's distance from the anchored side(s)

will not change. To maintain a constant position relative to one or more sides of your container, the control might have to be resized when the form's size changes.

By default, controls are anchored to the top and left, which makes them behave exactly as controls did in previous versions of Visual Basic. When you resize the form, they do not move or resize. If you want to create a TextBox that grows as you make your form wider, you can anchor it to the top, left, and right sides. If you want the TextBox to grow in height as well as width, anchor it to the bottom as well.

2.7.3 AutoScrolling Forms

In some cases there is a minimum size at which your form is usable, so resizing below that needs to be avoided. There are also situations when the content on your form is a fixed size, making resizing inappropriate. Windows forms provides a few additional features to allow you to deal with these situations. Forms have minimum/maximum height and width properties (allowing you to constrain resizing to a specific range of sizes) and the AutoScroll feature. AutoScroll allows a form to be resized by the users, but instead of shrinking the controls on the form, scroll bars appear to allow the users to view the entire form area even if they have resized the window. The form shown in Figure 2.21 is a perfect candidate for AutoScroll; it contains a large number of controls and buttons and cannot be resized using docking or anchoring.

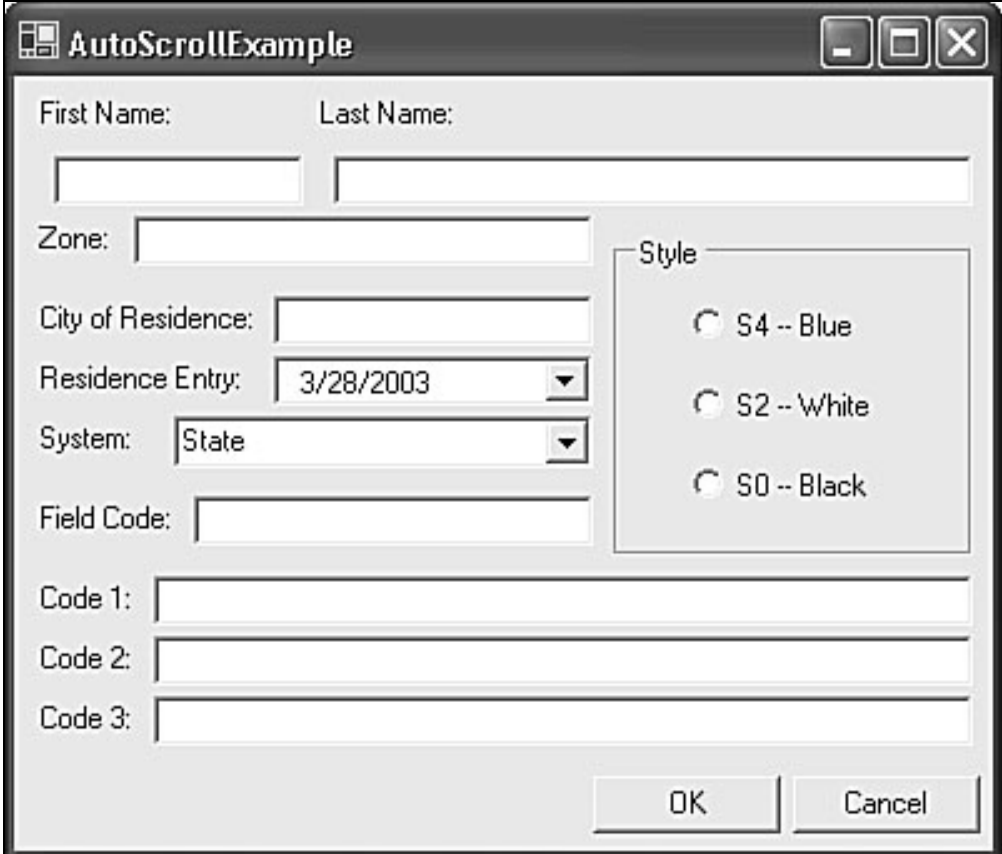
The image shows a Windows form titled "AutoScrollExample". It contains several input fields: "First Name:", "Last Name:", "Zone:", "City of Residence:", "Residence Entry:" (with a date "3/28/2003" and a dropdown arrow), "System:" (with "State" and a dropdown arrow), "Field Code:", "Code 1:", "Code 2:", and "Code 3:". To the right of these fields is a "Style" panel with three radio buttons: "S4 -- Blue", "S2 -- White", and "S0 -- Black". At the bottom right are "OK" and "Cancel" buttons. The form is designed to be large, demonstrating the need for AutoScrolling.

Figure 2.21: This form would be hard to resize, so the solution is to allow users to scroll

If the user were to resize this form, making it smaller than the area required for all of its controls, the AutoScroll feature of Windows forms will save the day by adding horizontal and/or vertical scroll bars as required (see Figure 2.22).

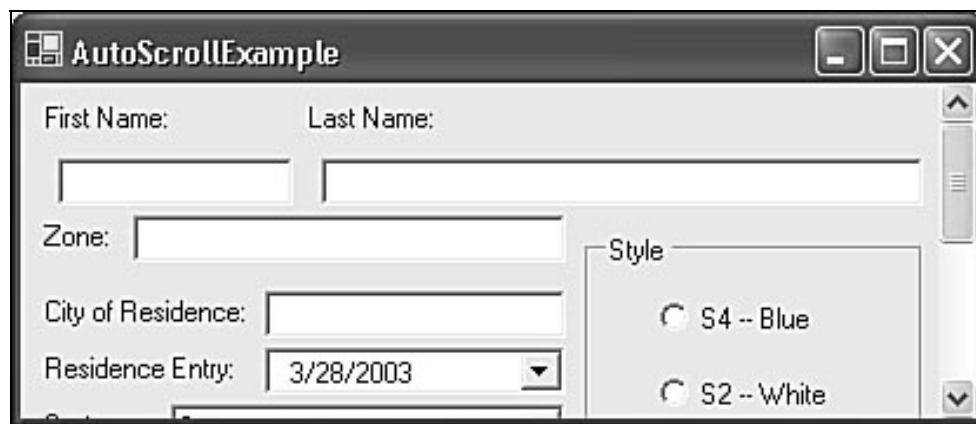


Figure 2.22: AutoScroll

In addition to the AutoScroll property, which you set to True to enable auto scrolling, there are two other properties, AutoScrollMargin and AutoScrollMinSize, that are used to configure exactly how scrolling occurs.

Check Your Progress 3

Fill in the blanks:

1. If the push pin is vertical, auto hide is _____.
2. A _____ item is attached to one of the edges of its container
3. _____ is the toolbar event in which you'll place your button code.

2.8 LET US SUM UP

Adding toolbars to your application has become fairly easy with the Toolbar control supplied with Visual Basic. The Toolbar control sets up the buttons of the actual toolbar displayed to users and handles user requests. AutoScroll allows a form to be resized by the users, but instead of shrinking the controls on the form, scroll bars appear to allow the users to view the entire form area even if they have resized the window. Customizing window and placing control on a form is a very important factor when we design our application (project). A form provides the interface for end users to interact with the application. Docking and anchoring are designed to relocate your controls when the form is resized. A property is a piece of information that characterizes a control.

2.9 LESSON END ACTIVITY

What are the properties that need to specify for each button?

2.10 KEYWORDS

Toolbar: It enables users to quickly get at an application's commonly used functions.

Docking: A docked item is attached to one of the edges of its container and it is designed to relocate the controls when the form is resized.

Anchoring: To anchor a control to a specific side means that the distance between the control and that side of its container becomes fixed.

AutoScroll: This feature allows a form to be resized by the users, but instead of shrinking the controls on the form by appearing scroll bars

ToolTips: It is the text which is displayed during runtime when the mouse pointer remains on a button for a short period of time

2.11 QUESTIONS FOR DISCUSSION

1. What is docking? What is its application?
2. How to add a toolbar to your project?
3. How to save an existing project?
4. Write short notes on
 - a. Auto scrolling form
 - b. Toolbar
 - c. Anchoring

Check Your Progress: Model Answers

CYP 1

1. bitmaps
2. zero
3. type

CYP 2

1. designing
2. handles

CYP 3

1. Turned off
2. Docked
3. ButtonClick

2.12 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*, Pearson edition, 2004

MSDN *Visual studio Library*.

Schneider, *An Introduction to Programming using Visual Basic .Net*, 5th Edition, PHI

Kant, *Visual Basic.Net— A beginners guide*, TMCH

LESSON

3

PROPERTY SETTING

CONTENTS

- 3.0 Aims and Objectives
- 3.1 Introduction
- 3.2 Setting Properties of Form and Control
 - 3.2.1 Properties Categories
- 3.3 Let us Sum up
- 3.4 Lesson End Activities
- 3.5 Keywords
- 3.6 Questions for Discussion
- 3.7 Suggested Readings

3.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- An overview different fields
- Different control objects and their properties

3.1 INTRODUCTION

Visual Basic .NET is Object-oriented. Everything we do in Visual Basic involves objects in some way or other and everything is based on the Object class. Controls, Forms, Modules, etc are all types of classes. Fields, Properties, Methods, and Events are members of the class. Fields and Properties represent information that an object contains.

3.2 SETTING PROPERTIES OF FORM AND CONTROL

A property is a piece of information that characterizes a control. It could be related to its location or size. It could be its color, its identification, or any visual aspect that gives it presence on the screen. The properties of an object can be changed either at design time or at run time. You can also manipulate these characteristics both at design and at run times. This means that you can set some properties at design time and some others at run time.

To manipulate the properties of a control at design time, first click the desired property from the Toolbox. Then add it to the form or to a container control. To change the properties of a control at design time, on the form, click the control to select it. Then use the Properties window.

The items in the Properties window display in a list set when installing Microsoft Visual Studio. In the beginning, you may be lost regular when looking for a particular control because the list is not arranged in a strict order of rules. You can rearrange the list. For example, you can cause the items to display in alphabetic order. To do this, in the title bar of the Properties window, click the Alphabetic button. To restore the list, you can click the categorized button.

3.2.1 Properties Categories

Each field in the Properties window has two sections: the property's name and the property's value:

The name of a property is represented on the left column. This is the official name of the property. Notice that the names of properties are in one word. You can use this same name to access the property in code.

The box on the right side of each property name represents the value of the property that you can set for an object. There are various kinds of fields you will use to set the properties. To know what particular kind a field is, you can click its name. But to set or change a property, you use the box on the right side of the property's name: the property's value, also referred to as the field's value.

Text Fields: There are fields that expect you to type a value. Most of these fields have a default value. To change the value of the property, click the name of the property, type the desired value, and press Enter. While some properties, such as the Text, would allow anything, some other fields expect a specific type of text, such as a numeric value.

Expandable Fields: Some fields have a + button. This indicates that the property has a set of sub-properties that actually belong to the same property and are set together. To expand such a field, click its + button and a – button will appear:

To collapse the field, click the – button. Some of the properties are numeric based, such as the above Size. With such a property, you can click its name and type two numeric values separated by a comma. Some other properties are created an enumerator or a class. If you expand such a field, it would display various options. Here is an example:

Boolean Fields: Some fields can have only a true or false value. To change their setting, you can either select from the combo box or double-click the property to toggle to the other value.

Action Fields: Some fields would require a value or item that needs to be set through an intermediary action. Such fields display an ellipsis button. When you click the button, a dialog box would come up and you can set the value for the field.

Boolean Fields: Some fields can have only a true or false value. After clicking the arrow, a list would display:

There are various types of selection fields. Some of them display just two items. To change their value, you can just double-click the field. Some other fields have more than two values in the field. To change them, you can click their arrow and select from the list. You can also double-click a few times until the desired value is selected.

Below are the properties of a Windows Form. Properties displayed below are categorized as seen in the properties window.

Check Your Progress 1

Fill in the blanks:

1. A property is a piece of information that _____ a control.
2. Each field in the Properties window has two sections: the _____ and _____.
3. _____ field allows you to type a value.

Appearance Properties	Description
BackColor	Gets/Sets the background color for the form
BackgroundImage	Get/Sets the background image in the form
Cursor	Gets/Sets the cursor to be displayed when the user moves the mouse over the form
Font	Gets/Sets the font for the form
ForeColor	Gets/Sets the foreground color of the form
FormBorderStyle	Gets/Sets the border style of the form
RightToLeft	Gets/Sets the value indicating if the alignment of the control's elements is reversed to support right-to-left fonts
Text	Gets/Sets the text associated with this form
Behavior Properties	Description
AllowDrop	Indicates if the form can accept data that the user drags and drops into it
ContextMenu	Gets/Sets the shortcut menu for the form
Enabled	Gets/Sets a value indicating if the form is enabled
ImeMode	Gets/Sets the state of an Input Method Editor
Data Properties	Description
DataBindings	Gets the data bindings for a control
Tag	Gets/Sets an object that contains data about a control
Design Properties	Description
Name	Gets/Sets name for the form
DrawGrid	Indicates whether or not to draw the positioning grid
GridSize	Determines the size of the positioning grid
Locked	Gets/Sets whether the form is locked
SnapToGrid	Indicates if the controls should snap to the positioning grid

Layout Properties	Description
AutoScale	Indicates if the form adjusts its size to fit the height of the font used on the form and scales its controls
AutoScroll	Indicates if the form implements autoscrolling
AutoScrollMargin	The margin around controls during auto scroll
AutoScrollMinSize	The minimum logical size for the auto scroll region
DockPadding	Determines the size of the border for docked controls
Location	Gets/Sets the co-ordinates of the upper-left corner of the form
MaximumSize	The maximum size the form can be resized to
MinimumSize	The minimum size the form can be resized to
Size	Gets/Sets size of the form in pixels
StartPosition	Gets/Sets the starting position of the form at run time
WindowState	Gets/Sets the form's window state
Misc Properties	Description
AcceptButton	Gets/Sets the button on the form that is pressed when the user uses the enter key
CancelButton	Indicates the button control that is pressed when the user presses the ESC key
KeyPreview	Determines whether keyboard controls on the form are registered with the form
Language	Indicates the current localizable language
Localizable	Determines if localizable code will be generated for this object
Window Style Properties	Description
ControlBox	Gets/Sets a value indicating if a control box is displayed
HelpButton	Determines whether a form has a help button on the caption bar
Icon	Gets/Sets the icon for the form
IsMdiContainer	Gets/Sets a value indicating if the form is a container for MDI child forms
MaximizeBox	Gets/Sets a value indicating if the maximize button is displayed in the caption bar of the form
Menu	Gets/Sets the MainMenu that is displayed in the form
MinimizeBox	Gets/Sets a value indicating if the minimize button is displayed in the caption bar of the form
Opacity	Determines how opaque or transparent the form is

ShowInTaskbar	Gets/Sets a value indicating if the form is displayed in the Windows taskbar
SizeGripStyle	Determines when the size grip will be displayed for the form
TopMost	Gets/Sets a value indicating if the form should be displayed as the topmost form of the application
TransparencyKey	A color which will appear transparent when painted on the form

Check Your Progress 2

Fill in the blanks:

1. _____ property helps to Gets/Sets the font for the form.
2. _____ property helps to Gets/Sets an object that contains data about a control.
3. _____ property helps to set the margin around controls during auto scroll

3.3 LET US SUM UP

A property is a piece of information that characterizes a control. It could be related to its location or size. It could be its color, its identification, or any visual aspect that gives it presence on the screen. The properties of an object can be changed either at design time or at run time. To manipulate the properties of a control at design time, first click the desired property from the Toolbox. To change the properties of a control at design time, on the form, click the control to select it. Then use the Properties window: Each field in the Properties window has two sections: the property's name and the property's value.

3.4 LESSON END ACTIVITIES

1. Design a data entry form for employee details database, which will be full screen and the close button will be inactive.
2. Add a toolbar in your project and customize it accordingly.

3.5 KEYWORDS

Property: A property is a piece of information that characterizes a control.

Text Fields: There are fields that expect you to type a value.

Expandable Fields: There are fields which have a + button and can be expanded.

Boolean Fields: There are fields which can have only a true or false value.

3.6 QUESTIONS FOR DISCUSSION

1. When and how properties of an object can be changed?
2. How to create a good user interface?
3. Write short notes on expandable field.

Check Your Progress: Model Answers

CYP 1

1. Characterizes
2. Property's name and the property's value.
3. Text Fields

CYP 2

1. Font
2. Tag
3. AutoScrollMargin

3.7 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*. Pearson edition, 2004

MSDN *Visual studio Library*.

Schneider, *An Introduction to Programming using Visual Basic .Net*, 5th Edition, PHI

Kant, *Visual Basic.Net— A Beginners Guide*, TMCH

UNIT II

LESSON

4

VB.NET VARIABLES

CONTENTS

- 4.0 Aims and Objectives
- 4.1 Introduction
- 4.2 VB.Net Variables
 - 4.2.1 Naming Variables
 - 4.2.2 Data Types
 - 4.2.3 The Variant Data Type
 - 4.2.4 Type Conversions
- 4.3 Data Type Constant
- 4.4 Building Project
 - 4.4.1 Creating a Project
 - 4.4.2 Writing Code
 - 4.4.3 Opening a Project
 - 4.4.4 Compiling and Executing a Project
- 4.5 Displaying Output
 - 4.5.1 Formatting Currency
 - 4.5.2 Formatting Numbers
 - 4.5.3 Formatting Percentages
 - 4.5.4 Formatting Dates and Times
 - 4.5.5 The Format() Function
 - 4.5.6 Formatting Numbers
 - 4.5.7 Formatting Dates and Times's Values
 - 4.5.8 User-defined Numeric Formats
 - 4.5.9 User-defined Date/Time Formats
- 4.6 Operators
 - 4.6.1 Arithmetic Operators
 - 4.6.2 Addition
 - 4.6.3 Subtraction
 - 4.6.4 Multiplication
 - 4.6.5 Division
 - 4.6.6 Integer Division
 - 4.6.7 Modulo Division
 - 4.6.8 Exponentiation
 - 4.6.9 Operator Precedence

Contd...

4.6.10	Arithmetic Assignment Operators
4.6.11	String Operators
4.6.12	String Concatenation
4.6.13	String Assignment Operator
4.6.14	Matching Strings
4.6.15	Relational Operators
4.6.16	Logical Operators
4.7	Let us Sum up
4.8	Lesson End Activities
4.9	Keywords
4.10	Questions for Discussion
4.11	Suggested Readings

4.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- How variables are used in VB.NET programming
- Different types of operators used in VB.NET programming

4.1 INTRODUCTION

Computer programs are processors of data. Therefore, they need of way of representing various types of data - numbers, alphabetic characters, special characters, and the like - inside the computer. Also, programs need a way to hold onto - to store - these data inside computer memory during processing. This lesson discusses these two most basic aspects of computer programming, how to represent and store information involved in computer processing.

4.2 VB.NET VARIABLES

Variables are named storage areas inside computer memory where a program places data during processing. A program sets aside these storage areas by declaring variables, that is, by assigning them a name and indicating what types of data will occupy these areas during processing.

Within a Visual Basic program, variables are declared using the Dim statement whose general format is shown below.

Dim variable as type – where variable is a programmer-supplied name for the storage area and type is one of the data types permissible in Visual Basic.

4.2.1 Naming Variables

Programmer-supplied variable names are also referred to as identifiers, and they must conform to Visual Basic naming conventions. A variable name must

- Begin with an alphabetic or numeric character, or the underscore (_) character.
- Be composed only of alphabetic, numeric, and underscore characters.

- Not contain embedded blank spaces.
- Not be the same as a Visual Basic keyword, words that comprise the Visual Basic language itself.

Variable names are not case sensitive, so the name expressed in upper-case characters refers to the same storage area as a name expressed in lower-case characters. However, for programming consistency, and to avoid unnecessary errors, it is best to express a variable as it is originally declared. Thus, valid variable names include X, My_Variable, MyVariable, My_Data_Variable, and other names using simple and meaningful identifiers.

4.2.2 Data Types

When declaring a variable an indication can be given about what type of data will be stored. There are many data types permissible under Visual Basic. The following are common for declaring the usual types of data involved in computer processing.

- Short. A non-decimal (whole) number with a value in the range of -32,768 to +32,767.
- Integer. A non-decimal (whole) number with a value in the range of -2,147,483,648 to +2,147,483,647.
- Long. A non-decimal (whole) number with a value in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.
- Single. A floating-point (decimal) positive or negative number with a value in the range of up to 3.402823E38 (scientific notation).
- Double. A floating-point (decimal) positive or negative number with a value in the range of up to 1.79769313486232E308.
- Decimal. A decimal number with a value in the range of up to +79,228,162,514,264,337,593,950,335. without decimal precision and up to +7.9228162514264337593543950335 with 28 places of decimal precision.
- String. Any number of alphabetic, numerical, and special characters.
- Date. Dates in the range between January 1, 0001 to December 31, 9999 and times in the range between 0:00:00 to 23:59:59.
- Boolean. The value True or False.

Variables, then, are declared by assigning a name and a data type. The following are examples of valid variable declarations that can be used to store data of the identified types.

```
Dim My_Counter As Integer
```

```
Dim My_Salary As Decimal
```

```
Dim My_Great-Big-Number As Double
```

```
Dim My_Name As String
```

```
Dim My_Birthday As Date
```

```
Dim My_Current_Time As Date
```

```
Dim My_Decision As Boolean
```

The Data types available in VB .NET, their size, type, description are summarized in the table below.

Data Type	Size in Bytes	Description	Type
Byte	1	8-bit unsigned integer	System.Byte
Char	2	16-bit Unicode characters	System.Char
Integer	4	32-bit signed integer	System.Int32
Double	8	64-bit floating point variable	System.Double
Long	8	64-bit signed integer	System.Int64
Short	2	16-bit signed integer	System.Int16
Single	4	32-bit floating point variable	System.Single
String	Varies	Non-Numeric Type	System.String
Date	8	Date Type	System.Date
Boolean	2	Non-Numeric Type	System.Boolean
Object	4	Non-Numeric Type	System.Object
Decimal	16	128-bit floating point variable	System.Decimal

Assigning Values to Variables

Variable declarations simply set aside memory storage areas as temporary containers of data -temporary because they are no longer accessible once the program ends. During processing, of course, these variables are occupied by data values.

Data for processing can come from many different sources. They can come from outside the program - from external files and databases; they can come from inside the program by explicitly setting a value for a variable or by storing processing results of manipulating other variables. In any case, variables take on values by assigning values to them.

Variables are assigned values with the Visual Basic assignment statement whose general format is shown below.

variable = value

The data value on the right of the equal sign is assigned to (is placed in) the variable identified on the left of the equal sign. Data values normally originate in one of three ways.

Assigning Constants to Variables

A literal data value - a particular number, string, date, or Boolean value - can be assigned to the variable. A number is assigned by specifying its value to the right of the equal sign:

My_Integer = 10

My_FloatingPoint = 12.34567

My_Decimal = 123.45

A string value is assigned to a variable by enclosing its value inside quotation marks:

My_String = "This is a string."

A date or time value is assigned to a variable by enclosing a particular date or time inside "#" symbols:

This_Date = #01/01/05#

This_Time = #12:00:00#

```
My_Birthday = #July 15, 1942#
```

Notice that you can store any common representation of dates, whether in the format #mm/dd/yy#, #mm/dd/yyyy#, #mm-dd-yy#, or #Month Day, Year#. Visual Basic converts all formats to its standard internal representation.

Incidentally, if there are two or more variables of the same type to be declared, you can declare them all on the same line by separating them with commas:

```
Dim This_Date, This_Time, My_BirthDay As Date
```

A Boolean value is assigned to a variable by specifying True or False:

```
Error_Flag = True
```

Variables that have been assigned literal values that do not get changed during program execution are called constants. When declaring constants it is common practice to combine the declaration and assignment into a single statement:

```
Dim My_Decimal As Decimal = 123.45
```

```
Dim My_String As String = "This is a string."
```

```
Dim My_Date As Date = #01/01/05#
```

Assigning Variables to Variables

Variables can be assigned to variables. That is, the data value contained in one storage area can be assigned to a second storage area.

```
Variable_B = Variable_A
```

When a variable is assigned to a variable, the value being assigned is copied to the target variable. The end result is that both variables contain the same value.

Assigning Expressions to Variables

The most common scenario is that the results of processing are assigned to a variable. Some type of arithmetic, string, or date manipulation operation takes place with the "answer" stored in the variable. Although these types of operations are discussed later, you should be able to understand the principles involved in the following assignment statements.

```
The_Answer = 2 + 2
```

```
The_Result = 3 * (Var_A + Var_B)
```

```
Full_Name = First_Name & " " & Middle_Name & " " & Last_Name
```

Variable The_Answer stores the value 4; variable The_Result stores 3 multiplied times the contents of Var_A plus the contents of Var_B; variable Full_Name stores the results of stringing together the contents of variables First_Name, Middle_Name, and Last_Name, placing literal blank spaces (" ") between them.

4.2.3 The Variant Data Type

If variable types are not explicitly declared - using As Integer, As Decimal, As String, etc. - then values assigned to the variable are stored as a Variant data type. Any type of data can be stored in the variable and any type of operation can be performed on it. It is a general-purpose data type that provides flexibility in storing different types of data in a single variable. It can be assigned a numeric value that is immediately replaced by a string value.

Although it does offer some programming flexibility, there are minor inefficiencies associated with converting different data types for representation in the same storage area. For this reason it is best always to declare specific data types for specific

variables. This practice of using strong typing of variables leads to more efficient programming and reduction in errors from inadvertently storing the wrong kinds of data in the wrong variables.

4.2.4 Type Conversions

Values of one data type can be converted to another data type. This operation is called casting from one type to another. Casting requires calling a conversion function to switch a data type. The resulting cast must be assigned to a variable of the cast type.

variable = Cname(value)

The Cname is one of the Visual Basic casting functions. For example, the following code converts an Integer to a Boolean type.

```
Dim A_Number As Integer
```

```
Dim A_Boolean As Boolean
```

```
A_Boolean = CBool(A_Number)
```

The following Cname functions are available in VB.NET.

- CBool(). Converts a string or numeric expression to True or False. A nonzero value is converted to True and a zero is converted to False.
- CDate(). Converts any valid date or time value to a Date type.
- CDbl(). Converts any number in the range of a Double value to a Double.
- CDec(). Converts any number in the range of a Decimal value to a Decimal.
- CInt(). Converts any number in the range of an Integer to an Integer.
- CLng(). Converts any number in the range of a Long value to a Long.
- CShort(). Converts any number in the range of a Short value to a Short.
- CSng(). Converts any number in the range of a Single value to a Single.
- CStr(). Converts a value to a String. For a value that can be interpreted as a Date type, the conversion is to a string representation of the date. For a value that can be interpreted as a Boolean type, the conversion is to "True" or "False". For a numeric value, a string representation of the number is returned.

Casting can also be done with the following general-purpose conversion function,

variable = CType(value, type)

where value is any literal value, variable, or expression that results in a value, and type is one of the data types.

```
Dim A_Number As Decimal = 123.45
```

```
Dim A_String As String
```

```
A_String = CType(A_Number, String)
```

In this example, variable A_String ends up storing the value "123.45".

Check Your Progress 1

Fill in the blanks:

1. _____ are named storage areas inside computer memory where a program places data during processing.
2. In Visual Basic program, variables are declared using the _____ statement.

4.3 DATA TYPE CONSTANT

Constants are very similar to variables. The main difference is that the value contained in memory cannot be changed once the constant is declared. When you declare a constant its value is also specified and this value cannot be changed during program execution.

Constants are used in situations where we need to keep the value in some memory location constant. If you use hard-coded values, and the value is changed then it has to be changed in all the locations in the code where it has been used. Instead if we are using constants, all we will need to do is to change the value of the constant. This would propagate the changes to our entire application.

Constants are declared as follows

Const x as Integer

4.4 BUILDING PROJECT

In the early days of Microsoft DOS, there was a language called Basic. It provided a simplified and easy way to create small applications using words very close to the English language. Since the language was easy, it became popular with the help of Microsoft operating systems gaining ground. To continue this tendency and provide more support for Basic, Microsoft used that language as the platform to create graphical user interface (GUI) applications. Once again, this move was welcomed and the language became the widely accepted Visual Basic.

The Microsoft Visual Basic programming environment became very popular for its ease of use and it was a candidate for serious productive applications. Because Visual Basic was not tied to the operating systems low-level operations, its programmers had to use library calls to access functions of the Win32 Application Programming Interface (API), the library that "defines" Microsoft Windows. This also accentuated the difference with other programming environments like Microsoft Visual C++ or other libraries like Microsoft Foundation Class (MFC). In fact, although Microsoft shipped Visual Studio 6 that combined various programming environments with different languages (Visual Basic, C++, ASP, Win32, etc), the only real thing they had in common was that they shipped in the same box (and the same DVD).

To unify the various languages or programming environments that Microsoft had developed for many years, the company created a new library aside from Win32. This was the birth of the .NET Framework. This library is used by, or shared among, different programming languages or environments so that programmers can benefit from a better collaboration. Now it is possible for people who "speak", that is, people who program in, different languages to work on the same project with less regard for compatibility issues. This is because (most of) the functions, classes, and resources, are used in conceptually the same way in the different languages.

Microsoft Visual Basic .NET is Microsoft's implementation of the .NET Framework for Visual Basic programmers. Although Visual Basic .NET is a "child" of Visual Basic 6.0, there are many differences that can be interpreted as a complete shift, with a lot of improvements. Because of these differences, many already Visual Basic 6.0 programmers resisted the move to this new environment (there were also many other considerations) but those programmers are catching up.

4.4.1 Creating a Project

A computer application, also called an application, also called a computer program, also called a program, is a series of instructions that present messages and choices to a

person, also called a user. This interaction allows the user to partially control the computer, including "telling" the computer what to do, when, and how. You, as the programmer, create instructions that allow a user to interact with the computer. The instructions are created in plain English using a language called (Basic or in this case) Visual Basic.

To create a project, you can display the New Project dialog box, select Visual Basic Projects, select the type of project, give it a name, specify its directory, and click OK.

4.4.2 Writing Code

The instructions used in a Visual Basic project are created in a normal, text-based, computer file. These instructions are also referred to as code. Besides being a regular text file, this file has the extension .vb (some other flavors of Visual Basic use other extensions, such as .vbs). A file that carries normal Visual Basic code is also called a module. There are two primary ways you can get a module depending on how you create your application:

- You can use Notepad, type your code, save the file with a .vb extension, and then add it to your application
- Probably the fastest way to get a module consists of using Microsoft Visual Studio .NET. To do this, on the main menu, you can click Project -> Add Module

If you create your application using Microsoft Visual Studio .NET, there are two main ways you can get a module. To create a module independent of any other object, on the main menu, you can click Project -> Add New Item..., select Module, give a name to the file, and click Open. If you create a project using a wizard and get a form, a module is automatically associated with it and it is used to carry the instructions of the form. On this site, we will use Visual Studio .NET to create our applications.

To indicate that a module contains code, the section that contains code starts with the Module keyword followed by a name for the module, and ends with the expression End module.

4.4.3 Opening a Project

As opposed to creating a new project, you can open a project that either you or someone else created. To open an existing project, on the main menu, you can click File -> Open -> Project... You can also click File -> Open Solution on the main menu. Alternatively, you can display the Start Page and click Open. In all cases, the action would display the Open Project dialog box. This allows you to select a Visual Project and open it.

When opening a project, if the application was created using Visual Basic 6.0, a wizard would guide you to upgrade the project.

4.4.4 Compiling and Executing a Project

As mentioned already, the instructions created for a Visual Basic project are written in plain English in a language easily recognizable to the human eye. To compile and execute a Visual Basic .NET project in one step, on the main menu, you can click Debug -> Start Without Debugging. Although there are other techniques or details in compiling (or debugging) and executing a project, for now, this is the only technique we will use until further notice.

4.5 DISPLAYING OUTPUT

Information stored in variables as various data types may not be in a format for visual presentation on a Web page. Visual Basic provides several built-in functions for presentation of this data.

4.5.1 Formatting Currency

The `FormatCurrency()` function formats numeric data for presentation as dollars and cents. Its general format is shown below.

`FormatCurrency(value [, trailing digits] [, leading digit] [, parentheses] [, group digits])` `value` is any expression that produces a number; `trailing digits` is an integer giving the number of digits following the decimal point; the default is rounding to 2 digits; `leading digit` is `True` or `False` to indicate whether a leading 0 is to appear before the decimal point for fractional values; `parentheses` is `True` or `False` to indicate whether negative numbers should be displayed inside parentheses; `group digits` is `True` or `False` to indicate whether numbers should be grouped between commas.

Format	Output
<code>FormatCurrency(12345.6789)</code>	\$12,345.68
<code>FormatCurrency(12345.6789, 4)</code>	\$12,345.6789
<code>FormatCurrency(12345.6789,,,False)</code>	\$12345.68
<code>FormatCurrency(-12345.6789)</code>	(\$12,345.68)
<code>FormatCurrency(-12345.6789,,,False)</code>	-\$12,345.68
<code>FormatCurrency(.6789)</code>	\$0.68
<code>FormatCurrency(.6789,,False)</code>	\$.68
<code>FormatCurrency(-.6789,,False,False)</code>	-\$0.68

4.5.2 Formatting Numbers

The `FormatNumber()` function returns a value formatted as a number. Its general format is shown below.

`FormatNumber(value [, trailing digits] [, leading digit] [, parentheses] [, group digits])` `value` is any expression that produces a number;

`trailing digits` is an integer giving the number of digits following the decimal point; the default is rounding to 2 digits;

`leading digit` is `True` or `False` to indicate whether a leading 0 is to appear before the decimal point for fractional values;

`parentheses` is `True` or `False` to indicate whether negative numbers should be displayed inside parentheses;

`group digits` is `True` or `False` to indicate whether numbers should be grouped between commas.

Format	Output
<code>FormatNumber(12345.6789)</code>	12,345.68
<code>FormatNumber(12345.6789,5)</code>	12,345.67890
<code>FormatNumber(12345.6789,,,False)</code>	12345.68

Contd...

FormatNumber(-12345.6789)	-12,345.68
FormatNumber(-12345.6789,,,True)	(12,345.68)
FormatNumber(.6789)	0.68
FormatNumber(.6789,,False)	.68
FormatNumber(-.6789,4)	-0.6789

4.5.3 Formatting Percentages

The FormatPercent() function returns a value formatted as a percentage, that is, multiplied by 100 with a trailing % character. Its general format is shown below.

FormatPercent(value [, trailing digits] [, leading digit] [, parentheses] [, group digits])
value is any expression that produces a number;

trailing digits is an integer giving the number of digits following the decimal point;
the default is rounding to 2 digits;

leading digit is True or False to indication whether a leading 0 is to appear before the decimal point for fractional values;

parentheses is True or False to indicate whether negative numbers should be displayed inside parentheses;

group digits is True or False to indicate whether numbers should be grouped between commas.

Format	Output
FormatPercent(.6789)	67.89%
FormatPercent(.6789,4)	67.8900%
FormatPercent(-.6789)	-67.89%
FormatPercent(-.6789,,,True)	(67.89%)

4.5.4 Formatting Dates and Times

The FormatDateTime() function returns a string expression representing a date/time value. Its general format is

FormatDateTime(value [, DateFormat.format])

Where value is a date or time value and *format* is one of the following values: GeneralDate, LongDate, ShortDate, LongTime, or ShortTime.

Format	Output
FormatDateTime(Now)	3/29/2008 5:12:34 PM
FormatDateTime(Today)	3/29/2008
FormatDateTime(TimeOfDay)	5:12:34 PM
FormatDateTime(Now,DateFormat.LongDate)	Saturday, March 29, 2008
FormatDateTime(Today,DateFormat.LongDate)	Saturday, March 29, 2008
FormatDateTime(Now,DateFormat.ShortDate)	3/29/2008
FormatDateTime(Today,DateFormat.ShortDate)	3/29/2008
FormatDateTime(Now,DateFormat.LongTime)	5:12:34 PM
FormatDateTime(TimeOfDay,DateFormat.LongTime)	5:12:34 PM
FormatDateTime(Now,DateFormat.ShortTime)	17:12
FormatDateTime(TimeOfDay,DateFormat.ShortTime)	17:12

4.5.5 The Format() Function

The Format() function is a general-purpose formatting function that returns a string value formatted according to a format string. The format strings duplicate numeric and date/time formats produced by the specialized formats described above. The general format for applying the Format() function is shown below.

Format(value, "format string")

4.5.6 Formatting Numbers

A format string for numeric values can use one of the predefined string values shown in the following table.

String	Description
General Number G g	Displays number with no thousand separator.
Currency C c	Displays number with thousand separator, if appropriate; display two digits to the right of the decimal separator.
Fixed F f	Displays at least one digit to the left and two digits to the right of the decimal separator.
Standard N n	Displays number with thousand separator, at least one digit to the left and two digits to the right of the decimal separator.
Percent	Displays number multiplied by 100 with a percent sign (%) appended immediately to the right; always displays two digits to the right of the decimal separator.
P p	Displays number with thousandths separator multiplied by 100 with a percent sign (%) appended to the right and separated by a single space; always displays two digits to the right of the decimal separator.

Examples of applying a format string to numeric values are shown in the following table.

Format	Output
Format(12345.6789,"g")	12345.6789
Format(12345.6789,"c")	\$12,345.68
Format(12345.6789,"f")	12345.68
Format(12345.6789,"n")	12,345.68
Format(-12345.6789,"g")	-12345.6789
Format(-12345.6789,"c")	(\$12,345.68)
Format(-12345.6789,"f")	-12345.68
Format(-12345.6789,"n")	-12,345.68
Format(.6789,"Percent")	67.89%
Format(.6789,"p")	67.89 %
Format(-.6789,"Percent")	-67.89%
Format(-.6789,"p")	-67.89 %

4.5.7 Formatting Dates and Time's Values

A format string for date/time values can use one of the predefined string values shown in the following table.

String	Description
Long Date D	Displays a date in long date format.
Short Date d	Displays a date in short date format.

Contd...

Long Time T	Displays a date in long date format.
Short Time t	Displays a date in short date format.
F	Displays the long date and long time.
F	Displays the long date and short time.
G	Displays the short date and short time.
M m	Displays the month and the day of a date.
Y y	Formats the date as the year and month.

Examples of applying a format string to date/time values are shown in the following table.

Format	Output
Format(Now,"D")	Saturday, March 29, 2008
Format(Now,"d")	3/29/2008
Format(Now,"T")	5:12:34 PM
Format(Now,"t")	5:12 PM
Format(Now,"F")	Saturday, March 29, 2008 5:12:34 PM
Format(Now,"f")	Saturday, March 29, 2008 5:12 PM
Format(Now,"g")	3/29/2008 5:12 PM
Format(Now,"m")	March 29
Format(Now,"y")	March, 2008

4.5.8 User-defined Numeric Formats

Formats can be defined for displaying numeric values by composing a string to describe the format. This user-defined string is applied through the Format() function. The characters shown in the following table are used to compose the format string.

Character	Description
0	Digit placeholder. Displays a digit or a zero. If the value has a digit in the position, then it displays; otherwise, a zero is displayed.
#	Digit placeholder. Displays a digit or a space. If the value has a digit in the position, then it displays; otherwise, a space is displayed.
.	Decimal placeholder; determines how many digits are displayed to the left and right of the decimal separator.
,	Thousand separator; separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Only a single "," is required in the format, between the first set of digit placeholders.
%	Percent placeholder. Multiplies the expression by 100. The percent character (%) is inserted in the position where it appears in the format string.
- + \$ ()	Literal characters; displayed exactly as typed in the format string.

Examples of applying a user-defined format string to numeric values are shown in the following table.

Format	Output
Format(012345.6789,"0.00")	12345.68
Format(012345.6789,"0,0.000")	12,345.679
Format(012345.6789,"00000,0.000000")	012,345.678900
Format(012345.6789,"#.###")	12345.68

Contd...

Format(012345.6789,"#,###")	12,345.68
Format(012345.6789,"\$ #,###")	\$ 12,345.68
Format(-012345.6789,"#,#####")	-12,345.6789
Format(-012345.6789,"\$ #,###")	-\$12,345.68
Format(.6789,"#,###")	.68
Format(.6789,"0,0.000")	00.679
Format(-.6789," 0.0000")	- 0.6789
Format(.6789,"0.00%")	67.89%

4.5.9 User-defined Date/Time Formats

Formats can be defined for displaying date and time values by composing a string to describe the format. This user-defined string is applied through the Format() function. The characters shown in the following table are used to compose the date/time format string.

Character	Description
:	Time separator.
/ -	Date separators.
%	Precedes a single-character format string.
d	Displays the day as a number without a leading zero.
dd	Displays the day as a number with a leading zero.
ddd	Displays the day name as an abbreviation.
dddd	Displays the day as a full name.
M	Displays the month as a number without a leading zero.
MM	Displays the month as a number with a leading zero.
MMM	Displays the month name as an abbreviation.
MMMM	Displays the month as a full name.
yy	Displays the year in two-digit format.
yyyy	Displays the year in four-digit format.
h	Displays the hour as a number without leading zeros using the 12-hour clock.
hh	Displays the hour as a number with leading zeros using the 12-hour clock.
H	Displays the hour as a number without leading zeros using the 24-hour clock.
HH	Displays the hour as a number with leading zeros using the 24-hour clock.
m	Displays the minute as a number without leading zeros.
mm	Displays the minute as a number with leading zeros.
s	Displays the seconds as a number without leading zeros.
ss	Displays the seconds as a number with leading zeros.
f...	Displays fractions of seconds using up to 7 characters to display fractional digits.
tt	Uses the 12-hour clock and displays an uppercase AM with any hour before noon; displays an uppercase PM with any hour between noon and 11:59 P.M.
	Additional characters and punctuation marks can be used within the format string. Characters that match any of the formatting characters must be preceded by "\".

Examples of applying a user-defined format string to date/time values are shown in the following table.

Format	Output
Format(Now,"M/d/yy")	3/29/08
Format(Now,"M-d-yyyy")	3-29-2008
Format(Now,"d-MMMM-yy")	29-March-08
Format(Now,"d MMMM, yyyy")	29 March, 2008
Format(Now,"MMMM d, yyyy")	March 29, 2008
Format(Now,"MMMM, yyyy")	March, 2008
Format(Now,"%d")	29
Format(Now,"h:m tt")	5:12 PM
Format(Now,"h:m:ss tt")	5:12:34 PM
Format(Now,"H:m")	17:12
Format(Now,"M/d/yy - h:mtt")	3/29/08 - 5:12PM
Format(Now,"H:m:ss.ffffff")	17:12:34.4307926
Format(Now,"To\da\y i\s MMMM d, yyyy.")	Today is March 29, 2008.

Check Your Progress 2

Fill in the blanks:

1. The _____ of the constant can not be changed once it is declared.
2. Full form of API is _____.

4.6 OPERATORS

Variables and literal data values are involved in processing operations in order to create new information from the original values. The two most common ways to manipulate constants and variables are through arithmetic and string operations. Visual Basic supplies arithmetic and string operators to construct these processing statements, or expressions.

Computer programs make extensive use of comparative operations to determine which of two or more processing routines to execute depending on a comparison of data values. If, for instance, one variable is larger in value than a second variable, then a particular group of processing statements is executed. On the other hand, if the first variable is smaller than the second variable, then a different group of processing statements is executed. This ability of programs to take alternative courses of action depending on comparative data values is made possible with relational operators and logical operators.

4.6.1 Arithmetic Operators

Much of information processing involves applying arithmetic to numeric data types. Values are added, subtracted, multiplied, and divided to generate results, either final results or intermediate values that are involved in further arithmetic operations. For these purposes Visual Basic supplies a set of seven arithmetic operators shown in the following table.

Operator	Use
+	Addition. Adds the values appearing on the two sides of the operator.
-	Subtraction. Subtracts the value on the right of the operator from the value on the left of the operator.
*	Multiplication. Multiplies the values appearing on the two sides of the operator.
/	Division. Divides the value on the left of the operator (dividend) by the value on the right of the operator (divisor).
\	Integer Division. Divides the value on the left of the operator by the value on the right of the operator, giving an Integer result.
Mod	Modulo Division. Returns the remainder of the division of the value on the left of the operator by the value on the right of the operator.
^	Exponentiation. Raises the value on the left of the operator to the power on the right of the operator.

Multiple operators can be combined with multiple constants and variables to create complex expressions.

4.6.2 Addition

Two numeric values are added by linking them with the addition operator. The result of the operation can be assigned to a variable of a compatible numeric type.

```
Dim Number_1 As Decimal = 123.45
```

```
Dim Number_2 As Decimal = 234.56
```

```
Dim Answer As Decimal
```

```
Answer = Number_1 + Number_2
```

4.6.3 Subtraction

Two numeric values are subtracted by linking them with the subtraction operator. The result of the operation can be assigned to a variable of a compatible numeric type.

```
Dim Number_1 As Decimal = 234.56
```

```
Dim Number_2 As Decimal = 123.45
```

```
Dim Answer As Decimal
```

```
Answer = Number_1 - Number_2
```

4.6.4 Multiplication

Two numeric values are multiplied by linking them with the multiplication operator. The result of the operation can be assigned to a variable of a compatible numeric type.

```
Dim Number_1 As Decimal = 234.56
```

```
Dim Number_2 As Decimal = 123.45
```

```
Dim Answer As Decimal
```

```
Answer = Number_1 * Number_2
```

4.6.5 Division

Two numeric values are divided by linking them with the division operator. The value on the left is the dividend; the value on the right is the divisor. The result of the operation can be assigned to a variable of a compatible numeric type.

Dim Number_1 As Decimal = 234.56

Dim Number_2 As Decimal = 123.45

Dim Answer As Decimal

Answer = Number_1 / Number_2

4.6.6 Integer Division

Two integer values are divided by linking them with the Integer Division operator. The value on the left is the dividend; the value on the right is the divisor. The result is the largest integer that is less than the absolute value of the quotient of the two operands. The result of the operation can be assigned to a variable of a compatible numeric type.

Dim Number_1 As Integer = 234

Dim Number_2 As Integer = 123

Dim Answer As Integer

Answer = Number_1 \ Number_2

4.6.7 Modulo Division

Two numeric values are divided by linking them with the Modulo Division operator. The value on the left is the dividend; the value on the right is the divisor. The result is the remainder of the division of the two values. The result of the operation can be assigned to a variable of a compatible numeric type.

Dim Number_1 As Integer = 10

Dim Number_2 As Integer = 3

Dim Answer As Integer

Answer = Number_1 Mod Number_2

4.6.8 Exponentiation

A numeric value is raised to a power by linking the value with a power value through the Exponentiation operator. The result of the operation can be assigned to a variable of a compatible numeric type.

Dim MyNumber As Integer = 3

Dim Squared As Integer

Dim Cubed As Integer

Squared = MyNumber ^ 2

Cubed = MyNumber ^ 3

4.6.9 Operator Precedence

When an arithmetic expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. Multiplication and division take precedence, and are performed first, over addition and subtraction. For example, in the expression

$x + y / z$

variable y is divided by variable z with the result then added to variable x. The expression is evaluated as $x + (y / z)$ because the / operator has higher precedence than the + operator. If you are not careful in combining operators you can end up with unexpected results. In the previous example, you would generate the wrong answer if your intent was to first add $x + y$ and then divided by z.

Complex arithmetic expressions created with these operators can be, and should be, enclosed within parentheses to explicitly control the order in which the operations are carried out. Explicitly code $x + (y / z)$ or $(x + y) / z$ depending on your intent.

Dim Answer1, Answer2 As Single

Answer1 = 1 + (2 / 3)

Answer2 = (1 + 2) / 3)

4.6.10 Arithmetic Assignment Operators

Normally, the results of arithmetic operations are assigned to a variable using the standard assignment operator "=". There are, in addition, special assignment operators that perform an arithmetic operation and a variable assignment at the same time. These operators are listed in the following table.

	Use
+=	Adds the value appearing on the right of the operator to the variable on the left of the operator. The statement $X += Y$ is equivalent to $X = X + Y$.
-=	Subtracts the value appearing on the right of the operator from the variable on the left of the operator. The statement $X -= Y$ is equivalent to $X = X - Y$.
*=	Multiplies the value appearing on the right of the operator times the variable on the left of the operator and <i>replaces</i> the variable with the new value. The statement $X *= Y$ is equivalent to $X = X * Y$.
/=	Divides the variable on the left of the operator by the value on the right of the operator and <i>replaces</i> the variable with the new value. The statement $X /= Y$ is equivalent to $X = X / Y$.
\=	Divides the Integer variable on the left of the operator by the Integer value on the right of the operator and <i>replaces</i> the variable with the new value. The statement $X \setminus= Y$ is equivalent to $X = X \setminus Y$.
^=	Raises the variable on the left of the operator to the power on the right of the operator and <i>replaces</i> the variable with the new value. The statement $X ^= Y$ is equivalent to $X = X ^ Y$.

A quick example of the += operator will give you an idea of how these operators work. This operator is often used to accumulate totals. Assume you have declared a variable named Total that is used to sum up a set arithmetic computations. In the following code, variable A is added to Total, variable B is added to Total, and variable C is added to Total. Following this program, variable Total contains the value 30.

Dim A As Integer = 5

Dim B As Integer = 10

Dim C As Integer = 15

Dim Total As Integer = 0

Total += A

Total += B

Total += C

Other popular uses for the += and -= operators are to increment a counter (Counter += 1) or to decrement a counter (Counter -= 1). In the first case, 1 is added to the Counter each time the statement is executed; in the second case, 1 is subtracted from the Counter.

4.6.11 String Operators

Visual Basic supplies the concatenation operator (&) to put together individual data strings to create a longer string of those combinations. The Like operator looks for matching characters in a string.

Operator	Use
& or +	Concatenation. Appends the string value appearing on the right of the operator to the string value on the left of the operator.
Like	Determines whether a string matches a given pattern.

4.6.12 String Concatenation

Concatenation of strings can take place with string variables and/or string constants. In the following example, three String variables are declared. A string expression concatenates the first two variables along with a literal blank space to create a complete sentence that is assigned to the third variable.

```
Dim String1, String2, String3 As String
```

```
String1 = "This is"
```

```
String2 = "a string."
```

```
String3 = String1 & " " & String2
```

Note that the plus sign (+) also can be used as a concatenation operator. This symbol is used for concatenation, rather than addition, because of the context in which it is applied. If two string are linked with the + symbol, then concatenation takes place because two strings cannot be added. If the + symbol links a string and a number, then concatenation also takes place. Since a string cannot be added, the number is automatically converted into a string and the two values are concatenated.

4.6.13 String Assignment Operator

Similar to arithmetic assignment operators, there are special string assignment operators that perform a concatenation operation and a variable assignment at the same time. These operators are listed in the following table.

Operator	Use
&= or +=	Concatenates the value appearing on the right of the operator to the variable on the left of the operator. The statement X &= Y is equivalent to X = X & Y.

Assume the following code:

```
Dim A As String = "This "
```

```
Dim B As String = "is "
```

```
Dim C As String = "a string."
```

```
Dim Sentence As String
```

```
Sentence &= A
```

```
Sentence &= B
```

```
Sentence &= C
```


At the end of this program, variable `Sentence` contains the full string "This is a string." Each statement appends a string value to the continually lengthening `Sentence` variable.

4.6.14 Matching Strings

The `Like` operator determines whether a string matches a given pattern. The operand on the left is the string being matched, the operand on the right is the pattern to match against. The result of the expression `string value Like string value` is the Boolean value `True` or `False` depending on the success of the match.

The default behavior of the `Like` operator is to evaluate the expression as an exact match. Therefore, the statements,

```
Dim String_A As String = "This is a string."
```

```
Dim String_B As String = "This is a string."
```

```
Dim Matched As Boolean
```

```
Matched = String_A Like String_B
```

result in the Boolean value `True` being assigned to variable `Matched`. The match is on a character-by-character, left-to-right basis; plus, the match is case sensitive. The string "THIS IS A STRING." does not match string "This is a string."

Wildcard characters can be used in the pattern for partial matches. These characters, along with special characters lists, are described in the following table.

Pattern	Meaning
?	Matches any single character at the position. For example, the expression "ABCDE" Like "AB?DE" returns <code>True</code> because it matches <i>any</i> character in the third position of the string as long as the remaining characters match exactly.
*	Matches zero or more characters at the position. For example, the expression "ABCDE" Like "*CDE" returns <code>True</code> because it matches <i>any</i> characters at the beginning of the string followed by "CDE" at the end of the string. Likewise, "ABCDE" Like "AB*" returns <code>True</code> since the two strings begin with "AB" irrespective of any following characters. The expression "ABCDE" Like "*D*" returns <code>True</code> because the string contains "D" preceded and followed by any number of characters.
#	Matches any single <i>digit</i> (0 - 9) at the position. For example, the expressions "12345" Like "12#45" and "12845" Like "12#45" return <code>True</code> because they match <i>any</i> decimal digit in the third position of the string as long as the remaining characters match exactly.
[char list]	Matches any <i>single</i> character in [char list]. For example, the expression "C" Like "[ABCDE]" returns <code>True</code> because the character "C" is found in the list of characters "ABCDE".
[!char list]	Matches any <i>single</i> character <i>not</i> in [char list]. For example, the expression "M" Like "[!ABCDE]" returns <code>True</code> because the character "M" is not found in the list of characters "ABCDE".

When matching against a [char list] the list can be specified as an ascending range of characters. To match against the upper-case alphabet, for instance, the character list is "[A-Z]"; to match against the decimal digits, the character list is "[0-9]".

The [char list] can be matched only against a single character. Thus, you cannot match a word to see if all its characters are included in the character list. You would need to compare each character individually with the character list by using a program loop.

4.6.15 Relational Operators

Visual Basic supplies a set of six relational operators to compare the magnitudes of two data values. Comparisons can be made with numeric or string values, with a Boolean value returned from the comparison. The general format for the comparison statement is

value operator value

The following table summarizes these relational operators.

Operator	Use
=	Tests whether two values are equal.
<>	Tests whether two values are <i>not</i> equal.
<	Tests whether the first value is less than the second value.
>	Tests whether the first value is greater than the second value.
<=	Tests whether the first value is less than or equal to the second value.
>=	Tests whether the first value is greater than or equal to the second value

In the following example, the Boolean value False is assigned to variable Result1 and the Boolean value True is assigned to Result2. Value1 is not equal to Value2; rather, Value1 is less than Value2.

```
Dim Value1 As Decimal = 10
```

```
Dim Value2 As Decimal = 20
```

```
Dim Result1 As Boolean
```

```
Dim Result2 As Boolean
```

```
Result1 = Value1 = Value2
```

```
Result2 = Value1 < Value2
```

Comparisons should be made between similar data types. If this is not the case, say in comparing a number with a string, Visual Basic attempts to make sense of the test and perform any type conversions necessary to carry out the comparison.

4.6.16 Logical Operators

Relational tests can be combined to test multiple conditions at the same time. The logical operators AND, OR, NOT, and Xor are used as conjunctions between multiple relational tests.

In the following example, all of the complex relational tests are True. Notice in the third relational test that parentheses are used, just as in composing arithmetic expressions, to control the order in which tests are made.

```
Dim Value1 As Decimal = 10
```

```
Dim Value2 As Decimal = 20
```

```
Dim Value3 As Decimal = -5
```

```
Dim Result1 As Boolean
```

```
Dim Result2 As Boolean
```

```
Dim Result3 As Boolean
```

Result1 = Value1 < Value2 AND Value1 > Value3

Result2 = Value1 > Value2 OR Value1 > Value3

Result3 = Value1 < Value2 AND (Value1 > Value3 OR Value2 < Value3)

The Xor (exclusive OR) operation tests whether one of two comparisons, but not the other, is True. Consider the following expressions based on the values shown above (parentheses are used to make the relational tests visually obvious).

Result4 = (Value1 < Value2) OR (Value1 > Value3)

Result5 = (Value1 < Value2) Xor (Value1 > Value3)

Both relational tests are True (Value1 < Value2 and Value1 > Value3). In the first statement, Result4 is True since either of the relational tests is True; it doesn't matter that both are True. In the second statement, Result5 is False since one or the other test, exclusively, must be True, but not both.

Check Your Progress 3

Fill in the blanks:

1. _____ function formats numeric data for presentation as dollars and cents.
2. _____ Returns the lower-case conversion of a string.
3. A variable name _____ begin with the underscore (_) character.

4.7 LET US SUM UP

Variables are named storage areas inside computer memory where a program places data during processing. The data value on the right of the equal sign is assigned to (is placed in) the variable identified on the left of the equal sign. Constants are very similar to variables. When you declare a constant its value is also specified and this value cannot be changed during program execution. The two most common ways to manipulate constants and variables are through arithmetic and string operations. Iteration and decision making is possible in VB.NET. Sometimes you may need to test multiple "mutually exclusive" conditions - a series of conditions only one of which will be true. The If...Elseif construct tests for as many such conditions as need to be tested. A Do...Loop creates an iteration. It offers the additional option of executing the statements in the loop at least one time. With the While...End While type of programming loop the number of iterations is known when it begins. A message box is a special dialog box used to display a piece of information to the user. As opposed to a regular form, the user cannot type anything on the dialog box. Function is a method which returns a value. A User-Defined Function, or UDF, is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment. Control is an object that can be drawn on to the Form to enable or enhance user interaction with the application. Values of one data type can be converted to another data type. This operation is called casting from one type to another.

4.8 LESSON END ACTIVITIES

1. Start a new project. Add a textbox, a Label and a button to your new Form. Then write a programme that does the following:

- a) Asks users to enter a number between 10 and 20.
 - b) The number will be entered into the Textbox.
 - c) When the Button is clicked, your Visual Basic code will check the number entered in the Textbox.
 - d) If it is between 10 and 20, then a message will be displayed.
 - e) The message box will display the number from the Textbox.
 - f) If the number entered is not between 10 and 20 then the user will be invited to try again, and whatever was entered in the Textbox will be erased
2. Add a Combo box and another button to your form. Create a list of items for your Combo Box. The list of items in your Combo box can be anything you like - pop groups, football teams, favourite foods, anything of your choice. Then try the following:
- a) Use a select case statement to test what a user has chosen from your drop-down list. Give the user a suitable message when the button was clicked.
 - b) Put two textboxes on your form. The first box asks users to enter a start position for a For Loop; the second textbox asks user to enter an end position for the For loop. When a button is clicked, the programme will add up the numbers between the start position and the end position. Display the answer in a message box. Get the startNumber and endNumber from the textboxes.
 - c) Amend your code to check that the user has entered numbers in the textboxes. You will need an If statement to do this. If there's nothing in the textboxes, you can halt the programme.

4.9 KEYWORDS

Variable: It is a named storage area inside computer memory where a program places data during processing.

Function: It is a method which returns a value

User-Defined Function: It is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment.

Control: It is an object that can be drawn on to the Form to enable or enhance user interaction with the application.

Casting: The operation which converts the values of one data type to another data type is called casting.

4.10 QUESTIONS FOR DISCUSSION

1. What are the different data types available in VB.NET? How the data is assigned to the variable?
2. Compare and contrast between conditional operator and logical operator.
3. Write short notes on
 - a) Constants

- b) Arithmetic Operators
- c) Matching String Operator
- d) MsgBox() function
- e) Control class

Check Your Progress: Model Answers

CYP 1

- 1. Variables
- 2. Dim

CYP 2

- 1. value
- 2. Application Programming Interface

CYP 3

- 1. FormatCurrency()
- 2. LCase()
- 3. can not

4.11 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*. Pearson edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic.Net*, 5th Edition, PHI

Kant, *Visual Basic.Net A Beginners Guide*, TMCH

LESSON

5

DECISION MAKING

CONTENTS

- 5.0 Aims and Objectives
- 5.1 Introduction
- 5.2 Conditional Statement
 - 5.2.1 If-then
 - 5.2.2 Select-Case
- 5.3 Looping
 - 5.3.1 Do
 - 5.3.2 While...End While
 - 5.3.3 For Next
 - 5.3.4 Nested loops
- 5.4 Let us Sum up
- 5.5 Lesson End Activities
- 5.6 Keywords
- 5.7 Questions for Discussion
- 5.8 Suggested Readings

5.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Concept of conditional statement
- Brief idea of looping
- Knowledge of nested looping

5.1 INTRODUCTION

The decision making statement is responsible for making a program an intelligent entity. However, not using the proper decision-making can cause a program to fail or possibly become dumb. This is known as a computer error. In Visual Basic, decision making is done by using an If...Then...Else or a Select Case statement.

5.2 CONDITIONAL STATEMENT

One of the powerful features in computer languages is the ability to make decisions. A script is written to test some condition - normally based on the value contained in a variable - and to take alternate courses of action based on whether the condition tested is evaluated as Boolean True or False. In other words, the program can branch to alternate sections of code rather than execute statement in a strict linear sequence.

5.2.1 If-then

The If Statement

Simple decision making in Visual Basic takes place with the If statement, which has the following general format.

If condition Then

...statements

End If

A condition - one that can be evaluated as either True or False - is posited. If evaluation of the condition turns out to be True, then one or more statements are executed; if the condition is False, the statements are not executed. Thus, depending on the condition, either some processing takes place or it does not.

Conditional Operators

Conditional expressions are set up using relational operators to compare values. These operators, which were presented earlier, are repeated in the following table.

Operator	Use
=	Tests whether two values are equal.
<>	Tests whether two values are <i>not</i> equal.
<	Tests whether the first value is less than the second value.
>	Tests whether the first value is greater than the second value.
<=	Tests whether the first value is less than or equal to the second value.
>=	Tests whether the first value is greater than or equal to the second value.

A conditional expression is formed by relating values - either variables or constants - with these operators. The comparison is evaluated as Boolean True or False, and the script reacts to that condition. The following script presents a simple example of script decision making.

```
Sub Button_Click (Src As Object, Args As EventArgs)
If Args.CommandName = "Button 1" Then
    ButtonResponse.Text = "You clicked Button 1"
End If
End Sub
```

Both button clicks call the Button_Click subprogram. The subprogram tests the CommandName argument passed to it to see if it is "Button 1". If so, then a message is written to the textbox. If a different CommandName is passed, then no action is taken. The condition test Args.CommandName = "Button 1" checks for equality of both sides of the relational operator.

Logical Operators

Sometimes you need to combine condition tests. For instance, you might need to know if a numeric value is within a particular range; that is, whether the number is greater than one value and less than another value. Or, you might want to see if the number is outside a range by testing whether it is less than one value or greater than a second value. These kinds of combination tests are made possible by using two or

more relational tests combined with the logical operators And, Or, and Not to form multiple comparisons. These operators were presented earlier and are repeated in the following table.

Logical Operator	Comparison
And	condition1 And condition2 The condition1 and condition2 tests both must be true for the expression to be evaluated as True.
Or	condition1 Or condition2 Either the condition1 or condition2 test must be true for the expression to be evaluated as True.
Not	Not condition The expression result is set to its opposite; a True condition is set to False and a False condition is set to True.
Xor	condition1 Xor condition2 Either condition1 is true or condition2 is true, but not both, for the condition test to be evaluated as True.

Below is an example that uses a multiple condition test. After entering a value in the text box, click the "Check" button. The script determines if you entered a number between 1 and 10 and, if so, confirms this value with a message.

```
Sub Check_Number (Src As Object, Args As EventArgs)
    If Number.Text <> "" Then
        If Number.Text >= 1 And Number.Text <= 10 Then
            NumberResponse.Text = "The number is between 1 and 10."
        End If
    End If
End Sub
```

This script has an If statement nested inside a containing If statement. The outer statement,

```
If Number.Text <> "" Then
```

checks whether a value has been entered into the textbox (it is not equal to empty, or null). If this condition is true, then the enclosed statements are executed. The enclosed statements form a second If test. In this case the value entered in the textbox is tested so see if it is greater than or equal to 1 and if it is less than or equal to 10. If these two conditions are true, the message is displayed.

The If...Else Statement

In the previous examples the If statement checks a condition and takes action on a True condition; it does not take any action on a False condition. An If...Else alternative form of the statement permits action on both True and False conditions. This format is shown below.

```
If condition Then
    ...statements
Else
    ...statements
End If
```


Here, the script takes alternative courses of action depending on the results of the condition test. If the expression is evaluated as True, then the immediately following set of statements is executed; if the expression is evaluated as False, the set of statements following the Else is executed. Thus, one of two different processing actions take place.

The following script is an example of the If...Else form of condition testing.

```
Sub Check_Button (Src As Object, Args As CommandEventArgs)
    If Args.CommandName = "First Button" Then
        Message.Text = "You clicked the first button."
    Else
        Message.Text = "You clicked the second button."
    End If
End Sub
```

The clicked button calls the Check_Button procedure, passing the CommandName of the button as an argument. The If statement tests this Args.CommandName. If its value is "First Button" then the first Message.Text assignment is made; otherwise (Else) the second Message.Text assignment is made.

Nesting If Statements

Two or more If statements can be nested. That is, an If statement can appear inside an If statement, which can appear inside an If statement, and so forth. In a general form these nested If statements appear as follows.

```
If condition Then
    If condition Then
        If condition Then
            ...statements
        Else
            ...statements
        End If
    Else
        If condition Then
            ...statements
        Else
            ...statements
        End If
    End If
End If
```

You need to be cautious, however, about nesting if statements too deeply. It is easy to get lost in the logic and the syntax. Often, you can reformulate the logic of comparisons and make simpler combinations of tests.

The If...ElseIf Statement

Sometimes you may need to test multiple "mutually exclusive" conditions - a series of conditions only one of which will be true. The If...ElseIf construct tests for as many such conditions as need to be tested. Its general format is shown below.

```

If condition1 Then
    ...statements1
ElseIf condition2 Then
    ...statements2
ElseIf condition3 Then
    ...statements3
...
[ Else
    ...statements ]
End If

```

If condition1 is evaluated as True, then the set of statements1 is executed; if condition1 is evaluated as False, then condition2 is evaluated. If this expression is evaluated as True, then the set of statements2 is executed; if the expression is evaluated as False, then condition3 is evaluated, and so forth for as many tests as are coded. The final, optional, Else condition is provided in case none of the previous tests is evaluated as True. It is the "default" condition. The point of the structure is that only one of the parallel tests can be evaluated as True, and only one set of statements is executed.

The following example uses a set of If...ElseIf statements to check whether an entered character is in the first, middle, or last third of the alphabet.

```

Sub Check_Letter (Src As Object, Args As EventArgs)
    If      UCase(Letter.Text) Like "[ABCDEFGHI]" Then
        Message.Text = "Character is in 1st third of alphabet."
    ElseIf UCase(Letter.Text) Like "[JKLMNOPQR]" Then
        Message.Text = "Character is in 2nd third of alphabet."
    ElseIf UCase(Letter.Text) Like "[STUVWXYZ]" Then
        Message.Text = "Character is in last third of alphabet."
    Else
        Message.Text = "Character is not alphabetic."
    End If
End Sub

```

The character entered is converted to upper case for testing with the string operator Like against a set of characters represent the first, middle, and last thirds of the alphabet. A set of If...ElseIf statements is used because only one of the parallel tests will be true, and an Else condition is provided in case a non-alphabetic character is entered.

Of course, a standard If...Else construct could be used for these tests. However, their construction is more difficult to write and to visualize, and each If statement must be paired with an End If statement.

```

Sub Check_Letter (Src As Object, Args As EventArgs)
    If UCase(Letter.Text) Like "[ABCDEFGHI]" Then
        Message.Text = "Character is in 1st third of alphabet."
    Else

```

```
If UCase(Letter.Text) Like "[JKLMNOPQR]" Then
    Message.Text = "Character is in 2nd third of alphabet."
Else
    If UCase(Letter.Text) Like "[STUVWXYZ]" Then
        Message.Text = "Character is in last third of alphabet."
    Else
        Message.Text = "Character is not alphabetic."
    End If
End If
End If
End Sub
```

5.2.2 Select-Case

A decision structure that is similar to If...ElseIf tests for a match against a discrete value. This Select...Case statement has the general format shown below.

```
Select test value
    Case match value1
        ...statements
    Case match value2
        ...statements
    Case match value3
        ...statements

    ...
[ Case Else
    ...statements ]
End Select
```

A test value is an expression that resolves to a single data value. Each of the match values in the Case statements is one of the possible values that matches the test value. The statements within the matching Case are executed. A Case Else option can be provided for a non-match.

The following example is a simple implementation of Select...Case to give you an idea of how it works.

```
Sub Check_Color (Src As Object, Args As EventArgs)
    Select Color.SelectedItem.Value
        Case "Red"
            Message.Text = "You chose Red."
        Case "Blue"
            Message.Text = "You chose Blue."
        Case "Green"
            Message.Text = "You chose Green."
        Case Else
```

```

        Message.Text = "You chose no color."
    End Select
End Sub

```

A selection from the drop-down list produces one of four discrete string values. This value, `Color.SelectedItem.Value`, is resolved in the `Select` statement and tested against the possible values of the selection in the `Case` statements. A match between the selected value and one of the match values produces an appropriate message.

When setting up a `Select...Case` decision structure you must make sure that the `Select` expression resolves to one of the basic data types - a single string or numeric value. The matching `Case` values can be single values of the same data type as in the above example, or they can be a list of values, a range of values, or a relational test that restricts a selection of values.

Matching Lists. To specify a list of matching values for a particular `Case`, separate the values with commas.

```
Case 1, 2, 3
```

```
...
```

```
Case 4, 5, 6
```

```
...
```

```
Case 7, 8, 9, 10
```

Matching Ranges. To specify a range of matching values for a particular `Case`, use an expression in the format "value To value" to indicate the range. Any value that falls within the collating sequence of the range matches the `Case`.

```
Case "A" To "I"
```

```
...
```

```
Case "J" To "R"
```

```
...
```

```
Case "S" To "Z"
```

It is not necessary to use single character strings. A match can be made on "soup" To "nuts" or "alpha" To "omega", testing for a value that falls within the sorted range of words.

Matching Expressions. To specify a relational test to specify a matching value, use an expression in the format "Is operator value" where operator is one of the relational operators (`=`, `<>`, `<`, `>`, `<=`, `>=`; the string operator `Like` cannot be used) and value is a data item of the same `Select` type.

```
Case Is < Some_Value
```

```
...
```

```
Case Is = Some_Value
```

```
...
```

```
Case Is > Some_Value
```

Also, you can combine any of the single-value, list-value, range-value, and expression-value tests for a single `Case`:

```
Case 1, 2, 3 To 9, Is < 10
```

Check Your Progress 1

True or False:

1. The program always executes statement in a strict linear sequence.
2. Conditional expressions are set up using relational operators to compare values.

5.3 LOOPING

Statements in a Visual Basic program are normally executed in the physical order of their appearance unless altered by a decision structure such as an If or Select statement. Even in these cases the statements are executed in a top-to-bottom sequence.

There are certain processing situations, however, that require a set of statements to be executed repeatedly - to iterate over and over again until a certain condition is met. You might, for instance, need to look through the items in an array, one at a time, until you find a matching value. The code to affect this search needs to execute again and again, each time indexing to the next element in the array.

5.3.1 Do

A Do...Loop creates an iteration. It offers the additional option of executing the statements in the loop at least one time.

```
Do [While|Until] condition
    ...statements
Loop
or
Do
    ...statements
Loop [While|Until] condition
```

In its Do While...Loop configuration the loop executes as long as a condition test at the beginning of the loop remains True. The condition may be False at the start of the loop, in which case the loop is exited before it begins. Somewhere inside the loop the condition becomes True for exiting the loop. As in the case with the While...End While loop, the condition test is for continuing the loop.

In its Do Until...Loop configuration the condition test is for ending the loop. Otherwise, the logic is the same as for the Do While...Loop.

If there is a situation in which the statements in the loop will be executed at least one time, then the Do...While Loop or Do...Until Loop configuration can be used. The condition test is not made until the end of the loop, after the statements have been executed the first time.

The Do...Loop presents special condition-test options that don't normally appear in routine data processing. For the most part, For...Next and While...End While statements can handle the bulk of processing. As an illustration, though, of the Do...Until Loop option, the following script loads the StatesArray and CodesArray arrays that are used in a previous example of the While...End While loop. It reads

records from an external text file to populate the arrays, then writes the information to this page.

```
Dim StatesArray(0) As String
Dim CodesArray(0) As String
Sub Page_Load
    Dim FileReader As StreamReader
    Dim LineIn As String
    Dim FieldArray() As String
    FileReader =
File.OpenText("e:\WebSite\tutorials\vbnet\vbnet04\StateCodes.txt")
    LineIn = FileReader.ReadLine()
    Do
        FieldArray = Split(LineIn, ",")
        StatesArray(UBound(StatesArray)) = FieldArray(0)
        CodesArray(UBound(CodesArray)) = FieldArray(1)
        LineIn = FileReader.ReadLine()
        If LineIn <> ""
            ReDim Preserve StatesArray(StatesArray.Length)
            ReDim Preserve CodesArray(CodesArray.Length)
        End If
    Loop Until LineIn = ""
    FileReader.Close()
    Dim i As Integer
    For i = 0 To UBound(StatesArray)
        CodesOut.Text &= StatesArray(i) & ", " & CodesArray(i) & "<br/>"
    Next
End Sub
```

This version of the script makes the assumption that there is at least one record in the external file from which the arrays are built. This is a proper assumption in this particular example, but should not be the premise for most external data stores. A better solution is to use the Do While...Loop construct, making the condition test prior to any processing. Yet, the While...End While serves this exact purpose.

As a general rule, use the While...End While statement for loops for which an explicit number of iterations is not known. Use the Do...Loop Until version if you are a balding 60-year-old renegade programmer with a pony tail.

5.3.2 While...End While

With the While...End While type of programming loop the number of iterations is known when it begins. If not known explicitly, it is known by a variable that resolves to an integer that sets its ending value. The loop iterates that many times unless exited on a condition that arises inside the loop. There may be situations, however, where it cannot be known in advance exactly how many times to execute the statements - to what value the ending value should be set.

The While...End While Loop

The While...End While loop is designed for these situations. Its general format is shown below.

```
While condition
    ...statements
End While
```

A While...End While loop does not execute a fixed number of times. It executes as long as a condition test at the beginning of the loop remains True. It could be that the condition is False at the start of the loop, in which case it doesn't get executed at all. Of course, it is important that somewhere inside the loop the condition becomes True; otherwise, the loop never ends. Whereas a For...Next loop presents a condition for ending the loop, a While...End While presents a condition for continuing the loop.

Before a more practical application is presented, take a look at how a While...End While loop can be set up to resemble a For...Next loop. As in a previous illustration an array is loaded with values and a loop displays its contents.

```
Sub Get_Fruit (Src As Object, Args As EventArgs)
    Dim Fruit() As String =
    {"apples", "oranges", "lemons", "grapes", "beer"}

    Dim i As Integer = 0
    While i <= Fruit.Length - 1
        FruitOut.Text &= Fruit(i) & " "
        i += 1
    End While
End Sub
```

It is important to notice two requirements for setting up the loop. First, the condition to be tested is established and initialized before the loop begins. In this case variable *i* is declared and initialized to 0 (since it is used to increment through the array, beginning with element 0). Second, provision is made inside the loop for ending the loop. In this case the counter *i* is incremented, ensuring that the loop will eventually end when it reaches the Length - 1 of the array. These provisions must be met for all While...End While loops.

5.3.3 For Next

Visual Basic provides this ability to iterate a set of statements with the For...Next statement whose general format is shown below.

```
For counter = start value To end value
    ...statements
Next
```

A For...Next loop establishes a counter to keep track of iterations through the enclosed statements. It is given a start value and an end value - integer values - to control the number of times the statements are executed, with the counter incremented by 1 each time through the loop.

The following code, for example, executes the statements in the For...Next loop 10 times.

```
Dim i As Integer
For i = 1 To 10
```

```
...statements  
Next
```

Variable *i* is defined as the counter. (This choice of counter names owes an historical debt to Fortran programming.) The counter is initialized to 1 when the loop is first entered. After the statements are executed the first time and the `Next` statement is encountered, program control returns to the matching `For` statement. The counter is incremented by 1 and a test is made to see if the counter has yet reached the end value 10. If not, the statements are executed a second time, and control again returns to the `For` statement where the counter is incremented again and is tested against the end value. This processing continues until finally the counter exceeds the end value and the loop comes to an end. Then, program control "skips over" the loop and continues in sequence with the statement following `Next`.

In the above example, start value and end value are given literal integer values. More likely, start value, and especially end value, are assigned through variables set elsewhere in the script.

The following script is a simple illustration of a `For...Next` loop. The user enters an integer value that is used as the ending value for the loop. Beginning with 1, the loop displays the sequence of integers to the ending value.

```
Sub Make_Numbers (Src As Object, Args As EventArgs)  
    If IsNumeric(EndValue.Text) Then  
        Dim i As Integer  
        For i = 1 To EndValue.Text  
            Numbers.Text &= i & " "  
        Next  
    End If  
End Sub
```

Since loop counter values must be numeric, a test is made at the beginning of the subprogram for a numeric value in the textbox. Each time through the loop, increasing values of counter *i* are concatenated to the previously displayed string of values, separated by a blank space.

5.3.4 Nested loops

A nested loop is a loop within a loop, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. Of course, a `break` within either the inner or outer loop would interrupt this process.

For example, the following demo procedure shows how you can nest different loop types, and can exit any loop to any level:

```
Sub LoopTest()  
    Dim dir As New DirectoryInfo("C:\")  
    Dim i As Integer = 0  
    Dim j As Integer = 0  
    For Each file As FileInfo In dir.GetFiles("*.*)")  
        Do While i < 10  
            Console.WriteLine("{0} {1}", i, file.FullName)  
            i++  
        Loop  
        j++  
    Next  
End Sub
```



```

While j < 100
    j += 1
    If j = 1 Then
        Exit While
    End If
    If j = 2 Then
        Exit Do
    End If
    If j = 3 Then
        Exit For
    End If
End While
Console.WriteLine("===After Do While loop")
i += 1
Loop
Console.WriteLine("===After While loop")
Next
Console.WriteLine("===After For loop")
End Sub

```

Check Your Progress 2

1. Give example of two conditional operators.

.....

2. Give example of two logical operators.

.....

3. What is the general rule to use the While...End While statement?

.....

5.4 LET US SUM UP

Iteration and decision making is possible in VB.NET. Sometimes you may need to test multiple "mutually exclusive" conditions - a series of conditions only one of which will be true. The If...ElseIf construct tests for as many such conditions as need to be tested. A Do...Loop creates an iteration. It offers the additional option of executing the statements in the loop at least one time. With the While...End While type of programming loop the number of iterations is known when it begins.

5.5 LESSON END ACTIVITIES

1. Start a new project. Add a textbox, a Label and a button to your new Form. Then write a programme that does the following:
 - a) Asks users to enter a number between 10 and 20.
 - b) The number will be entered into the Textbox.

- c) When the Button is clicked, your Visual Basic code will check the number entered in the Textbox.
- d) If it is between 10 and 20, then a message will be displayed.
- e) If the number entered is not between 10 and 20 then the user will be invited to try again, and whatever was entered in the Textbox will be erased

5.6 KEYWORDS

Conditional Statement: It is a script written to test some condition - normally based on the value contained in a variable - and to take alternate courses of action based on whether the condition tested is evaluated as Boolean True or False.

Looping: There are certain processing situations, however, that require a set of statements to be executed repeatedly - to iterate over and over again until a certain condition is met.

Nested Loop: It is a loop within a loop, an inner loop within the body of an outer one.

5.7 QUESTIONS FOR DISCUSSION

1. What is conditional statement? Why is it required?
2. Write a program code with Nested Loop.
3. Compare and contrast between:
 - (a) AND and OR logical operator
 - (b) Do Until...Loop and Do While...Loop.

Check Your Progress: Model Answers

CYP 1

1. False
2. True

CYP 2

1. > and <
2. AND & OR
3. As a general rule, use the While...End While statement for loops for which an explicit number of iterations is not known.

5.8 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*. Pearson edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic.Net*, 5th Edition, PHI

Kant, *Visual Basic.Net— A Beginners Guide*, TMCH

LESSON

6

FUNCTIONS

CONTENTS

- 6.0 Aims and Objectives
- 6.1 Introduction
- 6.2 Import Statement
- 6.3 MsgBox
 - 6.3.1 The MsgBox Function
- 6.4 Input Box
- 6.5 Function
 - 6.5.1 User Defined
 - 6.5.2 Calling Functions
 - 6.5.3 Built Functions
- 6.6 Controls
 - 6.6.1 Text Box Controls
 - 6.6.2 Label Controls
 - 6.6.3 Frame Controls
 - 6.6.4 Command Button
 - 6.6.5 Check Box
 - 6.6.6 Option Button
 - 6.6.7 List Box
 - 6.6.8 Combo Controls
 - 6.6.9 Picture Controls
 - 6.6.10 Image Controls
- 6.7 Let us Sum Up
- 6.8 Lesson End Activities
- 6.9 Keywords
- 6.10 Questions for Discussion
- 6.11 Suggested Readings

6.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Concept of Msgbox and Input Box
- User defined and Built functions
- Brief concept of controls

6.1 INTRODUCTION

To build a complete custom project interaction with user is required. User can enter their choice or input. They can receive the confirmation or error message. VB.net provides facility to interact with users. VB.net has a vast collection of functions. Apart from that user can create functions according to their choice and requirement.

6.2 IMPORT STATEMENT

A statement is a complete instruction. It can contain keywords, operators, variables, literals, expressions and constants. Each statement in Visual Basic should be either a declaration statement or an executable statement.

A declaration statement is a statement that can create a variable, constant, data type. They are the one's we generally use to declare our variables. On the other hand, executable statements are the statements that perform an action. They execute a series of statements. They can execute a function, method, loop, etc.

The Imports statement is used to import namespaces. Using this statement prevents you to list the entire namespace when you refer to them.

Example of Imports Statement

The following code imports the namespace System.Console and uses the methods of that namespace preventing us to refer to it every time we need a method of this namespace.

```
Imports System.Console  
  
Module Module1  
  
    Sub Main()  
  
        Write("Imports Statement")  
  
        WriteLine("Using Import")  
  
    End Sub  
  
End Module
```

The above two methods without an imports statement would look like this: System.Console.Write("Imports Statement") and System.Console.WriteLine("Using Import")

6.3 MSGBOX

A message box is a special dialog box used to display a piece of information to the user. As opposed to a regular form, the user cannot type anything on the dialog box. There are usually two kinds of message boxes you will create: one that simply displays information and one that expects the user to make a decision.

Although it displays as a dialog box, a message box is created from a built-in function. There are three functions you can use to create a message box.

6.3.1 The MsgBox Function

Microsoft Visual Basic provides the MsgBox() function used to easily create a message box. To display a simple message with just an OK button, use the MsgBox function with the following syntax:

```
MsgBox( )
```

In this case, the message to display must be passed as a string to the function. Here is an example:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button3.Click
    MsgBox("Welcome to the Wonderful World of Visual Basic")
End Sub
```

If you use this version of the MsgBox() function to create a message box, its title bar would display the name of the application that owns the message box:



Figure 6.1: A message box with one button

In reality, the MsgBox() function provides more arguments to create a complete message box with various options. The syntax of the MsgBox function is

```
Public Function MsgBox(ByVal Prompt As Object, _
                        Optional ByVal Options As
MsgBoxStyle = MsgBoxStyle.OKOnly, _
                        Optional ByVal Title As Object =
Nothing _
                        ) As MsgBoxResult
```

The Prompt argument is the string that the user will see displaying on the message box. You can pass it as a string. You can also create it from other pieces of strings. The Prompt argument can be made of up to 1024 characters. To display the Prompt on multiple lines, you can use either the constant vbCrLf or the combination Chr(10) & Chr(13) between any two strings.

The Options argument specifies what button or buttons and/or small icon should display on the message box. The available options are defined in the MsgBoxStyle enumerator. The buttons available are:

MsgBoxStyle	Value	Display
OKOnly	0	
OKCancel	1	
AbortRetryIgnore	2	
YesNoCancel	3	
YesNo	4	
RetryCancel	5	

To use any of these combinations of buttons, call the `MsgBoxStyle` enumerator and access the desired combination. Here is an example:

```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles Button3.Click  
    MsgBox("Your order appears to be correct" & vbCrLf & _  
        "Are you ready to provide your credit card  
information?", _  
        MsgBoxStyle.YesNoCancel)  
End Sub
```

This would produce:

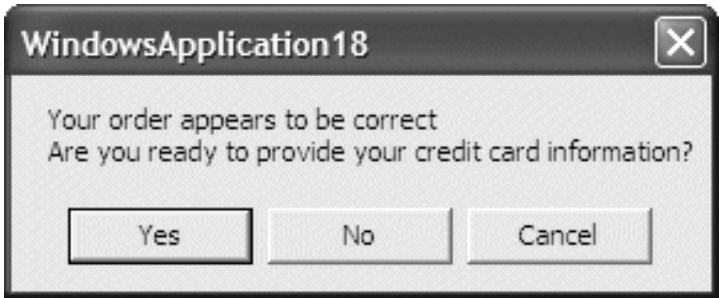


Figure 6.2: A message box with three buttons

Besides displaying the button(s), you can add other options. To combine options, you use the bit manipulation operator `OR`. For example, after specifying the buttons on the message box, you can decide which one would be the default, that is, which button would be activated if the user presses `Enter` instead of clicking. You can set the default argument using the following table

Option	Value	Example
DefaultButton1	0	
DefaultButton2	256	



Here is an example of combining styles:









```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button3.Click
```

```
    MsgBox("Your order appears to be correct" & vbCrLf & _
        "Are you ready to provide your credit card information?", _
        MsgBoxStyle.YesNoCancel Or MsgBoxStyle.DefaultButton2)
End Sub
```

These additional buttons can be used to further control what the user can do:

Constant	Value	Result
ApplicationModal	0	This creates a modal dialog box and the user must close the message box before continuing to use the application from where the message box was displayed
SystemModal	4096	This creates a modal dialog box that has an "accent" of modeless. The message box will stay on top of all other windows, including those from applications that have nothing to do with this message box. Like a normal message box, the user must close this message box to continue using the application that opened this message box

Besides the buttons, you can display an icon on the message box. The available icons are:

Icon	Value	Description	
Critical	16		
Question	32		
Exclamation	48		
Information	64		

Here is an example:

```
Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button3.Click
```

```
    MsgBox("Your order appears to be correct" & vbCrLf & _
        "Are you ready to provide your credit card information?", _
```

```

                                MsgBoxStyle.YesNoCancel
MsgBoxStyle.Information Or MsgBoxStyle.SystemModal)

End Sub

```

Or

This would produce:

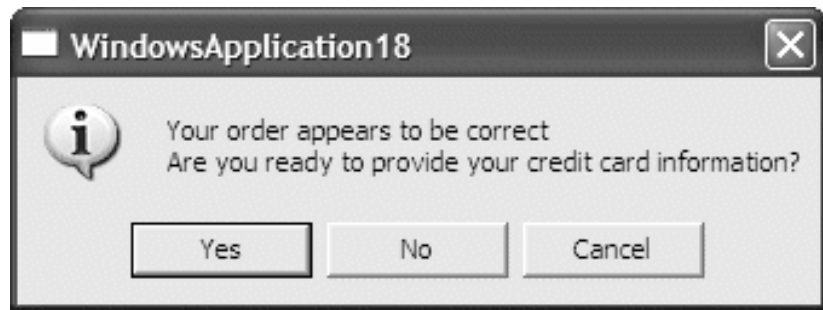
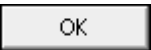



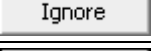
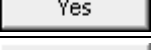
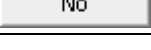


Figure 6.3: A message box

So far, all of the message boxes we have created displayed the name of the application on their title bar. If you want to display a customized title, use the Title argument: this would be the caption that would display on the title bar of the message box. It is a string whose word or words you can enclose between parentheses or that you can get from a created string.

MsgBox() is primarily a function. As such, it can return a value. This value corresponds to the button the user clicks on the message box. The return values are defined in the MsgBoxResult enumerator. Depending on the buttons the message box is displaying, after the user has clicked, the MsgBox() function can return one of the following values:

If the User Clicks	The Function Returns	Value
	MsgBoxResult.OK	1
	MsgBoxResult.Cancel	2
	MsgBoxResult.Abort	3
	MsgBoxResult.Retry	4
	MsgBoxResult.Ignore	5
	MsgBoxResult.Yes	6
	MsgBoxResult.No	7

The MessageBox Class

To support its own implementation of a message box, the .NET Framework provides the MessageBox class. This class has one method called Show() but overloaded in various versions. The Show() method is static, meaning that you don't need and must never declare a variable of type MessageBox to display a message box using this class.

To create a simple message box using the MessageBox class, you can pass a single string to its Show() method. In this case, the message box would display the name of the application as its caption. If you want to display your own caption, pass it as a second argument to the method.

The Win32 API's Message Box

Visual Basic provides as much flexibility as needed to create a message using either its own the MsgBox() function or the .NET Framework's MessageBox class. You still have one more option. The Win32 API provides a function called MessageBox. Its syntax is:

```
int MessageBox(HWND hWnd,
               LPCTSTR lpText,
               LPCTSTR lpCaption,
               UINT uType);
```

This function provides the same options as the MsgBox() function reviewed earlier. To use this function, you must include its library using Declare Auto Function as we review already. Here is an example:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Const MB_OK = &H0
    Const MB_ICONEXCLAMATION = &H30
    Declare Auto Function FMessageBox Lib "user32.dll" _
        Alias "MessageBox" (ByVal hWnd As Integer, _
                            ByVal txt As String, ByVal caption As String,
                            _
                            ByVal Typ As Integer) As Integer
    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()
        'This call is required by the Windows Form Designer.
        InitializeComponent()
        'Add any initialization after the InitializeComponent() call
    End Sub

    . . . No Change
#End Region

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
        FMessageBox(0, _
                    "The credit card information you
provided is not correct", _
                    "Sales Processing", _
                    MB_OK Or MB_ICONEXCLAMATION)
    End Sub
End Class
```

This would produce:



Figure 6.4: Another message box

Besides displaying a message, this function can also return a value depending on the button the user would have clicked.

Check Your Progress 1

Fill in the blanks:

1. There are two types of statements _____ and _____.
2. The Imports statement is used to import _____.

6.4 INPUT BOX

An input box is a specially designed dialog box that allows the programmer to request a value from the user and use that value as necessary. An input box displays a title, a message to indicate the requested value, a text box for the user, and two buttons: OK and Cancel. Here is an example:

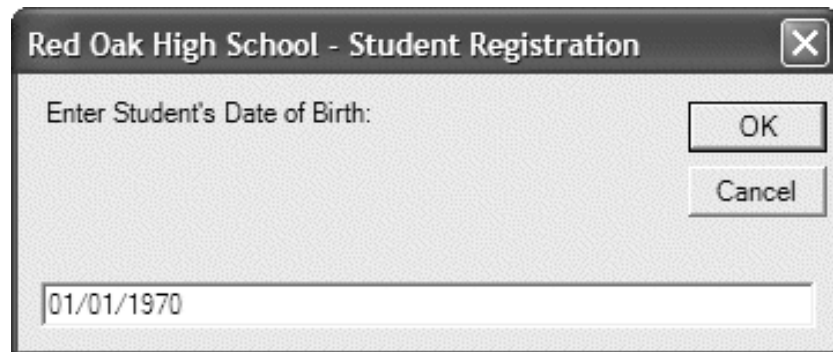


Figure 6.5: An input box

When an input box displays, it presents a request to the user who can then provide a value. After using the input box, the user can change his or her mind and press Esc or click Cancel. If the user provided a value and want to acknowledge it, he can click OK or press Enter. This would transfer the contents of the text box to the application that displayed the input box.

Creating an Input Box

To create an input box, you can use the `InputBox()` function. Its syntax is:

```
Public Function InputBox(ByVal Prompt As String, _
```

```

= "", _                                Optional ByVal Title As String
As String = "", _                      Optional ByVal DefaultResponse
= -1, _                                Optional ByVal XPos As Integer
= -1 _                                 Optional ByVal YPos As Integer

) As String

```

The only required argument of this function is the message that prompts the user. Here is an example:

```

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click

    InputBox("Enter Student's Date of Birth:")

End Sub

```

This would produce

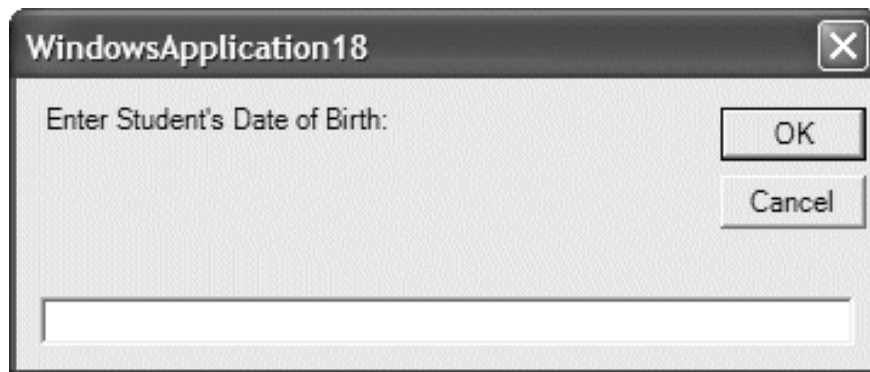


Figure 6.6: Another input box

When calling the InputBox() function, if you pass only the Prompt argument, the input box would display the name of the application in the title bar. If you want, you can specify your own caption through the Title argument. Here is an example:

```

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click

    InputBox("Enter Student's Date of Birth:", "Red Oak High
School - Student Registration")

End Sub

```

Upon reading the message on the Input box, the user is asked to enter a piece of information. The type of information the user is supposed to provide depends on you, the programmer. Therefore, there are two important things you should always do. First you should let the user know the type of information requested. Is it a number (what type of number)? Is it a string (such as the name of a country or a customer's name)? Is it the location of a file (such as C:\Program Files\Homework)? Are you expecting a Yes/No True/False type of answer (if so how should the user provide it)? Is it a date (if it is a date, what format is the user supposed to enter)? These questions indicate that you should state a clear request to the user. For example, instead of the above simple request, you can implement the InputBox() function as follows:

```

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click

```

```

        InputBox("Enter Student's Date of Birth (mm/dd/yyyy):", _
            "Red Oak High School - Student Registration")
    End Sub

```

This would produce:

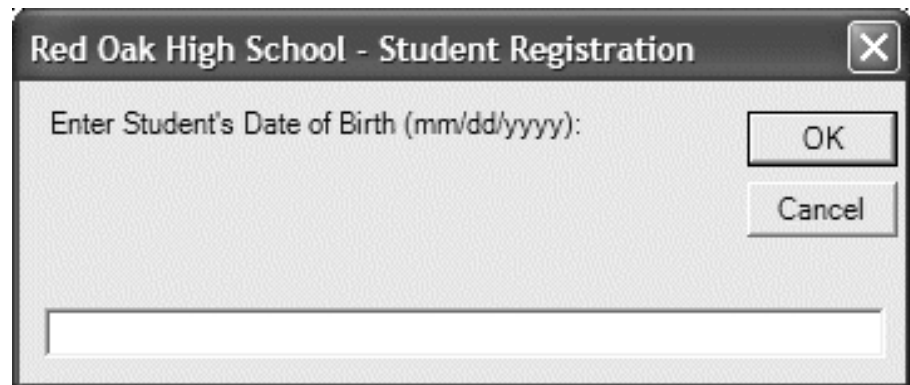


Figure 6.7: An input box

Sometimes, even if you provide an explicit prompt, the user might not provide a new value but click OK. The problem is that you would still need to get the value of the text box and you might want to involve it in an expression. You can solve this problem and that of providing an example to the user by filling the text box with a default value. This can be taken care of through the `DefaultResponse` argument. Here is an example:

```

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click
    InputBox("Enter Student's Date of Birth (mm/dd/yyyy):", _
        "Red Oak High School - Student Registration", "01/01/1970")
End Sub

```

The last two arguments, `XPos` and `YPos`, allow you to specify the default position of the input box when it comes up the first time.

After typing a value, the user would click one of the buttons: OK or Cancel. If the user clicks OK, you should retrieve the value the user would have typed. It is also your responsibility to find out whether the user typed a valid value or not. Because the `InputBox()` function returns a string, it has no mechanism of validating the user's entry. Based on this, you can retrieve the value of the Input Box when the user clicks OK. Here is an example:

```

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click
    Dim DOB As String = InputBox("Enter Student's Date of Birth:")
End Sub

```

6.5 FUNCTION

Function is a method which returns a value. Functions are used to evaluate data, make calculations or to transform data. Declaring a Function is similar to declaring a Sub procedure. Functions are declared with the `Function` keyword. There are two types of functions-user defined and built function.

6.5.1 User Defined

A User-Defined Function, or UDF, is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment.

Function Formats

The general format for a function is shown below.

```
Function name (argument As type [, argument As type]...) [As type]
    ...statements
    [Return value]
    [Exit Function]
End Function
```

A function is identified by the keyword `Function` at its beginning and the keywords `End Function` at its end. A function name is a programmer-supplied name that follows Visual Basic naming conventions. The name must be unique among the collection of functions on the page. Visual Basic statements are enclosed inside the function to carry out its processing.

There are two ways to set up a function to return a value. The function itself can be declared `As type`, a value is assigned to the function name, and an `Exit Function` statement returns the value to the calling statement. This setup is illustrated in the code below which assumes no arguments being passed to the function.

```
Function Get_Value() As Integer
    Get_Value = 10
    Exit Function
End Function
```

The second way to set up a function is to use a `Return` statement to explicitly pass an identified value back to the calling statement.

```
Function Get_Value()
    Dim Value As Integer
    Value = 10
    Return Value
End Function
```

6.5.2 Calling Functions

A function is called by a statement in the following general format.

[Call] Function name (argument [, argument]...)

The optional keyword `Call` is followed by a function name, followed by an argument list, if needed, separated by commas and contained inside a set of parentheses. If no arguments are supplied, the parentheses are not required. As in the case for subprograms, these arguments are usually names of variables whose values are passed to the called function.

The following example shows a function named `Get_Area` which calculates the area of a rectangle when supplied with height and width dimensions by the calling statement.

```
Sub Rectangle
    Dim Width As Integer = 10
    Dim Height As Integer = 5
    Dim Area As Integer
    Area = Get_Area(Width, Height)
End Sub

Function Get_Area (Width As Integer, Height As Integer) As Integer
    Get_Area = Width * Height
    Exit Function
End Function
```

Subprogram Rectangle sets the Width and Height of a rectangle; then it assigns to variable Area the returned value from function Get_Area, supplying it with the width and height values. Function Get_Area receives the passed values through its Width and Height arguments (local to the function and not to be confused with the like-named variables in subprogram Rectangle). The area calculation is assigned to the function name and is returned to the calling statement when the function exits.

As you can see, since a function returns a value, it can be treated just like a variable. In the above case, the function is assigned to a local variable, effectively assigning the returned value to the variable.

An alternative to the above coding is a function that explicitly returns its computed value.

```
Function Get_Area (Width As Integer, Height As Integer)
    Dim Area As Integer
    Area = Width * Height
    Return Area
End Function
```

You can use as many Exit Function and Return statements as needed. In more complex functions with alternate computational paths a value may be returned at different points in the function.

6.5.3 Built Functions

There is a large number of built-in functions. Many of the functions work on several types of arguments, whereas some only work for the correct types (e.g., numbers or strings). In the following description we'll discuss about different built-in functions.

Mathematical Functions

In addition to performing simple arithmetic and string operations with the arithmetic and string operators, Visual Basic programs can take advantage of several built-in mathematical functions and string functions to perform useful processing that, otherwise, could require highly complex original code.

Popular mathematical functions are summarized in the following table. Note that certain functions do not require the Math. prefix.

Function	Use
Math.Abs()	Returns the absolute value. Math.Abs(-10) returns 10.
Math.Ceiling()	Returns an integer that is greater than or equal to a number. Math.Ceiling(5.333) returns 6.
Fix()	Returns the integer portion of a number. Fix(5.3333) returns 5.
Math.Floor()	Returns an integer that is less than or equal to a number. Fix(5.3333) returns 5.
Int()	Returns the integer portion of a number. Int(5.3333) returns 5.
Math.Max()	Returns the larger of two numbers. Math.Max(5,7) returns 7.
Math.Min()	Returns the smaller of two numbers. Math.Min(5,7) returns 5.
Math.Pow()	Returns a number raised to a power. Math.Pow(12,2) returns 144.
Rnd()	Returns a random number between 0 and 1. Used in conjunction with Randomize statement to initialize the random number generator.
Math.Round()	Rounds a number to a specified number of decimal places. Rounds up on .5. Math.Round(1.1234567,5) returns 1.12346.
Math.Sign()	Returns the sign of a number. Returns -1 if negative and 1 if positive. Math.Sign(-5) returns -1.
Math.Sqrt()	Returns the square root of a positive number. Math.Sqrt(144) returns 12.

Random Numbers

The Rnd() function returns a random number between 0 and 1. More likely, the need is to generate a number within a particular range, between a given low and high number. This is accomplished with the following formula.

$\text{Math.floor}((\text{high} - \text{low} + 1) * \text{Rnd()} + \text{low})$

For instance, to generate a random number between 0 and 10 the formula becomes

$\text{Math.floor}((10 - 0 + 1) * \text{Rnd()} + 0)$

String Functions

Several built-in string functions perform string manipulations to augment simple concatenation with the "&" operator. These functions are summarized in the following table.

Function	Use
Asc()	Returns the character code of the first character of a string. Asc("A") returns 65.
Chr()	Returns the display character of a character code. Chr(65) returns "A".
GetChar()	Returns the character at a specified position in a string, counting from 1. GetChar("This is a string", 7) returns "s".
InStr()	Returns the starting position in a string of a substring, counting from 1. InStr("This is a string", "string") returns 11.

Contd...

InStrRev()	Returns the starting position in a string of a substring, searching from the end of the string. InStr("This is a string", "string") returns 11.
LCase()	Returns the lower-case conversion of a string. LCase("THIS IS A STRING") returns "this is a string".
Left()	Returns the left-most specified number of characters of a string. Left("This is a string", 4) returns "This".
Len()	Returns the length of a string. Len("This is a string") returns 16.
LTrim()	Removes any leading spaces from a string. LTrim(" This is a string") returns "This is a string".
Mid()	Returns a substring from a string, specified as the starting position (counting from 1) and the number of characters. Mid("This is a string", 6, 4) returns "is a".
Replace()	Replaces all occurrences of a substring in a string. Replace("This is a string", " s", " longer s") returns "This are a longer string" (replaces an "s" preceded by a blank space).
Right()	Returns the right-most specified number of characters of a string. Right("This is a string", 6) returns "string".
RTrim()	Removes any trailing spaces from a string. RTrim("This is a string ") returns "This is a string".
Str()	Returns the string equivalent of a number. Str(100) returns "100".
Space()	Fills a string with a given number of spaces. "This" & Space(5) & "string" returns "This string".
StrComp()	Compares two strings. Return values are 0 (strings are equal), 1 (first string has the greater value), or -1 (second string has the greater value) based on sorting sequence. StrComp("This is a string", "This string") returns -1.
StrReverse()	Reverses the characters in a string. StrReverse("This is a string") returns "gnirts a si sihT".
Trim()	Removes any leading and trailing spaces from a string. Trim(" This is a string ") returns "This is a string".
UCase()	Returns the upper-case conversion of a string. UCase("This is a string") returns "THIS IS A STRING".
Val()	Converts a numeric expression to a number. Val((1 + 2 + 3)^2) returns 36.

The above summaries give you a basic idea of the uses of these arithmetic and string functions. There are occasions throughout these tutorials to see them in action and in combination as they are applied to various processing needs.

Date and Time Functions

Visual Basic supplies a complete set of date and time functions to extract dates and times from the system clock and to manipulate these values for various computational and display purposes.

Date and Time Properties

When working with dates and times it is often necessary to know the current date and/or time. These values can be extracted from the server's system clock through the properties shown in the table below. These properties are used in several of the date and time functions described below.

Property	Value
DateString	Returns a String value representing the current date. DateString = 03-29-2008
Now	Returns a Date value containing the current date and time. Now = 3/29/2008 10:46:39 AM
TimeOfDay	Returns a Date value containing the current time of day (the date is set to 1/1/0001). TimeOfDay = 1/1/0001 10:46:39 AM
Timer	Returns a Double value representing the number of seconds elapsed since midnight. Timer = 38799.4338593
TimeString	Returns a String value representing the current time of day. TimeString = 10:46:39
Today	Returns or sets a Date value containing the current date. Today = 3/29/2008 12:00:00 AM

Date Functions

Date functions are summarized in the following table with functions described more fully below.

Function	Use
DateAdd()	Returns a Date value containing a date and time value to which a specified time interval has been added.
DateDiff()	Returns a Long value specifying the number of time intervals between two Date values.
DatePart()	Returns an Integer value containing the specified component of a given Date value.
DateSerial()	Returns a Date value representing a specified year, month, and day, with the time information set to midnight (00:00:00).
DateValue()	Returns a Date value containing the date information represented by a string, with the time information set to midnight (00:00:00).
Day()	Returns an Integer value from 1 through 31 representing the day of the month.
IsDate()	Returns a Boolean value indicating whether an expression can be converted to a date.
Month()	Returns an Integer value from 1 through 12 representing the month of the year.
MonthName()	Returns a String value containing the name of the specified month.
WeekDay()	Returns an Integer value containing a number representing the day of the week.

DateAdd() adds a given date interval to a date to produce a calculated date. Its general format is

String	Unit of time interval
d	Day of month (1 through 31)
y	Day of year (1 through 366)
m	Month
q	Quarter
w	Day of week (1 through 7)
ww	Week of year (1 through 53)
yyyy	Year

where interval is one of the string values shown in the accompanying table, number gives the number of intervals to add, and date is the date and time to which the

interval is to be added. For example, the following function returns the (formatted) date six months from today.

```
FormatDateTime(DateAdd("m", 6, Today),  
DateFormat.LongDate)
```

Six months from today is Monday, September 29, 2008.

DateDiff() returns a value giving the number of identified intervals between two dates. Its general format is

```
DateDiff(interval, date1, date2)
```

where interval is a string value representing an interval type using the date values shown in the table under the DateAdd() function, and date1 and date2 are the two dates between which the interval is calculated (date1 is subtracted from date2). The following example calculates the number of shopping days until Christmas.

```
DateDiff("d", Today, "12/24/" & DatePart("yyyy", Today))
```

There are 270 shopping days until Christmas.

Notice that the date2 argument passes "12/24" & DatePart("yyyy", Today). The DatePart() function (see below) extracts the current year from the current date and appends it to the string so that the calculation is always based on the current year.

DatePart() extracts a specified component of a given Date. Its general format is

```
DatePart(part, date)
```

where part is a string value representing a date component using the values shown in the table under the DateAdd() function, and date is any Date value. For example, the following function determines the day of the week on January 1, 2010.

```
WeekdayName(DatePart("w", "01/01/2010"))
```

January 1, 2010 is on a Friday

DatePart() returns an integer value (Sunday = 1) representing the day of the week. This value is converted into a weekday name with the WeekDayName() function (see below).

DateSerial() returns a Date value based upon integer values representing a year, month, and day. Its general format is

```
DateSerial(year, month, day)
```

For example, the function DateSerial(5, 7, 15) returns 7/15/2005 12:00:00 AM. You might wonder why the need to return a date if you already know the date! The point is that the passed values are integers, perhaps collected from a form in which a year, month, and day are selected from drop-down lists. These integer values are combined and converted into a Date type with the DateSerial() function.

DateValue() returns a Date type from a String value representing a date. This function works as a converter to a Date data type. For example, the function call DateValue("January 1, 2005") returns 1/1/2005 12:00:00 AM.

Day() returns an Integer representing the day of the month from a passed date. For instance, function Day(Today) returns 29. This is the same value as returned from the function DatePart("d", Today).

IsDate() returns a Boolean value indicating whether the passed value can be converted into a Date type. This function is handy when testing user input to verify that an actual date has been entered. A function call in the format IsDate("02/31/2005") returns False.

Month() returns an Integer value from 1 through 12 representing the month of the year. It works similar to the Day() function. The function call Month(Today) returns 3.

MonthName() accepts an Integer representing a month of the year and returns a String value containing the name of the month. Used in conjunction with the Month() function, the function call MonthName(Month(Today)) produces March.

WeekDay() returns an Integer value between 1 and 7 (Sunday = 1) representing the day of the week. The function WeekDay(Today) returns 7, which is identical to the value returned by DatePart("w", Today).

WeekDayName() accepts an Integer representing a day of the week and returns a String value containing the name of the day of the week. Used in conjunction with the Weekday() function, the function call WeekDayName(WeekDay(Today)) produces Saturday.

Time Functions

Time functions are summarized in the following table with functions described more fully below. Several of the date functions can be applied to time measurements.

Function	Use
DateAdd()	Returns a Date value containing a date and time value to which a specified time interval has been added.
DateDiff()	Returns a Long value specifying the number of time intervals between two Date values.
DatePart()	Returns an Integer value containing the specified component of a given Date value.
Hour()	Returns an Integer value from 0 through 23 representing the hour of the day.
Minute()	Returns an Integer value from 0 through 59 representing the minute of the hour.
Second()	Returns an Integer value from 0 through 59 representing the second of the minute.
TimeSerial()	Returns a Date value representing a specified hour, minute, and second, with the date information set relative to January 1 of the year 1.
TimeValue()	Returns a Date value containing the time information represented by a string, with the date information set to January 1 of the year 1.

DateAdd() adds a given time interval to a time to produce a calculated time. Its general format is

DateAdd(interval, number, time) String Unit of time interval

String	Unit of time interval
h	Hour
n	Minute
s	Second

where interval is one of the string values shown in the accompanying table, number gives the number of intervals to add, and time is the date and time to which the interval is to be added. For example, the following function returns the (formatted) time twelve hours from now.

FormatDateTime(DateAdd("h", 12, Now), DateFormat.LongDate)

Twelve hours from now is 10:46:39 PM.

DateDiff() can be used to return a value giving the number of identified intervals between two times. Its general format is

`DateDiff(interval, time1, time2)`

where `interval` is a string value representing an interval type using the time values shown in the table under the `DateAdd()` function, and `time1` and `time2` are the two times between which the interval is calculated (`time1` is subtracted from `time2`). The following example calculates the number of minutes until midnight.

`DateDiff("n", TimeString, "11:59:59 PM") + 1`

There are 794 minutes until midnight.

It can be a bit tricky working with times because of the roll-over that takes place at midnight. If the value "00:00:00 AM" were to be used in the above example, it would produce -645 minutes, which are the number of minutes from the beginning of the current day. To get around this problem, one minute is added to the time difference until "11:59:59 PM" of the current day.

`DatePart()` can be used to extract a specified component of a given time. Its general format is

`DatePart(part, time)`

where `part` is a string value representing a time component using the time values shown in the table under the `DateAdd()` function, and `time` is any Date value. For example, the following function determines the current hour (24-hour clock) of the current day.

`DatePart("h", Now)`

The current hour of the day is 10

`Hour()` returns an Integer representing the hour of the day (24-hour clock). For instance, function `Hour(Now)` returns 10. This is the same value as returned from the function `DatePart("h", Now)`.

`Minute()` returns an Integer representing the minute of the hour. For instance, function `Minute(Now)` returns 46. This is the same value as returned from the function `DatePart("n", Now)`.

`Second()` returns an Integer representing the second of the minute. For instance, function `Second(Now)` returns 39. This is the same value as returned from the function `DatePart("s", Now)`.

`TimeSerial()` returns a Date value representing a specified hour, minute, and second, with the date information set relative to January 1 of the year 1. Its general format is

`TimeSerial(hour, minute, second)`

For example, the function `TimeSerial(5, 30, 45)` returns 1/1/0001 5:30:45 AM. As in the case for the `DateSerial()` function, integer values are combined and converted into a Date type.

`TimeValue()` returns a Date type from a String value representing a time. This function works as a converter to a Date data type. For example, the function call `TimeValue("4:35:17 PM")` returns 1/1/0001 4:35:17 PM.

Some of the date and time functions appear trivial in isolation. However, you will find them crucial when you begin combining them in applications that rely on accurate reporting of and calculations involving dates and times.

Comparison Functions

It is often necessary to test the contents of variables and other data containers to determine if they contain values and, if so, whether they contain a particular data type. For example, the user may be expected to enter data into a textbox for processing by a script, with processing contingent on having a proper data type available.

Testing for the presence of data can be performed with a relational operation such as `TextBox.Text = ""`. This test returns `True` if there is no value in the textbox; the textbox is empty or contains a null value. It returns `False` if data is present. (Usually, this test is formulated as `TextBox.Text <> ""`, testing for the expected condition that data is present.)

In testing for expected data types, Visual Basic supplies a couple of functions. `IsNumeric()` tests for a numeric value; `IsDate()` tests for a date/time data type. Thus, the test `IsNumeric(TextBox.Text)` returns `True` if the textbox contains a number; it returns `False` if the textbox is empty or if it does not contain a number. The same return values are produced when testing for a date/time type.

Check Your Progress 2

Fill in the blanks:

1. _____ function returns a Date value containing the current date and time
2. _____ function returns an Integer value containing the specified component of a given Date value.
3. _____ function returns an Integer value from 0 through 23 representing the hour of the day.

6.6 CONTROLS

Control is an object that can be drawn on to the Form to enable or enhance user interaction with the application. Examples of these controls, TextBoxes, Buttons, Labels, Radio Buttons, etc. All these Windows Controls are based on the Control class, the base class for all controls. Visual Basic allows us to work with controls in two ways: at design time and at runtime. Working with controls at design time means, controls are visible to us and we can work with them by dragging and dropping them from the Toolbox and setting their properties in the properties window. Working at runtime means, controls are not visible while designing, are created and assigned properties in code and are visible only when the application is executed. There are many new controls added in Visual Basic .NET.

Notable properties of most of these Windows Controls which are based on the Control class itself are summarized in the table below. You can always find the properties of the control with which you are working by pressing F4 on the keyboard or by selecting View->Properties Window from the main menu.

The Control Class

The Control class is in the `System.Windows.Forms` namespace. It is a base class for the Windows Controls. The class hierarchy is shown below.

Object

 MarshalByRefObject

```
Component
Control
    ButtonBase, Etc, Etc
        Button, Etc, Etc
```

Main class is the Object class from which MarshalByRefObject class is derived and the Component class is derived from the MarshalByRefObject class and so on.

Control Tab Order

To move focus from one control to other quickly using the keyboard we can use the Tab key. We can set the order in which the focus is transferred by setting the tab order. The tab order is the order in which controls on the form receive the focus and is specified by the TabIndex property. To change the order in which a control receives focus we need to set the TabIndex property to different value for each control on the form. Lower values receive the focus first and proceed numerically through higher values. If there is a tie between TabIndex values, the focus first goes to the control that is closest to the front of the form.

We can also set the tab order graphically with Visual Studio by selecting Tab Index from the View menu. Boxes containing current tab order appear in each control when you select Tab Index from View menu. Click each control to set the correct tab order in which you want the controls to receive focus.

VB.Net has some basic controls. The creator of the control is responsible for the programming that enables the interface: for making sure that properties appear in the properties window, that events are fired when they are supposed to be fired etc.

The user of the control doesn't see the internal implementation of the control's application. The users interaction with the control depends on the interface settings that the application developer has made and on code added by the developer.

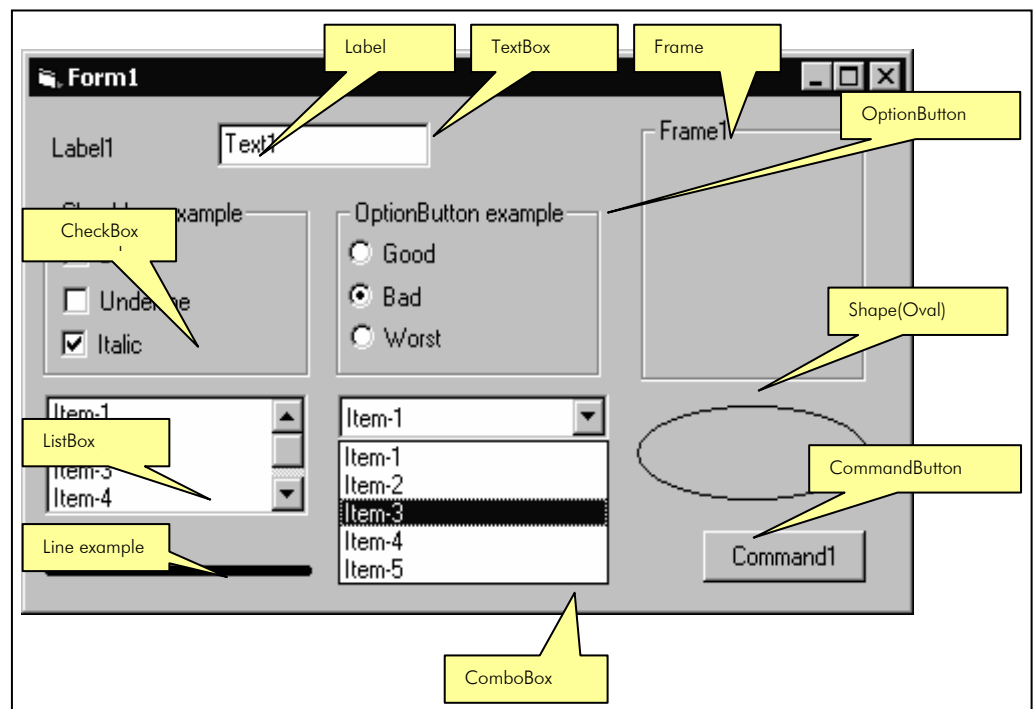


Figure 6.8: Different basic controls

The icons, which you see on the toolbox when you start a VB project, are all standard controls provided by Microsoft.



Figure 6.9: Standard Microsoft controls

6.6.1 Text Box Controls

This control is used to accept some data from the user. User can type in the textbox and that value can be accessed in any procedure by referring to the Text property of the text box control.

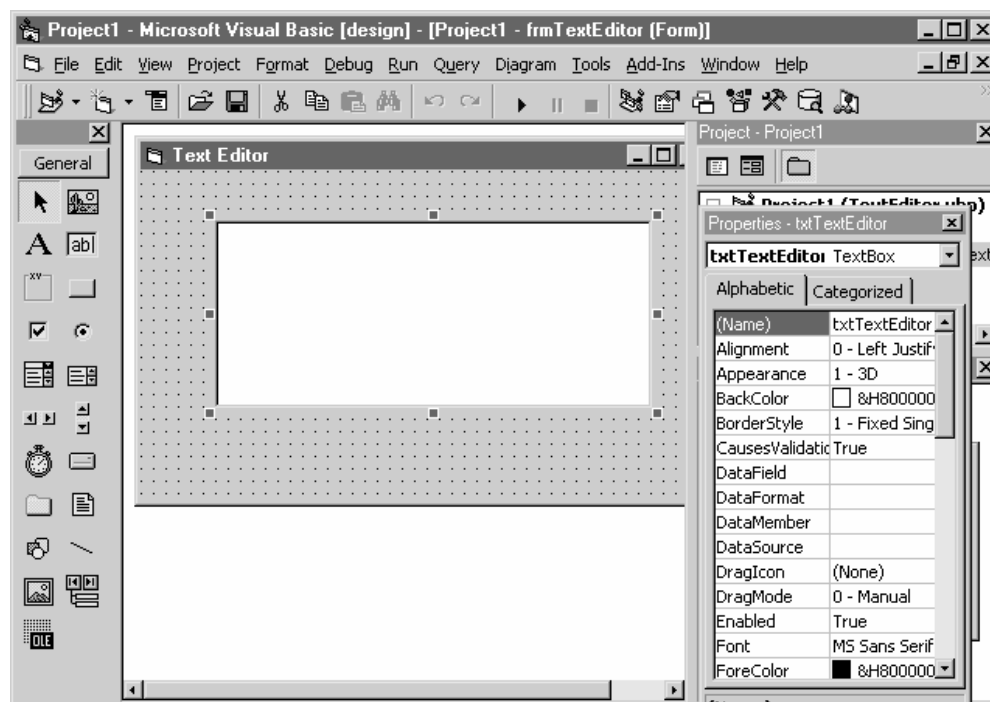


Figure 6.10: The text box when it is being designed

The text box is the standard control that is used to receive input from the user as well as to display the output. It can handle string (text) and numeric data but not images or pictures. String in a text box can be converted to a numeric data by using the function `Val(text)`. The following example illustrates a simple program that processes the inputs from the user.

Example

In this program, two text boxes are inserted into the form together with a few labels. The two text boxes are used to accept inputs from the user and one of the labels will be used to display the sum of two numbers that are entered into the two text boxes. Besides, a command button is also programmed to calculate the sum of the two numbers using the plus operator. The program use creates a variable `sum` to accept the summation of values from text box 1 and text box 2. The procedure to calculate and to display the output on the label is shown below. The output is shown in Figure 6.11.

```
Private Sub Command1_Click()  
    'To add the values in text box 1 and text box 2  
    Sum = Val(Text1.Text) + Val(Text2.Text)  
    'To display the answer on label 1  
    Label1.Caption = Sum  
End Sub
```

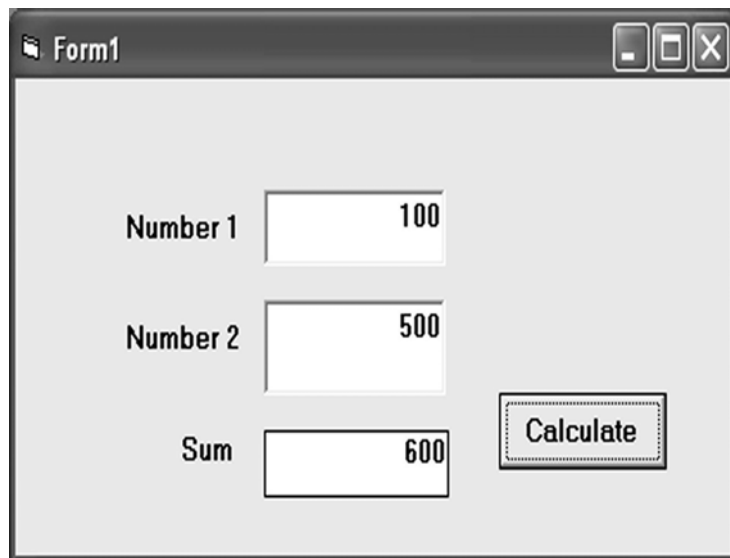
The image shows a Windows application window titled "Form1". Inside the window, there are three text boxes arranged vertically on the left. The first text box is labeled "Number 1" and contains the value "100". The second text box is labeled "Number 2" and contains the value "500". The third text box is labeled "Sum" and contains the value "600". To the right of the "Sum" text box, there is a button labeled "Calculate". The button has a dashed border and a 3D effect. The window has a standard Windows title bar with minimize, maximize, and close buttons.

Figure 6.11: Text Box

6.6.2 **Label Controls**

Labels are one of the most commonly used controls. They are also the simplest to use. You use labels to (yes you've guessed it) label things such as text boxes, where the text does not need to be changed by the user. This control is used to display some static (i.e. user cannot change it) text on the screen. This is generally placed against the controls (like textbox) which accept some input from the user, to convey the message what data to enter.

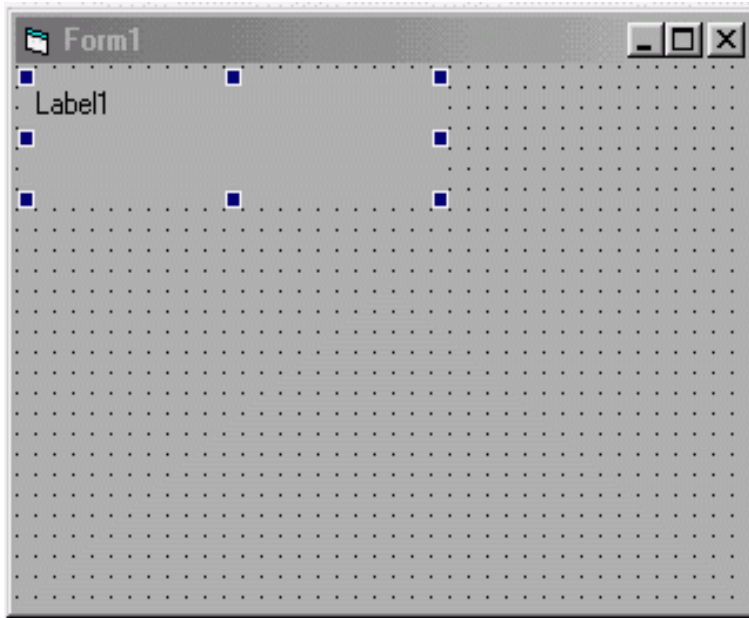


Figure 6.12: Label Control

6.6.3 Frame Controls

A frame control is a box which is used to group other controls together. It is useful in form design because you can move all the controls it contains by moving the frame. A frame has a Caption property and an Enabled property. You can set the enabled property to false to disable all the controls that the frame contains.



Figure 6.13: Frame Control

6.6.4 Command Button

The Command button is a common control frequently used in VB.Net programming. This is one of the first controls that beginners learn to place onto a form and then to code an "event procedure" for the command control object.

An event procedure is something that happens in response to a user action such as a click or a "mouse over". (If an object on a form has no corresponding code written by the programmer, nothing will happen if the user clicks on the control or attempts any other action with that control.)

This control is used to get the user's response depending on which some specific action can be taken. For example, you can see the Ok and Cancel buttons on the screens of the products of Microsoft.

The Cancel Property and the Default Property were inherent only to the Command button. Many properties available to the command button may also be set with other control objects from the toolbar, and even objects not from the toolbar. The ToolTips property is available for setting with any of the control objects.

The ToolTips property is a clever way to have the user read a message about a control object simply by hovering with the mouse pointer over that object for a couple of seconds.

Setting the message of this property is very simple. Just click on the Command button or another object to select it and then in the Properties Window (F4), scroll down to ToolTips and double click in the space to the right of ToolTips and start typing your message without quotation marks.

You will need to run the program (F5 shortcut), and hold your mouse pointer over the object in order for your ToolTip message to appear. Always check your ToolTips in the Run mode just to make certain they are functioning the way you want them to. Remember, set this Property, preferably at Design time.

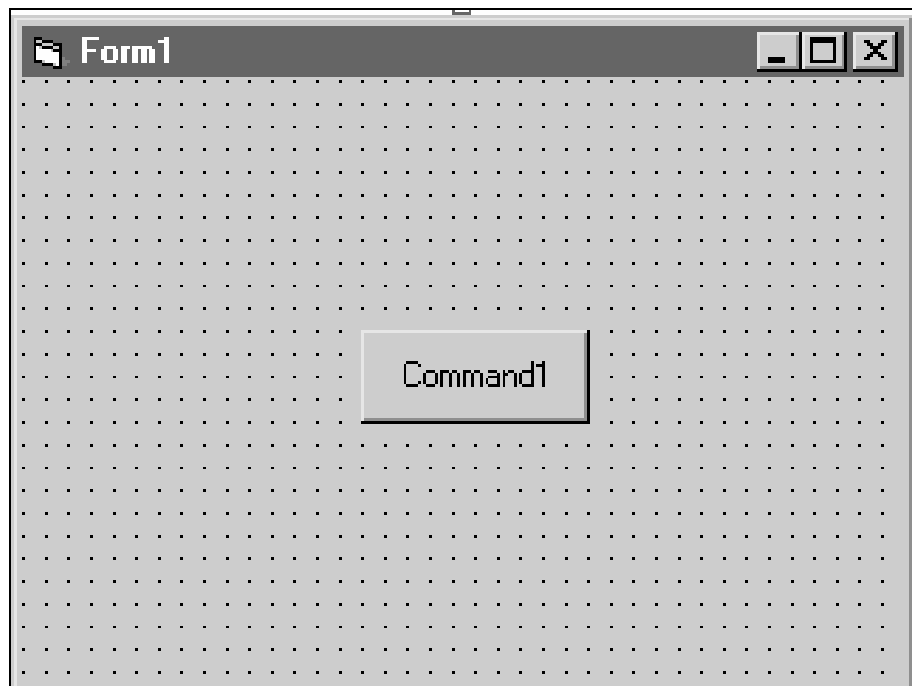


Figure 6.14: The Command Button at Design-time

6.6.5 Check Box

This is typically used for turning on/off some particular feature of your program. The Check Box control lets the user to select or unselect an option. When the Check Box is checked, its value is set to 1 and when it is unchecked, the value is set to 0. You can include the statements `Check1.Value=1` to mark the Check Box and `Check1.Value=0` unmark the Check Box, and use them to initiate certain actions.

For example, the program will change the background color of the form to red when the check box is unchecked and it will change to blue when the check box is checked. You will learn about the conditional statement `If....Then....Elesif` in later lesson. `VbRed` and `vbBlue` are color constants and `BackColor` is the background color property of the form.

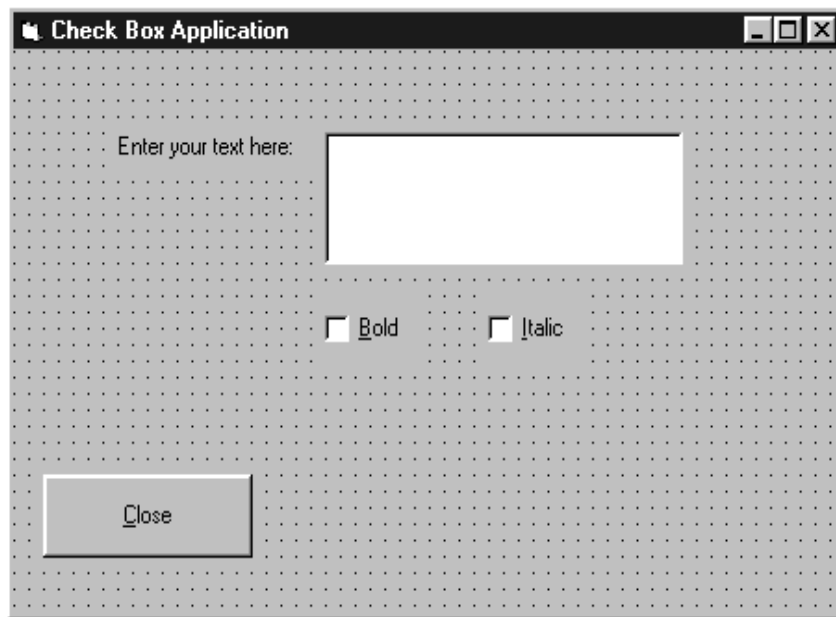


Figure 6.15: Check Box

6.6.6 Option Button

It is used in groups. VB.Net handles the feature that only 1 option button can be selected at a time. The Option Buttons control also lets the user selects one of the choices. However, two or more Option Buttons must work together because as one of the Option Button is selected, the other Option Buttons will be unselected. In fact, only one Option Button can be selected at one time. When an Option Button is selected, its value is set to “True” and when it is unselected; its value is set to “False”.

In the following example, the shape control is placed in the form together with six Option Buttons. When the user clicks on different Option Buttons, different shapes will appear. The values of the shape control are 0, 1, and 2,3,4,5 which will make it appear as a rectangle, a square, an oval shape, a rounded rectangle and a rounded square respectively.

Example

```
Private Sub Option1_Click ( )  
Shape1.Shape = 0  
End Sub
```

```
Private Sub Option2_Click()  
Shape1.Shape = 1  
End Sub  
  
Private Sub Option3_Click()  
Shape1.Shape = 2  
End Sub  
  
Private Sub Option4_Click()  
Shape1.Shape = 3  
End Sub  
  
Private Sub Option5_Click()  
Shape1.Shape = 4  
End Sub  
  
Private Sub Option6_Click()  
Shape1.Shape = 5  
End Sub
```

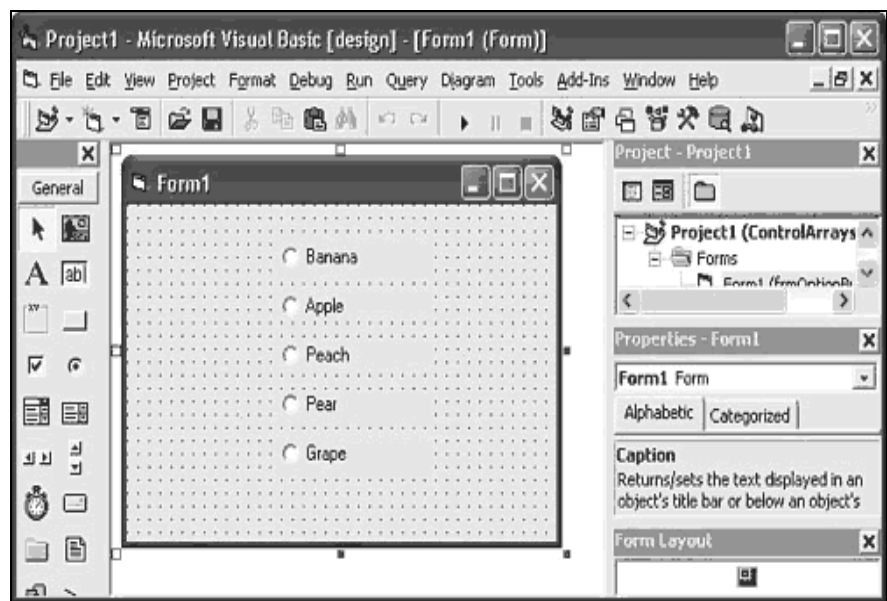


Figure 6.16: Option Button

Check Your Progress 3

Fill in the blanks:

1. A ____ control is a box which is used to group other controls together.
2. _____ control is used to display some static (i.e. user cannot change it) text on the screen.
3. An _____ procedure is something that happens in response to a user action such as a click or a "mouse over".

6.6.7 List Box

List boxes are controls that give the users choices from which they can select. Your application is responsible for initializing list boxes with values. The user cannot add items to a list box. This control is used to display a list. If the List is long and doesn't fit in the ListBox then scroll bars appears automatically.

The function of the List Box is to present a list of items where the user can click and select the items from the list. In order to add items to the list, we can use the AddItem method. For example, if you wish to add a number of items to list box 1, you can key in the following statements

Example

```
Private Sub Form_Load ( )  
List1.AddItem "Lesson1"  
List1.AddItem "Lesson2"  
List1.AddItem "Lesson3"  
List1.AddItem "Lesson4"  
End Sub
```

The items in the list box can be identified by the ListIndex property, the value of the ListIndex for the first item is 0, the second item has a ListIndex 1, and the second item has a ListIndex 2 and so on

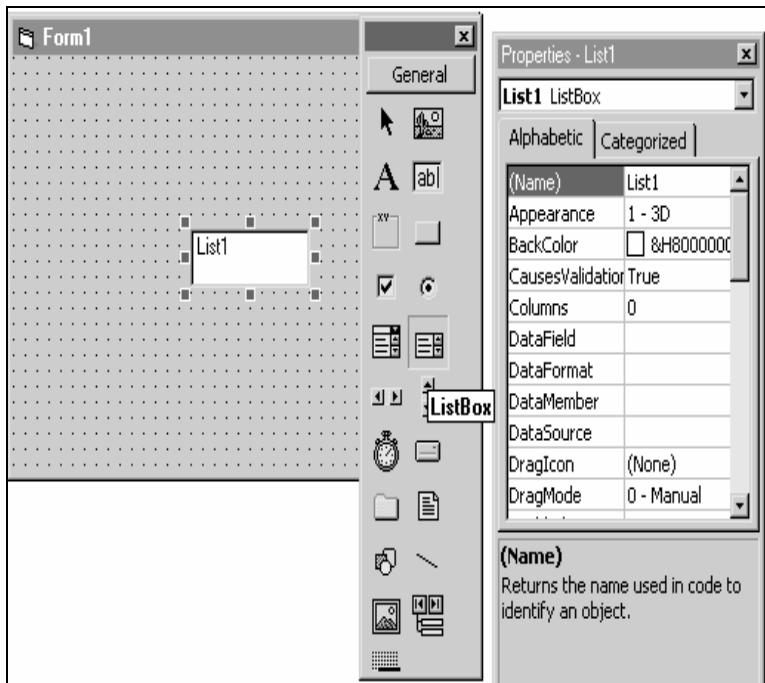


Figure 6.17: List Box

6.6.8 Combo Controls

Combo boxes are so-named because they "combine" the features found in both text boxes and list boxes. Combo boxes are also commonly referred to as "drop-down boxes" or "drop-down lists".

There are three combo box styles:

0 Drop Down Combo

1 Simple Combo

2 Drop Down List

At design-time, you set the style of the combo box with its Style property.

Of the three combo box styles, the one most similar to a ListBox is "2 - Drop-Down List". In fact, everything discussed in the last several pages regarding ListBoxes applies to drop-down style combo boxes EXCEPT the following:

The combo box displays the selected item in the text box portion of the combo box. The ListBox portion of the combo box remains hidden until the combo box receives focus and the user clicks the down arrow on the text box.

You can change the width of a drop-down combo box, but not its height.

Like a ListBox, and unlike the other combo box styles, the user can only select a value that is in the list they cannot type in a new value. However, if the user does press a keyboard key, the closest item in the list alphabetically will be selected (i.e., if the user types the letter "B", and the word "Banana" is in the list, then "Banana" will be selected).

Multiple selections cannot be made. Only one item may be selected from the list.

Unlike a ListBox, VB.Net does not automatically pre-select the first value from a combo box. Therefore, the initial value of the ListIndex property is 1. If you do not pre-select a value in code (say in the Form_Load procedure), then be sure to check the ListIndex property for a value of 1 when you check to see which item the user selected.

The "0 Drop Down Combo" style of combo box operates the same way as the Drop Down List, except that the user may type a new value in the text box portion of the combo box. Please note that if the user types a non-list value, this value is NOT automatically added to the list, and the value of ListIndex would be 1. You should check the Text property to access the user's selection or entry.

The "1 Simple Combo" style of combo box operates in a manner similar to the Drop Down Combo in that the user can either select an item from the list or type a new entry. The difference is that the list does not drop down like a ListBox, you determine the height of the list portion at design-time. This is the only combo-box style that allows you to adjust the height of the list portion.

This control is used to display a list in compact form. This control consumes less space and still can have all the elements of list in it.

The function of the Combo Box is also to present a list of items where the user can click and select the items from the list. However, the user needs to click on the small arrowhead on the right of the combo box to see the items which are presented in a drop-down list. In order to add items to the list, you can also use the AddItem method. For example, if you wish to add a number of items to Combo box 1, you can key in the following statements

Example

```
Private Sub Form_Load ( )  
    Combol.AddItem "Item1"  
    Combol.AddItem "Item2"
```

```

Combo1.AddItem "Item3"
Combo1.AddItem "Item4"
End Sub

```

The combo box control actually turns into two different kinds of controls on the form, depending on how you set up the combo box. The three kinds of combo boxes are

- **Dropdown combo boxes:** The dropdown combo box control offers a handy way for the user to view and add items to a list. The dropdown combo box gives you the advantage of listing items for the user without taking up screen space that is needed for other things. Your user can display the entire list by clicking the down arrow.
- **Simple combo boxes:** The simple combo box control does exactly the same thing as the dropdown combo box except that the simple combo box is always displayed in its dropdown form. In other words, if screen space is not a problem, you might want to use a simple combo box to display and collect values by using a list that does not require the user's extra keypress to open the list.
- **Drop-down list box:** It doesn't let the user enter new items, so it's similar to a list box. Unlike a list box, however, the drop-down list box normally appears closed to a single line until the user clicks the down arrow button to open the list box to its full size. Technically, drop-down list boxes are not combo box controls but work more like list boxes. The reason drop-down list boxes fall in the combo box control family is that you place drop-down list boxes on forms by clicking the combo box control and setting the Style combo box property.

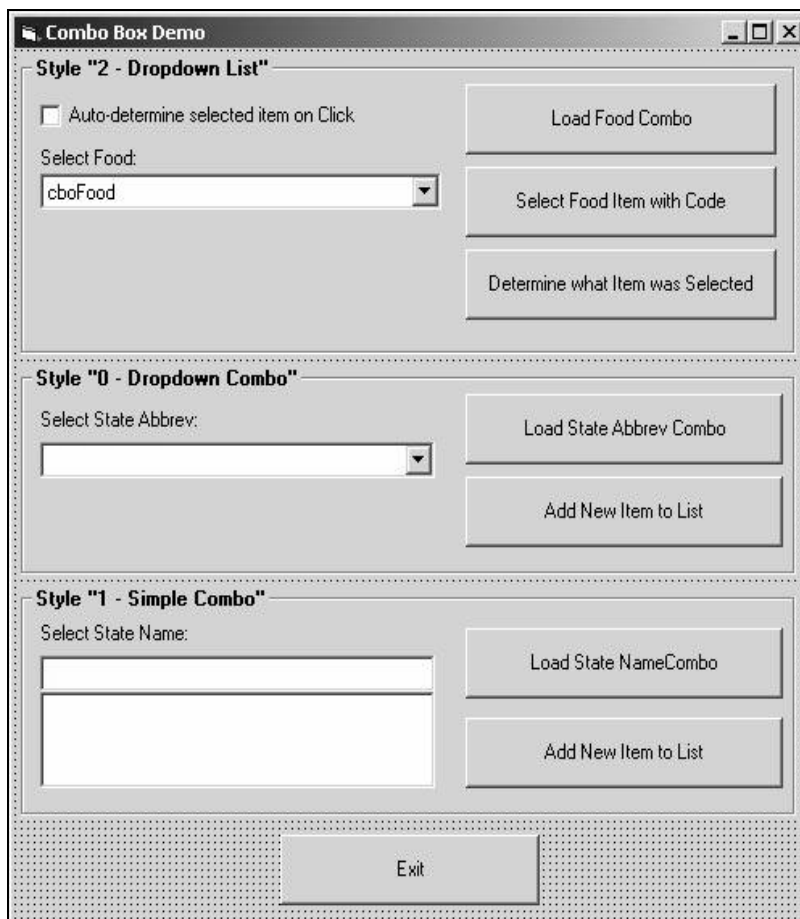


Figure 6.18: Combo Box

6.6.9 Picture Controls

The Picture Box is one of the controls that used to handle graphics. You can load a picture at design phase by clicking on the picture item in the properties window and select the picture from the selected folder. You can also load the picture at runtime using the LoadPicture method. For example, the statement will load the picture grape.gif into the picture box.

```
Picture1.Picture=LoadPicture ("C:\VB program\Images\grape.gif")
```

The image in the picture box is not resizable. It displays pictures. It also acts as an area on which you can print text and graphics. Use it for home-grown graphics or print previews.

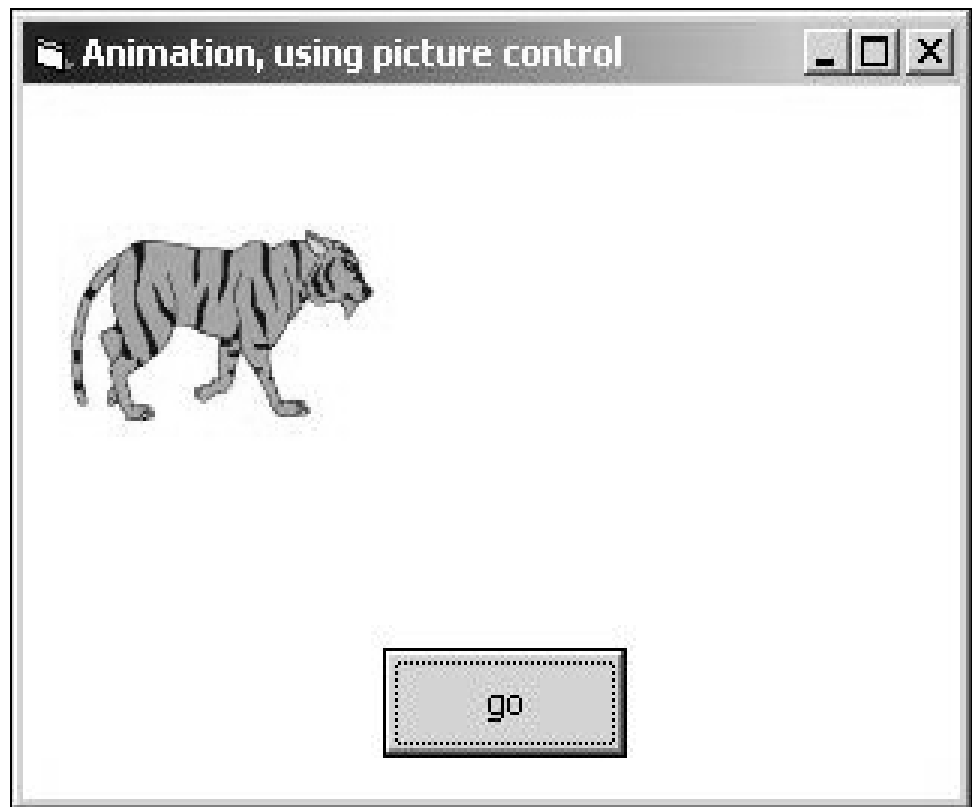


Figure 6.19: Picture Control

6.6.10 Image Controls

The Image Box is another control that handles images and pictures. Use this to display a picture. Use it over the PictureBox because it takes less operating system resources. It functions almost identically to the picture box. However, there is one major difference, the image in an Image Box is stretchable, which means it can be resized. This feature is not available in the Picture Box.

Similar to the Picture Box, it can also use the LoadPicture method to load the picture. For example, the statement loads the picture grape.gif into the image box.

```
Image1.Picture=LoadPicture ("C:\VB program\Images\grape.gif")
```

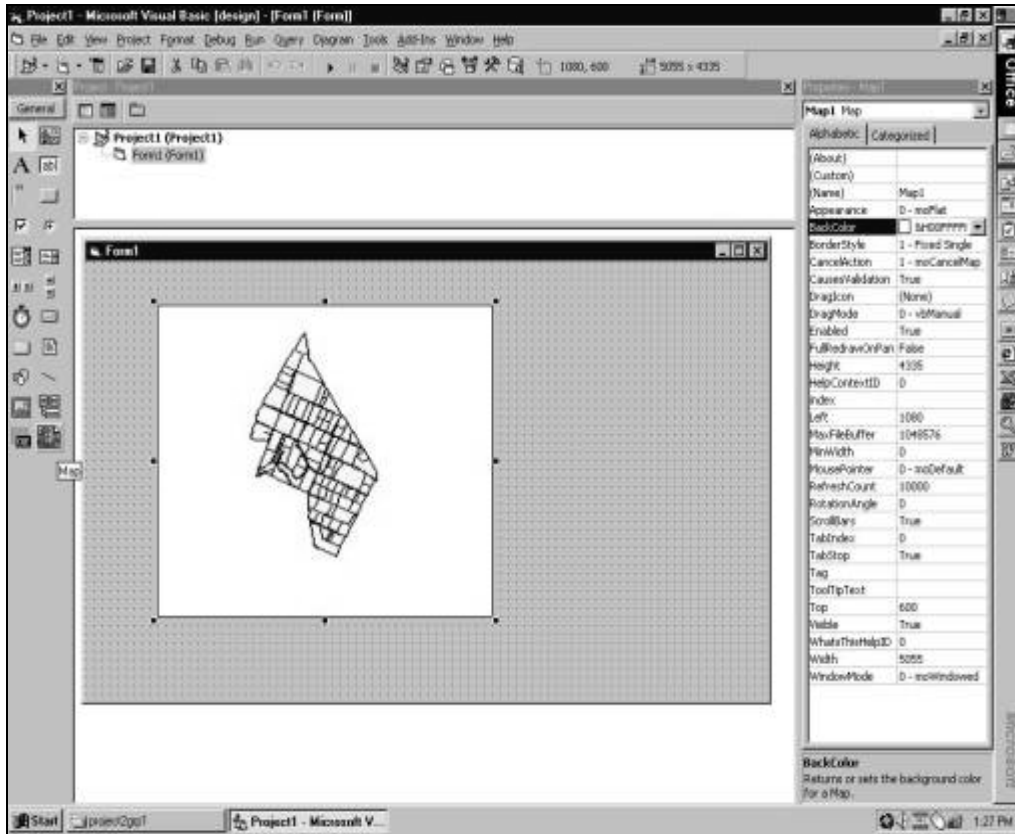



Figure 6.20: Image Control

Check Your Progress 4

Fill in the blanks:

1. _____ function formats numeric data for presentation as dollars and cents.
2. _____ Returns the lower-case conversion of a string.
3. Visual Basic allows us to work with controls in two ways: _____ and _____.

6.7 LET US SUM UP

A message box is a special dialog box used to display a piece of information to the user. As opposed to a regular form, the user cannot type anything on the dialog box. Function is a method which returns a value. A User-Defined Function, or UDF, is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment. Control is an object that can be drawn on to the Form to enable or enhance user interaction with the application. Values of one data type can be converted to another data type. This operation is called casting from one type to another.

6.8 LESSON END ACTIVITIES

1. Start a new project. Add a textbox, a Label and a button to your new Form. Then write a programme that does the following:
 - a) Asks users to enter a number between 10 and 20.
 - b) The number will be entered into the Textbox.
 - c) When the Button is clicked, your Visual Basic code will check the number entered in the Textbox.
 - d) If it is between 10 and 20, then a message will be displayed.
 - e) The message box will display the number from the Textbox.
 - f) If the number entered is not between 10 and 20 then the user will be invited to try again, and whatever was entered in the Textbox will be erased.
2. Add a Combo box and another button to your form. Create a list of items for your Combo Box. The list of items in your Combo box can be anything you like - pop groups, football teams, favourite foods, anything of your choice. Then try the following:
 - a) Use a select case statement to test what a user has chosen from your drop-down list. Give the user a suitable message when the button was clicked.
 - b) Put two textboxes on your form. The first box asks users to enter a start position for a For Loop; the second textbox asks user to enter an end position for the For loop. When a button is clicked, the programme will add up the numbers between the start position and the end position. Display the answer in a message box. Get the startNumber and endNumber from the textboxes.
 - c) Amend your code to check that the user has entered numbers in the textboxes. You will need an If statement to do this. If there's nothing in the textboxes, you can halt the programme.

6.9 KEYWORDS

Statement: It is a complete instruction.

Declaration Statement: It is a statement that can create a variable, constant, data type.

Executable Statement: It is a statement that performs an action.

Imports Statement: It is a statement that is used to import namespaces.

Message Box: It is a special dialog box used to display a piece of information to the user.

Input Box: It is a specially designed dialog box that allows the programmer to request a value from the user and use that value as necessary.

Function: It is a method which returns a value.

User-Defined Function: It is a function provided by the user of a program or environment, in a context where the usual assumption is that functions are built into the program or environment.

Control: It is an object that can be drawn on to the Form to enable or enhance user interaction with the application.

Casting: The operation which converts the values of one data type to another data type is called casting.

6.10 QUESTIONS FOR DISCUSSION

1. What is imports statement? Give an example.
2. What is msgbox? What are the buttons available with it?
3. How to create an input box?
4. Compare and contrast between built-in function and user-define function.

Check Your Progress: Model Answers

CYP 1

1. declaration statement, executable statement.
2. namespaces

CYP 2

1. now
2. DatePart()
3. Hour()

CYP 3

1. frame
2. Label
3. event

CYP 4

1. FormatCurrency()
2. LCase()
3. Design time, runtime

6.11 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*, Pearson edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic.Net*, 5th Edition, PHI

Kant, *Visual Basic.Net—A Beginners Guide*, TMCH

UNIT III

LESSON

7

ARRAY

CONTENTS

- 7.0 Aims and Objectives
 - 7.1 Introduction
 - 7.2 Array
 - 7.3 Menus and Dialog Boxes
 - 7.3.1 Dialog Boxes
 - 7.4 Let us Sum Up
 - 7.5 Lesson End Activities
 - 7.6 Keywords
 - 7.7 Questions for Discussion
 - 7.8 Suggested Readings
-

7.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- The use of array and collection
 - Brief description of Menus
 - Concept of Dialog Boxes
-

7.1 INTRODUCTION

Variables are the basic storage units for holding data values during processing. A problem arises, however, with large collections of data values for which it is impractical to declare a separate variable for each of them. Consider a need to store the names or codes for all the states. It would require declaring and assigning values to 50 different variables. Fortunately, though, there is a way to handle these large sets of similar data items through use of arrays and collections.

In this lesson overview of array will be discussed.

7.2 ARRAY

Arrays are programming constructs that store data and allow us to access them by numeric index or subscript. Array helps us create shorter and simpler code in many situations. Arrays in Visual Basic .NET inherit from the Array class in the System namespace. All arrays in VB are zero based, meaning, the index of the first element is zero and they are numbered sequentially. You must specify the number of array elements by indicating the upper bound of the array. The upper bound is the number that specifies the index of the last element of the array. Arrays are declared using Dim, ReDim, Static, Private, Public and Protected keywords. An array can have one

dimension (linear arrays) or more than one (multidimensional arrays). The dimensionality of an array refers to the number of subscripts used to identify an individual element. In Visual Basic we can specify up to 32 dimensions. Arrays do not have fixed size in Visual Basic.

Using Arrays

An array is a collection of data values, all of which are accessible through the same variable name. An array can be thought of as a "table" with data values occupying its "cells." For instance, you can visualize an array named Letters containing the first five letters of the Greek alphabet as in the following table.

Letters (array)	Index
alpha	(0)
Beta	(1)
gamma	(2)
delta	(3)
epsilon	(4)

A major advantage of using an array to store a collection of values is that all values can be referenced through the single array name; it is not necessary to individually declare and name separate variables for all the items. Rather, the items in an array are indexed by their position in the array, numbered beginning with 0 (zero) for the first element in the array. Thus, the value "gamma" in the above array is identified by the reference Letters(2). The name of the array is followed by an index value enclosed in parentheses.

Multidimensional Arrays

Arrays can have more than one dimension to hold pairs, or even triplets, of values. The following array has two dimensions, the first "column" of which contains the Greek letters, the second of which contains equivalent English letters.

Letters		Index
(0)	(1)	
Alpha	A	(0)
Beta	B	(1)
gamma	C	(2)
Delta	D	(3)
Epsilon	E	(4)

Now, a particular data value is referenced by a pair of indexes representing its row and column position, in that order, with both indexes beginning with 0. The value "gamma" is referenced by Letters(2,0); the value "e" is referenced by Letters(4,1). The row and column indexes are separated by a comma.

Declaring Arrays

Similar to declaring variables, an array is declared with a Dim statement. The array name is followed by a pair of parentheses to indicate an array variable. For a single dimension array (one column) the parentheses are empty; for a multidimensional array commas are used for each additional column. An array is typed in the same way as variables to declare the type of data to occupy the array. The following statements

declare a single- and a two-dimensional array, respectively, each of which contain string data.

```
Dim Letters() As String
```

```
Dim Letters(,) As String
```

When arrays are declared, they do not have physical sizes. They have dimensions, but the number of rows and columns is unknown. If you know in advance the number of values that will occupy an array, you can include the size as part of its declaration. The two previous Letters arrays can be declared as

```
Dim Letters(4) As String
```

```
Dim Letters(4,1) As String
```

to indicate an array with 5 rows, and an array with 5 rows and 2 columns, respectively. Note that, because array indexing begins with 0, the dimensions of an array are always 1 less than the actual number of array elements. You can determine the actual size of an array through the `arrayName.Length` property, which returns the total number of elements in the array.

Initializing Arrays

There are numerous ways to assign values to the elements of an array. If the values are known, and they are modest in size, you can populate an array during its declaration. Values are assigned by enclosing them within braces, separated by commas. The following statement is one way to initially load the single-dimension Letters array. (A dimension size is not required, being determined from the number of values.)

```
Dim Letters() As String = {"alpha","beta","gamma","delta","epsilon"}
```

For two-dimensional arrays, pairs of values are loaded by enclosing them inside braces, which are enclosed inside a pair of braces representing the array as a whole.

```
Dim Letters(,) As String = {  
{"alpha","a"}, {"beta","b"}, {"gamma","c"}, {"delta","d"}, {"epsilon","e"} }
```

A second method of loading an array of known dimensions is inside a procedure. The following `Page_Load` subprogram dimensions an array and loads values with simple assignment statements. In this case the size of the array is required since it must have a physical size before assigning values to its elements.

```
Sub Page_Load  
    Dim Letters(4) As String  
    Letters(0) = "alpha"  
    Letters(1) = "beta"  
    Letters(2) = "gamma"  
    Letters(3) = "delta"  
    Letters(4) = "epsilon"  
End Sub
```

In a similar way a two-dimensional array can be loaded.

```
Sub Page_Load  
    Dim Letters(4,1) As String  
    Letters(0,0) = "alpha"
```



```
Letters(1,0) = "beta"  
Letters(2,0) = "gamma"  
Letters(3,0) = "delta"  
Letters(4,0) = "epsilon"  
Letters(0,1) = "a"  
Letters(1,1) = "b"  
Letters(2,1) = "c"  
Letters(3,1) = "d"  
Letters(4,1) = "e"  
  
End Sub
```

Arrays of Unknown Dimensions

It is not unusual that you might not know in advance the number of values to be loaded into an array. Often times these values are loaded from external files with changing sizes. One solution is to dimension the array to some assumed maximum size and don't worry about it being too large. Another solution is to redimension it to increasing sizes as needed. This topic is taken up later in the discussion about accessing arrays.

Using Collections

Similar to arrays for storing sets of data items are Visual Basic collections. A collection is a one-dimensional storage area in which values of mixed data types can be placed, although there are few occasions for doing so. Collections are increased in size simply by adding items to it.

A collection is declared with a Dim statement that creates a new Collection object. The following statement declares a collection for storing the state codes.

Dim States As New Collection

Then, values are added to the collection with the collection.Add() method.

```
States.Add("AL")  
States.Add("AK")  
States.Add("AZ")  
States.Add("AR")  
States.Add("CA")  
...
```

Values are retrieved from a collection by reference to a collection.Item(index). Unlike arrays, items in a collection are indexed beginning with 1. Therefore, a reference to States.Item(5) in the above collection produces the value "CA". (The keyword .Item is optional when referring to a collection element; States(5) or States("CA") will work.)

One of the main advantages of collections is that items can be indexed by values other than their physical locations in the collection. Each item in a collection can have both a value and a key, where the key is used as the index to an associated value. Values and keys are established when the collection is loaded by using the Add() method in the following format.

```
collection.Add(value, key)
```

For example, to load the States collection with state codes and to establish state names as the keys to them, the following code is used.

```
Dim States As New Collection
States.Add("AL", "Alabama")
States.Add("AK", "Alaska")
States.Add("AZ", "Arizona")
States.Add("AR", "Arkansas")
States.Add("CA", "California")
...
```

Now, to retrieve the state code associated with the state name "California" the following reference is made,

```
States.Item("California")
```

and the value "CA" is retrieved from the collection. (The values and keys can be reversed to look up a state name from a state code.)

Although collections permit only single-dimension "arrays" of values, they can be easier to work with than arrays for storing modest collections of data values.

Example 1

Inserting items into a collection by index

```
Public Class Tester
    Public Shared Sub Main
        Dim wordCollection As New Collection

        wordCollection.Add("This")
        wordCollection.Add("is")
        wordCollection.Add("a")
        wordCollection.Add("collection")
        wordCollection.Add("of")
        wordCollection.Add("words")
        ' ----- Insert a word after item 3.
        wordCollection.Add("slightly", , , 3)

        ' ----- Insert a word before item 5.
        wordCollection.Add("longer", , 5)

        For Each word As String In wordCollection
            Console.WriteLine(word)
        Next word
    End Sub
End Class
```

Example 2

Add key value pair to Collection

```
Option Strict On
Imports System.Collections
Public Class Collect
    Public Shared Sub Main()
        Dim sta As New Collection

        sta.Add("New York", "NY")
        sta.Add("Michigan", "MI")
        sta.Add("New Jersey", "NJ")
        sta.Add("Massachusetts", "MA")

        For Each stcode As String In sta
            Console.WriteLine(stcode)
        Next
    End Sub
End Class
```

Example 3

Store objects in Collection and retrieve by key and index

```
Imports System
Public Class Employee
    Private myEmpID As Integer

    Public Sub New(ByVal empID As Integer)
        Me.myEmpID = empID
    End Sub 'New

    Public Overrides Function ToString( ) As String
        Return myEmpID.ToString( )
    End Function 'ToString

    Public Property EmpID( ) As Integer
        Get
            Return myEmpID
        End Get
        Set(ByVal Value As Integer)
            myEmpID = Value
        End Set
    End Property
End Class
```

```
End Set
End Property
End Class 'Employee
```

```
Class Tester
```

```
Shared Sub Main( )
```

```
    Dim intCollection As New Collection( )
```

```
    Dim empCollection As New Collection( )
```

```
    Dim empCollection2 As New Collection( )
```

```
    Dim i As Integer
```

```
    For i = 0 To 4
```

```
        empCollection.Add(New Employee(i + 100))
```

```
        intCollection.Add((i * 5))
```

```
    Next i
```

```
    empCollection2.Add(New Employee(1), "G")
```

```
    empCollection2.Add(New Employee(2), "J")
```

```
    empCollection2.Add(New Employee(3), "T")
```

```
    For Each i In intCollection
```

```
        Console.WriteLine("{0} ", i.ToString( ))
```

```
    Next i
```

```
    Console.WriteLine( )
```

```
    Console.WriteLine("Employee collection...")
```

```
    Dim e As Employee
```

```
    For Each e In empCollection
```

```
        Console.WriteLine("{0} ", e.ToString( ))
```

```
    Next e
```

```
    Console.WriteLine( )
```

```
    Console.WriteLine("Employee collection 2...")
```

```
    For Each e In empCollection2
```

```
        Console.WriteLine("{0} ", e.ToString( ))
```

```
    Next e
```

```
    Console.WriteLine( )
```

```
    Dim emp As Employee
```

```
    emp = empCollection2.Item("J")
```

```
    Console.WriteLine(emp.ToString( ))
```

```
emp = empCollection2.Item(1)

Console.WriteLine("Index(1)      retrieved      empID      {0}",
emp.ToString( ))

End Sub

End Class
```

Check Your Progress 1

True or False:

1. Arrays always have only one dimension.
2. A collection is declared with a Dim statement.

7.3 MENUS AND DIALOG BOXES

Everyone should be familiar with Menus. Menus (File, Edit, Format etc in all windows applications) are those that allow us to make a selection when we want to perform some action with the application, for example, to format the text, open a new file, print and so on.

In VB .NET MainMenu is the container for the Menu structure of the form. Menus are made of MenuItem objects that represent individual parts of a menu (like File->New, Open, Save, Save As etc). The two main classes involved in menu handling are, MainMenu and MenuItem.

The MainMenu class let's us assign objects to a form's menu class and MenuItem is the class which supports the items in a menu system. Menus like File, Edit, Format etc and the items in those Menus are supported by this MenuItem class. It's this MenuItem's click event that makes these Menus work. For a MenuItem to be displayed, we need to add it to a MainMenu object.

Event of the MenuItem

The default event of the MenuItem is the Click event which looks like this in code:

```
Private Sub MenuItem1_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles MenuItem1.Click

End Sub
```

Notable properties of the MenuItem class are summarized below.

Under the Miscellaneous Section of the properties window:

Checked: Default value is set to False. Changing it to True makes a checkmark appear towards the left of the Menu.

DefaultItem: Default value is set to False. Changing it to True makes this menu item default menu item.

RadioCheck: Changing it to True makes a menu item display a radio button instead of a checkmark.

Shortcut: Enables to set a short cut key from a list of available shortcuts for the menu item.

Working with Menus

Creating Menus is simple. Drag a MainMenu component from the toolbar onto the form. When you add a MainMenu component to the form it appears in the component tray below the form. Windows form designer will add the MenuItem's for this by default, you need not add this. Once when you finish adding a MainMenu component to the form you will notice a "TypeHere" box towards the top-left corner of the form. To create a menu all you have to do is click on the "TypeHere" text which opens up a small textbox allowing you to enter text for the menu. You can view that in the image below. You can use the arrow keys on the keyboard to create a submenu or add other items to that menu or click on the first menu item and use the left/right arrow keys on the keyboard to create a new menu item. That's all it takes to add a menu to the form.

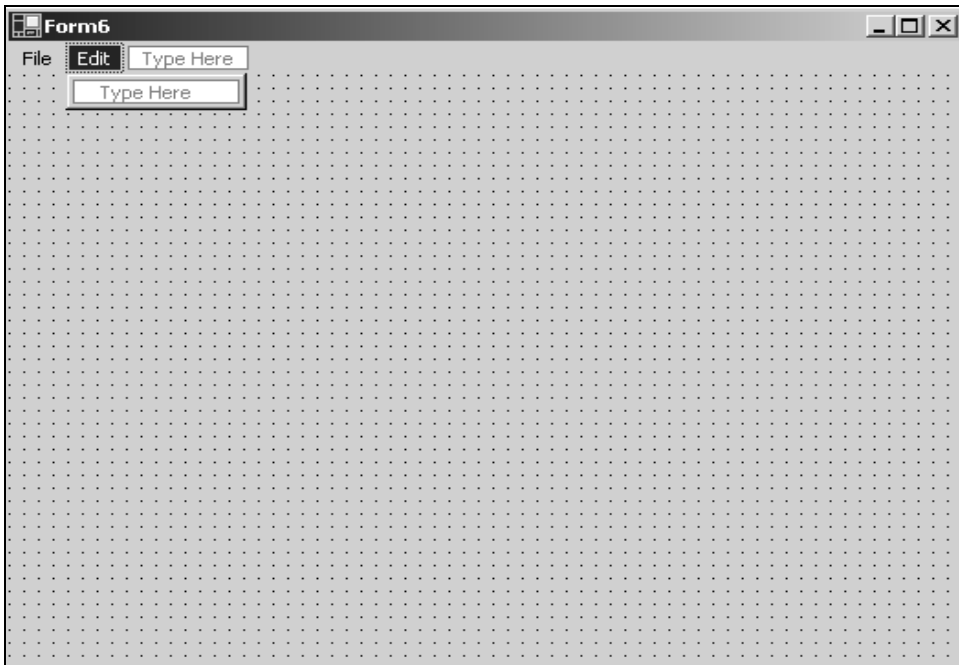


Figure 7.1: A form with a menu

Working with an example

Let's work with an example to understand Menus. Drag a MainMenu and a TextBox onto the form. In the "Type Here" part, type File and under file type "New" and "Exit". Our intention here is to display "Welcome to Menus" in the TextBox when "New" is clicked and close the form when "Exit" is clicked. The Menu which we will create should look like this File->New, Exit (New and Exit below File). The code for that looks like this:

```
Public Class Form3 Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "
Private Sub MenuItem2_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs)_ Handles MenuItem2.Click
TextBox1.Text = "Welcome to Menus"
End Sub
```

```
Private Sub MenuItem3_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles MenuItem3.Click  
  
Me.Close()  
  
'Me refers to the current object (form)  
  
End Sub  
  
End Class
```

Seperating Menu Items

We can separate menu items with a seperator. A separator is a horizontal line between items on a menu. We can use separator bars to divide menu items into groups on menus that contain multiple items. You add a separator to menus by entering a hyphen (-) as the text of a menu item. It will be displayed as a separator. The image below displays a separator bewteen the Close and Exit menu items.

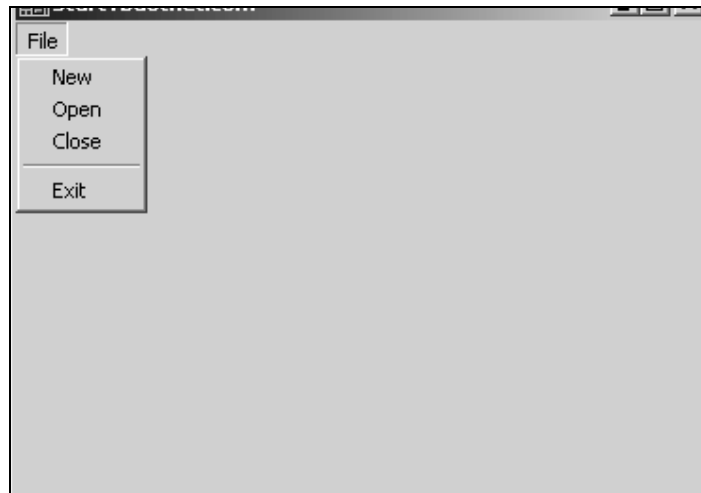


Figure 7.2: A screen with menu

Cloning Menus

Cloning menus is making a copy of existing menu items. For example, we can clone the File menu item displayed in the image below to serve as a context menu for a control. To clone a menu we should use the CloneMenu method. The CloneMenu method creates a copy of the specified menu and all of it's members. This includes their properties and event handlers. Thus, all the events that are handled by the menu item will be handled by the cloned menu. The cloned menu can then be assigned to a control.

Shortcut Keys

You can assign shortcut keys to enable instant access to menu commands. To assign a shortcut to the menu item, in the properties window select the shortcut property and choose the appropriate shortcut key combination from the drop-down menu.

Context Menus

Context menus are menus that appear when an item is right-clicked. In any windows application when you right-click your mouse you get a menu which might display some shortcuts from the Edit Menu, for example, cut, copy, paste, paste special and so on. All these menu items which are available when you right-click are called Context Menus. In Visual Basic we create context menus with the ContextMenu component.

The ContextMenu component is edited exactly the same way the MainMenu component is edited. The ContextMenu appears at the top of the form and you can add menu items by typing them. To associate a ContextMenu with a particular form or control we need to set the ContextMenu property of that form or control to the appropriate menu.

7.3.1 Dialog Boxes

Most Windows applications request for user input. Dialog boxes are one means of requesting users for specific kinds of inputs. Therefore, VB.NET allows its designers to create a number of different types of dialog boxes. Standard Dialog boxes are included in classes that fall within the purview of the CommonDialog.

- OpenFileDialog
- ColorDialog
- FontDialog
- PageSetupDialog
- PrintDialog

Let us now briefly study the features of the CommonDialog boxes

OpenFileDialogClass

Open File Dialog's are supported by the OpenFileDialog class and they allow us to select a file to be opened. Below is the image of an OpenFileDialog.

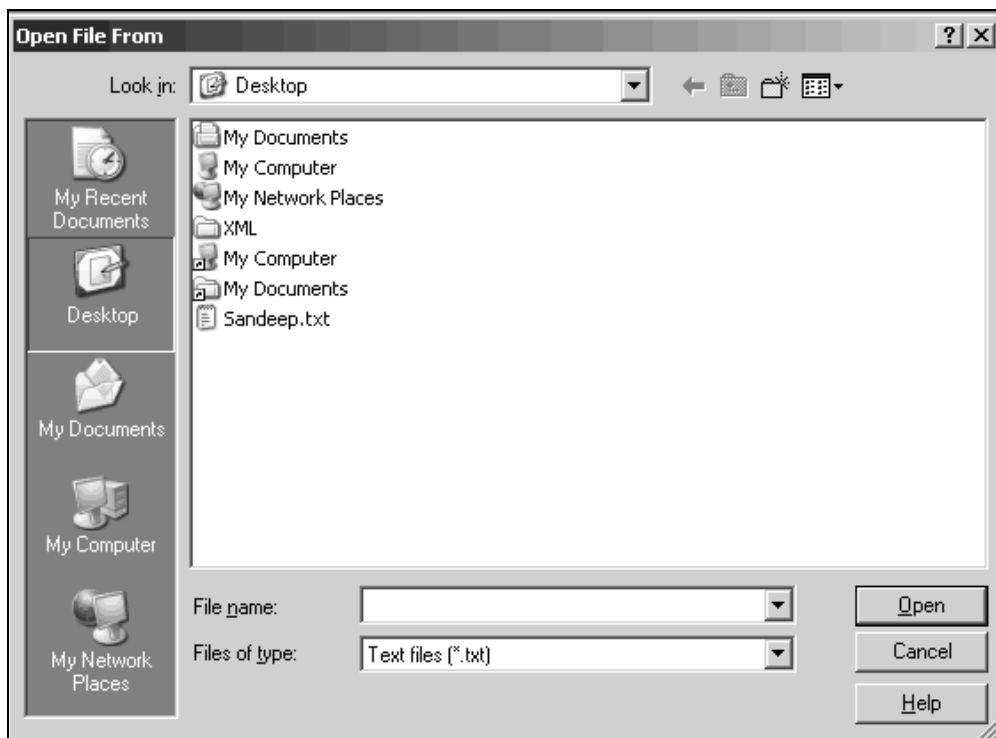


Figure 7.3: Open file Dialog Box

This class provides users with the file selection capability. The properties and methods of this dialog boxes are given below:

Property or Method	Description
ShowDialog	Displays the dialog
MultiSelect	Sets/unsets the selection of <u>multiple files</u>
ShowReadOnly	Sets/unsets the read-only check box checked
Filter	Sets the type of files that will appear in the dialog box
FilterIndex	Sets the index of the filter selected in the dialog box

SaveFileDialog Class

Save File Dialog's are supported by the SaveFileDialog class and they allow us to save the file in a specified location. Below is the image of a SaveFileDialog.

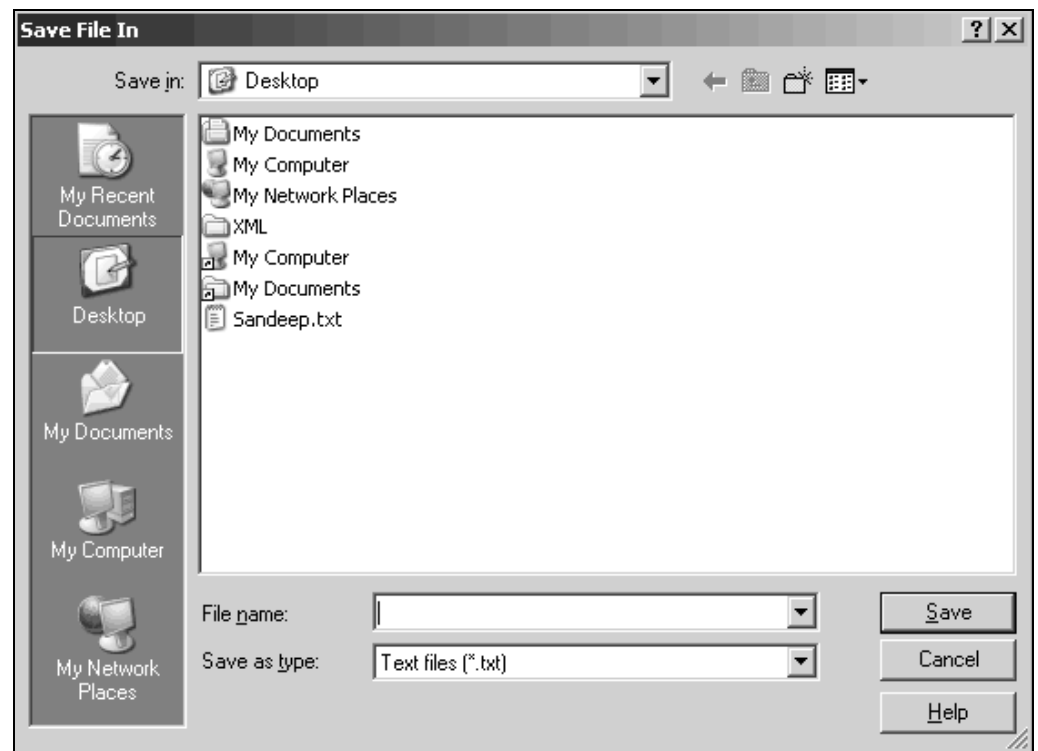


Figure 7.4: Save file Dialog Box

The SaveFileDialog class offers you the standard window that we see while saving the file. The methods and properties of this dialog box are given below:

Property or Method	Description
ShowDialog	Displays the message
CheckFileExists	Checks for the existence of file specified
FileName	Determines the file name selected by the user
Filter	Condition for files to be shown in the dialog box
FilterIndex	Determine the index of the filter selected in the dialog box

The ColorDialog Class

Color Dialog's are supported by the ColorDialog Class and they allow us to select a color. The image below displays a color dialog.

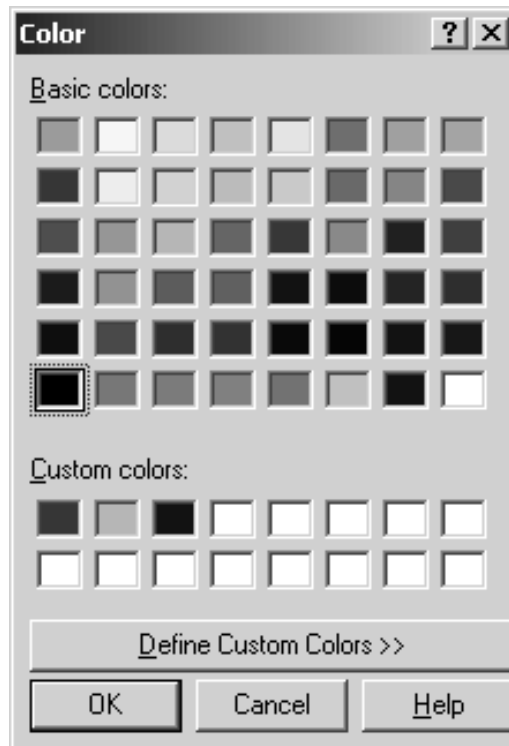


Figure 7.5: Color Dialog Box

This dialog box shows the color palette for allowing user to select a color and add that color to the palette. The properties of the Class are given below:

Property or Method	Description
ShowDialog	Displays the dialog box
Color	Determines the color selected by the user
AllowFullOpen	Specifies if the user can add <u>custom colors</u> to the box
SolidColorOnly	Determines if the user can use dithered colors

Check Your Progress 2

Fill in the blanks:

1. An array is a collection of _____ all of which are accessible through the same variable name.
2. Arrays in Visual Basic .NET inherit from the _____ class
3. An array is declared with a _____ statement.
4. Items in a collection are indexed beginning with _____.

7.4 LET US SUM UP

Arrays are programming constructs that store data and allow us to access them by numeric index or subscript. The upper bound is the number that specifies the index of the last element of the array. An array can have one dimension (linear arrays) or more

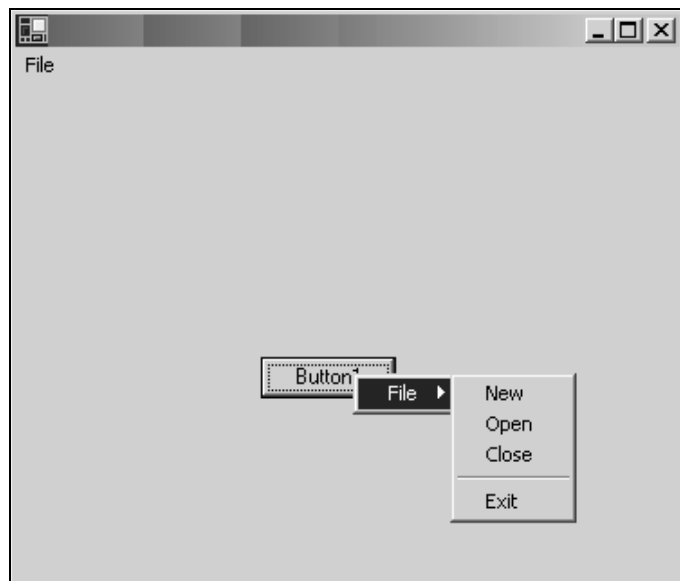
than one (multidimensional arrays). The dimensionality of an array refers to the number of subscripts used to identify an individual element. A collection is a one-dimensional storage area in which values of mixed data types can be placed. VB.NET allows its designers to create a number of different types of dialog boxes. Standard Dialog boxes are included in classes that fall within the purview of the CommonDialog.

7.5 LESSON END ACTIVITIES

1. Create a dialog box which is as shown in the following figure.



2. Create a form which is as shown in the following figure.



7.6 KEYWORDS

Array: It is a programming constructs that store data and allow us to access them by numeric index or subscript.

Upper bound: The upper bound is the number that specifies the index of the last element of the array.

Dimensionality of an array: The dimensionality of an array refers to the number of subscripts used to identify an individual element.

Collection: It is a one-dimensional storage area in which values of mixed data types can be placed.

7.7 QUESTIONS FOR DISCUSSION

1. Compare and contrast between array and collection.
2. How to create Arrays of Unknown Dimensions?
3. Write short notes on
 - a) OpenFileDialog
 - b) Context menu
 - c) shortcut keys

Check Your Progress: Model Answers

CYP 1

1. False
2. True

CYP 2

1. Data values
2. Array
3. Dim
4. 1

7.8 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net* Pearson edition, 2004

MSDN Visual studio Library.

Schneider, "An introduction to programming using Visual Basic .Net, 5th Edition , PHI

Kant, *Visual Basic.Net A Beginners Guide*, TMCH

LESSON

8

VB.NET - PROGRAMMING

CONTENTS

- 8.0 Aims and Objectives
 - 8.1 Introduction
 - 8.2 Structured Programming
 - 8.2.1 File-Level Programming Elements
 - 8.2.2 Namespace-Level Programming Elements
 - 8.2.3 Module-Level Programming Elements
 - 8.2.4 Procedure-Level Programming Elements
 - 8.3 Object-oriented Programming
 - 8.3.1 A Namespace
 - 8.3.2 A Class
 - 8.3.3 An Object
 - 8.3.4 Modules
 - 8.3.5 Access Types
 - 8.3.6 Encapsulation
 - 8.3.7 Data Hiding or Abstraction
 - 8.3.8 Shared Functions
 - 8.3.9 Overloading
 - 8.3.10 Inheritance
 - 8.3.11 MustInherit
 - 8.3.12 NotInheritable
 - 8.3.13 Overriding
 - 8.3.14 Polymorphism
 - 8.3.15 Constructors and Destructors
 - 8.3.16 Property Routines
 - 8.3.17 A Simple Program
 - 8.4 Let us Sum up
 - 8.5 Lesson End Activity
 - 8.6 Keywords
 - 8.7 Questions for Discussion
 - 8.8 Suggested Readings
-

8.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Structured Programming features of VB.net
- Why VB.net can be considered as object oriented programming

8.1 INTRODUCTION

Visual Basic offers a programming language and development environment. It is simple and easy to use. That makes it an extremely attractive choice for the programmers. With the release of its new .NET platform, Microsoft also released a new version of the Visual Basic language, Visual Basic .NET. A number of new features are added. Vb.net is a complete object oriented programming and it is to be written in structured fashion.

8.2 STRUCTURED PROGRAMMING

A Visual Basic program is built up from standard building blocks. A solution comprises one or more projects. A project in turn can contain one or more assemblies. Each assembly is compiled from one or more source files. A source file provides the definition and implementation of classes, structures, modules, and interfaces, which ultimately contain all your code.

8.2.1 File-Level Programming Elements

When you start a project or file and open the code editor, you see some code already in place and in the correct order. Any code that you write should follow the following sequence:

Option statements

Imports statements

Namespace statements and namespace-level elements

If you enter statements in a different order, compilation errors can result.

A program can also contain conditional compilation statements. You can intersperse these in the source file among the statements of the preceding sequence.

Option Statements

Option statements establish ground rules for subsequent code, helping prevent syntax and logic errors. The Option Explicit Statement (Visual Basic) ensures that all variables are declared and spelled correctly, which reduces debugging time. The Option Strict Statement helps to minimize logic errors and data loss that can occur when you work between variables of different data types. The Option Compare Statement specifies the way strings are compared to each other, based on either their Binary or Text values.

Imports Statements

You can include an Imports Statement to import names defined outside your project. An Imports statement allows your code to refer to classes and other types defined within the imported namespace, without having to qualify them. You can use as many Imports statements as appropriate.

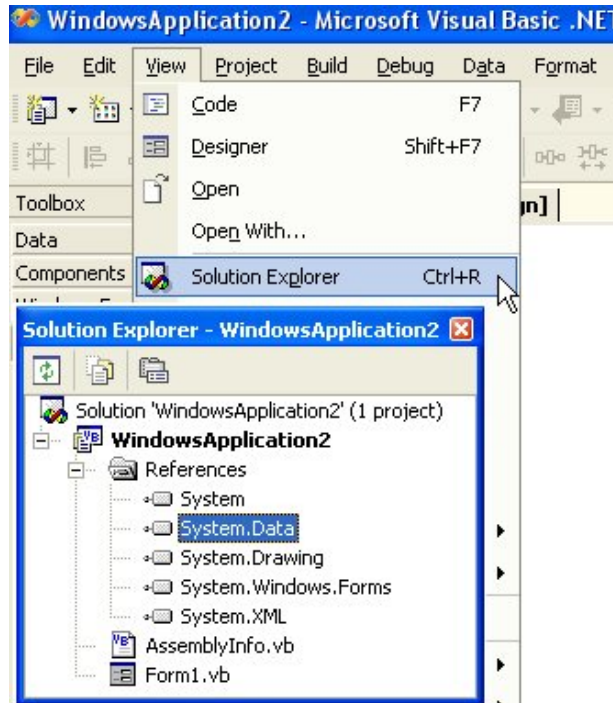
The actual effect of the Imports statement in VB.NET is often a source of confusion for people just learning the language. And the interaction with VB.NET References makes for even more confusion.

Here's the whole story very briefly ... then we'll go over the details in depth.

A Reference to a VB.NET namespace is a requirement and must be added to a project before the objects in the namespace can be used. The Imports statement is not a requirement and is simply a coding convenience that allows shorter names to be used.

Now let's look at an actual example. To illustrate this idea, we're going to use the System.Data namespace - which provides the new ADO.NET data technology.

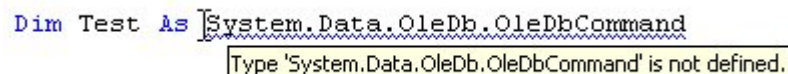
System.Data is added to Windows applications as a Reference by default in VB.NET.



Adding a namespace to the Reference collection in a project makes the objects in that namespace available to the project. The most visible effect of this is that the Visual Studio "Intellisense" helps you find the objects in popup menu boxes.



If you attempt to use an object in your program without a Reference, the line of code generates an error.



The Imports statement, on the other hand, is never required. The only thing it does is allow the name to be resolved without being fully qualified. In other words ...

```
Imports System.Data

Public Class Form1

    Inherits System.Windows.Forms.Form

    Private Sub Form1_Load( ...

        Dim Test As OleDb.OleDbCommand

    End Sub

End Class
```

```
Imports System.Data.OleDb

Public Class Form1

    Inherits System.Windows.Forms.Form

    Private Sub Form1_Load( ...

        Dim Test As OleDbCommand

    End Sub

End Class
```

are both correct. But ...

```
Imports System.Data

Public Class Form1

    Inherits System.Windows.Forms.Form

    Private Sub Form1_Load( ...

        Dim Test As OleDbCommand

    End Sub

End Class
```

results in a syntax error ("Type 'OleDbCommand' is not defined") because the Imports namespace qualification System.Data doesn't provide enough information to find the object OleDbCommand.

Namespace Statements

Namespaces help you organize and classify your programming elements for ease of grouping and accessing. You use the Namespace Statement to classify the following statements within a particular namespace.

The concept of a namespace has been around for quite a while in programming but has only become a requirement for Visual Basic programmers to know about since XML and .NET became critical technologies. The traditional definition of a namespace is a name that uniquely identifies a set of objects so there is no ambiguity when objects from different sources are used together. The type of example that you usually see is something like the Dog namespace and the Furniture namespace both have Leg objects so you can refer to a Dog.Leg or a Furniture.Leg and be very clear about which one you mean.

In practical .NET programming, however, a namespace is just the name that is used to refer to Microsoft's libraries of objects. For example, both System.Data and System.XML are typical References in default VB .NET Windows Applications and the collection of objects they contain are referred to as the System.Data namespace and the System.XML namespace.

The reason "made-up" examples like "Dog" and "Furniture" are used in other definitions is that the "ambiguity" problem really only comes up when you define

your own namespace, not when you're using Microsoft's object libraries. For example, try to find object names that are duplicated between System.Data and System.XML.

When you're using XML, a namespace is a collection of element type and attribute names. These element types and attribute names are uniquely identified by the name of the XML namespace of which they are a part. In XML, a namespace is given the name of a Uniform Resource Identifier (URI) - such as a Web site's address - both because the namespace could be associated with the site and because a URI is a unique name. When it's used this way, the URI is not required to be used other than as a name and there doesn't have to be a document or XML schema at that address.

Conditional Compilation Statements

Conditional compilation statements can appear almost anywhere in your source file. They cause parts of your code to be included or excluded at compile time depending on certain conditions. You can also use them for debugging your application, because conditional code runs in debugging mode only.

8.2.2 Namespace-Level Programming Elements

Classes, structures, and modules contain all the code in your source file. They are namespace-level elements, which can appear within a namespace or at the source file level. They hold the declarations of all other programming elements. Interfaces, which define element signatures but provide no implementation, also appear at module level.

Class Statement

Class statement defines a new data type. A class is a fundamental building block of object-oriented programming (OOP). You can use Class only at namespace or module level. This means the declaration context for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block

Structure Statement

The Structure statement defines a composite value type that you can customize. A structure is a generalization of the user-defined type (UDT) of previous versions of Visual Basic.

Structures support many of the same features as classes. For example, structures can have properties and procedures, they can implement interfaces, and they can have parameterized constructors. However, there are significant differences between structures and classes in areas such as inheritance, declarations, and usage. Also, classes are reference types and structures are value types.

You can use Structure only at namespace or module level. This means the declaration context for a structure must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block

Module Statement

A Module statement defines a reference type available throughout its namespace. A module (sometimes called a standard module) is similar to a class but with some important distinctions. Every module has exactly one instance and does not need to be created or assigned to a variable. Modules do not support inheritance or implement interfaces. Notice that a module is not a type in the sense that a class or structure is — you cannot declare a programming element to have the data type of a module.

You can use Module only at namespace level. This means the declaration context for a module must be a source file or namespace, and cannot be a class, structure, module, interface, procedure, or block. You cannot nest a module within another module, or within any type.

A module has the same lifetime as your program. Because its members are all Shared, they also have lifetimes equal to that of the program.

Interface Statement

An interface defines a set of members, such as properties and procedures, that classes and structures can implement. The interface defines only the signatures of the members and not their internal workings.

A class or structure implements the interface by supplying code for every member defined by the interface. Finally, when the application creates an instance from that class or structure, an object exists and runs in memory.

You can use Interface only at namespace or module level. This means the declaration context for an interface must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block.

8.2.3 Module-Level Programming Elements

Procedures, operators, properties, and events are the only programming elements that can hold executable code (statements that perform actions at run time). They are the module-level elements of your program. Data elements at module level are variables, constants, enumerations, and delegates.

Function Statement

All executable code must be inside a procedure. Each procedure in turn is declared within a class, structure, or module, which is called the containing class, structure, or module.

Use a Function procedure when you need to return a value to the calling code. Use a Sub procedure when you do not need to return a value.

You can use Function only at module level. This means the declaration context for a function must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block.

Function procedures default to public access. You can adjust their access levels with the access modifiers.

You can use a Function procedure on the right side of an expression when you want to use the value returned by the function. You use the Function procedure the same way you use any library function such as Sqrt, Cos, or ChrW.

You call a Function procedure by using the procedure name, followed by the argument list in parentheses, in an expression. You can omit the parentheses only if you are not supplying any arguments. However, your code is more readable if you always include the parentheses.

A function can also be called using the Call statement, in which case the return value is ignored.

Sub Statement

All executable code must be inside a procedure. Use a Sub procedure when you do not need to return a value to the calling code. Use a Function procedure when you need to return a value.

You can use Sub only at module level. This means the declaration context for a sub procedure must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. Sub procedures default to public access. You can adjust their access levels with the access modifiers.

Declare Statement

Sometimes you need to call a procedure defined in a file (such as a DLL or code resource) outside your project. When you do this, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly, such as where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The Declare statement creates a reference to an external procedure and supplies this necessary information.

You can use Declare only at module level. This means the declaration context for an external reference must be a class, structure, or module, and cannot be a source file, namespace, interface, procedure, or block.

External references default to Public (Visual Basic) access. You can adjust their access levels with the access modifiers.

Operator Statement

You can use Operator only in a class or structure. This means the declaration context for an operator cannot be a source file, namespace, module, interface, procedure, or block.

All operators must be Public Shared. You cannot specify ByRef, Optional, or ParamArray for either operand.

You cannot use the operator symbol or identifier to hold a return value. You must use the Return statement, and it must specify a value. Any number of Return statements can appear anywhere in the procedure.

Defining an operator in this way is called operator overloading, whether or not you use the Overloads keyword. The following table lists the operators you can define.

Type	Operators
Unary	+, -, IsFalse, IsTrue, Not
Binary	+, -, *, /, \, &, ^, >>, <<, =, <>, >, >=, <, <=, And, Like, Mod, Or, Xor
Conversion (unary)	CType

Note that the = operator in the binary list is the comparison operator, not the assignment operator.

When you define CType, you must specify either Widening or Narrowing.

Matched Pairs

You must define certain operators as matched pairs. If you define either operator of such a pair, you must define the other as well. The matched pairs are the following:

= and <>

> and <

>= and <=

IsTrue and IsFalse

Data Type Restrictions

Every operator you define must involve the class or structure on which you define it. This means that the class or structure must appear as the data type of the following:

- The operand of a unary operator.
- At least one of the operands of a binary operator.
- Either the operand or the return type of a conversion operator.

Certain operators have additional data type restrictions, as follows:

- If you define the IsTrue and IsFalse operators, they must both return the Boolean type.
- If you define the << and >> operators, they must both specify the Integer type for the operandtype of operand2.

The return type does not have to correspond to the type of either operand. For example, a comparison operator such as = or <> can return Boolean even if neither operand is Boolean.

Logical and Bitwise Operators

The And, Or, Not, and Xor operators can perform either logical or bitwise operations in Visual Basic. However, if you define one of these operators on a class or structure, you can define only its bitwise operation.

You cannot define the AndAlso operator directly with an Operator statement. However, you can use AndAlso if you have fulfilled the following conditions:

- You have defined And on the same operand types you want to use for AndAlso.
- Your definition of And returns the same type as the class or structure on which you have defined it.
- You have defined the IsFalse operator on the class or structure on which you have defined And.

Similarly, you can use OrElse if you have defined Or on the same operands, with the return type of the class or structure, and you have defined IsTrue on the class or structure.

Property Statement

The Property statement introduces the declaration of a property. A property can have a Get procedure (read only), a Set procedure (write only), or both (read-write).

You can use Property only at module level. This means the declaration context for a property must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block.

Properties default to public access. You can adjust a property's access level with an access modifier on the Property statement, and you can optionally adjust one of its property procedures to a more restrictive access level.

Visual Basic passes a parameter to the Set procedure during property assignments. If you do not supply a parameter for Set, the integrated development environment (IDE) uses an implicit parameter named *value*. This parameter holds the value to be assigned to the property. You typically store this value in a private local variable and return it whenever the Get procedure is called.

Event Statement

To handle an event, you must associate it with an event handler subroutine using either the *Handles* or *AddHandler* statement. The signatures of the subroutine and the event must match. To handle a shared event, you must use the *AddHandler* statement.

You can use *Event* only at module level. This means the declaration context for an event must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block.

In most circumstances, you can use the first syntax in the Syntax section of this topic for declaring events. However, some scenarios require that you have greater control over the detailed behavior of the event. The last syntax in the Syntax section of this topic, which uses the *Custom* keyword, provides that control by allowing you to define custom events. In a custom event, you specify exactly what happens when code adds or removes an event handler to or from the event, or when code raises the event.

8.2.4 Procedure-Level Programming Elements

Most of the contents of procedure-level elements are executable statements, which constitute the run-time code of your program. All executable code must be in some procedure (*Function*, *Sub*, *Operator*, *Get*, *Set*, *AddHandler*, *RemoveHandler*, *RaiseEvent*). Data elements at procedure level are limited to local variables and constants.

The Main Procedure

The *Main* procedure is the first code to run when your application has been loaded. *Main* serves as the starting point and overall control for your application. There are four varieties of *Main*:

Sub Main()

Sub Main(ByVal cmdArgs() As String)

Function Main() As Integer

Function Main(ByVal cmdArgs() As String) As Integer

The most common variety of this procedure is *Sub Main()*.

Check Your Progress 1

True or False:

1. The *Option Explicit* Statement (Visual Basic) makes the variable declaration optional.
2. Vb.net is a complete object oriented programming.

8.3 OBJECT-ORIENTED PROGRAMMING

VB.NET is completely object-oriented. Objects are central to Visual Basic programming: Forms and controls are objects. Databases are objects. If you have used Visual Basic for a while, or if you have worked through the examples in the documentation, then you have already programmed with objects, but there is a lot more to objects than what you have seen so far.

8.3.1 A Namespace

In VB.NET, classes and other data structures for a specific purpose are grouped together to form a namespace. You can use the classes in a namespace, by simply importing the namespace. The Imports keyword is used to import a namespace to your project. .NET framework provides a rich set of built in classes, grouped together to various namespaces. In this lesson, we are using the System namespace.

8.3.2 A Class

Probably, you are already familiar with classes and objects. Simply speaking, a Class is a definition of a real life object. For example, Human is a class for representing all human beings. Dog is a class to represent all Dogs. Classes can contain functions too. Animals is a namespace.

Namespace Animals

Dog is a class in the namespace Animals:

```
Class Dog
    Bark is a function in this Class:
        Function Bark()
            Console.WriteLine ("Dog is barking")
        End Function
End Class
End Namespace
```

8.3.3 An Object

An object is an instance of a Class. For example, Jimmy is an object of type Dog. We will create an object in the next section. Read on.

8.3.4 Modules

You can use modules to write common functions. A Module is a group of functions. Unlike functions in classes, Public functions in modules can be called directly from anywhere else. VB provides Functions and Subroutines. Functions and Subroutines are almost the same, but the difference is that a subroutine can't return a value.

```
Public Module modMain
    Execution will start from the Main() subroutine:
    Sub Main()
        'Call our function. See below
        OurFunction()
    End sub
    OurFunction: Our own little function to use the class Dog:
    Function OurFunction()
        'Here is how we declare a variable Jimmy of type Dog.
```

```
'We use Animals.Dog because, the class Dog is in the
'namespace Animals (see above).
Dim Jimmy as Animals.Dog
'Create an object. Unlike in VB 6, it is not required to use
'the 'set' keyword.
Jimmy = new Animals.Dog()
'Another way to create an object is
'Dim Jimmy as new Dog
'Call Jimmy's Main Function
Jimmy.Bark()
End Function
End module
```

8.3.5 Access Types

The major access types are Public, Private, Friend and Protected. A Class may contain functions, variables etc., which can be either Public or Private or Protected or Friend. If they are Public, they can be accessed by creating objects of the Class. Private and Protected members can be accessed only by the functions inside the Class. Protected members are much like Private members, but they have some special use while inheriting a Class. Friend members can be accessed only by elements of the same project, and not by the ones outside the current project. Let us expand our dog class.

Import the System namespace (already available in .NET).

```
Imports System
Animals is a namespace.
Namespace Animals
Dog is a class in the namespace Animals.
Public Class Dog
    'A public variable
    Public AgeOfDog as Integer
    Bark is a function in this class. It is Public:
    Public Function Bark()
        Console.WriteLine ("Dog is barking")
    End Function
    Walk is a function in this class. It is Private.
    Private Function Walk()
        Console.WriteLine ("Dog is walking")
    End Function
End Class
End Namespace
Our Module:
Public Module modMain
    Execution will start from the Main() subroutine:
    Sub Main()
        'Call our function. See below
```

```

        OurFunction()
End sub

'OurFunction: Called from Main()
Function OurFunction()
    Dim Jimmy as Animals.Dog
    Jimmy=new Animals.Dog()
    'This will work, because Bark & Ageofdog are public
    Jimmy.Bark
    Jimmy.AgeOfDog=10
    'Calling the Walk function will not work here, because
    'Walk() is outside the class Dog
    'So this is wrong. Uncomment this and try to compile, it will
    'cause an error.
    'Jimmy.Walk
End Function
End Module

```

8.3.6 Encapsulation

Putting all the data and related functions in a Class is called Encapsulation. Encapsulation is the exposure of properties and methods of an object while hiding the actual implementation from the outside world. In other words, the object is treated as a black box—developers who use the object should have no need to understand how it actually works.

Encapsulation allows developers to build objects that can be changed without affecting the client code that uses them. The key is that the interface of the object, the set of exposed properties and methods of the object, doesn't change even if the internal implementation does.

8.3.7 Data Hiding or Abstraction

Normally, in a Class, variables used to hold data (like the age of a dog) is declared as Private. Functions or property routines are used to access these variables. Protecting the data of an object from outside functions is called Abstraction or Data Hiding. This prevents accidental modification of data by functions outside the class.

8.3.8 Shared Functions

The shared members in a class (both functions and variables) can be used without creating objects of a class as shown. The Shared modifier indicates that the method does not operate on a specific instance of a type and may be invoked directly from a type rather than through a particular instance of a type.

```

Import the System namespace (already available in .NET).
Imports System
Animals is a namespace.
Namespace Animals
    Dog is a class in the namespace Animals.
    Class Dog

```



```
Bark is a now a Public, shared function in this class.
Public Shared Function Bark()
    Console.WriteLine ("Dog is barking")
End Function
Walk is a Public function in this class. It is not shared.
Public Function Walk()
    Console.WriteLine ("Dog is walking")
End Function
End Class
End Namespace
Our Module:
Public Module modMain
Execution will start from the Main() subroutine.
Sub Main()
    'We can call the Bark() function directly,
    'with out creating an object of type Dog -
    'because it is shared.
    Animals.Dog.Bark()
    'We can call the Walk() function only
    'after creating an object, because
    'it is not shared.
    Dim Jimmy as Animals.Dog
    Jimmy=new Animals.Dog()
    Jimmy.Walk()
    'Now Guess? The WriteLine() function we used so far
    'is a shared function in class Console :)
    'Also, we can write the Main() function itself as a shared
    'function in a class. i.e Shared Sub Main(). Try
    'moving Main() from this module to the above class
End sub
End Module
```

Check Your Progress 2

1. What are the major access types?

.....
.....

2. What is Encapsulation?

.....
.....

8.3.9 Overloading

Overloading is a simple technique, to enable a single function name to accept parameters of different type. Let us see a simple Adder class. Import the System namespace (already available in .NET).

Imports System

Class Adder

Here, we have two Add() functions. This one adds two integers. Convert.ToString is equivalent to the good old CStr.

```
Overloads Public Sub Add(A as Integer, B as Integer)
    Console.WriteLine ("Adding Integers: " + Convert.ToString(a + b))
End Sub
```

This one adds two strings.

```
Overloads Public Sub Add(A as String, B as String)
    Console.WriteLine ("Adding Strings: " + a + b)
End Sub
```

'And both have the same name. This is possible because, we used the

'Overloads keyword, to overload them.

'Here, we have the Main Function with in this class. When you write.

'your main function inside the class, it should be a shared function.

```
Shared Sub Main()
    Dim AdderObj as Adder
    'Create the object
    AdderObj=new Adder
    'This will invoke first function
    AdderObj.Add(10,20)
    'This will invoke second function
    AdderObj.Add("hello"," how are you")
End Sub
```

End Class

8.3.10 Inheritance

Inheritance is the property in which, a derived class acquires the attributes of its base class. In simple terms, you can create or 'inherit' your own class (derived class), using an existing class (base class). You can use the Inherits keyword for this.

Let us see a simple example. Import the System namespace (already available in .NET).

Imports System

Our simple base class:

Class Human

```
'This is something that all humans do
Public Sub Walk()
```

```
        Console.WriteLine ("Walking")
    End Sub
End Class
```

Now, let us derive a class from Human.

A Programmer is a Human.

```
Class Programmer
    Inherits Human
    'We already have the above Walk() function
    'This is something that all programmers do ;)
    Public Sub StealCode()
        Console.WriteLine ("Stealing code")
    End Sub
End Class
```

Just a MainClass.

```
Class MainClass
    'Our main function
    Shared Sub Main()
        Dim Tom as Programmer
        Tom=new Programmer

        'This call is okie because programmer got this function
        'from its base class
        Tom.Walk()

        'This is also correct because Tom is a programmer
        Tom.StealCode()
    End Sub
End Class
```

8.3.11 MustInherit

The MustInherit keyword specifies that a class cannot be instantiated and can be used only as a base class. I.e., if you declare our Human class as "MustInherit Class Human", then you can't create objects of type Human without inheriting it.

8.3.12 NotInheritable

The NotInheritable keyword specifies that a class cannot be inherited. I.e., if you specify 'NotInheritable Class Human', no derived classes can be made from the Human class.

8.3.13 Overriding

By default, a derived class Inherits methods from its base class. If an inherited property or method needs to behave differently in the derived class it can be overridden; that is, you can define a new implementation of the method in the derived class. The Overridable keyword is used to mark a function as overridable. The keyword Overrides is used to mark that a function is overriding some base class function. Let us see an example.

Import the System namespace (already available in .NET).

```
Imports System
```

Our simple base class:

```
Class Human
```

```
    'Speak() is declared Overridable
```

```
    Overridable Public Sub Speak()
```

```
        Console.WriteLine ("Speaking")
```

```
    End Sub
```

```
End Class
```

Now, let us derive a class from Human:

An Indian is a Human:

```
Class Indian
```

```
    Inherits Human
```

```
    'Let us make Indian speak Hindi, the National Language
```

```
    'in India
```

```
    'Speak() is overriding Speak() in its base class (Human)
```

```
    Overrides Public Sub Speak()
```

```
        Console.WriteLine ("Speaking Hindi")
```

```
    'Important: As you expect, any call to Speak() inside this class
```

```
    'will invoke the Speak() in this class. If you need to
```

```
    'call Speak() in base class, you can use MyBase keyword.
```

```
    'Like this
```

```
    'Mybase.Speak()
```

```
    End Sub
```

```
End Class
```

Just a class to put our Main().

```
Class MainClass
```

```
    'Our main function
```

```
Shared Sub Main()  
    'Tom is a generic Human  
    Dim Tom as Human  
    Tom=new Human  
    'Tony is a human and an Indian  
    Dim Tony as Indian  
    Tony=new Indian  
    'This call will invoke the Speak() function  
    'in class Human  
    Tom.Speak()  
    'This call will invoke the Speak() function  
    'in class Indian  
    Tony.Speak()  
End Sub  
End Class
```

8.3.14 Polymorphism

Polymorphism is the property in which a single object can take more than one form. For example, if you have a base class named Human, an object of Human type can be used to hold an object of any of its derived type. When you call a function in your object, the system will automatically determine the type of the object to call the appropriate function. For example, let us assume that you have a function named speak() in your base class. You derived a child class from your base class and overloaded the function speak(). Then, you create a child class object and assign it to a base class variable. Now, if you call the speak() function using the base class variable, the speak() function defined in your child class will work. On the contrary, if you are assigning an object of the base class to the base class variable, then the speak() function in the base class will work. This is achieved through runtime type identification of objects. See the example.

Import the System namespace (already available in .NET).

Imports System

This example is exactly the same as the one we saw in the previous lesson. The only difference is in the Shared Sub Main() in the class MainClass. So scroll down and see an example:

Our simple base class:

```
Class Human  
    'Speak() is declared Overridable  
    Overridable Public Sub Speak()  
        Console.WriteLine ("Speaking")  
    End Sub  
End Class
```

Now, let us derive a class from Human.

An Indian is a Human.

```
Class Indian
```

```
Inherits Human
```

```
'Let us make Indian speak Hindi, the National Language
```

```
'in India
```

```
'Speak() is overriding Speak() in its base class (Human)
```

```
Overrides Public Sub Speak()
```

```
    Console.WriteLine ("Speaking Hindi")
```

```
'Important: As you expect, any call to Speak() inside this class
```

```
'will invoke the Speak() in this class. If you need to
```

```
'call Speak() in base class, you can use MyBase keyword.
```

```
'Like this
```

```
'Mybase.Speak()
```

```
End Sub
```

```
End Class
```

Carefully examine the code in Main():

```
Class MainClass
```

```
'Our main function
```

```
Shared Sub Main()
```

```
    'Let us define Tom as a human (base class)
```

```
    Dim Tom as Human
```

```
    'Now, I am assigning an Indian (derived class)
```

```
    Tom=new Indian
```

```
    'The above assignment is legal, because
```

```
    'Indian IS_A human.
```

```
    'Now, let me call Speak as
```

```
    Tom.Speak()
```

```
    'Which Speak() will work? The Speak() in Indian, or the
```

```
    'Speak() in human?
```

```
    'The question arises because, Tom is declared as a Human,
```

```
    'but an object of type Indian is assigned to Tom.
```

```
    'The Answer is, the Speak() in Indian will work. This is because,
```

```
    'most object oriented languages like Vb.net can automatically
```

```
    'detect the type of the object assigned to a base class variable.
```

```
    'This is called Polymorphism
```

```
End Sub
```

```
End Class
```

8.3.15 Constructors and Destructors

Import the System namespace (already available in .NET).

Imports System

A Constructor is a special function which is called automatically when a class is created. In VB.NET, you should use useNew() to create constructors. Constructors can be overloaded, but unlike the functions, the Overloads keyword is not required. A Destructor is a special function which is called automatically when a class is destroyed. In VB.NET, you should use useFinalize() routine to create Destructors. They are similar to Class_Initialize and Class_Terminate in VB 6.0.

Dog is a class:

```
Class Dog
```

```
    'The age variable
```

```
    Private Age as integer
```

The default constructor:

```
    Public Sub New()
```

```
        Console.WriteLine ("Dog is Created With Age Zero")
```

```
        Age=0
```

```
    End Sub
```

The parameterized constructor:

```
    Public Sub New(val as Integer)
```

```
        Console.WriteLine ("Dog is Created With Age " +  
Convert.ToString(val))
```

```
        Age=val
```

```
    End Sub
```

This is the destructor:

```
    Overrides Protected Sub Finalize()
```

```
        Console.WriteLine ("Dog is Destroyed")
```

```
    End Sub
```

```
    'The Main Function
```

```
    Shared Sub Main()
```

```
        Dim Jimmy, Jacky as Dog
```

```
        'Create the objects
```

```
        'This will call the default constructor
```

```
        Jimmy=new Dog
```

```
        'This will call the parameterized constructor
```

```
        Jacky=new Dog(10)
```

```
    End Sub
```

```
    'The Destruction will be done automatically, when
```

```
    'the program ends. This is done by the Garbage
```

```
    'Collector.
```

```
End Class
```

8.3.16 Property Routines

You can use both properties and fields to store information in an object. While fields are simply Public variables, properties use property procedures to control how values are set or returned. You can use the Get/Set keywords for getting/setting properties. See the following example. Import the System namespace (already available in .NET).

```
Imports System

Dog is a class.

Public Class Dog
    'A private variable    to hold the value
    Private mAgeOfDog as Integer
This is our property routine:
Public Property Age() As Integer
    'Called when someone tries to retrieve the value
Get
    Console.WriteLine ("Getting Property")
    Return mAgeOfdog
End Get
Set(ByVal Value As Integer)
    'Called when someone tries to assign a value
    Console.WriteLine ("Setting Property")
    mAgeOfDog=Value
End Set
End Property
End Class
```

Another class:

```
Class MainClass
    'Our main function. Execution starts here.
    Shared Sub Main()
        'Let us create an object.
        Dim Jimmy as Dog
        Jimmy=new Dog
        'We can't access mAgeofDog directly, so we should
        'use Age() property routine.
        'Set it. The Age Set routine will work
        Jimmy.Age=30
        'Get it back. The Age GET routine will work
        Dim curAge=Jimmy.Age()
    End Sub
End Class
```

8.3.17 A Simple Program

Let us analyze a simple program. First, let us import the required namespaces:

```
Imports System
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Drawing
```



```
'We are inheriting a class named SimpleForm, from the
'class System.Windows.Forms.Form
'
'i.e, Windows is a namespace in system, Forms is a
'namespace in Windows, and Form is a class in Forms.
Public Class SimpleForm
Inherits System.Windows.Forms.Form
    'Our constructor
Public Sub New()
    'This will invoke the constructor of the base
    'class
MyBase.New()
```

Set the text property of this class. We inherited this property from the base class:

```
Me.Text = "Hello, How Are You?"
End Sub
End Class
Public Class MainClass
    Shared Sub Main()
        'Create an object from our SimpleForm class
        Dim sf as SimpleForm
        sf=new SimpleForm

        'Pass this object to the Run() function to start
        System.Windows.Forms.Application.Run(sf)
    End Sub
End Class
```

Check Your Progress 3

Fill in the blanks:

1. In VB.net each _____ is compiled from one or more source files.
2. Function procedures default to _____ access.
3. All operators must be _____ Shared.
4. You can use Event only at _____ level.

8.4 LET US SUM UP

VB.NET is completely object oriented. A class is a fundamental building block of object-oriented programming (OOP). You can use Class only at namespace or module level. This means the declaration context for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. Also, classes are reference types and structures are value types. A Module statement defines

a reference type available throughout its namespace. This means the declaration context for a module must be a source file or namespace, and cannot be a class, structure, module, interface, procedure, or block. An interface defines a set of members, such as properties and procedures, that classes and structures can implement. Each procedure in turn is declared within a class, structure, or module, which is called the containing class, structure, or module. Function procedures default to public access. Databases are objects. In simple terms, you can create or 'inherit' your own class (derived class), using an existing class (base class).

8.5 LESSON END ACTIVITY

VB.NET is completely object oriented. Explain.

8.6 KEYWORDS

Object: An object is an instance of a Class.

Module: A Module is a group of functions.

Encapsulation: Putting all the data and related functions in a Class is called Encapsulation.

Abstraction: Protecting the data of an object from outside functions is called Abstraction or Data Hiding.

Overloading: It is a simple technique, to enable a single function name to accept parameters of different type.

Inheritance: It is the property in which, a derived class acquires the attributes of its base class.

Polymorphism: It is the property in which a single object can take more than one form.

Constructor: It is a special function which is called automatically when a class is created.

Destructor: It is a special function which is called automatically when a class is destroyed.

8.7 QUESTIONS FOR DISCUSSION

1. What is Option Explicit Statement? Compare it with Option Compare Statement.
2. What are the namespace-level elements?
3. Describe Module statement.

Check Your Progress: Model Answers

CYP 1

1. False
2. True

CYP 2

1. The major access types are Public, Private, Friend and Protected..
2. Putting all the data and related functions in a Class is called Encapsulation.

Contd...

CYP 3

1. Assembly
2. Public
3. Public
4. Module

8.8 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*, Pearson edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic.Net*, 5th Edition, PHI

Kant, *Visual Basic.Net—A Beginners Guide*, TMCH

LESSON

9

FILE HANDLING

CONTENTS

- 9.0 Aims and Objectives
- 9.1 Introduction
- 9.2 Files Classification
- 9.3 Handling Files using Function and Classes
 - 9.3.1 Directory Class
- 9.4 File Class
- 9.5 File Processing
- 9.6 Let us Sum up
- 9.7 Lesson End Activity
- 9.8 Keywords
- 9.9 Questions for Discussion
- 9.10 Suggested Readings

9.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- An overview of file classification
- Concept of file class and directory class
- File handling using function and classes
- File processing

9.1 INTRODUCTION

File handling in Visual Basic is based on System.IO namespace with a class library that supports string, character and file manipulation. These classes contain properties, methods and events for creating, copying, moving, and deleting files. Since both strings and numeric data types are supported, they also allow us to incorporate data types in files. In this lesson overview of file handling will be discussed.

9.2 FILES CLASSIFICATION

Windows Explorer represents the contents of your hard drive graphically using file and folder icons. Many of the actions that you perform in Explorer (such as copying or renaming files) also can be accomplished in Visual Basic .NET via the System.IO namespace. The System assembly should be automatically referenced in your project, so you just need to add the following Imports statement at the top of your form or module:

Imports System.IO

The most useful classes for performing file operations are

- **FileSystemInfo:** The `FileSystemInfo` class represents an entry in the file system, whether it is a file or a folder. You cannot create an instance of a `FileSystemInfo` object with the `New` keyword but, as you will see in a moment, it is useful when processing both files and folders at the same time.
- **FileInfo:** You use the `FileInfo` class to manipulate operating system files. Each `FileInfo` object you create represents a file on the disk. This class is inherited from the `FileSystemInfo` class and contains additional, file-specific members.
- **DirectoryInfo:** The `DirectoryInfo` class allows you to control the folders of your file system. Like the `FileInfo` class, the `DirectoryInfo` class is inherited from the `FileSystemInfo` class and contains additional, directory-specific members.
- **File:** The `File` class contains static methods for performing operations on operating system files.
- **Directory:** The `Directory` class contains static methods for performing operations on operating system folders.

You may have already noticed there is some duplication in the previous list of classes. For example, both the `File` and `FileInfo` classes contain methods for determining whether a file exists, deleting a file, and so on. The reason is that the `Directory` and `File` classes contain all the static methods, while the `DirectoryInfo` and `FileInfo` classes contain all the instance methods.

When dealing with files, static methods can be more efficient if you do not need to perform multiple operations on the same file or directory, or access its properties. By avoiding the overhead of declaring a variable and storing an object instance, you use fewer of your system resources.

9.3 HANDLING FILES USING FUNCTION AND CLASSES

VB.NET offers two handy classes—`DirectoryInfo` and `FileSystemInfo`—that allow you to access directories, check their properties, and perform other basic functions on the file system. `DirectoryInfo` lets you get information about a directory, and create, move, and enumerate through directories and subdirectories. `FileSystemInfo` has methods that allow you to manipulate both files and directories; therefore, a `FileSystemInfo` object can represent either a directory or a file. In this tip, I'll show you a way to list all the files in a particular directory and then access the files' properties using `DirectoryInfo` and `FileSystemInfo` classes.

9.3.1 Directory Class

Microsoft documentation says the `Directory` class exposes static methods for creating, moving, and enumerating through directories and subdirectories. In addition, you may access and manipulate various directory attributes such as creation or last modified time stamps, along with the Windows access control list.

The `Directory` class in VB .NET allows us to create and work with Folders/Directories. With this class we can create, edit and delete folders and also maintain drives on the machine.

The basics

The `Directory` class provides numerous static methods for working with directories. These static methods perform security checks on all methods. I'll examine most of these methods, with the first group consisting of the basics of working with directories:

- **CreateDirectory:** Allows you to create new directories or subdirectories. Also, you may establish directory security with its creation.
- **Delete:** Allows you to delete a specific empty directory from the system. This method throws an `IOException` if the directory specified in the path parameter contains files or subdirectories.
- **Exists:** Allows you to determine if a specific directory exists. The path parameter is permitted to specify relative or absolute path information. Relative path information is interpreted as relative to the current working directory.
- **Move:** Allows you to move a file or directory to a new location. It accepts two parameters: the directory or file to move and its destination.

The VB.NET code is mentioned below which uses these methods to create, move, and delete directories. It uses the `Exists` method throughout the code to determine if a directory exists. All calls to the `Directory` class methods are enclosed in try/catch blocks. The code begins with a directory called `test` on the C drive. It uses the `Path` class to determine if the string containing the directory name has a file extension. If so, it determines it is a path to a file and not a directory and the application halts. This directory is created if it does not already exist. Next, the directory is renamed to `techrepublic` and finally deleted.

```
Imports System.IO
Module Module1
Sub Main()
Dim testPath As StringtestPath = "c:\test"
If (Path.HasExtension(testPath)) ThenConsole.WriteLine("This is a file
path, not a directory.")
Else
If (Directory.Exists(testPath)) ThenConsole.WriteLine("This directory
already exists.")
Else
TryDirectory.CreateDirectory(testPath)
Catch ex As IOExceptionConsole.WriteLine("Error creating directory {0} -
{1}", testPath, ex.Message.ToString())
End Try
If (Directory.Exists(testPath)) Then
TryDirectory.Move(testPath, "c:\techrepublic")
Catch ex As IOExceptionConsole.WriteLine("Error moving directory {0} -
{1}", testPath, ex.Message.ToString())
End Try
End If
If (Directory.Exists("c:\techrepublic")) Then
```

```
TryDirectory.Delete("c:\techrepublic", True)

Catch ex As IOException Console.WriteLine("Error deleting directory
c:\\techrepublic - {0}", ex.Message.ToString())

End Try

End If

End If

End If

End Sub

End Module
```

Directories and their contents

Once a directory is created, it is often necessary to peruse its contents or examine the directory itself. The Directory class makes it easy to access information on the directory as well as its contents via various methods and properties. These methods are outlined in the following list:

- GetCurrentDirectory: Returns the current application's working directory.
- GetDirectories: Returns a list of subdirectories for a specific directory as an array of string values.
- GetDirectoryRoot: Returns the volume information, root information, or both for the specified path.
- GetFiles: Returns a string array containing the filenames within a specific directory.
- GetFileSystemEntries: Returns a string array containing the filenames and directories within a specific directory.
- GetParent: Retrieves the parent directory of the specified path, including both absolute and relative paths as a string value.
- GetLogicalDrives: Retrieves the names of the logical drives on the computer in the form <drive letter>:\ within a string array.

The VB.NET code mentioned below provides a sample of using these methods. Basically, it displays all information regarding the current application directory, all subdirectories and files, the root directory, and all logical drives on the current system.

```
using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace TRDirectory {

class Program {

static void Main(string[] args) {

string[]

subDirs;int i;DateTime dtCreated;DateTime dtAccessed;DateTime dtWrite;

try {

subDirs = Directory.GetDirectories("c:\\");

for (i = 0; i <= subDirs.GetUpperBound(0); i++)
```

```

{dtCreated = Directory.GetCreationTime(subDirs[i]);dtAccessed =
Directory.GetLastAccessTime(subDirs[i]);dtWrite =
Directory.GetLastWriteTime(subDirs[i]);Console.WriteLine("Subdirectory
{0}", subDirs[i]);Console.WriteLine("Created: {0}",
dtCreated.ToString());Console.WriteLine("Last accessed: {0}",
dtAccessed.ToString());Console.WriteLine("Last changed: {0}",
dtWrite.ToString());Directory.SetLastAccessTime(subDirs[i],
DateTime.Now);}}
catch (IOException e)
{Console.WriteLine("Exception {0}", e.Message.ToString());}}}}

```

Time and date stamps

It is often advantageous to know when a directory was created or the last time it was accessed or modified. The Directory class provides the following methods for working with time and date stamps:

- **GetCreationTime:** Gets the creation date and time of a directory.
- **GetLastAccessTime:** Returns the date and time of the specified file or directory that was last accessed.
- **GetLastWriteTime:** Returns the date and time of the specified file or directory that was last written to.
- **SetCreationTime:** Sets the creation date and time for the specified file or directory.
- **SetLastAccessTime:** Sets the date and time of the specified file or directory that was last accessed.
- **SetLastWriteTime:** Sets the date and time a directory was last written to.

The below mentioned VB.NET code provides a list of all directories on the C drive with date and times when they were last accessed, changed, and initially created. The last access time is reset to the current date and time for each directory.

```

Imports System.IO
Module Module1
Sub Main()
Dim subDirs() As String
Dim i As Integer
Dim dtCreated As DateTime
Dim dtAccessed As DateTime
Dim dtWrite As DateTime
TrysubDirs = Directory.GetDirectories("c:\")
For i = 0 To subDirs.GetUpperBound(0)dtCreated =
Directory.GetCreationTime(subDirs(i))dtAccessed =
Directory.GetLastAccessTime(subDirs(i))dtWrite =
Directory.GetLastWriteTime(subDirs(i))Console.WriteLine("Subdirectory
{0}", subDirs(i))Console.WriteLine("Created: {0}",
dtCreated.ToString())Console.WriteLine("Last accessed: {0}",
dtAccessed.ToString())Console.WriteLine("Last changed: {0}",
dtWrite.ToString())Directory.SetLastAccessTime(subDirs(i),
DateTime.Now)
Next i
Catch e As IOExceptionConsole.WriteLine("Exception {0}",
e.Message.ToString())

```



```
End Try
End Sub
End Module
```

Check Your Progress 1

Fill in the blanks:

1. _____ represents the contents of your hard drive graphically using file and folder icons.
2. The _____ class represents an entry in the file system, whether it is a file or a folder.

9.4 FILE CLASS

The File class provides methods to copy, delete, and rename files.

- **Copy:** File.Copy can be used to create a backup copy of a file:

File.Copy("c:\temp\quotes.doc", "C:\temp\quotesbackup.doc", True)The parameters of the File.Copy method are

Source File— The source file you want to copy. If the file does not exist, an exception will be thrown.

Destination File— The path to the destination file you want to create.

Overwrite Existing File— This parameter is optional and indicates whether you want to overwrite an existing destination file. If this parameter is omitted or set to False and the destination file already exists, the method will throw an exception.

As with many of the shared methods, there is a corresponding method for use with an object instance. If you have already created a FileInfo object, you can use the CopyTo method and simply provide the destination and overwrite parameters:

```
Dim filTemp As New File("C:\temp\quotes.doc")
filTemp.CopyTo("C:\temp\newquotes.doc", False)
```

Note

The Copy and CopyTo methods will throw an exception if you attempt to overwrite an existing read-only file, even if you have the overwrite parameter set to True. To overwrite an existing read-only file, you first need to turn off its read-only attribute.

The DirectoryInfo class does not have a Copy or CopyTo method. However, you can create a new directory using the Create method and then copy the files within it. As an exercise, adapt the recursive function from Listing 24.3 to copy all the files and subdirectories contained within a given directory.

The rename and delete methods are identical for files and directories:

- **Move:** Static method to rename a file or directory. Its parameters are a source and a destination.
- **MoveTo:** Instance method to rename a file or directory.

- **Delete:** Deletes the specified file or directory. Wildcards are not allowed. If the read-only attribute of the file you want to delete is set, the method call will throw an exception.

9.5 FILE PROCESSING

To support file processing, the .NET Framework provides the System.IO namespace that contains many different classes to handle almost any type of file operation you may need to perform. The parent class of file processing is Stream. With Stream, you can store data to a stream or you can retrieve data from a stream. Stream is an abstract class, which means that you cannot use it to declare a variable in your application.

As an abstract class, Stream is used as the parent of the classes that actually implement the necessary operations. You will usually use a combination of classes to perform a typical stream operation. For example, some classes are used to create a stream object while some others are used to write data to the created stream.

Before performing file processing, one of your early decisions will consist of specifying the type of operation you want the user to perform. For example, you may want to create a brand new file. You may want to open an existing file. Or you may want to perform a routine operation on a file. In all or most cases, whether you are creating a new file or manipulating an existing one, you must specify the name of the file. You can do this by declaring a String variable. Most classes used to create a stream as we will learn later can take a string that represents the name of the file.

The File's Path

If you declare a string variable as done above and use it as the name of the file, the file will be created in the same folder as the application. Otherwise, you can create your new file anywhere in the hard drive. To do that, you must provide a complete path where the file will reside.

A path is a string that specifies the drive (such as A:, C:, or D:). The sections of a complete path string are separated by a backslash. For example, a path can be made of the name of a folder followed by the name of the file. An example would be

`C:\Palermo.tde`

A path can also consist of a drive followed by the name of the folder in which the file will be created. Here is an example:

`C:\Program Files\Palermo.tde`

A path can also indicate that the file will be created in a folder that itself is inside of another folder. In this case, remember that the names of folders must be separated by backslashes.

When providing a path to the file, if the drive you specify doesn't exist or cannot be read, the compiler would consider that the file doesn't exist. If you provide a folder that doesn't exist in the drive, the compiler would consider that the file doesn't exist. This also means that the compiler will not create the folder(s) (the .NET Framework provides all means to create a folder but you must ask the compiler to create it simply specifying a folder that doesn't exist will not automatically create it, even if you are creating a new file). As we will see in the next sections, the compiler can check the existence of a file or path.

File Existence

While Stream is used as the parent of all file processing classes, the .NET Framework provides the File class equipped with methods to create, save, open, copy, move, delete, or provide detailed information about, files. Based on its functionality, the File class is typically used to assist the other classes with their processing operations. To effectively provide this support, all File's methods are static, which means that you will not need to declare a File variable to access them.

One of the valuable operations that the File class can perform is to check the existence of the file you want to use. For example, if you are creating a new file, you may want to make sure that it doesn't exist already because if you try to create a file that exists already, the compiler may first delete the old file before creating the new one. This could lead to unpredictable results, especially because such a file is not sent to the Recycle Bin. On the other hand, if you are trying to open a file, you should first make sure that the file exists, otherwise the compiler will not be able to open a file it cannot find.

To check the existence of a file, the File class provides the Exists() method. Its syntax is:

Public Shared Function Exists(ByVal path As String) As Boolean

If you provide only the name of the file, the compiler would check it in the folder of the application. If you provide the path to the file, the compiler would check its drive, its folder(s) and the file itself. In both cases, if the file exists, the method returns true. If the compiler cannot find the file, the method returns false. It's important to know that if you provided a complete path to the file, any slight mistake in the path such as one backslash instead of two or a misspelling somewhere, would produce a false result.

Access to a File

In order to perform an operation on a file, you must specify to the operating system how to proceed. One of the options you have is to indicate the type of access that will be granted on the file. This access is specified using the FileAccess enumerator. The members of the FileAccess enumerator are:

- FileAccess.Write: New data can be written to the file
- FileAccess.Read: Existing data can be read from the file
- FileAccess.ReadWrite: Existing data can be read from the file and new data be written to the file

File Sharing

In standalone workstations, one person is usually able to access and open a file then perform the necessary operations on it. In networked computers, you may create a file that different people can access at the same time or you may make one file access another file to retrieve information. For example, suppose you create an application for a fast food restaurant that has two or more connected workstations and all workstations save their customers orders to a common file. In this case, you must make sure that any of the computers can access the file to save an order. An employee from one of these workstations must also be able to open the file to retrieve a customer order for any necessary reason. You can also create a situation where one file holds an inventory of the items of a store and another file holds the customers orders. Obviously one file would depend on another. Based on this, when an operation

must be performed on a file, you may have to specify how a file can be shared. This is done through the FileShare enumerator.

The values of the FileShare enumerator are:

- FileShare.Inheritable: Allows other file handles to inherit from this file
- FileShare.None: The file cannot be shared
- FileShare.Read: The file can be opened and read from
- FileShare.Write: The file can be opened and written to
- FileShare.ReadWrite: The file can be opened to write to it or read from it

The Mode of a File

Besides the access to the file, another option you will most likely specify to the operating system is referred to as the mode of a file. It is specified through the FileMode enumerator. The members of the FileMode Enumerator are:

- FileMode.Append: If the file already exists, the new data will be added to its end. If the file doesn't exist, it will be created and the new data will be added to it
- FileMode.Create: If the file already exists, it will be deleted and a new file with the same name will be created. If the file doesn't exist, then it will be created
- FileMode.CreateNew: If the file already exists, the compiler will throw an error. If the file doesn't exist, it will be created
- FileMode.Open: If the file exists, it will be opened. If the file doesn't exist, an error would be thrown
- FileMode.OpenOrCreate: If the file already exists, it will be opened. If the file doesn't exist, it will be created
- FileMode.Truncate: If the file already exists, its contents will be deleted completely but the file will be kept, allowing you to write new data to it. If the file doesn't exist, an error would be thrown.

Binary File Streaming

File streaming consists of performing one of the routine operations on a file, such as creating or opening it. This basic operation can be performed using a class called FileStream. You can use a FileStream object to get a stream ready for processing. As one of the most complete classes of file processing of the .NET Framework, FileStream is equipped with all necessary properties and methods. To use it, you must first declare a variable of it. The class is equipped with nine constructors.

One of the constructors (the second) of the FileStream class has the following syntax:

```
Public Sub New(ByVal path As String, ByVal mode As FileMode)
```

This constructor takes as its first argument the name of the file or its path. The second argument specifies the type of operation to perform on the file. Here is an example:

```
Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) _  
    Handles btnStart.Click  
    Dim fileName As String = "Persons.spr"
```

```

        Dim fstPersons As FileStream = New FileStream(fileName,
        FileMode.Create)

    End Sub

```

Stream Writing

A streaming operation is typically used to create a stream. Once the stream is ready, you can write data to it. The writing operation is performed through various classes. One of these classes is `BinaryWriter`.

The `BinaryWriter` class can be used to write values of primitive data types (`char`, `int`, `float`, `double`, etc). To use a `BinaryWriter` value, you can first declare its pointer. To do this, you would use one of the class' three constructors. The first constructor is the default, meaning it doesn't take an argument. The second constructor has the following syntax:

```
Public Sub New(ByVal output As Stream)
```

This constructor takes as argument a `Stream` value, which could be a `FileStream` variable. Here is an example:

```

Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnStart.Click

    Dim fileName As String = "Persons.spr"

    Dim fstPersons As FileStream = New FileStream(fileName,
    FileMode.Create)

    Dim wrtPersons As BinaryWriter = New BinaryWriter(fstPersons)

End Sub

```

Most classes that are used to add values to a stream are equipped with a method called `Write`. This is also the case for the `BinaryWriter` class. This method takes as argument the value that must be written to the stream. The method is overloaded so that there is a version for each primitive data type. Here is an example that adds strings to a newly created file:

```

Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnStart.Click

    Dim fileName As String = "Persons.spr"

    Dim fstPersons As FileStream = New FileStream(fileName,
    FileMode.Create)

    Dim wrtPersons As BinaryWriter = New BinaryWriter(fstPersons)

    wrtPersons.Write("James Bloch")

    wrtPersons.Write("Catherina Wallace")

    wrtPersons.Write("Bruce Lamont")

    wrtPersons.Write("Douglas Truth")

End Sub

```

Stream Closing

When you use a stream, it requests resources from the operating system and uses them while the stream is available. When you are not using the stream anymore, you should free the resources and make them available again to the operating system so that other services can use them. This is done by closing the stream.

To close a stream, you can call the `Close()` method of the class(es) you were using. Here are examples:

```
Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnStart.Click
    Dim fileName As String = "Persons.spr"
    Dim fstPersons As FileStream = New FileStream(fileName,
FileStream.Create)
    Dim wrtPersons As BinaryWriter = New BinaryWriter(fstPersons)
    wrtPersons.Write("James Bloch")
    wrtPersons.Write("Catherina Wallace")
    wrtPersons.Write("Bruce Lamont")
    wrtPersons.Write("Douglas Truth")
    wrtPersons.Close()
    fstPersons.Close()
End Sub
```

Stream Reading

As opposed to writing to a stream, you may want to read existing data from it. Before doing this, you can first specify your intent to the streaming class using the `FileMode` enumerator. This can be done using the `FileStream` class as follows:

```
Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnStart.Click
    Dim fileName As String = "Persons.spr"
    '
    Dim fstPersons As FileStream = New
FileStream(fileName, FileMode.Create)
    '
    Dim wrtPersons As BinaryWriter = New
BinaryWriter(fstPersons)
    '
    wrtPersons.Write("James Bloch")
    '
    wrtPersons.Write("Catherina Wallace")
    '
    wrtPersons.Write("Bruce Lamont")
    '
    wrtPersons.Write("Douglas Truth")
    '
    '
    '
    wrtPersons.Close()
    '
    fstPersons.Close()
    Dim fstPersons As New FileStream(fileName, FileMode.Open)
End Sub
```

Once the stream is ready, you can get prepared to read data from it. To support this, you can use the `BinaryReader` class. This class provides two constructors. One of the constructors (the first) has the following syntax:

```
Public Sub New(ByVal input As Stream)
```

This constructor takes as argument a `Stream` value, which could be a `FileStream` object. After declaring a `FileStream` variable using this constructor, you can read data from it. To support this, the class provides an appropriate method for each primitive data type.

After using the stream, you should close it to reclaim the resources it was using. This is done by calling the Close() method.

Here is an example of using the mentioned methods:

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

    Dim fileName As String = "Persons.spr"

    '          Dim fstPersons As FileStream = New
FileStream(fileName, FileMode.Create)

    '          Dim wrtPersons As BinaryWriter = New
BinaryWriter(fstPersons)

    '          wrtPersons.Write("James Bloch")
    '          wrtPersons.Write("Catherina Wallace")
    '          wrtPersons.Write("Bruce Lamont")
    '          wrtPersons.Write("Douglas Truth")
    '
    '          wrtPersons.Close()
    '          fstPersons.Close()

    Dim fstPersons As New FileStream(fileName, FileMode.Open)
    Dim rdrPersons As New BinaryReader(fstPersons)
    rdrPersons.Close()
    fstPersons.Close()

End Sub
```

Character and String Streaming

As mentioned earlier, the items to save on a document are primarily considered as bits. This makes it possible to save small pieces of information such as those that would fit a byte, a char, a short, or a double value. To support the ability to write one character at a time to a medium, the System.IO namespace provides the TextWriter class. This class is abstract, meaning you can't initialize a variable with it. Instead, you can use one of its derived classes, particularly the StreamWriter class.

Stream Writing

The operation used to save a character to a stream is performed through the StreamWriter class that is based on this. Therefore, to perform this operation, you can declare a pointer to StreamWriter or to its parent the TextWriter class. The StreamWriter class is equipped with various constructors. This allows you to initialize a stream with an option of your choice. For example, to initialize it using a FileStream object, you can use the following constructor:

```
Public Sub New(ByVal stm As Stream)
```

This constructor expects a Stream-type of object as argument. Here is an example of how this constructor can be used:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _

    Handles Button1.Click

    Dim stmRecords As New FileStream("Records.rcd",
FileMode.CreateNew, FileAccess.Write)
```

```

        Dim txtMemo As New StreamWriter(stmRecords)
End Sub

```

If you don't (yet) have a stream you can use, you can directly initialize a file using its name. To do this, you can use the following constructor:

```
Public Sub New(ByVal path As String)
```

When using this constructor, you can pass it the name of the file or its (complete) path.

Eventually, after using a Stream-based class, always remember to release its resources by calling its Close() method. Before calling Close(), call the class' Flush() method to clear the memory areas that the object was using when performing its operations.

After creating a character stream, you can write values to it. To support this, the StreamWriter class is equipped with the Write() method. This method is overloaded with four versions. One of the versions uses the following syntax:

```
Overrides Overloads Public Sub Write(ByVal value As Char)
```

This method expects as argument a character. This Write() method writes one character, then another immediate call to Write() writes the next character on the same line. This can be convenient in many cases. In some other cases, you may want each character to be written on its own line. To write each character on its own line, use the WriteLine() method instead. The WriteLine() method is inherited from the TextWriter class that is the parent of StreamWriter. It can be used as follows:

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

```

    Dim stmWeekdays As New FileStream("Weekdays.txt",
    FileMode.Create, FileAccess.Write, FileShare.None)
```

```
    Dim stmWriter As New StreamWriter(stmWeekdays)
```

```
    stmWriter.WriteLine("M")
```

```
    stmWriter.WriteLine("T")
```

```
    stmWriter.WriteLine("W")
```

```
    stmWriter.WriteLine("T")
```

```
    stmWriter.WriteLine("F")
```

```
    stmWriter.WriteLine("S")
```

```
    stmWriter.WriteLine("S")
```

```
    stmWriter.Flush()
```

```
    stmWeekdays.Flush()
```

```
    stmWeekdays.Close()
```

```
End Sub
```

Instead of writing one character at a time, you can write an array of characters to a stream. To do this, you would use the following version of the Write() method:

```
Overrides Overloads Public Sub Write(ByVal buffer() As Char)
```

This version takes as argument an array of Char values. This version would write a series of characters of the buffer argument on the same line or the same paragraph. If you make another call to this Write() method to add another series of characters, they would be added to the same line. If you want to write each array in its own

paragraph, you can call the `TextWriter.WriteLine()` method that the `StreamWriter` class inherits from its parent. Here is an example:

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

    Dim stmQuarter As New FileStream("Quarter.txt",
    FileMode.Create, FileAccess.Write, FileShare.None)

    Dim stmWriter As New StreamWriter(stmQuarter)

    Dim jan As Char() = {"J", "a", "n", "u", "a", "r", "y"}
    Dim feb As Char() = {"F", "e", "b", "r", "u", "a", "r", "y"}
    Dim mar As Char() = {"M", "a", "r", "c", "h"}

    stmWriter.WriteLine(jan)
    stmWriter.WriteLine(feb)
    stmWriter.WriteLine(mar)

    stmWriter.Flush()
    stmQuarter.Flush()
    stmQuarter.Close()

End Sub
```

Another way you can write characters to a stream is to proceed from a string. This operation is performed through the following version of the `Write()` method:

Overrides Overloads Public Sub Write(ByVal value As String)

This version expects a `String` pointer as argument and writes it wholly to the stream. Here is an example from a form equipped with a `Multiline TextBox` named `textBox1` and a `Button` control named `btnSave`. When the user clicks the `btnSave` button, each line, treated as a `String` value, of the text box is written to a stream:

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

    Dim stmRecords As New FileStream("Conserve.txt",
    FileMode.Create, FileAccess.Write, FileShare.None)

    Dim stmWriter As New StreamWriter(stmRecords)

    Dim i As Integer
    For i = 0 To Me.textBox1.Lines.Length - 1 Step 1
        stmWriter.WriteLine(Me.textBox1.Lines(i).ToString())
    Next
    stmWriter.Flush()
    stmWriter.Close()

End Sub
```

The last version of the `StreamWriter.Write()` method allows you also to write an array of characters to a stream but this time, you don't have to include the whole array, just a set number of characters of the array. In other words, you can select a section of the array and write it to the stream. This method uses the following syntax:

Overrides Overloads Public Sub Write(ByVal buffer() As Char, ByVal index As Integer, ByVal count As Integer)

This method expects an array of Char as the first argument. When writing the characters of this argument to a stream, the second argument specifies the starting character from the array. The last argument specifies the number of characters from the starting index to the end of the array, that would be written to stream.

Check Your Progress 2

Fill in the blanks:

1. To check the existence of a file, the File class provides the _____ method.
2. Most classes that are used to add values to a stream are equipped with a method called _____.

Stream Reading

As opposed to writing characters, you may want to read them from a stream. To perform this operation, you can use the StreamReader class, which is derived from the TextReader class. To read a character from a stream, the StreamReader class is equipped with the Read() method which is overloaded in two versions. One of the versions uses the following syntax:

Overrides Overloads Public Function Read() As Integer

This method reads one a character from the stream, checks the next character and returns it as an integer. If the next character has a value higher than -1, then the method reads it and advances to the next. Once, or when, the method finds out that the next character has a value of -1, then it stops. Here is an example that reads characters from a stream, one character at a time:

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

    Dim fileReader As New FileStream("Conserve.txt", FileMode.Open,
FileAccess.Read)

    Dim stmReader As New StreamReader(fileReader)

    Dim lstChars As New StringBuilder

    Dim iChar As Integer = stmReader.Read()

    lstChars.Append(Convert.ToChar(iChar))

    Do While stmReader.Peek() >= 0

        iChar = stmReader.Read()

        lstChars.Append(Convert.ToChar(iChar))

        Me.TextBox1.Text = lstChars.ToString()

    Loop

    stmReader.Close()

End Sub
```

As mentioned earlier, the Write() method of the StreamWriter class writes its values on the same line. This means that, when you read them back using the StreamReader.Read() method, all characters on the same line are read at once. We also mentioned that an alternative was to write each character on its own line using the TextWriter.WriteLine() method. In the same way, to read characters one line at a time, you can call the ReadLine() method. The TextReader.ReadLine() method returns a pointer to String and not a character. This returned value is actually a number that

represents the Unicode number of the character, such as 87 but if you access 87 using the `StreamReader.Read()` method, it would consider that 87 is a string and would therefore read only 8 as the character. This means that, if you had used the `StreamWriter.WriteLine()` method, you should also use the `StreamReader.ReadLine()` method to read the number as a string. After reading the line, before using the character, remember to cast it appropriately in order to retrieve the number stored in it. Here is an example that reads the characters written on the `Weekdays.txt` file we saved earlier:

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

    Dim stmWeekdays As New FileStream("Weekdays.txt",
    FileMode.Create, FileAccess.Write, FileShare.None)

    Dim stmWriter As New StreamWriter(stmWeekdays)
    stmWriter.WriteLine("M")
    stmWriter.WriteLine("T")
    stmWriter.WriteLine("W")
    stmWriter.WriteLine("T")
    stmWriter.WriteLine("F")
    stmWriter.WriteLine("S")
    stmWriter.WriteLine("S")
    stmWriter.Flush()
    stmWeekdays.Flush()
    stmWeekdays.Close()

End Sub

Private Sub btnOpen_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles btnOpen.Click

    Dim stmWeekdays As New FileStream("Weekdays.txt",
    FileMode.Open, FileAccess.Read)

    Dim stmReader As New StreamReader(stmWeekdays)
    Dim strDay As String = stmReader.ReadLine()
    Me.TextBox1.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox2.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox3.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox4.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox5.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox6.Text = strDay
    strDay = stmReader.ReadLine()
    Me.TextBox7.Text = strDay
    stmWeekdays.Flush()
```

```
stmWeekdays.Close()
```

```
End Sub
```

In the same way, if you had written characters to a stream using arrays of characters and you want to read each line or paragraph as an entity, you can call the `TextReader.ReadLine()` method. This time, you may not need to cast the returned value(s). Here is an example:

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

```
    Dim stmQuarter As New FileStream("Quarter.txt",
    FileMode.Create, FileAccess.Write, FileShare.None)
```

```
    Dim stmWriter As New StreamWriter(stmQuarter)
```

```
    Dim jan As Char() = {"J", "a", "n", "u", "a", "r", "y"}
```

```
    Dim feb As Char() = {"F", "e", "b", "r", "u", "a", "r", "y"}
```

```
    Dim mar As Char() = {"M", "a", "r", "c", "h"}
```

```
    stmWriter.WriteLine(jan)
```

```
    stmWriter.WriteLine(feb)
```

```
    stmWriter.WriteLine(mar)
```

```
    stmWriter.Flush()
```

```
    stmQuarter.Flush()
```

```
    stmQuarter.Close()
```

```
End Sub
```

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

```
    Dim stmQuarter As New FileStream("Quarter.txt", FileMode.Open,
    FileAccess.Read)
```

```
    Dim stmReader As New StreamReader(stmQuarter)
```

```
    Dim strMonth As String = stmReader.ReadLine()
```

```
    Me.TextBox1.Text = strMonth
```

```
    strMonth = stmReader.ReadLine()
```

```
    Me.TextBox2.Text = strMonth
```

```
    strMonth = stmReader.ReadLine()
```

```
    Me.TextBox3.Text = strMonth
```

```
    stmQuarter.Flush()
```

```
    stmQuarter.Close()
```

```
End Sub
```

After retrieving the characters from the array, instead of reading the whole line or paragraph, you may just want a selected number of characters. To read such a section of the line or paragraph, you can call the second version of the `StreamReader.Read()` method whose syntax is:

```
Overrides Overloads Public Function Read( _
```

```
<InteropServices.In(), _
```

```
Out()> ByVal buffer() As Char, _  
    ByVal index As Integer, _  
    ByVal count As Integer _  
) As Integer
```

The first argument of this method is the variable that will return an array of characters. The second argument is the index of the first character that will be considered from the array. The third argument is the number of characters that will be considered.

Check Your Progress 3

Fill in the blanks:

1. The parent class of file processing is _____.
2. The .NET Framework provides the _____ namespace to support file processing,
3. The sections of a complete path string are separated by a _____.

9.6 LET US SUM UP

Windows Explorer represents the contents of your hard drive graphically using file and folder icons. Many of the actions that you perform in Explorer (such as copying or renaming files) also can be accomplished in Visual Basic .NET via the System.IO namespace. VB.NET offers two handy classes—DirectoryInfo and FileSystemInfo—that allow you to access directories, check their properties, and perform other basic functions on the file system. To support file processing, the .NET Framework provides the System.IO namespace that contains many different classes to handle almost any type of file operation you may need to perform. The parent class of file processing is Stream. With Stream, you can store data to a stream or you can retrieve data from a stream. Stream is an abstract class, which means that you cannot use it to declare a variable in your application.

9.7 LESSON END ACTIVITY

Briefly discuss the file processing in VB.NET.

9.8 KEYWORDS

Windows Explorer: It represents the contents of your hard drive graphically using file and folder icons.

FileSystemInfo Class: It represents an entry in the file system, whether it is a file or a folder

FileInfo Class: It is a class to manipulate operating system files.

DirectoryInfo Class: It is a class that allows you to control the folders of your file system.

File Class: It is a class that contains static methods for performing operations on operating system files.

Directory Class: It is a class that contains static methods for performing operations on operating system folders.

File Path: It is a string that specifies the drive (such as A:, C:, or D:).

9.9 QUESTIONS FOR DISCUSSION

1. Compare and contrast between DirectoryInfo and FileSystemInfo class.
2. What is Binary File Streaming? Explain it.
3. Write short notes on
 - a) Directory Class
 - b) File Class
 - c) System.IO namespace

Check Your Progress: Model Answers

CYP 1

1. Windows Explorer
2. FileSystemInfo

CYP 2

1. Exists()
2. Write

CYP 3

1. Stream
2. System.IO
3. Backslash (/)

9.10 SUGGESTED READINGS

Shirish Chavan, *Visual Basic.Net*, Pearson edition, 2004

MSDN Visual studio Library.

Schneider, *An Introduction to Programming using Visual Basic.Net*, 5th Edition, PHI

Kant, *Visual Basic.Net– A Beginners Guide*, TMCH

UNIT IV

LESSON

10

VISUAL C++ PROGRAMMING

CONTENTS

- 10.0 Aims and Objectives
- 10.1 Introduction
- 10.2 MFC and Windows
- 10.3 MFC Fundamentals
- 10.4 MFC Class Hierarchy
- 10.5 MFC Member and Global Functions
 - 10.5.1 MFC Class Member Functions
 - 10.5.2 MFC Global Functions
 - 10.5.3 Some Important Global Functions
- 10.6 Let us Sum up
- 10.7 Lesson End Activity
- 10.8 Keywords
- 10.9 Questions for Discussion
- 10.10 Suggested Readings

10.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- An overview of MFC-MFC Hierarchy
- The concept of MFC Functions
- Brief idea about MFC Library
- Knowledge of MFC Resources

10.1 INTRODUCTION

Microsoft Visual C++ (referred to as VC++) is a software development environment that allows a programmer to write, compile, link, execute, test, and debug C++ programs. Visual C++ is a programming environment used to create computer applications for the Microsoft Windows family of operating systems. To assist it, the Microsoft Foundation Class Library, or MFC, was created as an adaptation of Win32 in MS Visual Studio. The MFC library is a set of data types, functions, classes, and constants used to create applications for the Microsoft Windows family of operating systems. You must understand the Microsoft Foundation Class (MFC) hierarchy. This class hierarchy encapsulates the user interface portion of the Windows API, and makes it significantly easier to create Windows applications in an object oriented way. This hierarchy is available for and compatible with all versions of Windows. The code you create in MFC is extremely portable.

This lesson introduces the fundamental concepts behind MFC and how to use the VC++ environment.

10.2 MFC AND WINDOWS

The MFC is a library and can be used to manually develop programs. To make the use of MFC friendlier, Microsoft developed Microsoft Visual C++. This is a graphical programming environment that allows designing Windows objects and writing code to implement their behavior. As its name indicates, this environment takes C++ as its base language. Fortunately, using MFC, it goes over some of the limitations of C++ and takes advantage of MFC's classes.

The basic functionality of a window is defined in a class called `CWnd`. An object created from `CWnd` must have a parent. In other words, it must be a secondary window, which is a window called from another, existing, window. Therefore, we will come back to it later.

The `CWnd` class gives birth to a class called `CFrameWnd`. This class actually describes a window or defines what a window looks like. The description of a window includes items such as its name, size, location, etc. To create a window using the `CFrameWnd` class, you must create a class derived from the `CFrameWnd` class.

As in real world, a Microsoft Windows window can be identified on the monitor screen because it has a frame. To create such a window, you can use the `CFrameWnd` class. `CFrameWnd` provides various alternatives to creating a window. For example, it is equipped with a method called `Create()`. The `Create()` method uses a set of arguments that define a complete window.

The syntax of the `CFrameWnd::Create()` method is as follows:

```
BOOL Create(LPCTSTR          lpszClassName,
            LPCTSTR          lpszWindowName,
            DWORD             dwStyle      = WS_OVERLAPPEDWINDOW,
            const RECT& rect  = rectDefault,
            CWnd*             pParentWnd   = NULL,
            LPCTSTR          lpszMenuName = NULL,
            DWORD             dwExStyle    = 0,
            CCreateContext*   pContext     = NULL );
```

As you can see, only two arguments are necessary to create a window, the others, though valuable, are optional.

As you should have learned from C++, every class in a program has a name. The name lets the compiler know the kind of object to create. The name of a class follows the conventions used for C++ names. That is, it must be a null-terminated string. Many classes used in Win32 and MFC applications are already known to the Visual C++ compiler. If the compiler knows the class you are trying to use, you can specify the `lpszClassName` as `NULL`. In this case, the compiler would use the characteristic of the `CFrameWnd` object as the class' name. Therefore, the `Create()` function can be implemented as follows:

```
Create(NULL);
```

Every object in a computer has a name. In the same way, a window must have a name. The name of a window is specified as the `lpszWindowName` argument of the `Create()` method. In computer applications, the name of a window is the one that displays on

the top section of the window. This is the name used to identify a window or an application. For example, if a window displays Basic Windows Application, then the object is called the "Basic Windows Application Window". Here is an example:

```
Create(NULL, "Basic Windows Application");
```



Figure 10.1: Basic Windows Application Window

In all of your windows, you should specify a window name. If you omit it, the users of your window would have difficulty identifying it. Here is an example of such a window:



Figure 10.2: A window without a name

After creating a window, to let the application use it, you can use a pointer to the frame class used to create the window. In this case, that would be (a pointer to) `CFrameWnd`. To use the frame window, assign its pointer to the `CWinThread::m_pMainWnd` member variable. This is done in the `InitInstance()` implementation of your application.

Macros

When creating objects that derive directly or indirectly from `CObject`, such as `CFrameWnd`, you should let the compiler know that you want your objects to be dynamically created. To do this, use the `DECLARE_DYNCREATE` and the `IMPLEMENT_DYNCREATE` macros. The `DECLARE_DYNCREATE` macro is provided in the class' header file and takes as argument the name of your class. An example would be `DECLARE_DYNCREATE(CTheFrame)`. Before implementing the

class or in its source file, use the `IMPLEMENT_DYNCREATE` macro, passing it two arguments: the name of your class and the name of the class you derived it from.

We mentioned that, to access the frame, a class derived from `CFrameWnd`, from the application class, you must use a pointer to the frame's class. On the other hand, if you want to access the application, created using the `CWinApp` class, from the windows frame, you use the `AfxGetApp()` global function.

Windows Styles

Windows styles are characteristics that control such features as its appearance, its borders, its minimized, its maximized, or its resizing state, etc.

After creating a window, to display it to the user, you can apply the `WS_VISIBLE` style to it. Here is an example:

```
Create(NULL, "Windows Application", WS_VISIBLE);
```

The top section of a window has a long bar called the title bar. This is a section we referred to above and saw that it is used to display the name of the window. This bar is added with the `WS_CAPTION` value for the style argument. Here is an example:

```
Create(NULL, "Windows Application", WS_CAPTION);
```

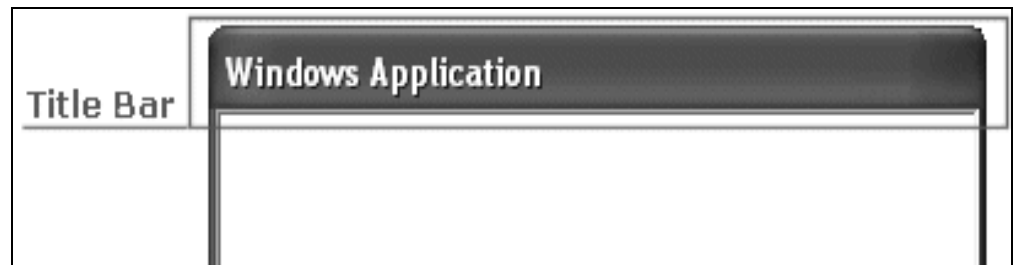


Figure 10.3: Title bar

A window is identified by its size which can be seen by its borders. Borders are given to a window through the `WS_BORDER` value for the style. As you can see from the above window, using the `WS_CAPTION` style also gives borders to a window.

A shortcut to creating a window that has a caption and a border is done by using the `WS_OVERLAPPED` style:

```
Create(NULL, "Windows Application", WS_OVERLAPPED);
```

As you will see shortly, various styles can be applied to the same window. To combine styles, you use the bitwise OR operator represented by the beam symbol `|`. For example, if you have a style called `WS_ONE` and you want to combine it with the `WS_OTHER` style, you can combine them as `WS_ONE | WS_OTHER`.

The title bars serves various purposes. For example, it displays the name of the window. In this case the window is called `Window Application`. The title bar also allows the user to move a window. To do this, the user would click and drag a section of the title bar while moving in the desired direction.

After using a window, the user may want to close it. This is made possible by using or adding the `WS_SYSMENU` style. This style adds a button with X that is called the System Close button. Here is an example:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU);
```

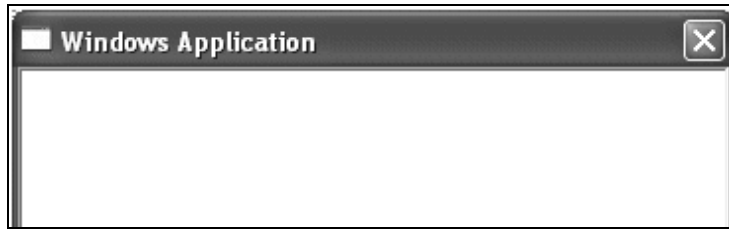


Figure 10.4: A Window with a close button

When a window displays, it occupies a certain area of the screen. To access other windows, for example to reveal a window hidden behind, the user can shrink the size of the window completely and make it disappear from the screen without closing the window. This is referred to as minimizing the window. The ability to minimize a window is made possible with the presence of a system button called Minimize. To allow this, you provide or add the `WS_MINIMIZEBOX` style:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU |  
WS_MINIMIZEBOX);
```

As opposed to minimizing the window, the user can change its size to use the whole area of the screen. This is referred to as maximizing the window. This feature is provided through the system Maximize button. To allow this functionality, provide or add the `WS_MAXIMIZEBOX` style. If you create a window with the `WS_SYSMENU` and the `WS_MINIMIZEBOX` style but not the `WS_MAXIMIZEBOX` style, the Maximize button would be disabled and the user would not be able to maximize the window:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU |  
WS_MINIMIZEBOX);
```



Figure 10.5: A Window with minimize maximize button

Therefore, if you want to provide all three system buttons, provide their styles.

You can also control whether the window appears minimized or maximized when it comes up. To minimize the window at startup, apply the `WS_MINIMIZE` style. On the other hand, to have a maximized window when it launches, use the `WS_MAXIMIZE` style. This style also assures that a window has borders. If you had not specified the `WS_CAPTION`, you can make sure that a window has borders using the `WS_BORDER` value.

One of the effects the user may want to control on a window is its size. For example, the user may want to narrow, enlarge, shrink, or heighten a window. To do this, the user would position the mouse on one of the borders, click and drag in the desired direction. This action is referred to as resizing a window. For the user to be able to change the size of a window, the window must have a special type of border referred to as a thick frame. To provide this border, apply or add the `WS_THICKFRAME` style:

```
Create(NULL, "Windows Application",
```

```
    WS_VISIBLE | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX  
    | WS_THICKFRAME);
```

Because many windows will need this functionality, a special style can combine them and it is called `WS_OVERLAPPEDWINDOW`. Therefore, you can create a resizable window as follows:

```
Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW);
```

Windows Location

To locate things that display on the monitor screen, the computer uses a coordinate system similar to the Cartesian's but the origin is located on the top left corner of the screen. Using this coordinate system, any point can be located by its distance from the top left corner of the screen of the horizontal and the vertical axes:

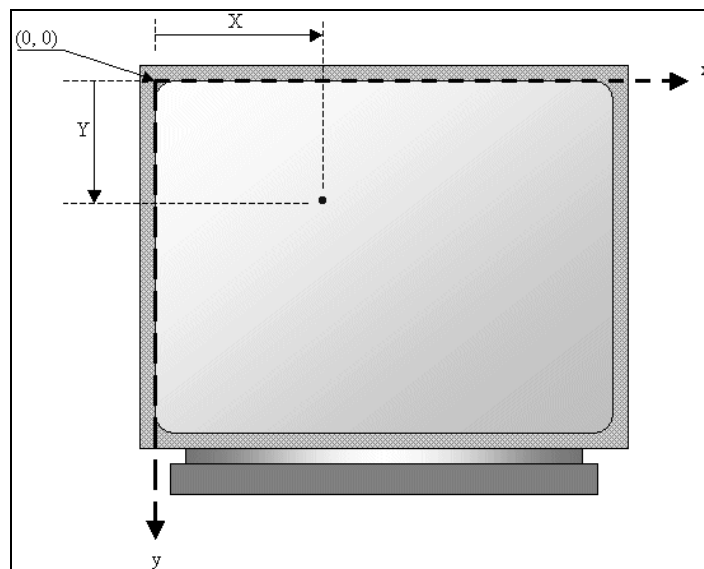


Figure 10.6: Window Location

To manage such distances, the operating system uses a point that is represented by the horizontal and the vertical distances. The Win32 library provides a structure called `POINT` and defined as follows:

```
typedef struct tagPOINT {  
    LONG x;  
    LONG y;  
} POINT;
```

The `x` member variable is the distance from the left border of the screen to the point. The `y` variable represents the distance from the top border of the screen to the point.

Besides the Win32's `POINT` structure, the Microsoft Foundation Classes (MFC) library provides the `CPoint` class. This provides the same functionality as the `POINT` structure. As a C++ class, it adds more functionality needed to locate a point.

The `CPoint::CPoint()` default constructor can be used to declare a point variable without specifying its location. If you know the `x` and `y` coordinates of a point, you can use the following constructor to create a point:

```
CPoint(int X, int Y);
```

While a point is used to locate an object on the screen, each window has a size. The size provides two measurements related to an object. The size of an object can be represented as follows:

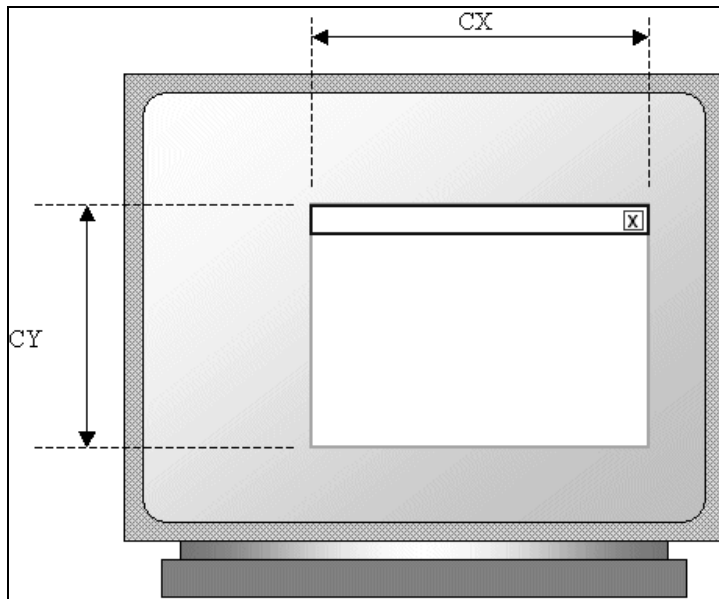


Figure 10.7: Window size

The width of an object, represented as CX, is the distance from its left to its right borders and is provided in pixels.

The height of an object, represented as CY is the distance from its top to its bottom borders and is given in pixels.

To represent these measures, the Win32 library uses the SIZE structure defined as follows:

```
typedef struct tagSIZE {  
    int cx;  
    int cy;  
} SIZE;
```

Besides the Win32's SIZE structure, the MFC provides the CSize class. This class has the same functionality as SIZE but adds features of a C++ class. For example, it provides five constructors that allows you to create a size variable in any way of your choice. The constructors are:

```
CSize();  
CSize(int initCX, int initCY);  
CSize(SIZE initSize);  
CSize(POINT initPt);  
CSize(DWORD dwSize);
```

The default constructor allows you to declare a CSize variable whose values are not yet available. The constructor that takes two arguments allows you to provide the width and the height to create a CSize variable. If you want another CSize or a SIZE

variables, you can use the `CSize(SIZE initSize)` constructor to assign its values to your variable. You can use the coordinates of a `POINT` or a `CPoint` variable to create and initialize a `CSize` variable. When we study the effects of the mouse, we will know you can use the coordinates of the position of a mouse pointer. These coordinates can help you define a `CSize` variable.

Besides the constructors, the `CSize` class is equipped with different methods that can be used to perform various operations on `CSize` and `SIZE` objects. For example, you can add two sizes to get a new size. You can also compare two sizes to find out whether they are the same.

Windows Dimensions

When a window displays, it can be identified on the screen by its location with regards to the borders of the monitor. A window can also be identified by its width and height. These characteristics are specified or controlled by the `rect` argument of the `Create()` method. This argument is a rectangle that can be created through the Win32 `RECT` structure or the MFC's `CRect` class. First, you must know how Microsoft Windows represents a rectangle.

A rectangle is a geometric figure with four sides or borders: top, right, bottom, and left. A window is also recognized as a rectangle. Therefore, it can be represented as follows:

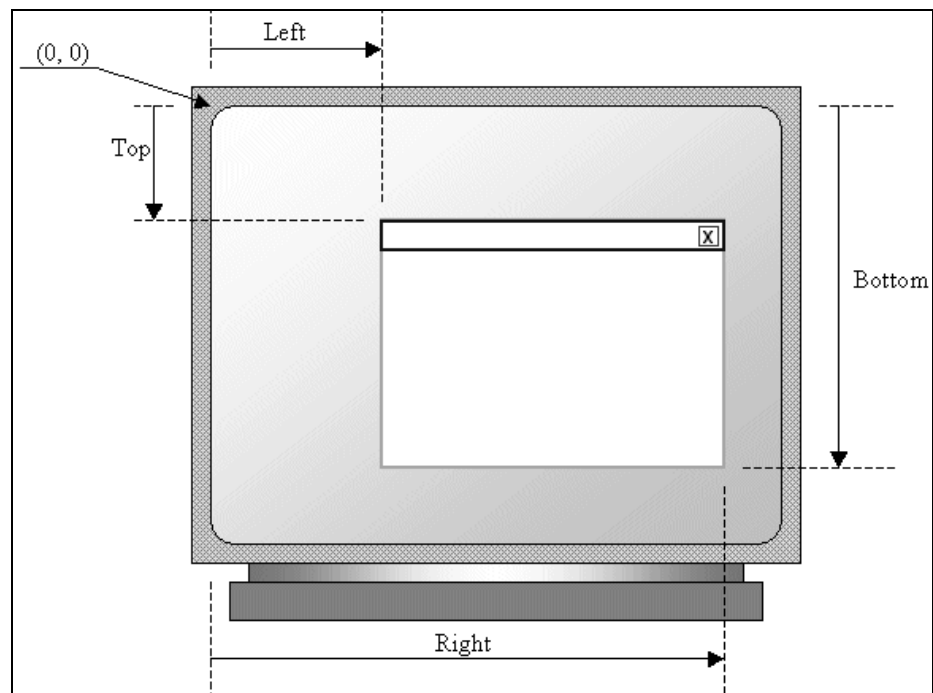


Figure 10.8: Window Dimension

Microsoft Windows represents items as on a coordinate system whose origin (0, 0) is located on the top-left corner of the screen. Everything else is positioned from that point. Therefore:

- the distance from the left border of the screen to left border of a window represents the left measurement
- the distance from the top border of the screen to the top border of a window is the top measurement.

- the distance from the left border of the screen to right border of a window represents the right measurement
- the distance from the top border of the screen to the bottom border of a window is the top measurement.

To represent these distances, the Win32 API provides a structure called RECT and defined as follows:

```
typedef struct _RECT {  
    LONG left;  
    LONG top;  
    LONG right;  
    LONG bottom;  
} RECT, *PRECT;
```

This structure can be used to control the location of a window on a coordinate system. It can also be used to control or specify the dimensions of a window. To use it, specify a natural numeric value for each of its member variables and pass its variable as the rect argument of the Create() method. Here is an example:

```
RECT Recto;  
Recto.left    = 100;  
Recto.top     = 120;  
Recto.right   = 620;  
Recto.bottom  = 540;  
  
Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW, Recto);
```

Besides the Win32 RECT structure, the MFC provides an alternate way to represent a rectangle on a window. This is done as follows:

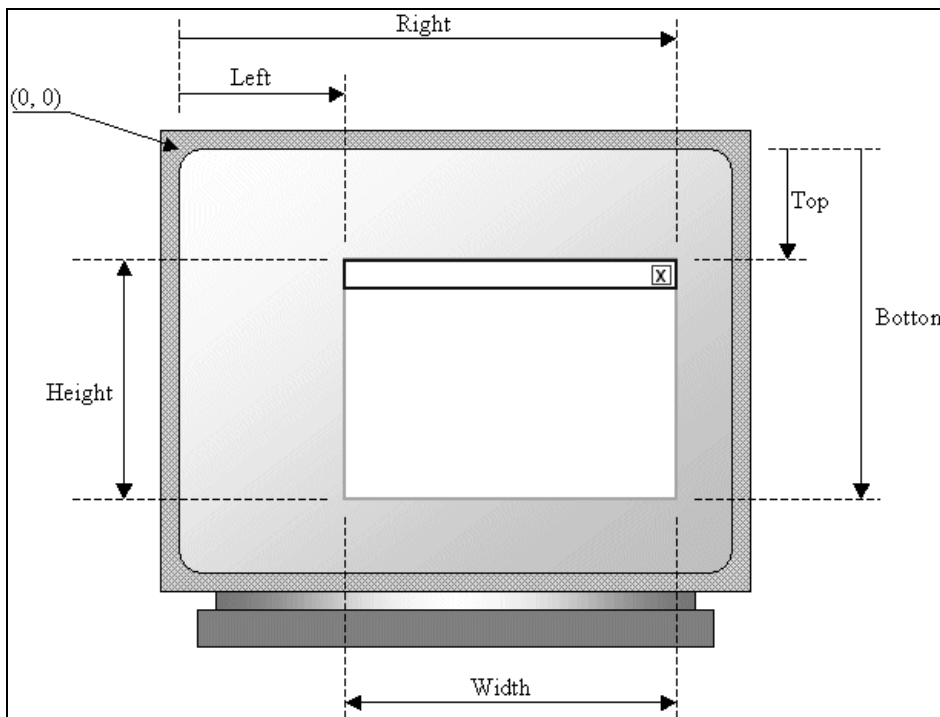


Figure 10.9: A window is also recognized for its width and its height

Besides the left, top, right, and bottom measurements, a window is also recognized for its width and its height. The width is the distance from the left to the right borders of a rectangle. The height is the distance from the top to the bottom borders of a rectangle. To recognize all these measures, the Microsoft Foundation Classes library provides a class called `CRect`. This class provides various constructors for almost any way of representing a rectangle. The `CRect` class provides the following constructors:

```
CRect();  
CRect(int l, int t, int r, int b);  
CRect(const RECT& srcRect);  
CRect(LPCRECT lpSrcRect);  
CRect(POINT point, SIZE size);  
CRect(POINT topLeft, POINT bottomRight);
```

The default construct is used to declare a `CRect` variable whose dimensions are not known. On the other hand, if you know the left, the top, the right, and the bottom measures of the rectangle, as seen on the `RECT` structure, you can use them to declare and initialize a rectangle. In the same way, you can assign a `CRect`, a `RECT`, a pointer to `CRect`, or a pointer to a `RECT` variable to create or initialize a `CRect` instance.

If you do not know or would not specify the location and dimension of a window, you specify the value of the `rect` argument as `rectDefault`. In this case, the compiler would use its own internal value for this argument.

Windows Parents

Many applications are made of different windows, as we will learn eventually. When an application uses various windows, most of these windows depend on a particular one. It could be the first window that was created or another window that you designated. Such a window is referred to as the parent window. All the other windows depend on it directly or indirectly.

If the window you are creating is dependent of another, you can specify that it has a parent. This is done with the `pParentWnd` argument of the `CFrameWnd::Create()` method. If the window does not have a parent, pass the argument with a `NULL` value. If the window has a parent, specify its name as the `pParentWnd` argument.

Check Your Progress 1

1. What is MFC library?

.....
.....

2. What is the parent class of `CFrameWnd`?

.....
.....

10.3 MFC FUNDAMENTALS

The classes in MFC are loosely organized into several major categories. Of these, the two major categories are Application Architecture classes and Window Support

classes. Other categories contain classes that encapsulate a variety of system, GDI, or miscellaneous services.

Most classes in MFC are derived from a common root, the `CObject` class. The `CObject` class implements two important features: serialization and run-time type information. (Note that the `CObject` class predates RTTI, the new C++ run-time type information mechanism; the `CObject` class does not use RTTI.) However, there are several simple support classes that are not derived from `CObject`.

The CObject Class and Serialization

The `CObject` class, the "mother of all classes" (well, almost) implements serialization and run-time type information.

Serialization is the conversion of an object to and from a persistent form. Or, in simpler terms, it means writing an object to disk or reading it from the disk or any other forms of persistent storage.

Why is serialization necessary? Why not just write

```
cout << myObject;
```

and get it over with? For one thing, everybody knows that writing anything to disk that involves pointers can be tricky. When you later read that disk file, chances are that whatever your pointer pointed to has either been moved or is no longer present in memory at all. But this is not the end of the story.

MFC objects are not only written to disk files. Serialization is also used to place an object on the clipboard or to prepare the object for OLE embedding.

The MFC Library uses `CArchive` objects for serialization. A `CArchive` object represents persistent storage of some kind. When an object is about to be serialized, `CArchive` calls the object's `Serialize` member function, one of the overridable functions in `CObject`. Thus, the underlying philosophy is that it is the object that knows best how to prepare itself for persistent storage, while it is the `CArchive` object that knows how to transfer the resulting data stream to persistent media.

But let an example do the talking. This example implements something simple, a string class `CMyString`. (Note that this has nothing to do with the sophisticated MFC `CString` class; the sole purpose of this exercise is to demonstrate `CObject` serialization.)

`CMyString` has two data members; one represents the length of the string, the other is a pointer to the string data. Unlike C strings, a `CMyString` can contain embedded null characters and does not require a terminating null character. The declaration of the `CMyString` class would thus look like this (only the data members and the `Serialize` member function are shown):

```
class CMyString
{
private:
    WORD m_nLength;
    LPSTR m_pString;
public:
    virtual void Serialize(CArchive &ar);
};
```

Why the Windows type WORD is used instead of declaring m_nLength an integer? There is a very important reason. Windows guarantees that the WORD type will represent a 16-bit integer on all present and future versions of Windows. This is important when it comes to storing data on persistent storage; it ensures that data files written under one operating system specific version of our application remain readable under another. Had we used int instead, we would be facing the problem that an int is a 16-bit type under Windows 3.1, a 32-bit type under Windows 95 and Windows NT, and who knows what under future versions of Windows. Thus, data files created under these different operating systems would be incompatible.

The Serialize member function is responsible for actually writing data to, and reading data from, a CArchive object. However, we cannot simply just write m_nLength and m_pString to the archive. Instead, we have to write the data m_pString points to, that is, the string itself. When it comes to reading the data, we must first determine the length of the string, allocate memory for it, and then read the string itself:

```
CMYString::Serialize(CArchive &ar)
{
    if (ar.IsStoring())
    {
        ar << m_nLength;
        ar.Write(m_pString, m_nLength);
    }
    else
    {
        ar >> m_nLength;
        m_pString = new char[m_nLength];
        ar.Read(m_pString, m_nLength);
    }
}
```

In order for this code to compile and run correctly, it is also necessary to use a few helper macros. For a class to be serializable, one must use the DECLARE_SERIAL macro in the class declaration and the IMPLEMENT_SERIAL macro somewhere in the class's implementation file. One specific feature that these macros add to a class is MFC run-time type information.

Why is type information necessary for successful serialization? Well, consider what happens when data is read from persistent storage. Before reading an object, we know nothing about it other than the fact that it is CObject derived. Run-time type information that has been serialized together with the object helps to determine the actual type of the object.

Once type information has been obtained, the CArchive object can create an object of the new type and call its Serialize member function to read in object-specific data. Without run-time type information this would not be possible.

Run-Time Type Information

MFC maintains run-time type information with the help of the CRuntimeClass class and several helper macros.

The `CRuntimeClass` class has member variables holding the name of the class and the size of an object belonging to that class. This information not only identifies the class but also assists in serialization.

Applications rarely use `CRuntimeClass` directly. Instead, they rely on a series of macros that embed a `CRuntimeClass` object in the declaration of a `CObject`-derived class, and provide an implementation.

There are three pairs of macros, summarized in the following table.

Symbolic constant	Description
<code>DECLARE_DYNAMIC</code> and <code>IMPLEMENT_DYNAMIC</code>	Adds run-time information to the class. Enables the use of the <code>IsKindOf</code> member function.
<code>DECLARE_DYNCREATE</code> and <code>IMPLEMENT_DYNCREATE</code>	Renders the class dynamically creatable through <code>CRuntimeClass::CreateObject</code> .
<code>DECLARE_SERIAL</code> and <code>IMPLEMENT_SERIAL</code>	Adds serialization capability to the class; enables the use of <code><<</code> and <code>>></code> operators with a <code>CArchive</code>

You only need to use one set of these macros at any time. The functionality of `DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE` is a superset of the functionality of `DECLARE_DYNAMIC/IMPLEMENT_DYNAMIC`; and the functionality of `DECLARE_SERIAL/IMPLEMENT_SERIAL` is a superset of the functionality of `DECLARE_DYNAMIC/IMPLEMENT_DYNAMIC`.

To use these macros, you embed the `DECLARE_` macro in the declaration of your class and you add the `IMPLEMENT_` macro to your implementation file. To create a `CMyString` class that is `CObject`-derived and supports serialization, you would therefore declare the class as follows:

```
class CMyString : public CObject
{
    DECLARE_SERIAL(CMyString)
    ...
};
```

In the implementation file, you would add the following macro (outside any member functions):

```
IMPLEMENT_SERIAL(CMyString, CObject, 0)
```

MFC and Multiple Inheritance

A frequently asked question with respect to MFC concerns the use of the classes of MFC with multiple inheritance. Generally, the answer is that although using multiple inheritance with MFC is possible, doing so is not recommended.

In particular, the `CRuntimeClass` class does not support multiple inheritance. As `CRuntimeClass` is used by `CObject` for run-time class information, dynamic object creation, and serialization, this limitation has a serious effect on any attempt to use multiple inheritance in an MFC program.

10.4 MFC CLASS HIERARCHY

MFC is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. MFC is compatible across the entire Windows family. As each new version of Windows comes out, MFC gets modified so that old code compiles and works under the new system. MFC also gets extended, adding new capabilities to the hierarchy and making it easier to create complete applications.

The advantage of using MFC and C++ - as opposed to directly accessing the Windows API from a C program-is that MFC already contains and encapsulates all the normal "boilerplate" code that all Windows programs written in C must contain. Programs written in MFC are therefore much smaller than equivalent C programs. On the other hand, MFC is a fairly thin covering over the C functions, so there is little or no performance penalty imposed by its use. It is also easy to customize things using the standard C calls when necessary since MFC does not modify or hide the basic structure of a Windows program.

The best part about using MFC is that it does all of the hard work for you. The hierarchy contains thousands and thousands of lines of correct, optimized and robust Windows code. Many of the member functions that you call invoke code that would have taken you weeks to write yourself. In this way MFC tremendously accelerates your project development cycle.

MFC is fairly large. For example, Version 4.0 of the hierarchy contains something like 200 different classes. Fortunately, you don't need to use all of them in a typical program. In fact, it is possible to create some fairly spectacular software using only ten or so of the different classes available in MFC. The hierarchy is broken into several different class categories which include (but is not limited to):

- Application Architecture
- Graphical Drawing and Drawing Objects
- File Services
- Exceptions
- Structures - Lists, Arrays, Maps
- Internet Services
- OLE 2
- Database
- General Purpose

Most classes in MFC derive from a base class called CObject. This class contains data members and member functions that are common to most MFC classes. The second thing to notice is the simplicity of the list. The CWinApp class is used whenever you create an application and it is used only once in any program. The CWnd class collects all the common features found in windows, dialog boxes, and controls. The CFrameWnd class is derived from CWnd and implements a normal framed application window. CDialog handles the two normal flavors of dialogs: modeless and modal respectively. CView is used to give a user access to a document through a window. Finally, Windows supports six native control types: static text, editable text, push buttons, scroll bars, lists, and combo boxes (an extended form of list). Once you understand this fairly small number of pieces, you are well on your way to a complete understanding of MFC. The other classes in the MFC hierarchy implement other

features such as memory management, document control, data base support, and so on.

To create a program in MFC, you either use its classes directly or, more commonly, you derive new classes from the existing classes. In the derived classes you create new member functions that allow instances of the class to behave properly in your application. Both CHelloApp and CHelloWindow are derived from existing MFC classes.

10.5 MFC MEMBER AND GLOBAL FUNCTIONS

10.5.1 MFC Class Member Functions

- Most functions called by an app will be members of an MFC class
- Examples:

 ShowWindow()--a member of CWnd class

 TextOut()--a member of CDC

 LoadBitmap()--a member of CBitmap

- Applications can also call API functions directly
- Use Global Scope Resolution Operator (::), for
- Example:

 UpdateWindow(hWnd);

- Usually more convenient to use MFC member functions

10.5.2 MFC Global Functions

- Not members of any MFC class
- Always begin with Afx prefix (Application FrameworkS)
- Independent of or span MFC class hierarchy
- Example:

 AfxMessageBox()--

 Message boxes are predefined windows

 Can be activated independently from rest an application

10.5.3 Some Important Global Functions

- ❖ AfxAbort () -- uconditionally terminate an app
- ❖ AfxBeginThread() -- Create & run a new thread
- ❖ AfxGetApp() -- Returns a pointer to the application object
- ❖ AfxGetMainWnd() -- Returns a pointer to application's main window
- ❖ AfxGetInstanceHandle() -- Returns handle to applications's current instance
- ❖ AfxRegisterWndClass() -- Register a custom WNDCLASS for an MFC app

Check Your Progress 2

Fill in the blanks:

1. The basic functionality of a window is defined in a class called _____.
2. The _____ class is known as the "mother of all classes".
3. MFC maintains run-time type information with the help of the _____.

10.6 LET US SUM UP

Microsoft Visual C++ (referred to as VC++) is a software development environment that allows a programmer to write, compile, link, execute, test, and debug C++ programs. Visual C++ is a programming environment used to create computer applications for the Microsoft Windows family of operating systems. The MFC library is a set of data types, functions, classes, and constants used to create applications for the Microsoft Windows family of operating systems. The basic functionality of a window is defined in a class called CWnd. Windows styles are characteristics that control such features as its appearance, its borders, its minimized, its maximized, or its resizing state, etc. The classes in MFC are loosely organized into several major categories. Of these, the two major categories are Application Architecture classes and Window Support classes. Other categories contain classes that encapsulate a variety of system, GDI, or miscellaneous services. c. Or, in simpler terms, it means writing an object to disk or reading it from the disk or any other forms of persistent storage. MFC is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. MFC is compatible across the entire Windows family. As each new version of Windows comes out, MFC gets modified so that old code compiles and works under the new system. MFC supports property sheets through two classes: CPropertySheet and CPropertyPage. Resource is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc. Resource files define the visual appearance of an application. Resources include dialogs, menus, bitmaps, icons, cursors, text strings, and several other types.

10.7 LESSON END ACTIVITY

1. Design the form as follows:

The image shows a Windows-style dialog box titled "Solas Property Rental - New Property". It contains the following fields and controls:

- Property #:** A text input field.
- Property Type:** A dropdown menu currently showing "Unknown".
- Property Condition:** A dropdown menu currently showing "Unknown".
- Bedrooms:** A text input field containing the value "0".
- Bathrooms:** A text input field containing the value "0.00".
- Monthly Rent:** A text input field containing the value "0.00".
- Buttons:** "OK" and "Cancel" buttons are located on the right side of the dialog.

10.8 KEYWORDS

MFC: Microsoft Foundation Class is a library and can be used to manually develop programs.

Microsoft Visual C++: It is also referred to as VC++; a software development environment that allows a programmer to write, compile, link, execute, test, and debug C++ programs.

CWnd: It is a class where the basic functionality of a window is defined.

CFrameWnd: This is a class which actually describes a window or defines what a window looks like.

10.9 QUESTIONS FOR DISCUSSION

1. Discuss the use of the classes of MFC with multiple inheritance.
2. What is the syntax of the CFrameWnd::Create() method?
3. How to access the frame?
4. How to locate things that display on the monitor screen?
5. How MFC maintains run-time type information?

Check Your Progress: Model Answers

CYP 1

1. The MFC library is a set of data types, functions, classes, and constants used to create applications for the Microsoft Windows family of operating systems.
2. CWnd

CYP 2

1. CWnd
2. CObject
3. CRuntimeClass

10.10 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up*, 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth, *Visual C++6 Unleashed*, Second edition, Techmedia.

LESSON

11

OBJECT PROPERTIES

CONTENTS

- 11.0 Aims and Objectives
- 11.1 Introduction
- 11.2 Various Object Properties
 - 11.2.1 Constructing Property Pages
 - 11.2.2 Adding a Property Sheet Object
 - 11.2.3 CPropertyPage Member Functions
 - 11.2.4 Modeless Property Sheets
- 11.3 MFC Library
 - 11.3.1 CObject
 - 11.3.2 CArchive
 - 11.3.3 CWinApp
 - 11.3.4 CWnd
 - 11.3.5 CFile
 - 11.3.6 CGDIObject
 - 11.3.7 CExcept
 - 11.3.8 CDialog
 - 11.3.9 CString
 - 11.3.10 CEdit
 - 11.3.11 CList
- 11.4 Let us Sum Up
- 11.5 Lesson End Activity
- 11.6 Keywords
- 11.7 Questions for Discussion
- 11.8 Suggested Readings

11.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- The concept of object properties
- Overview of MFC Library

11.1 INTRODUCTION

MFC stands for Microsoft Foundation Class library. In its core is a collection of libraries, a set of data types, functions, classes, and constants used to create applications for the Microsoft Windows family of operating systems. Thus it was created by Microsoft to make writing applications for Windows easier and also for spreading the popularity of the operating system. The basic of MFC and the Visual C++ development environment is explained in the previous lesson. In this lesson the MFC library will be viewed thoroughly.

11.2 VARIOUS OBJECT PROPERTIES

A Property is a piece of information that is associated with an object. Storing and retrieving all of the properties belonging to an object through a single interface provides a convenient and generic way to deal with any property.

Property sheets are several overlapping dialogs in one. The user selects one of the dialogs, or property pages, by clicking on the corresponding tab in a tab control.

MFC supports property sheets through two classes: `CPropertySheet` and `CPropertyPage`. `CPropertySheet` corresponds to the property sheet; `CPropertyPage`-derived classes correspond to the individual property pages within the property sheet.

Using a property sheet requires several steps. First, the property pages must be constructed; next, the property sheet must be created.

The following simple example reviews this process. A new application, PRP, is used to display a property sheet, as shown in Figure 11.1. Like our earlier application, DLG, PRP is also a standard SDI application created by AppWizard.

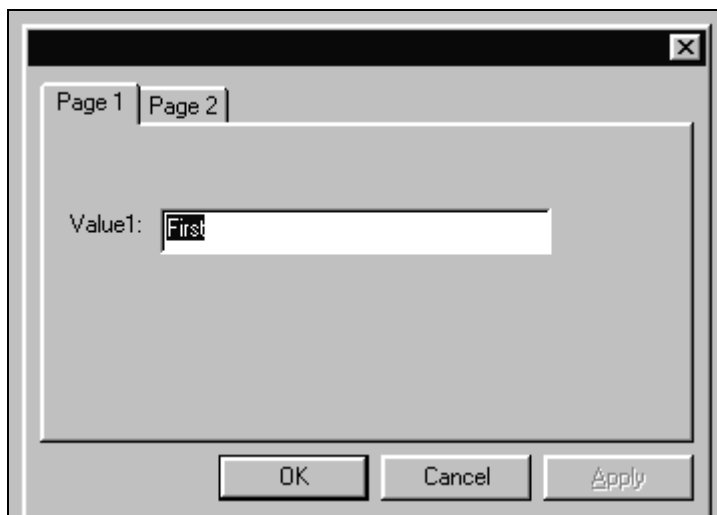


Figure 11.1: Property Sheet

11.2.1 Constructing Property Pages

Constructing a property page is very similar to constructing dialogs. The first step is to construct the dialog template resource for every property page that you wish to add to the property sheet.

There are a few special considerations when constructing a dialog template resource for a property page object:

The dialog's caption should be set to the text that you wish to see appear in the tab corresponding to the dialog.

- The dialog's style should be set to child.
- The dialog's border style should be set to thin.
- The Titlebar style should be checked.
- The Disabled style should be checked.

Although the property pages in a property sheet will overlap, it is not necessary to create them with the same size. The MFC Library will automatically adjust the size of property pages to match the size of the largest property page.

In this example we construct two property pages for our application—nothing fancy, just a simple text field in both of them. The first property page, titled "Page 1," is shown in Figure 11.2. To insert a blank property page template similar to the one shown here, use the IDR_PROPPAGE_SMALL subtype of the Dialog resource type in the Insert Resource dialog. Afterwards, you can add the controls as shown.

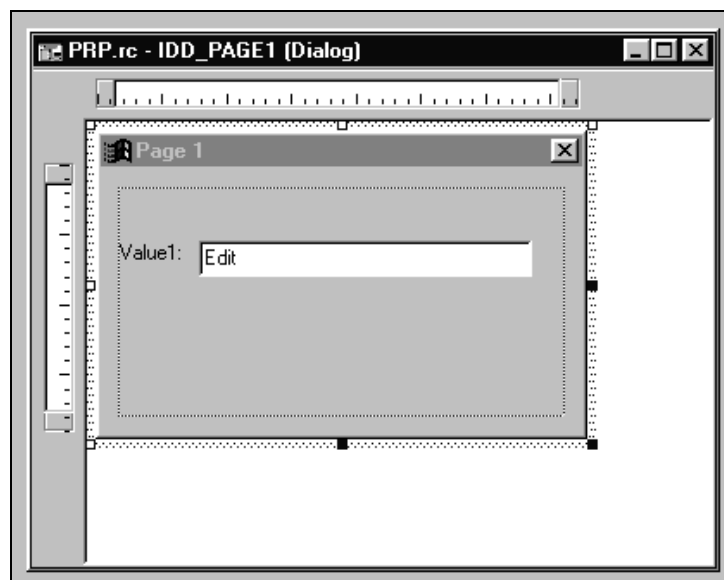


Figure 11.2: The first property page, titled "Page 1"

The identifier of the dialog template resource should be set to IDD_PAGE1; the identifier of the edit control should be set to IDC_EDIT1. Make sure that the dialog template's properties are set correctly. To set the dialog's caption, double click on the dialog to invoke the Dialog Properties property sheet (Figure 11.3).

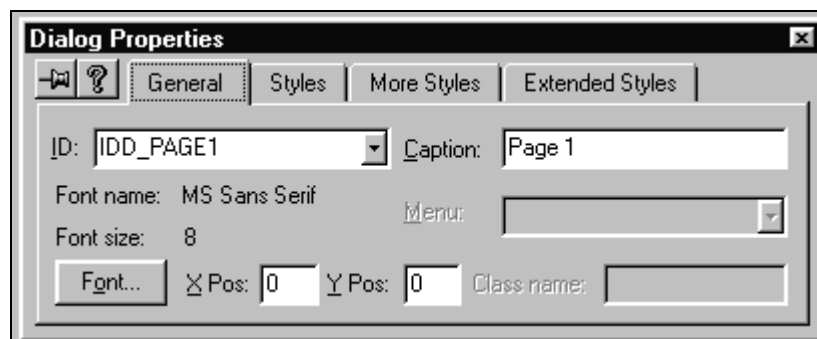


Figure 11.3: Dialog Properties property sheet

To set the style, border style, and titlebar setting, select the Styles tab in the property sheet of the dialog resource (Figure 11.4).

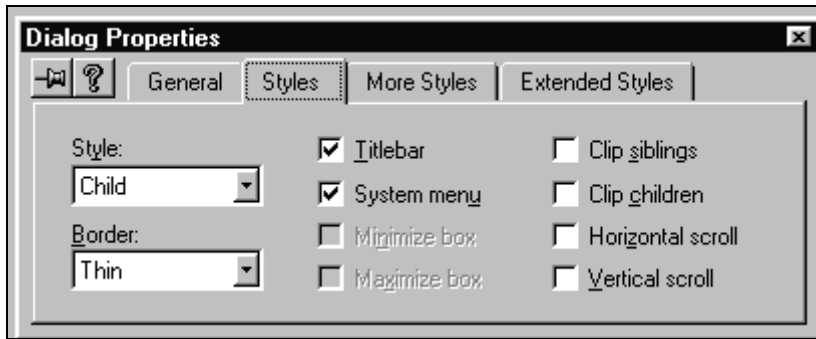


Figure 11.4: Property page dialog resource styles

To set the Disabled style of the dialog resource, use the More Styles tab in the dialog resource property sheet (Figure 11.5).



Figure 11.5: Setting the property page dialog resource to Disabled

The second property page in our simple example is, for the sake of simplicity, nearly identical to the first. In fact, you can create the dialog resource for this second property page by simply making a copy of the first. Make sure that the identifier of the new dialog resource is set to `IDD_PAGE2` and that the identifier of the edit control within it is `IDC_EDIT2`. (It would be perfectly legal to use the same identifier for controls in separate property pages; they act and behave like separate dialogs. Nevertheless, I prefer to use distinct identifiers; this helps reduce the possibility for errors.)

Once both property page dialog resources have been constructed, it is time to invoke the ClassWizard and construct classes that correspond to these property pages. To do so, invoke the ClassWizard while the focus is on the first property page dialog resource while it is open for editing. As with dialogs, the ClassWizard will recognize that the dialog template has no corresponding class and offer you the opportunity to create a new class.

In the Create New Class dialog, specify a name for the class corresponding to the dialog template (for example, `CMyPage1`). More importantly, make sure that this new class is based on `CPropertyPage` (and not the default `CDialog`). Once the correct settings have been entered, create the class.

While in ClassWizard, you should also add a member variable that corresponds to the edit control in the property page. Name this variable `m_sEdit1`.

These steps should be repeated for the second property page. The class for this property page should be named CMyPage2, and the member variable corresponding to its edit control should be named m_sEdit2.

Construction of our property pages is now complete. Take a brief look at the code generated by ClassWizard. The declaration of CMyPage1 is shown in the following listing (the declaration of CMyPage2 is virtually identical).

CMyPage1 declaration.

```
class CMyPage1 : public CPropertyPage
{
    DECLARE_DYNCREATE(CMyPage1)
// Construction
public:
    CMyPage1();
    ~CMyPage1();
// Dialog Data
   //{{AFX_DATA(CMyPage1)
    enum { IDD = IDD_PAGE1 };
    CString m_sEdit1;
    }//}}AFX_DATA
// Overrides
    // ClassWizard generate virtual function overrides
   //{{AFX_VIRTUAL(CMyPage1)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    }//}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
   //{{AFX_MSG(CMyPage1)
        // NOTE: the ClassWizard will add member functions here
    }//}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

As you can see, there is very little difference between this declaration and the ClassWizard-generated declaration of a CDialog-derived dialog class. Most importantly, CPropertyPage-derived classes can use Dialog Data Exchange functions just as classes derived from CDialog.

The implementation of CMyPage1 member functions is also no different from the implementation of similar functions in a CDialog-derived class. Perhaps the one notable difference is that this class has been declared as dynamically creatable with the help of the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros.

CMyPage1 implementation.

```
IMPLEMENT_DYNCREATE(CMyPage1, CPropertyPage)

CMyPage1::CMyPage1() : CPropertyPage(CMyPage1::IDD)
{
   //{{AFX_DATA_INIT(CMyPage1)
    m_sEdit1 = _T("");
   //}}AFX_DATA_INIT
}

CMyPage1::~CMyPage1()
{
}

void CMyPage1::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CMyPage1)
    DDX_Text(pDX, IDC_EDIT1, m_sEdit1);
   //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CMyPage1, CPropertyPage)
   //{{AFX_MSG_MAP(CMyPage1)
    // NOTE: the ClassWizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

As its declaration, the implementation of CMyPage2 is virtually identical to that of CMyPage1.

11.2.2 Adding a Property Sheet Object

Now that the property pages have been constructed, the one remaining task is to create the property sheet. Again, we need to invoke the new property sheet when the user selects a new menu command, Property Sheet, from the application's View menu. Add this command to the menu using the resource editor, and invoke the ClassWizard to add a corresponding member function, CMainFrame::OnViewPropertySheet, to the CMainFrame class.

In this member function, we have to perform a series of tasks. First, a property sheet object must be constructed. Next, the property pages must be added to it using the AddPage member function; and finally, the property sheet must be invoked using the DoModal member function.

The following listing contains the implementation of CMainFrame::OnViewPropertySheet that performs all these tasks.

The CMainFrame::OnViewPropertySheet function.

```
void CMainFrame::OnViewPropertySheet()
{
    // TODO: Add your command handler code here
```

```
CPropertySheet myPropSheet;  
CMyPage1 myPage1;  
CMyPage2 myPage2;  
myPage1.m_sEdit1 = "First";  
myPage2.m_sEdit2 = "Second";  
myPropSheet.AddPage(&myPage1);  
myPropSheet.AddPage(&myPage2);  
myPropSheet.DoModal();  
}
```

Do not forget to include the header files `MyPage1.h` and `MyPage2.h` in `MainFrm.cpp`; otherwise, you will not be able to declare objects of type `CMyPage1` or `CMyPage2` and the function in the above listing will not compile.

At this time, the application is ready to be compiled and run.

Although in this example we made no use of the property page member variables after the property sheet is dismissed, we could access them simply through the property page objects `myPage1` and `myPage2`.

11.2.3 CPropertyPage Member Functions

Our simple example did not utilize many of the advanced capabilities of the `CPropertyPage` class.

For example, in a more realistic application, you may wish to override the `CancelToClose` member function whenever a change is made to a property page. This member function changes the OK button to Close and disables the Cancel button in the property sheet. This function is best used after an irreversible change has been made in a property page.

Another frequently used property page function is the `SetModified` function. This function can be used to enable the Apply Now button in the property sheet.

Other property page overridables include `OnOK` (called when the OK, Apply Now, or Close button is clicked in the property sheet), `OnCancel` (called when the cancel button is clicked in the property sheet), and `OnApply` (called when the Apply Now button is clicked in the property sheet).

Property sheets can also be used to implement wizard-like behavior; that is, behavior similar to the behavior of the ubiquitous wizards that can be found in many Microsoft applications. Wizard mode can be enabled by calling the `SetWizardMode` member function of the property sheet; in the property pages, override the `OnWizardBack`, `OnWizardNext`, and `OnWizardFinish` member functions.

11.2.4 Modeless Property Sheets

Using the `DoModal` member function of a property sheet implies modal behavior. As is the case with dialogs, it is also possible to implement a modeless property sheet.

To accomplish this, it is first of all necessary to derive our own property sheet class. This is important because at the very least, we must override its `PostNcDestroy` member function to ensure that objects of this class are destroyed when the modeless property sheet is dismissed.

The new property sheet class can be created using ClassWizard. Create a new class derived from CPropertySheet, and name it CMySheet. While in ClassWizard, add the PostNcDestroy member function.

The declaration of CMySheet (in the file MySheet.h), as generated by ClassWizard, is shown in the listing below.

CMySheet declaration.

```
class CMySheet : public CPropertySheet
{
    DECLARE_DYNAMIC(CMySheet)
// Construction
public:
    CMySheet(UINT nIDCaption, CWnd* pParentWnd = NULL,
             UINT iSelectPage = 0);
    CMySheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL,
             UINT iSelectPage = 0);
// Attributes
public:
// Operations
public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CMySheet)
protected:
    virtual void PostNcDestroy();
    //}AFX_VIRTUAL
// Implementation
public:
    virtual ~CMySheet();
    // Generated message map functions
protected:
    //{AFX_MSG(CMySheet)
        // NOTE - the ClassWizard will add and remove member
        // functions here.
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

In the implementation file, MySheet.cpp, it is necessary to modify the PostNcDestroy member function to destroy not only the property sheet object, but also any property pages associated with it. The implementation of this function, together with other, ClassWizard-supplied member function implementations for the CMySheet class, is shown in the listing below.

```

CMySheet declaration.

// CMySheet
IMPLEMENT_DYNAMIC(CMySheet, CPropertySheet)

CMySheet::CMySheet(UINT nIDCaption, CWnd* pParentWnd,
                  UINT iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
}

CMySheet::CMySheet(LPCTSTR pszCaption, CWnd* pParentWnd,
                  UINT iSelectPage)
    :CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
}

CMySheet::~CMySheet()
{
}

BEGIN_MESSAGE_MAP(CMySheet, CPropertySheet)
    //{{AFX_MSG_MAP(CMySheet)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// CMySheet message handlers
void CMySheet::PostNcDestroy()
{
    // TODO: Add your specialized code here and/or call the base class
    CPropertySheet::PostNcDestroy();
    for (int i = 0; i < GetPageCount(); i++)
        delete GetPage(i);
    delete this;
}

```

A modeless property sheet does not have OK, Cancel, and Apply Now buttons by default. If any buttons are required, these must be added by hand. We are not going to worry about these now; the modeless property sheet can still be dismissed by closing it through its control menu.

How is the modeless property sheet invoked? Obviously, we have to modify the `OnViewPropertysheet` member function in our `CMainFrame` class, as using `DoModal` is no longer appropriate. Nor is it appropriate to create the property sheet or any of its property pages on the stack, as we do not want them destroyed when the `OnViewPropertysheet` function returns. The new `OnViewPropertysheet` function is shown below.

Invoking a modeless property sheet

```
void CMainFrame::OnViewPropertySheet()  
{  
    // TODO: Add your command handler code here  
    CMySheet *pMyPropSheet;  
    CMyPage1 *pMyPage1;  
    CMyPage2 *pMyPage2;  
    pMyPropSheet = new CMySheet("");  
    pMyPage1 = new CMyPage1;  
    pMyPage2 = new CMyPage2;  
    pMyPage1->m_sEdit1 = "First";  
    pMyPage2->m_sEdit2 = "Second";  
    pMyPropSheet->AddPage(pMyPage1);  
    pMyPropSheet->AddPage(pMyPage2);  
    pMyPropSheet->Create();  
}
```

In order for CMainFrame::OnViewPropertySheet to compile in its new form, it is necessary to add the include file MySheet.h to MainFrm.cpp; otherwise, the attempt to declare an object of type CMySheet will fail.

Check Your Progress 1

Fill in the blanks:

1. _____ member function changes the OK button to Close and disables the Cancel button in the property sheet.
2. A _____ property sheet does not have OK, Cancel, and Apply buttons by default.

11.3 MFC LIBRARY

The Microsoft Foundation Class Library (also Microsoft Foundation Classes or MFC) is a library that wraps portions of the Windows API in C++ classes, including functionality that enables them to use a default application framework. Classes are defined for many of the handle-managed Windows objects and also for predefined windows and common controls.

The following classes are included in the MFC Library.

11.3.1 CObject

CObject is the root base class for most of the Microsoft Foundation Class Library (MFC). It serves as the root not only for library classes such as CFile and CObList, but also for the classes that you write. The CObject class contains many useful features that you may want to incorporate into your own program objects, including serialization support, run-time class information, and object diagnostic output.

CObject provides basic services, including

- Serialization support
- Run-time class information

- Object diagnostic output
- Compatibility with collection classes

Note that CObject does not support multiple inheritance. Your derived classes can have only one CObject base class, and that CObject must be leftmost in the hierarchy. It is permissible, however, to have structures and non-CObject-derived classes in right-hand multiple-inheritance branches.

You will realize major benefits from CObject derivation if you use some of the optional macros in your class implementation and declarations.

The first-level macros, DECLARE_DYNAMIC and IMPLEMENT_DYNAMIC, permit run-time access to the class name and its position in the hierarchy. This, in turn, allows meaningful diagnostic dumping.

The second-level macros, DECLARE_SERIAL and IMPLEMENT_SERIAL, include all the functionality of the first-level macros, and they enable an object to be "serialized" to and from an "archive."

11.3.2 CArchive

CArchive Class allows you to save a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted.

CArchive does not have a base class. Later you can load the objects from persistent storage, reconstituting them in memory. This process of making data persistent is called "serialization."

You can think of an archive object as a kind of binary stream. Like an input/output stream, an archive is associated with a file and permits the buffered writing and reading of data to and from storage. An input/output stream processes sequences of ASCII characters, but an archive processes binary object data in an efficient, nonredundant format.

You must create a CFile object before you can create a CArchive object. In addition, you must ensure that the archive's load/store status is compatible with the file's open mode. You are limited to one active archive per file.

When you construct a CArchive object, you attach it to an object of class CFile (or a derived class) that represents an open file. You also specify whether the archive will be used for loading or storing. A CArchive object can process not only primitive types but also objects of CObject-derived classes designed for serialization. A serializable class usually has a Serialize member function, and it usually uses the DECLARE_SERIAL and IMPLEMENT_SERIAL macros, as described under class CObject.

The overloaded extraction (>>) and insertion (<<) operators are convenient archive programming interfaces that support both primitive types and CObject-derived classes.

CArchive also supports programming with the MFC Windows Sockets classes CSocket and CSocketFile. The IsBufferEmpty member function supports that usage.

11.3.3 CWinApp

The main application class in MFC encapsulates the initialization, running, and termination of an application for the Windows operating system. An application built on the framework must have one and only one object of a class derived from CWinApp. This object is constructed before windows are created.

CWinApp is derived from CWinThread, which represents the main thread of execution for your application, which might have one or more threads. In recent versions of MFC, the InitInstance, Run, ExitInstance, and OnIdle member functions are actually in class CWinThread. These functions are discussed here as if they were CWinApp members instead, because the discussion concerns the object's role as application object rather than as primary thread.

Like any program for the Windows operating system, your framework application has a WinMain function. In a framework application, however, you do not write WinMain. It is supplied by the class library and is called when the application starts up. WinMain performs standard services such as registering window classes. It then calls member functions of the application object to initialize and run the application. (You can customize WinMain by overriding the CWinApp member functions that WinMain calls.)

To initialize the application, WinMain calls your application object's InitApplication and InitInstance member functions. To run the application's message loop, WinMain calls the Run member function. On termination, WinMain calls the application object's ExitInstance member function.

11.3.4 CWnd

CWnd Class provides the base functionality of all window classes in the Microsoft Foundation Class Library.

A CWnd object is distinct from a Windows window, but the two are tightly linked. A CWnd object is created or destroyed by the CWnd constructor and destructor. The Windows window, on the other hand, is a data structure internal to Windows that is created by a Create member function and destroyed by the CWnd virtual destructor. The DestroyWindow function destroys the Windows window without destroying the object.

The CWnd class and the message-map mechanism hide the WndProc function. Incoming Windows notification messages are automatically routed through the message map to the proper OnMessage CWnd member functions. You override an OnMessage member function to handle a member's particular message in your derived classes.

The CWnd class also lets you create a Windows child window for your application. Derive a class from CWnd, then add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

You create a child window in two steps. First, call the constructor CWnd to construct the CWnd object, then call the Create member function to create the child window and attach it to the CWnd object.

When the user terminates your child window, destroy the CWnd object, or call the DestroyWindow member function to remove the window and destroy its data structures.

Within the Microsoft Foundation Class Library, further classes are derived from CWnd to provide specific window types. Many of these classes, including CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, and CDialog, are designed for further derivation. The control classes derived from CWnd, such as CButton, can be used directly or can be used for further derivation of classes.

11.3.5 CFile

CFile is the base class for Microsoft Foundation Class file classes. It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes. CFile works in conjunction with the CArchive class to support serialization of Microsoft Foundation Class objects.

The hierarchical relationship between this class and its derived classes allows your program to operate on all file objects through the polymorphic CFile interface. A memory file, for example, behaves like a disk file.

Use CFile and its derived classes for general-purpose disk I/O. Use ofstream or other Microsoft iostream classes for formatted text sent to a disk file.

Normally, a disk file is opened automatically on CFile construction and closed on destruction. Static member functions permit you to interrogate a file's status without opening the file.

11.3.6 CGDIObject

The CGdiObject class provides generic support for GDI objects in the form of a series of member functions. The Attach and Detach member functions can be used to attach a CGdiObject-derived MFC object to a GDI object or detach it from the GDI object. The handle of the object, stored in the m_hObject member variable, can be retrieved through the "safe" function GetSafeHandle. (This function can also be used with null CGdiObject pointers.) A pointer to a CGdiObject that corresponds to a Windows GDI object handle can be obtained by calling the static member function FromHandle. To obtain a GDI object's type, use the GetObjectType member function.

The CreateStockObject member function can be used to create a stock pen, brush, font, or palette. Note that this function should be called with a CGdiObject-derived object that is of the appropriate class (CPen, CBrush, CFont, or CPalette).

The UnrealizeObject member function can be used to reset the origin of a brush or reset a palette. Do not use this member function for objects of any other type.

The DeleteObject member function deletes the GDI object that the CGdiObject-derived MFC object is attached to. The DeleteTempMap function, called usually from the idle-time handler of your application's CWinApp object, is used to delete any temporary CGdiObject objects that were created by the FromHandle member function.

11.3.7 CExcept

CException is the base class for all exceptions in the Microsoft Foundation Class Library. The derived classes and their descriptions are listed below:

CMemoryException	Out-of-memory exception
CNotSupportedException	Request for an unsupported operation
CArchiveException	Archive-specific exception
CFileException	File-specific exception
CResourceException	Windows resource not found or not createable
COleException	OLE exception
CDBException	Database exception (that is, exception conditions arising for MFC database classes based on Open Database Connectivity)
COleDispatchException	OLE dispatch (automation) exception

Contd...

CUserException	Exception that indicates that a resource could not be found
CDaoException	Data access object exception (that is, exception conditions arising for DAO classes)
CInternetException	Internet exception (that is, exception conditions arising for Internet classes).

These exceptions are intended to be used with the `THROW`, `THROW_LAST`, `TRY`, `CATCH`, `AND_CATCH`, and `END_CATCH` macros.

To catch a specific exception, use the appropriate derived class. To catch all types of exceptions, use `CException`, and then use `CObject::IsKindOf` to differentiate among `CException`-derived classes. Note that `CObject::IsKindOf` works only for classes declared with the `IMPLEMENT_DYNAMIC` macro, in order to take advantage of dynamic type checking. Any `CException`-derived class that you create should use the `IMPLEMENT_DYNAMIC` macro, too.

You can report details about exceptions to the user by calling `GetErrorMessage` or `ReportError`, two member functions that work with any of `CException`'s derived classes.

If an exception is caught by one of the macros, the `CException` object is deleted automatically; do not delete it yourself. If an exception is caught by using a catch keyword, it is not automatically deleted.

`CException` is an abstract base class. You cannot create `CException` objects; you must create objects of derived classes. If you need to create your own `CException` type, use one of the derived classes listed above as a model. Make sure that your derived class also uses `IMPLEMENT_DYNAMIC`.

11.3.8 CDialog

`CDialog` Class is the base class used for displaying dialog boxes on the screen. Dialog boxes are of two types: modal and modeless. A modal dialog box must be closed by the user before the application continues. A modeless dialog box allows the user to display the dialog box and return to another task without canceling or removing the dialog box.

A `CDialog` object is a combination of a dialog template and a `CDialog`-derived class. Use the dialog editor to create the dialog template and store it in a resource, then use the Add Class wizard to create a class derived from `CDialog`.

A dialog box, like any other window, receives messages from Windows. In a dialog box, you are particularly interested in handling notification messages from the dialog box's controls since that is how the user interacts with your dialog box. Use the Properties window to select which messages you wish to handle and it will add the appropriate message-map entries and message-handler member functions to the class for you. You only need to write application-specific code in the handler member functions.

If you prefer, you can always write message-map entries and member functions manually.

In all but the most trivial dialog box, you add member variables to your derived dialog class to store data entered in the dialog box's controls by the user or to display data for the user. You can use the Add Variable wizard to create member variables and associate them with controls. At the same time, you choose a variable type and

permissible range of values for each variable. The code wizard adds the member variables to your derived dialog class.

A data map is generated to automatically handle the exchange of data between the member variables and the dialog box's controls. The data map provides functions that initialize the controls in the dialog box with the proper values, retrieve the data, and validate the data.

To create a modal dialog box, construct an object on the stack using the constructor for your derived dialog class and then call `DoModal` to create the dialog window and its controls. If you wish to create a modeless dialog, call `Create` in the constructor of your dialog class.

You can also create a template in memory by using a `DLGTEMPLATE` data structure as described in the Platform SDK. After you construct a `CDialog` object, call `CreateIndirect` to create a modeless dialog box, or call `InitModalIndirect` and `DoModal` to create a modal dialog box.

The exchange and validation data map is written in an override of `CWnd::DoDataExchange` that is added to your new dialog class.

Both the programmer and the framework call `DoDataExchange` indirectly through a call to `CWnd::UpdateData`.

The framework calls `UpdateData` when the user clicks the OK button to close a modal dialog box. (The data is not retrieved if the Cancel button is clicked.) The default implementation of `OnInitDialog` also calls `UpdateData` to set the initial values of the controls. You typically override `OnInitDialog` to further initialize controls. `OnInitDialog` is called after all the dialog controls are created and just before the dialog box is displayed.

You can call `CWnd::UpdateData` at any time during the execution of a modal or modeless dialog box.

If you develop a dialog box by hand, you add the necessary member variables to the derived dialog-box class yourself, and you add member functions to set or get these values.

A modal dialog box closes automatically when the user presses the OK or Cancel buttons or when your code calls the `EndDialog` member function.

When you implement a modeless dialog box, always override the `OnCancel` member function and call `DestroyWindow` from within it. Don't call the base class `CDialog::OnCancel`, because it calls `EndDialog`, which will make the dialog box invisible but will not destroy it. You should also override `PostNcDestroy` for modeless dialog boxes in order to delete this, since modeless dialog boxes are usually allocated with `new`. Modal dialog boxes are usually constructed on the frame and do not need `PostNcDestroy` cleanup.

11.3.9 CString

`CString` does not have a base class. A `CString` object consists of a variable-length sequence of characters. `CString` provides functions and operators using a syntax similar to that of Basic. Concatenation and comparison operators, together with simplified memory management, make `CString` objects easier to use than ordinary character arrays.

`CString` is based on the `TCHAR` data type. If the symbol `_UNICODE` is defined for your program, `TCHAR` is defined as type `wchar_t`, a 16-bit character type; otherwise,

it is defined as `char`, the normal 8-bit character type. Under Unicode, then, `CString` objects are composed of 16-bit characters. Without Unicode, they are composed of 8-bit `char` type.

When not using `_UNICODE`, `CString` is enabled for multibyte character sets (MBCS, also known as double-byte character sets, DBCS). Note that for MBCS strings, `CString` still counts, returns, and manipulates strings based on 8-bit characters, and your application must interpret MBCS lead and trail bytes itself.

- `CString` objects also have the following characteristics:
- `CString` objects can grow as a result of concatenation operations.
- `CString` objects follow “value semantics.” Think of a `CString` object as an actual string, not as a pointer to a string.
- You can freely substitute `CString` objects for `const char*` and `LPCTSTR` function arguments.
- A conversion operator gives direct access to the string’s characters as a read-only array of characters (a C-style string).

Where possible, allocate `CString` objects on the frame rather than on the heap. This saves memory and simplifies parameter passing.

`CString` assists you in conserving memory space by allowing two strings sharing the same value also to share the same buffer space. However, if you attempt to change the contents of the buffer directly (not using MFC), you can alter both strings unintentionally. `CString` provides two member functions, `CString::LockBuffer` and `CString::UnlockBuffer`, to help you protect your data. When you call `LockBuffer`, you create a copy of a string, then set the reference count to -1, which “locks” the buffer. While the buffer is locked, no other string can reference the data in that string, and the locked string will not reference another string. By locking the string in the buffer, you ensure that the string’s exclusive hold on the data will remain intact. When you have finished with the data, call `UnlockBuffer` to reset the reference count to 1.

11.3.10 CEdit

`CEdit` Class provides the functionality of a Windows edit control. An edit control is a rectangular child window in which the user can enter text.

You can create an edit control either from a dialog template or directly in your code. In both cases, first call the constructor `CEdit` to construct the `CEdit` object, then call the `Create` member function to create the Windows edit control and attach it to the `CEdit` object.

Construction can be a one-step process in a class derived from `CEdit`. Write a constructor for the derived class and call `Create` from within the constructor.

`CEdit` inherits significant functionality from `CWnd`. To set and retrieve text from a `CEdit` object, use the `CWnd` member functions `SetWindowText` and `GetWindowText`, which set or get the entire contents of an edit control, even if it is a multiline control. Text lines in a multiline control are separated by `‘\r\n’` character sequences. Also, if an edit control is multiline, get and set part of the control’s text by calling the `CEdit` member functions `GetLine`, `SetSel`, `GetSel`, and `ReplaceSel`.

If you want to handle Windows notification messages sent by an edit control to its parent (usually a class derived from `CDialog`), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

```
ON_Notification( id, memberFxn )
```

where `id` specifies the child window ID of the edit control sending the notification, and `memberFxn` is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

```
afx_msg void memberFxn( );
```

Following is a list of potential message-map entries and a description of the cases in which they would be sent to the parent:

ON_EN_CHANGE The user has taken an action that may have altered text in an edit control. Unlike the **EN_UPDATE** notification message, this notification message is sent after Windows updates the display.

ON_EN_ERRSPACE The edit control cannot allocate enough memory to meet a specific request.

ON_EN_HSCROLL The user clicks an edit control's horizontal scroll bar. The parent window is notified before the screen is updated.

ON_EN_KILLFOCUS The edit control loses the input focus.

ON_EN_MAXTEXT The current insertion has exceeded the specified number of characters for the edit control and has been truncated. Also sent when an edit control does not have the **ES_AUTOHSCROLL** style and the number of characters to be inserted would exceed the width of the edit control. Also sent when an edit control does not have the **ES_AUTOVSCROLL** style and the total number of lines resulting from a text insertion would exceed the height of the edit control.

ON_EN_SETFOCUS Sent when an edit control receives the input focus.

ON_EN_UPDATE The edit control is about to display altered text. Sent after the control has formatted the text but before it screens the text so that the window size can be altered, if necessary.

ON_EN_VSCROLL The user clicks an edit control's vertical scroll bar. The parent window is notified before the screen is updated.

If you create a **CEdit** object within a dialog box, the **CEdit** object is automatically destroyed when the user closes the dialog box.

If you create a **CEdit** object from a dialog resource using the dialog editor, the **CEdit** object is automatically destroyed when the user closes the dialog box.

If you create a **CEdit** object within a window, you may also need to destroy it. If you create the **CEdit** object on the stack, it is destroyed automatically. If you create the **CEdit** object on the heap by using the `new` function, you must call `delete` on the object to destroy it when the user terminates the Windows edit control. If you allocate any memory in the **CEdit** object, override the **CEdit** destructor to dispose of the allocations.

To modify certain styles in an edit control (such as **ES_READONLY**) you must send specific messages to the control instead of using `ModifyStyle`.

11.3.11 CList

CList Class supports ordered lists of non-unique objects accessible sequentially or by value. CList lists behave like doubly-linked lists.

A variable of type POSITION is a key for the list. You can use a POSITION variable as an iterator to traverse a list sequentially and as a bookmark to hold a place. A position is not the same as an index, however.

Element insertion is very fast at the list head, at the tail, and at a known POSITION. A sequential search is necessary to look up an element by value or index. This search can be slow if the list is long.

If you need a dump of individual elements in the list, you must set the depth of the dump context to 1 or greater.

Certain member functions of this class call global helper functions that must be customized for most uses of the CList class.

Check Your Progress 2

Fill in the blanks:

1. MFC supports property sheets through two classes _____ and _____.
2. Using the _____ member function of a property sheet implies modal behavior.
3. _____ Class supports ordered lists of non-unique objects accessible sequentially or by value.

11.4 LET US SUM UP

The new property sheet class can be created using ClassWizard. On termination, WinMain calls the application object's ExitInstance member function. CWnd Class provides the base functionality of all window classes in the Microsoft Foundation Class Library. The DestroyWindow function destroys the Windows window without destroying the object. CFile is the base class for Microsoft Foundation Class file classes. You cannot create CException objects; you must create objects of derived classes. CDialog Class is the base class used for displaying dialog boxes on the screen. A CDialog object is a combination of a dialog template and a CDialog-derived class. The code wizard adds the member variables to your derived dialog class. If you wish to create a modeless dialog, call Create in the constructor of your dialog class.

11.5 LESSON END ACTIVITY

1. Design the dialog box as shown on next page.

Palace Ice Scream - Employees Records

First Name: Edit Last Name: Edit

Address: Edit

City: Edit State: Edit

ZIP Code: Edit Hourly Salary: Edit

Employment Status

Part Time Full Time

OK Cancel

11.6 KEYWORDS

Property: It is a piece of information that is associated with an object.

Property Sheet: It is a collection of several overlapping dialogs in one.

Microsoft Foundation Class Library: It is also known as Microsoft Foundation Classes or MFC; a library that wraps portions of the Windows API in C++ classes, including functionality that enables them to use a default application framework.

CObject: It is the root base class for most of the Microsoft Foundation Class Library (MFC).

Serialization: It is the conversion of an object to and from a persistent form.

11.7 QUESTIONS FOR DISCUSSION

1. Discuss the use of the classes of MFC with multiple inheritance.
2. How to create property page?
3. Why CException is known as an abstract base class?
4. What is CObject? What are the basic services provided by it?
5. Write short note on
 - a) CObject
 - b) CString

Check Your Progress: Model Answers

CYP 1

1. CancelToClose
2. modeless

CYP 2

1. CPropertySheet and CPropertyPage
2. DoModal
3. CList

11.8 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up*, 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth, *Visual C++6 Unleashed*, Second edition, Techmedia.

LESSON

12

DOCUMENT/VIEW ARCHITECTURE

CONTENTS

- 12.0 Aims and Objectives
- 12.1 Introduction
- 12.2 Resources
 - 12.2.1 Menus
 - 12.2.2 Accelerators
 - 12.2.3 Dialog
 - 12.2.4 Icon
 - 12.2.5 Bitmaps
 - 12.2.6 Versions
- 12.3 Message Maps
- 12.4 Document/View Architecture
 - 12.4.1 The View
 - 12.4.2 The Document
 - 12.4.3 The Frame
 - 12.4.4 The Document/View Approach
 - 12.4.5 Overview of the Single Document Interface (SDI)
 - 12.4.6 Creating a Single Document Interface
 - 12.4.7 Overview of the Multiple Document Interface (MDI)
 - 12.4.8 Creating a Multiple Document Interface
- 12.5 Let us Sum up
- 12.6 Lesson End Activity
- 12.7 Keywords
- 12.8 Questions for Discussion
- 12.9 Suggested Readings

12.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Overview of resource files
- Brief idea about Message Maps
- Concept of SDI and MDI

12.1 INTRODUCTION

MFC makes it easy to work with both single-document interface (SDI) and multiple-document interface (MDI) applications. SDI applications allow only one open document frame window at a time. For instance, the program `notepad.exe` is a SDI application. Netscape is also an SDI application. MDI applications allow multiple document frame windows to be open in the same instance of an application. An MDI application has a window within which multiple MDI child windows, which are frame windows themselves, can be opened, each containing a separate document. Word for Windows and the VC++ developer studio are MDI applications. In these you can have several documents opened at once. This is particularly useful when you want to cut and paste between documents. In some applications, the child windows can be of different types, such as chart windows and spreadsheet windows. In that case, the menu bar can change as MDI child windows of different types are activated. A graphing application comes to mind where in one window you have a spreadsheet-like data list, and in another window you have a plot of the data. For small applications, a SDI application will usually be all you need. After you master it, the jump to MDI is a snap. Let's go over the document view architecture.

12.2 RESOURCES

Resource is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc. Resource files define the visual appearance of an application. Resources include dialogs, menus, bitmaps, icons, cursors, text strings, and several other types. Microsoft Visual C++ makes creating a resource file particularly easy by providing the necessary tools in the same environment used to program, meaning you usually do not have to use an external application to create or configure this file.

Although an application can use various resources that behave independently of each other, these resources are grouped into a text file that has the `.rc` extension. You can create this file manually and fill out all necessary parts but it is advantageous to let Visual C++ create it for you. To do this, you add or create one resource at a time when designing it. After saving a resource, it is automatically added to the `.rc` file. To make your resource recognizable to the other files of the program, you must also create a header file usually called `resource.h`. This header file must provide a constant integer that identifies each resource and makes it available to any part that needs it. This also means that most, if not all, of your resources will be represented by an identifier.

An identifier is a constant integer whose name usually starts with `ID`. Although in Win32 programming you usually can use the name of a resource as a string, in MFC applications, resources are usually referred to by their identifier. To make an identifier name (considered a string) recognizable to an MFC (or Win32) function, you use a macro called `MAKEINTRESOURCE`. Its syntax is:

```
LPTSTR MAKEINTRESOURCE(WORD IDentifier);
```

This macro takes the identifier of the resource and returns a string that is given to the function that called it.

In the strict sense, after creating the resource file, it must be compiled to create a new file that has the extension `.res`. Fortunately, Visual C++ automatically compiles the file and links it to the application so that your efforts are reduced.

12.2.1 Menus

Menu statements in the resource script can be used to specify menu bars or popup menus. Menu statements contain one or more menu definition statements enclosed between a BEGIN-END keyword pair. Consider the following example:

```
MyMenu MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N", ID_FILE_NEW
        MENUITEM SEPARATOR
        MENUITEM "E&xit", ID_FILE_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "Cu&t\tCtrl+X", ID_EDIT_CUT
        MENUITEM "&Copy\tCtrl+C", ID_EDIT_COPY
        MENUITEM "&Paste\tCtrl+V", ID_EDIT_PASTE
    END

    POPUP "&Help"
    BEGIN
        MENUITEM "&About", ID_HELP_ABOUT
    END
END
```

The identifiers (for example., ID_FILE_NEW) in this example are assumed to be defined elsewhere and refer to numeric values.

A menu definition statement can specify a menu item or a submenu. To define a menu item, use the MENUITEM keyword. This is followed either by the text of the menu item and the menu identifier or by the SEPARATOR keyword; the latter specifies a separator that is a vertical line for menu bars or a horizontal line for popup (sub) menus.

A menu item's identifier may be followed by a list of options. Options are separated by commas or spaces and include the following keywords: CHECKED, GRAYED, HELP, INACTIVE, MENUBARBREAK, MENUBREAK.

To specify a submenu, use the POPUP keyword. The POPUP keyword is followed by the title of the submenu, and a set of menu items enclosed between a BEGIN-END keyword pair. A submenu may contain nested submenus.

12.2.2 Accelerators

Accelerators are keystrokes that represent shortcuts to specific tasks. For example, when you use Ctrl+C to copy an item to the clipboard, you are using an accelerator.

Enclosed between the BEGIN-END keyword pair, an accelerator statement contains an arbitrary number of keyboard events followed by the identifier of the accelerator. Consider the following example:

```
MyAcc ACCELERATOR
BEGIN
    "C", ID_EDIT_COPY, VIRTKEY, CONTROL, NOINVERT
    "V", ID_EDIT_PASTE, VIRTKEY, CONTROL, NOINVERT
    "X", ID_EDIT_CUT, VIRTKEY, CONTROL, NOINVERT
END
```

This example assumes that the symbolic constants `ID_EDIT_COPY`, `ID_EDIT_PASTE`, and `ID_EDIT_CUT` are defined elsewhere (in a header file) and refer to numeric identifiers.

Accelerators are interpreted when an application calls the `TranslateAccelerator` function after retrieving a message through `GetMessage` (or `PeekMessage`). `TranslateAccelerator` translates `WM_KEYDOWN` (or `WM_SYSKEYDOWN`) messages into `WM_COMMAND` (or `WM_SYSCOMMAND`) messages. The identifiers that follow accelerator keys in an accelerator statement will become the command identifiers in the `WM_COMMAND` messages.

12.2.3 Dialog

Together with menu statements, dialog statements define what are perhaps the most visible parts of a typical application. A dialog statement defines the layout and appearance of a dialog box.

The statement consists of several lines. The first line contains an identifier, the `DIALOG` keyword, and four numeric parameters that define the position of the dialog's upper-left corner and the size of the dialog box.

All size and position information in a dialog statement is measured in dialog units. Dialog units are derived from the size of the font specified for the dialog. Dialog base units represent the average height and width of a character in the selected font. Four horizontal dialog units amount to one horizontal dialog base unit; eight vertical dialog units amount to one vertical dialog base unit.

For dialogs that use the system font, applications can obtain the size of dialog base units, in pixels, by calling `GetDialogBaseUnits`. For dialogs using other fonts, it may be necessary to explicitly calculate the average size of characters to obtain the dialog base units.

Once the dialog base units are known, applications can convert between dialog units and pixels using the following formulae:

```
pixelX = (dialogunitX * baseunitX) / 4
pixelY = (dialogunitY * baseunitY) / 8
dialogunitX = (pixelX * 4) / baseunitX
dialogunitY = (pixelY * 8) / baseunitY
```

Following the line containing the `DIALOG` keyword, a dialog statement may contain several optional instructions. These may include commonly used optional instructions or dialog-specific optional instructions.

The **CAPTION** optional instruction is followed by a string specifying the title of the dialog box. The default is a dialog with no title.

The **STYLE** optional instruction specifies the style of the dialog. Style values are usually predefined in the header file `windows.h`. Several values can be combined using the logical or (`|`) operator. The default style for dialogs containing no **STYLE** instruction is `WS_POPUP | WS_BORDER | WS_SYSMENU`.

The **EXSTYLE** optional instruction is similar to **STYLE** and specifies extended styles.

The **CLASS** optional instruction can be used to specify a special window class for a dialog. Using the **CLASS** instruction should be used sparingly, as it redefines the behavior of the dialog.

The **FONT** statement specifies the font to be used in the dialog. The default is the system font.

The **MENU** statement identifies the menu resource that defines the menu of the dialog box. In the absence of this statement, the dialog box will be created without a menu bar.

Enclosed between the **BEGIN-END** keyword pair is a series of control statements specifying the controls within the dialog. There are several types of control statements. Each control statement contains the control type, control text, a control identifier (text or integer), control position, and control style and extended style parameters:

CONTROL-STATEMENT control-text, identifier, x, y, width, height [, style [, extended-style]]

An edit control is defined by the **EDITTEXT** control statement.

A static control is defined by the **LTEXT**, **CTEXT**, **RTEXT**, or **ICON** control statement. The first three of these control statements define a left-aligned, centered, or right-aligned static control, respectively. The last specifies a static control with the `SS_ICON` style.

A button control is defined by one of the following keywords: **AUTO3STATE**, **AUTOCHECKBOX**, **AUTORADIOBUTTON**, **CHECKBOX**, **DEFPUSHBUTTON**, **GROUPBOX**, **PUSHBOX**, **PUSHBUTTON**, **RADIOBUTTON**, **STATE3**, **USERBUTTON**.

The **COMBOBOX** control statement defines a combo box control.

The **LISTBOX** control statement can be used to specify a list box.

The **SCROLLBAR** control statement defines, what else? A scrollbar.

The **CONTROL** control statement can be used to define a generic control. The syntax of this statement is somewhat different from the syntax used for other control statements:

CONTROL control-text, identifier, class-name, x, y, width, height [, extended-style]

The class-name parameter specifies the window class for the control. This can be one of the Windows control classes. Thus, the **CONTROL** statement can be used as an alternative syntax for all the other control statements.

A variant of the **DIALOG** statement is the **DIALOGEX** statement. It extends the syntax of the **DIALOG** statement in the following ways:

- Help identifiers can be specified for the dialog itself as well as any controls within it.
- Font weight and italic settings can be put in the FONT instruction.
- Control-specific data can be added to control statements (enclosed by the BEGIN-END keyword pair).
- The keywords BEDIT, HEDIT, and IEDIT can be used for pen controls.

12.2.4 Icon

An icon is a small picture used on a window. It is used in two main scenarios. On a window's frame, it display on the left side of the window name on the title bar. In Windows Explorer, on the Desktop, in My Computer, or in the Control Panel windows, an icon is used to represent an application:

On a window, the icon is a 16x16 pixels size of picture to accommodate the standard height of the title bar. In window displays such as Windows Explorer or My Computer, the applications can be represented as tiles or as a list. Each display uses a different size of icon. Therefore, an icon is created in two sizes that share the same name but the operating system can manage that concept. This means that you will sometimes create two designs for one icon, a 16x16 pixel and a 32x32 pixel.

An icon is a graphical object made of two categories of colors. One category represents the artistic side of the design. The other is the color that would be used as background so that if the icon is positioned on top of another picture, the background color would be used as the transparent color to show the sections that are not strictly part of the icon. In reality, Microsoft Windows imposes the color that is used as background on an icon.

The icon statement specifies a binary icon file (edited by an icon editor) that defines an icon resource. For example:

```
MyIcon ICON myicon.ico
```

12.2.5 Bitmaps

A bitmap resource is a bitmap image file (.bmp) stored within a resource file. Each bitmap resource is identified by a unique index or resource ID. Bitmaps are extracted from a resource file at run-time using the LoadResPicture function, which takes the index and the format (0 or vbResBitmap) as arguments.

Bitmap resources provide a convenient way to store multiple images for an application while keeping the source files hidden from the user. You can use the Windows Paint application to edit or create bitmap files.

The BITMAP statement specifies a bitmap file (edited separately with a bitmap editor) that is to define a bitmap resource. For example:

```
MyBitmap BITMAP mybitmap.bmp
```

12.2.6 Versions

The version information resource can be used to identify the version of a binary file (typically, an executable or library file). Version information is used by File Installation Library functions.

The version information resource statement contains several version statements that identify the version number of the file and the product, provide additional version information, and specify the language and the target operating system.

Check Your Progress 1

1. What is resource?

.....
.....

2. What is menu statement?

.....
.....

12.3 MESSAGE MAPS

Some of your applications will be made of various objects. Most of the time, more than one application is running on the computer. These two scenarios mean that the operating system is constantly asked to perform some assignments. Because there can be so many requests presented unpredictably, the operating system leaves it up to the objects to specify what they want, when they want it, and what behavior or result they expect.

The Microsoft Windows operating system cannot predict what kinds of requests one object would need to be taken care of and what type of assignment another object would need. To manage all these assignments and requests, the objects send messages, one message at a time, to the operating system. For this reason, Microsoft Windows is said to be a message-driven operating system.

The messages are divided in various categories but as mentioned already, each object has the responsibility to decide what message to send and when. Therefore, most of the messages we will review here are part of a window frame. Others will be addressed when necessary.

Once a control has composed a message, it must send it to the right target which could be the operating system. In order to send a message, a control must create an event. It is also said to fire an event. To make a distinction between the two, a message's name usually starts with WM_ which stands for Window Message. The name of an event usually starts with On which indicates an action. Remember, the message is what needs to be sent. The event is the action of sending the message.

For the compiler to manage messages, they should be included in the class definition. The list of messages starts on a section driven by, but that ends with, the DECLARE_MESSAGE_MAP macro. The section can be created as follows:

```
#include <afxwin.h>

class CSimpleFrame : public CFrameWnd
{
public:
    CSimpleFrame();
    DECLARE_MESSAGE_MAP()
};
```

The `DECLARE_MESSAGE_MAP` macro should be provided at the end of the class definition. The actual messages (as we will review them shortly) should be listed just above the `DECLARE_MESSAGE_MAP` line. This is mostly a suggestion. In some circumstances, and for any reason, you may want, or have, to provide one message or a few messages under the `DECLARE_MESSAGE_MAP` line.

To implement the messages, you should/must create a table of messages that your program is using. This table uses two delimiting macros. It starts with a `BEGIN_MESSAGE_MAP` and ends with an `END_MESSAGE_MAP` macros. The `BEGIN_MESSAGE_MAP` macro takes two arguments, the name of your class and the MFC class you derived your class from. An example would be:

```
BEGIN_MESSAGE_MAP(CSimpleFrame, CFrameWnd)
```

Like the `DECLARE_MESSAGE_MAP` macro, `END_MESSAGE_MAP` takes no argument. Its job is simply to specify the end of the list of messages. The table of messages can be created as follows:

```
#include <afxwin.h>
#include "resource.h"

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();
    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    LoadFrame(IDR_MAINFRAME);
}

class CMainApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
END_MESSAGE_MAP()

BOOL CMainApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

CMainApp theApp;
```

There various categories of messages the operating system receives. Some of them come from the keyboard, some from the mouse, and some others from various other origins. For example, some messages are sent by the application itself while some other messages are controlled by the operating.

12.4 DOCUMENT/VIEW ARCHITECTURE

The Document/View architecture is the foundation used to create applications based on the Microsoft Foundation Classes library. It allows you to make distinct the different parts that compose a computer program including what the user sees as part of your application and the document a user would work on. This is done through a combination of separate classes that work as an ensemble.

The parts that compose the Document/View architecture are a frame, one or more documents, and the view. Put together, these entities make up a usable application.

12.4.1 The View

A view is the platform the user is working on to do his or her job. For example, while performing word processing, the user works on a series of words that compose the text. If a user is performing calculations on a spreadsheet application, the interface the user is viewing is made of small boxes called cells. Another user may be in front of a graphic document while drawing lines and other geometric figures. The thing the user is starring at and performing changes is called a view. The view also allows the user to print a document.

To let the user do anything on an application, you must provide a view, which is an object based on the CView class. You can either directly use one of the classes derived from CView or you can derive your own custom class from CView or one of its child classes.

12.4.2 The Document

A document is similar to a bucket. It can be used to hold or carry water and that water can be retrieved when needed. For a computer application, a document holds the user's data. For example, after working on a text processor, the user may want to save the file. Such an action creates a document and this document must reside somewhere. In the same way, to use an existing file, the user must locate it, open it, and make it available to the application. These two jobs and many others are handled behind the scenes as a document.

To create the document part of this architecture, you must derive an object from the CDocument class.

12.4.3 The Frame

As its name suggests, a frame is a combination of the building blocks, the structure (in the English sense), and the borders of an item. A frame gives "physical" presence to a window. A frame defines the location of an object with regards to the Windows desktop. A frame provides borders to a window, borders that the user can grab to move, size, and resize the object. The frame is also a type of desk that holds the tools needed on an application.

An application cannot exist without a frame. As we saw in previous lessons, to provide a frame to an application, you can derive a class from CFrameWnd.

12.4.4 The Document/View Approach

To create an application, you obviously should start by providing a frame. This can be taken care of by deriving a class from `CFrameWnd`. Here is an example:

```
class CMainFrame : public CFrameWnd
{
    DECLARE_DYNCREATE(CMainFrame)
    DECLARE_MESSAGE_MAP()
};
```

To give "physical" presence to the frame of an application, you can declare an `OnCreate()` method. Here is an example:

```
class CMainFrame : public CFrameWnd
{
    DECLARE_DYNCREATE(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    DECLARE_MESSAGE_MAP()
};
```

The easiest way you can implement this method is to call the parent class, `CFrameWnd`, to create the window. As we have seen in the past, if this method returns 0, the frame has been created. It returns -1, this indicates that the window has been destroyed. Therefore, you can create a frame as follows:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // Call the base class to create the window
    if( CFrameWnd::OnCreate(lpCreateStruct) == 0)
        return 0;
    // else is implied
    return -1;
}
```

To allow users to interact with your application, you should provide a document. To do this, you can derive a class from `CDocument` so you can take advantage of this class. If you do not plan to do anything with the document, you can just make it an empty class. Here is an example:

```
class CExerciseDoc : public CDocument
{
    DECLARE_DYNCREATE(CExerciseDoc)
    DECLARE_MESSAGE_MAP()
};
```

Besides the few things we have learned so far, your next big decision may consist on the type of application you want to create. This is provided as a view. The most fundamental class of the view implementations in the MFC is `CView`. Because `CView` is an abstract class, you cannot directly use it in your application. You have two main alternatives. You can derive your own class based on `CView` (the `CView` class itself is

based on CWnd) or you can use one of the many view classes that the MFC provides. The classes that are readily available to you are:

Class	Description
CEditView	Used for a basic text editing application
CRichEditView	Allows creating rich documents that perform text and paragraph formatting
CScrollView	Provides the ability to scroll a view horizontally and vertically
CListView	Provides a view that can display a list of items
CTreeView	Allows displaying a list of items arranged as a tree
CFormView	Used to create a view that resembles a dialog box but provides the document/view features
CDaoRecordView	Provides a view that resembles a dialog box but used for database controls
CCTRLView	Provides parental guidance to the CEditView, CListView, CTreeView, and CRichEditView

Once you have a frame, a document, and a view, you can create an application, which, as we have learned so far, is done by deriving a class from CWinApp and overriding the InitInstance() method. In the InitInstance() implementation, you must let the compiler know how a document is created in your application. To do this, you must provide a sample document, called a template, that defines the parts that constitute a document for your particular type of application. This is done using a pointer variable declared from CDocTemplate or one of its derived classes.

12.4.5 Overview of the Single Document Interface (SDI)

The expression Single Document Interface or SDI refers to a document that can present only one view to the user. This means that the application cannot display more than one document at a time. If the user wants to view another type of document of the current application, he or she must another instance of the application. Notepad and WordPad are examples of SDI applications:

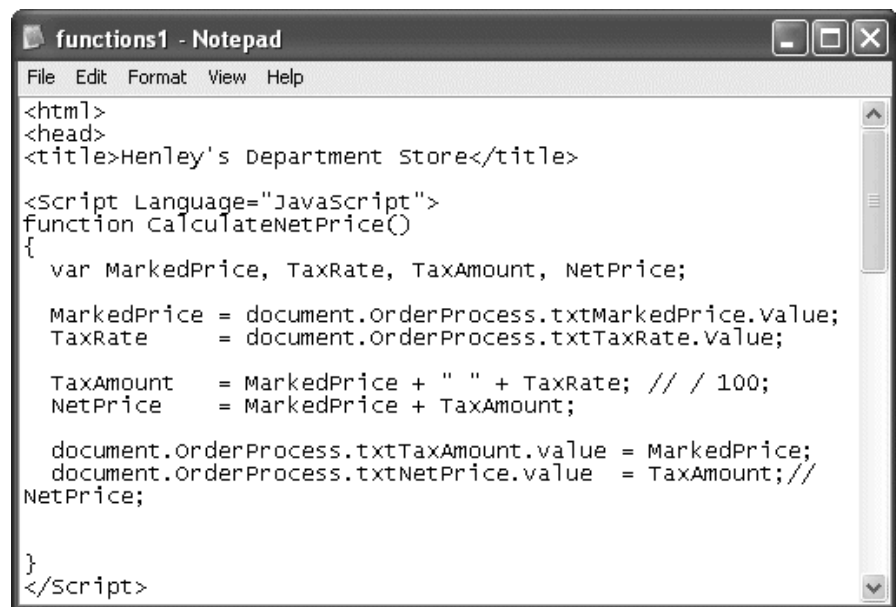


Figure 12.1: Notepad- An example of SDI

Notepad can be used to open only one document such as an HTML file, to view another file, the user would either replace the current document or launch another copy of Notepad.

To create an SDI, Microsoft Visual C++ provides the MFC wizard which provides all the basic functionality that an application needs, including the resources and classes.

12.4.6 Creating a Single Document Interface

As mentioned earlier, after creating a frame, a document, and a view, you can create an application by deriving a class from `CWinApp` and overriding the virtual `InitInstance()` member function. In `InitInstance()`, you must provide a template for your type of application. This is done using a `CDocTemplate` type of object.

To create an SDI, the `CDocTemplate` class provides the `CSingleDocTemplate` used to create an application that provides only one view. Therefore, you can declare a pointer variable to `CSingleDocTemplate`. Using this pointer and the new operator, use the `CSingleDocTemplate` constructor to provide the template. The syntax of the `CSingleDocTemplate` constructor is:

```
CSingleDocTemplate(UINT nIDResource,
    CRuntimeClass* pDocClass,
    CRuntimeClass* pFrameClass,
    CRuntimeClass* pViewClass);
```

The `CSingleDocTemplate` constructor needs the common identifier for the resources of your application. We saw in Lesson 3 that this can be done by using `IDR_MAINFRAME` as the common name of most or all main resources of an application. This is provided as the `nIDResource` argument.

The second argument, `pDocClass`, is the name of the class you derived from `CDocument`, as mentioned earlier.

The third argument, `pFrameClass`, is the frame class you derived from either `CFrameWnd` or one of its children.

The `pViewClass` argument can be an MFC `CView`-derived class. It can also be a class you created based on `CView`.

Each of these arguments must be passed as a pointer to `CRuntimeClass`. This can be taken care of by using the `RUNTIME_CLASS` macro. Its syntax is:

```
RUNTIME_CLASS(ClassName);
```

Each one of the classes you want to use must be provided as the `ClassName` argument. The `RUNTIME_CLASS` macro in turn returns a pointer to `CRuntimeClass`. To effectively use the `RUNTIME_CLASS` macro, you should make sure that the (each) class is created and implemented using the `DECLARE_DYNAMIC`, the `DECLARE_DYNCREATE`, or the `DECLARE_SERIAL` macros.

To actually create the application so it can be displayed to the user, the `CWinApp` class is equipped with the `AddDocTemplate()` method. Therefore, After creating a template, pass the `CSingleDocTemplate` pointer to the `CWinApp::AddDocTemplate()` method. Its syntax is:

```
void AddDocTemplate(CDocTemplate *pTemplate);
```

Everything else is subject to how you want your application to provide a useful experience to the user.

Check Your Progress 2

True or False:

1. In order to send a message, a control must create a class.
2. An application cannot exist without a frame.

12.4.7 Overview of the Multiple Document Interface (MDI)

An application is referred to as Multiple Document Interface, or MDI, if the user can open more than one document in the application without closing it. To provide this functionality, the application provides a parent frame that acts as the main frame of the computer program. Inside of this frame, the application allows creating views that each uses its own frame, making it distinct from the other. Here is Microsoft Word 97 displaying as a classic MDI application:

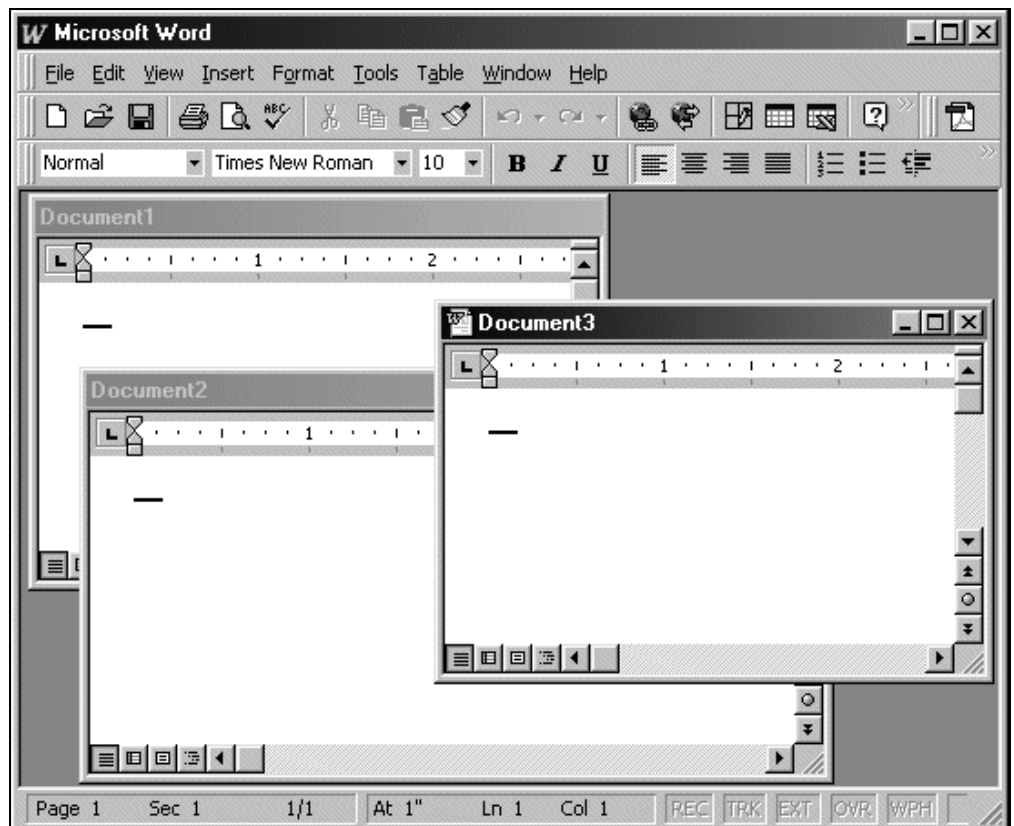


Figure 12.2: Example of MDI

When using an MDI application, a user can create a document, open another while the first is still displaying, and even create new documents or open additional ones. The documents are stacked in a Z-order axis of a 3-D coordinate system. There are two ways to display the documents of an MDI. If one document is maximized, all documents are maximized. In this case, the main menu of the application would display the system buttons on its right, which creates two ranges of system buttons:

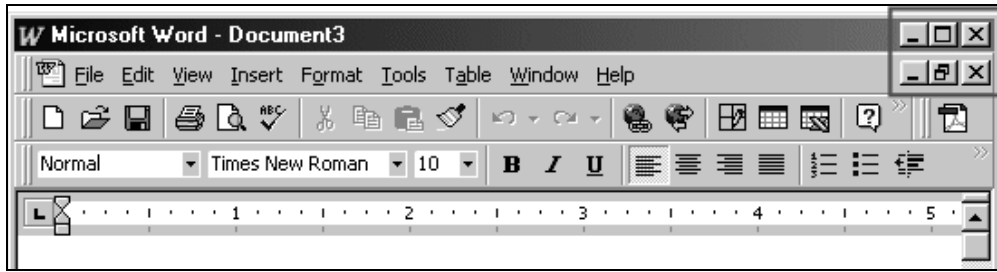


Figure 12.3: The system button in MDI

If no document is maximized, each document can assume one of two states: minimized or restored. While the child documents are confined to the borders of the main frame, if a document is or gets minimized, it would display its button inside the main frame. If a document is not minimized, it would display inside the main frame either with its original size or the size the user had given it:

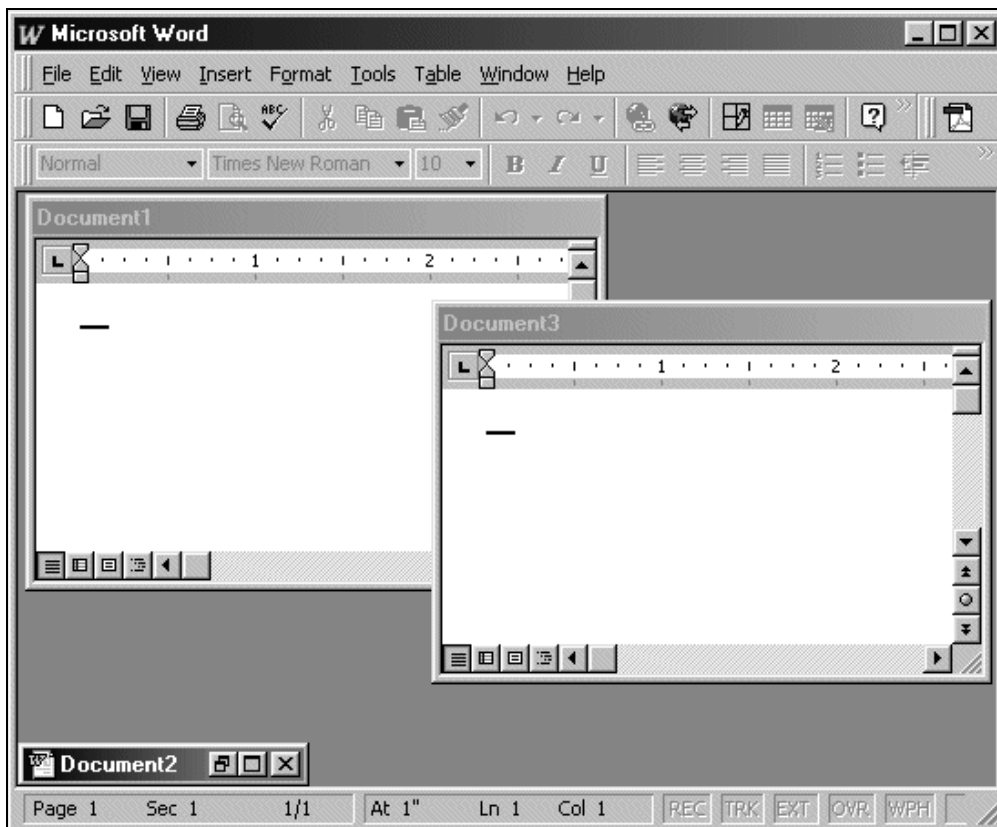


Figure 12.4: Minimized or restored child windows in MDI

There are two main ways the user can access the different documents of an MDI. If they are maximized (remember that if the user maximizes one document, all the others get maximized also), the menu of the main frame, which we always call the main menu in this book, has an item called Window. When the Window menu item is accessed, it would display a list of the currently opened documents. The user can then select from that list:

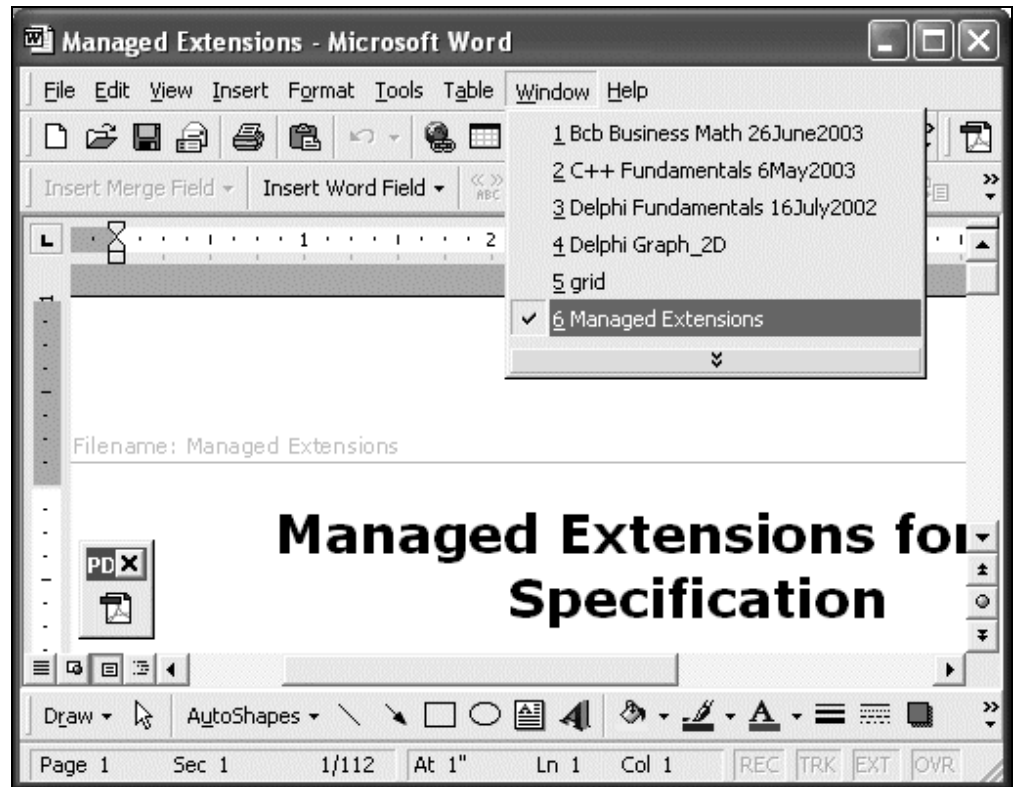


Figure 12.5: List of the currently opened documents

The separation of the parent and the child frames allows the user to close one child document or all documents. This would still leave the main frame of the application but the user cannot use it as a document. Still, the main frame allows the user to either create a new document or open an existing one.

To manage the differentiation between the parent and its children, the application uses two different menus: one for the main frame and one for a (or each) child. Both menus use the same menu bar, meaning that the application cannot display two frame menus at one time. It displays either the parent menu or a (the) child menu. When at least one document is displaying, the menu applies to the document. If no document is displaying, the main frame is equipped with a simplified menu with limited but appropriate operations. The user can only create a new document, only open an existing document, or simply close the application.

12.4.8 Creating a Multiple Document Interface

We have mentioned that an MDI is an application made of a parent frame and at least one child. Therefore, to create an MDI, because it requires a frame of its own, you should first create a series of resource objects that share a common name as `IDR_MAINFRAME`. A basic application should have an icon and a menu. The icon can be designed any way you like and you are recommended to create one made of a 32x32 and a 16x16 versions. The menu, since it will be used only when no document is available, should provide functionality that does not process a document. It should allow the user to create a new document, to open an existing document, and to quit the application. Such a menu can have the following items:

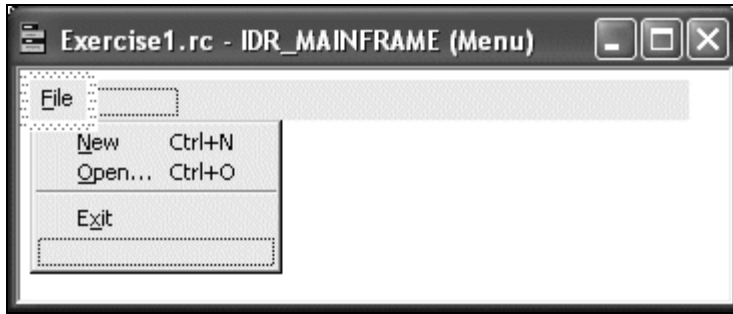


Figure 12.6: Resource objects for MDI

Two resources are sufficient to create an application since you can call the `CFrameWnd::LoadFrame()` method. To create a frame for a Multiple Document Interface, you must derive your frame class from `CMDIFrameWnd`:

```

BOOL CExercise1App::InitInstance()
{
    // Create a frame for the window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    CCommandLineInfo cmdInfo;

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}

```

The above code allows only creating a window frame for the parent window. To allow the user to interact with the computer through your MDI, you must provide a template for a document. The child document must have its own frame, distinct from that of the parent. To provide this frame, you have two main alternatives. You can directly use the `CMDIChildWnd` class or you can derive your own class from `CMDIChildWnd`:

```

class CChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CChildFrame)
    DECLARE_MESSAGE_MAP()
};

```

As opposed to a Single Document Interface application, to create a Multiple Document Interface (MDI) application, you use the `CMultiDocTemplate` class which,

like the `CSingleDocTemplate` class, is derived from `CDocTemplate`. Therefore, you must declare a pointer variable to `CMultiDocTemplate` using the new operator, then use its constructor to initialize the template. The syntax of the `CMultiDocTemplate` constructor is:

```
CMultiDocTemplate(UINT nIDResource,
                  CRuntimeClass* pDocClass,
                  CRuntimeClass* pFrameClass,
                  CRuntimeClass* pViewClass);
```

To make a document different from the parent, you must create an additional series of resources that share a common name other than `IDR_MAINFRAME`. The most basic resources you should create are a menu and an icon. There are two main types of MDI applications.

One kind of application may use only one particular category of documents such as only text-based. Because text-based documents can include ASCII text files, rich text documents (RTF), HTML files, script-based documents (JavaScript, VScript, Perl, PHP, etc), etc, such an application may be configured to open only that type of document. To create such an MDI application, in the constructor of the `CMultiDocTemplate` object that you are using, specify a `CView` derived class (`CEditView`, `CListView`, etc) or your own class you derived from `CView` or one of its children. Here is an example:

```
BOOL CMultiEdit1App::InitInstance()
{
    CMultiDocTemplate* pDocEdit;
    pDocEdit = new CMultiDocTemplate(
        IDR_EDITTYPE,
        RUNTIME_CLASS(CMultiEdit1Doc),
        RUNTIME_CLASS(CChildFrame),
        RUNTIME_CLASS(CEditView));
    AddDocTemplate(pDocEdit);

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    CCommandLineInfo cmdInfo;

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();
```

```

return TRUE;
}

```

In this case, the user can create and/or open as many text files as the computer memory would allow.

Another type of MDI application you can create would allow the user to create or open more than one type of document. To provide this functionality, you must specify a template for each type. Again, there are various types of techniques you can use. You can ask the user to select the type of document he or she wants to create when the application starts:

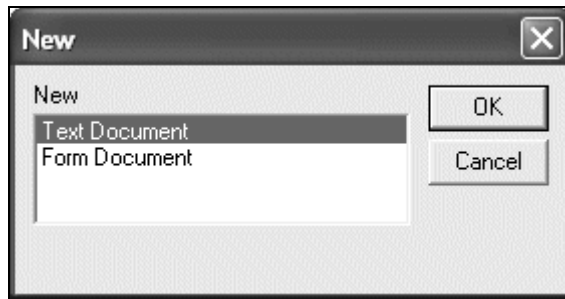


Figure 12.7: Selection of different types of application for MDI

To do this, you can create a template for each type of document using a `CDocMultiDocTemplate` constructor for each:

```

BOOL CMultiEdit1App::InitInstance()
{
    CMultiDocTemplate* pDocEdit;
    pDocEdit = new CMultiDocTemplate(
        IDR_EDITTYPE,
        RUNTIME_CLASS(CMultiEdit1Doc),
        RUNTIME_CLASS(CChildFrame),
        RUNTIME_CLASS(CEditView));
    AddDocTemplate(pDocEdit);

    CMultiDocTemplate* pDocForm;
    pDocForm = new CMultiDocTemplate(
        IDR_FORMTYPE,
        RUNTIME_CLASS(CMultiEdit1Doc),
        RUNTIME_CLASS(CChildFrame),
        RUNTIME_CLASS(CEmplRecords));
    AddDocTemplate(pDocForm);

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
}

```

```
m_pMainWnd = pMainFrame;  
  
CCommandLineInfo cmdInfo;  
  
// Dispatch commands specified on the command line  
if (!ProcessShellCommand(cmdInfo))  
    return FALSE;  
  
pMainFrame->ShowWindow(m_nCmdShow);  
pMainFrame->UpdateWindow();  
  
return TRUE;  
}
```

You can also display an empty main frame when the application starts and let the user create a document based on the available types from the main menu. Here is such a menu from Borland Image Editor:

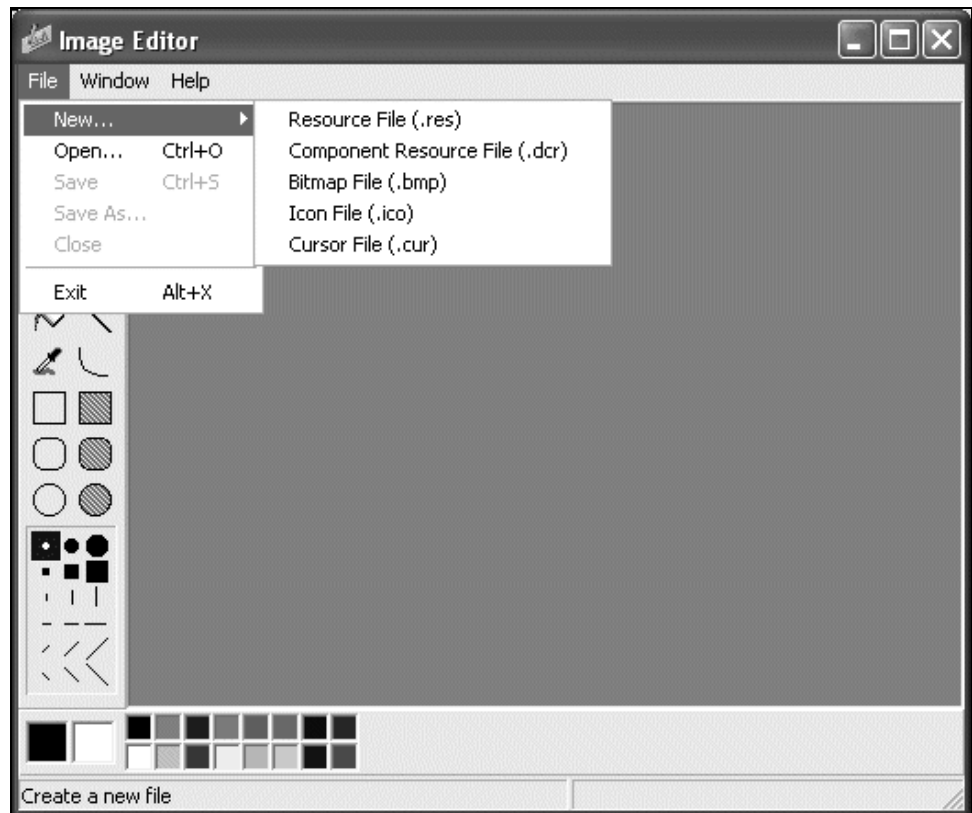


Figure 12.8: Borland Image Editor for empty main frame

As you can see, creating an MDI is not necessarily too difficult but it can involve a few more steps than an SDI.

Check Your Progress 3

Fill in the blanks:

1. The extension of resource file is _____.
2. _____ in the resource script can be used to specify menu bars or popup menus.
3. A dialog statement defines the layout and appearance of a _____.
4. An icon is a graphical object made of _____ categories of colors.

12.5 LET US SUM UP

Resource files define the visual appearance of an application. It includes dialogs, menus, bitmaps, icons, cursors, text strings, and several other types. It is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc. Resource files define the visual appearance of an application. For a computer application, a document holds the user's data. The expression Single Document Interface or SDI refers to a document that can present only one view to the user. This means that the application cannot display more than one document at a time. In the Multiple Document Interface, you can open more than one document at a time.

12.6 LESSON END ACTIVITY

Discuss the different resources of windows environment.

12.7 KEYWORDS

Resource: It is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc.

Identifier: It is a constant integer whose name usually starts with ID.

Accelerators: Those are keystrokes that represent shortcuts to specific tasks.

Icon: It is a small picture used on a window.

Bitmap Resource: It is a bitmap image file (.bmp) stored within a resource file.

Version Information Resource: It is a resource file which can be used to identify the version of a binary file.

View: It is the platform the user is working on to do his or her job.

Document: It holds the user's data.

Frame: It is an object which defines the location of an object with regards to the Windows desktop.

SDI: It refers to a document that can present only one view to the user.

Multiple Document Interface (MDI): It refers to a frame where the user can open more than one document in the application without closing it.

Serialization: It is the conversion of an object to and from a persistent form.

12.8 QUESTIONS FOR DISCUSSION

1. Compare and contrast SDI and MDI.
2. Explain message maps.
3. What are the various parts of Document/View architecture?
4. How to create SDI?
5. Write short note on
 - a) Accelerators
 - b) Message Maps
 - c) Icon
 - d) Menu

Check Your Progress: Model Answers

CYP 1

1. Resource is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc.
2. Menu statement is one type of resource script which can be used to specify menu bars or popup menus.

CYP 2

1. False
2. True

CYP 3

1. .rc
2. Menu statements
3. Dialog box
4. Two

12.9 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up*, 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth, *Visual C++6 Unleashed*, Second edition Techmedia.

UNIT V

LESSON

13

DATA HANDLING IN VC++

CONTENTS

- 13.0 Aims and Objectives
- 13.1 Introduction
- 13.2 Connecting to Data Source
 - 13.2.1 DAO
 - 13.2.2 ODBC
- 13.3 Let us Sum up
- 13.4 Lesson End Activity
- 13.5 Keywords
- 13.6 Questions for Discussion
- 13.7 Suggested Readings

13.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Concept of Data Handling
- Brief idea of DAO and ODBC

13.1 INTRODUCTION

Most business applications work with data. They maintain, manipulate, and access records of data that are stored in databases. If you build business applications, odds are that you will need to be able to access a database with your applications.

In this chapter, you'll explore the technique of database access.

13.2 CONNECTING TO DATA SOURCE

The Microsoft Foundation Classes (MFC) encompass several classes that you can use to provide a simpler C++ interface to databases. These classes are particularly useful when quickly generating applications that present a simple, consistent interface to the user.

MFC provides classes for using the open database connectivity (ODBC) API to interface with ODBC data sources, as well as classes for working with object linking and embedding databases (OLE DB) and Data Access Objects (DAO) to work with desktop databases. In this chapter, you will look at the ODBC classes specifically, although the OLE DB and DAO classes are very similar.

13.2.1 DAO

Data Access Objects, or DAOs, are Microsoft's latest invention in database access technology. This technology is used for database access in Microsoft's Visual Basic 4, Microsoft Access, and Visual Basic for Applications; and now, with the help of a set of specialized MFC classes, it is also available to the Visual C++ programmer.

DAO Overview

Data Access Objects enable you to access and manipulate databases through the Microsoft Jet database engine. Through this engine, you can access data in Microsoft Access database files (MDB files). The technology also enables you to access local and remote databases through ODBC drivers.

Data Access Object technology is based on OLE. Figure 13.1 depicts the hierarchy of Data Access Objects. This hierarchy is greatly simplified by the DAO classes in MFC.

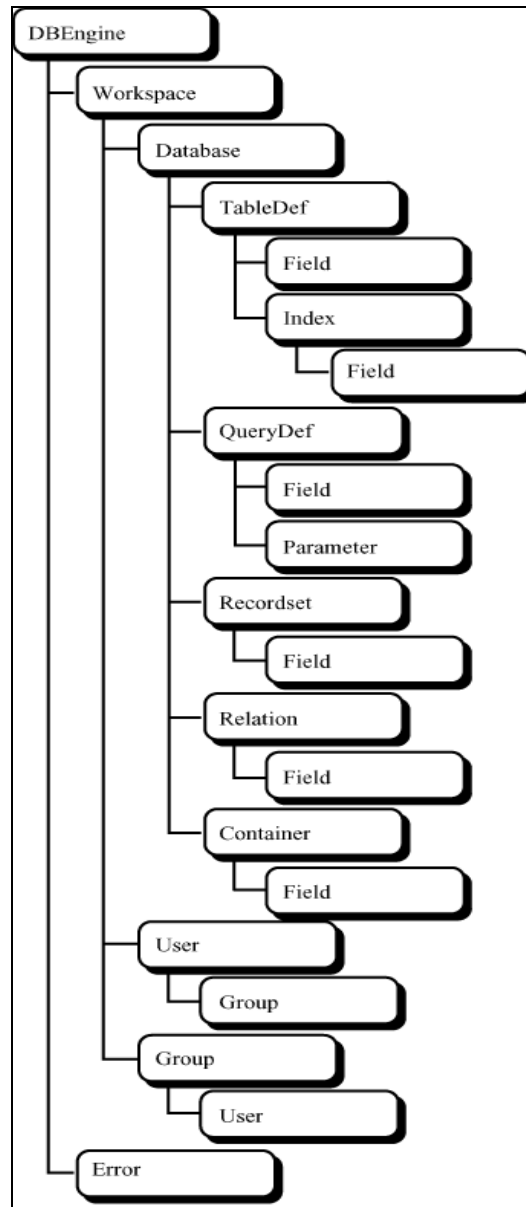


Figure 13.1: DAO object hierarchy

Many DAO functions utilize Structured Query Language (SQL) statements. You can use the SQL SELECT statement to retrieve data from a database, or the SQL UPDATE, INSERT, and DELETE statements to modify the contents of the database. An easy way to create SQL statements for use with DAO objects is to create the query from within Microsoft Access, save the query in the database, and access the query through a QueryDef object.

Visual C++ provides extensive support for building DAO applications through the AppWizard. In addition to ODBC, AppWizard enables you to create applications that are based on DAO Classes. Our tour of DAO begins with the creation of a simple DAO application and exploration of its behavior.

Building a DAO Application

Building a DAO application is quite simple. First, if it does not exist yet, we must create a data source. For the application demonstrated here, the data source is a simple Access database of two tables. Next, the skeleton application must be created using AppWizard; and finally, we must customize this application as appropriate.

The application is a simple browser; it browses a row set that is created as a relational join of two tables.

The Database

The database used in this example contains two tables. One table contains the first names, last names, and ages of employees; the other table contains the names of benefit packages offered to employees and the maximum qualifying age for each package. The purpose of our application, which we decided to call ADAO, is simple: display, for each employee, all benefits packages he or she qualifies for.

The database, `adao.mdb`, is constructed using Microsoft Access. To construct the database, start Access and create a blank database named `adao.mdb` in the directory of your choice. Upon successful creation, select the Table tab in the Database window, and click the New button. Select New Table.

Figure 13.2 shows the newly constructed Employees table just before it is saved. As you can see, three fields were added (LastName, FirstName, and Age) of which the first two are 50-character-wide text fields, the third is a number field. The table's primary key was set to the combination of the LastName and FirstName fields.

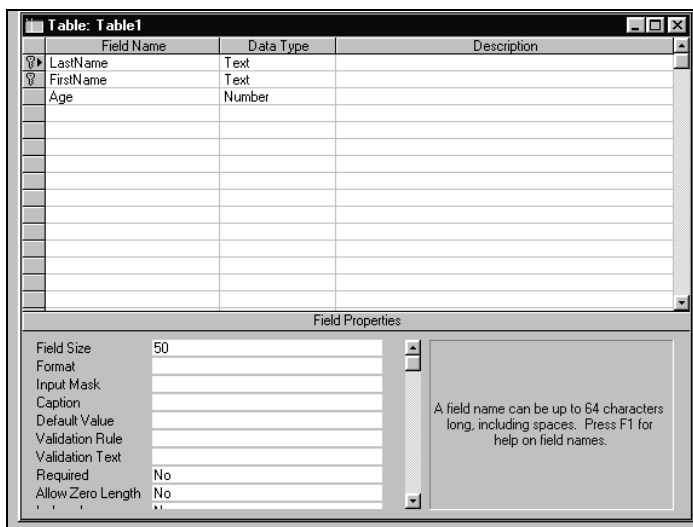


Figure 13.2: Creating the Employees table

Save the new table under the name `Employees` and repeat this procedure to create a second table (Figure 13.3). This table contains information about benefit packages. It contains two fields, the first of which, `Name`, serves as the table's primary key. This table should be saved under the name `Plans`.

Table: Table1

Field Name	Data Type	Description
Name	Text	
MaxAge	Number	

Field Properties

Field Size	50
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	No

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Figure 13.3: Creating the Plans table

When your work creating these tables is done, the Database window should show two tables, as seen in Figure 13.4.

Database: ADAO

New Open Design

Table

Query

Form

Report

Macro

Module

Tables

- Employees
- Plans

Figure 13.4: Tables in adao.mdb

The next step is to add data to these tables. You can do so by simply double-clicking the table's name in the Database window.

Figure 13.5 shows the four records those are added to Employees. Figure 13.6 shows the three records those are added to the Plans table.

	LastName	FirstName	Age
	Doe	John	29
	Doe	Jane	26
	Brown	Joe	44
	Smith	Joseph	24
*			0

Record: 4 of 4

Figure 13.5: Records in the Employees table

	Name	MaxAge
	Plan 'A'	25
	Plan 'B'	40
	Plan 'C'	65
*		0

Record: 3 of 3

Figure 13.6: Records in the Plans table

This is all we need to do with Microsoft Access. Our MDB file is now ready for use from within a C++ DAO application.

Creating the Skeleton Application

To create the ADAO application, launch the AppWizard and start creating a single-document-based project. Database support is specified in AppWizard Step 2; select the Database view without file support option (Figure 13.7).

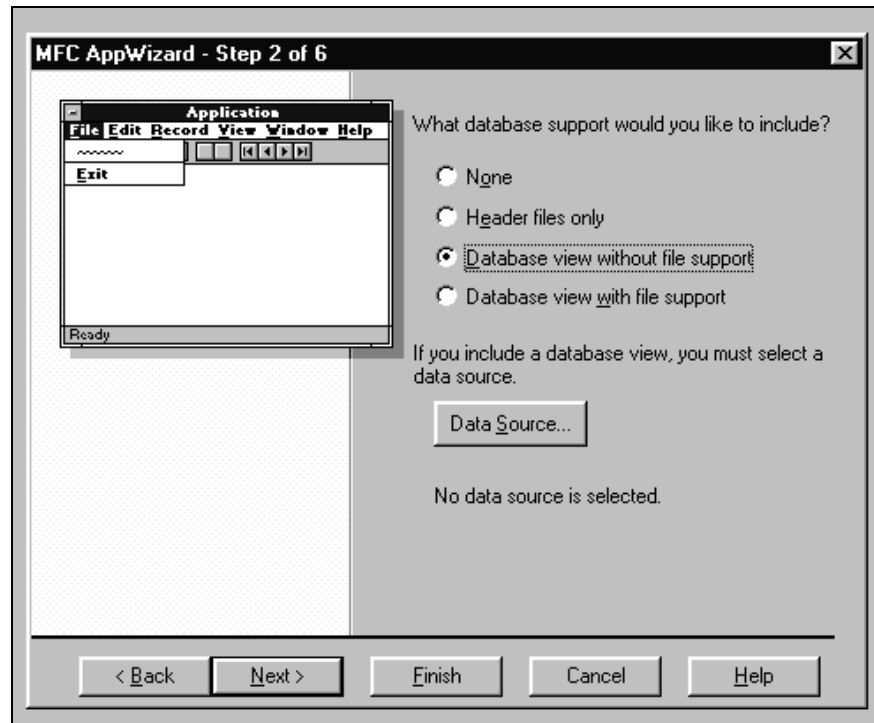


Figure 13.7: Adding database support to a skeleton application

Before you can proceed from this step, you must specify a data source. To do so, click on the Data Source button. In the Database Options dialog that appears, select DAO as the data source (Figure 13.8).

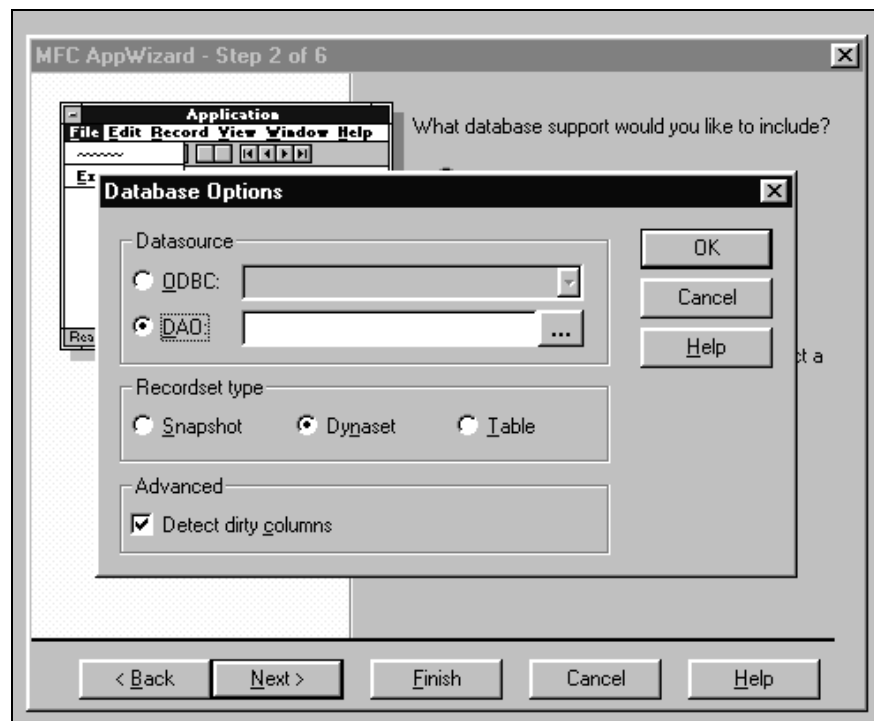


Figure 13.8: Adding a DAO data source

Clicking on the ellipsis button next to the DAO field enables you to specify the actual database file. It brings up a standard File Open dialog, where you can select the file `adao.mdb`. Select this file and when the Database Options dialog reappears, click the OK button. This should display another dialog where the tables of the database can be selected. Select both the Employees and the Plans table and click OK (Figure 13.9).

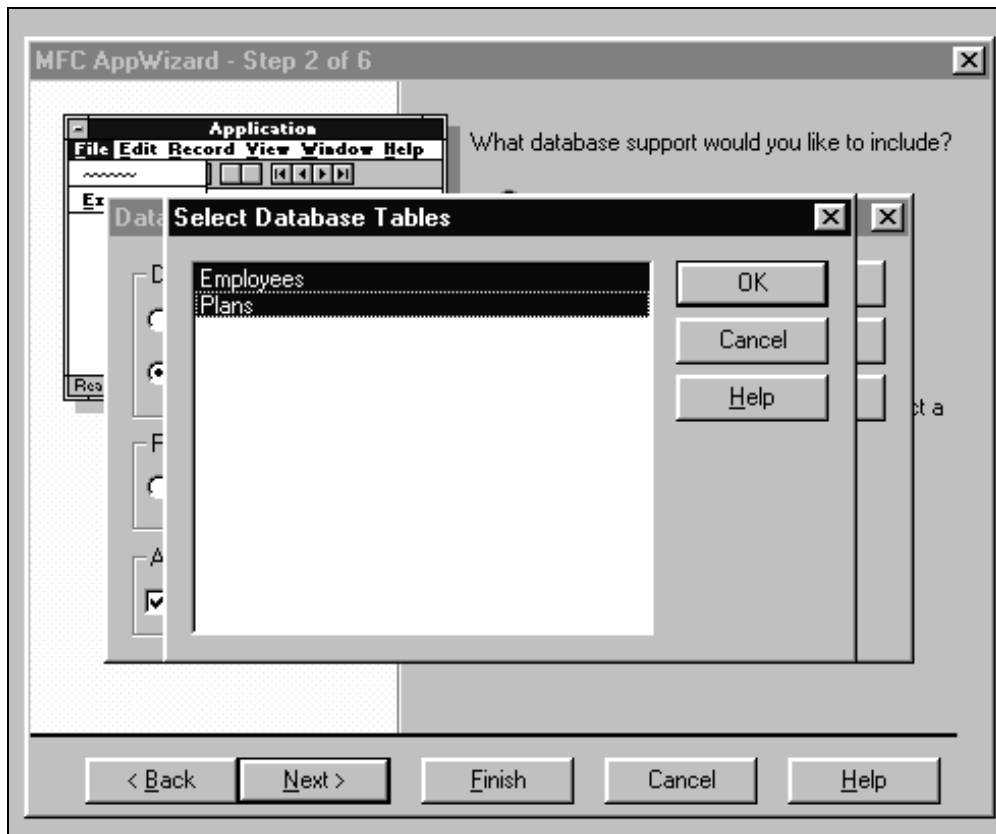


Figure 13.9: Selecting tables

At this time, all dialogs should disappear except for the AppWizard Step 2 dialog; this dialog should now display our data source selection.

All other AppWizard settings can remain at their default values; therefore, you can quickly complete creating the skeleton application by clicking on the Finish button.

Exploring the DAO Application Skeleton

The classes of the skeleton application created by AppWizard are shown in Figure 13.10. When compared to an application with no database support, this application offers one extra class and a few additional member variables and functions in its document and view classes. Not evidently visible, but also a notable difference, is the fact that the view class is derived from `CDaoRecord View`.

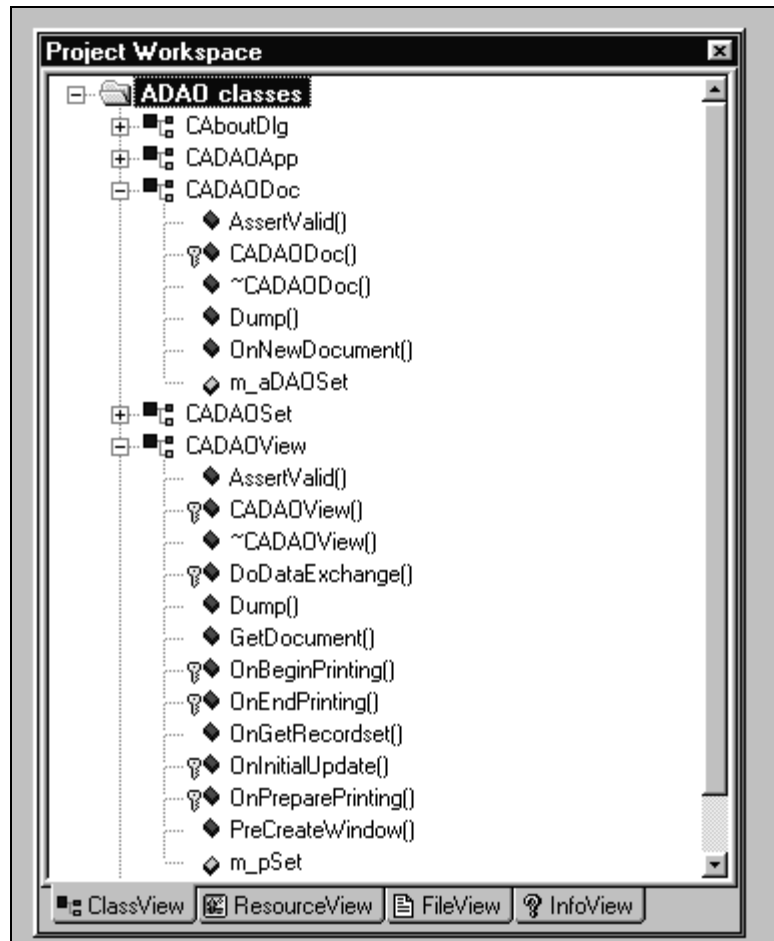


Figure 13.10: Skeleton application classes

The new class, CADAOSet, is derived from CDaoRecordset and represents the row set that we will select from the join of the Employees and Plans table. Looking at this class's declaration, you can see that the AppWizard already inserted member variables that correspond to the columns (fields) in the two tables.

```
CDAOSet class declaration.
class CDAOSet : public CDaoRecordset
{
public:
    CDAOSet(CDaoDatabase* pDatabase = NULL);
    DECLARE_DYNAMIC(CDAOSet)
// Field/Param Data
    //{AFX_FIELD(CDAOSet, CDaoRecordset)
    CString m_LastName;
    CString m_FirstName;
    double m_Age;
    CString m_Name;
    double m_MaxAge;
    //}AFX_FIELD
```

```
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CADAOSet)
public:
virtual CString GetDefaultDBName();
virtual CString GetDefaultSQL();
virtual void DoFieldExchange(CDaoFieldExchange* pFX);
//}}AFX_VIRTUAL
// Implementation
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
};
```

A look at the implementation of CADAOSet shows how these member variables are initialized in the class's constructor. The variables are also referred to in the AppWizard-generated implementation of the DoFieldExchange member function. This function exchanges data between member variables in the class and fields in the database.

CADAOSet class implementation.

```
IMPLEMENT_DYNAMIC(CADAOSet, CDaoRecordset)
```

```
CADAOSet::CADAOSet(CDaoDatabase* pdb)
```

```
: CDaoRecordset(pdb)
```

```
{
    //{{AFX_FIELD_INIT(CADAOSet)
    m_LastName = _T("");
    m_FirstName = _T("");
    m_Age = 0.0;
    m_Name = _T("");
    m_MaxAge = 0.0;
    m_nFields = 5;
    //}}AFX_FIELD_INIT
    m_nDefaultType = dbOpenDynaset;
}

CString CADAOSet::GetDefaultDBName()
{
    return _T("G:\\\\ADAO\\\\adao.mdb");
}

CString CADAOSet::GetDefaultSQL()
{
    return _T("[Employees],[Plans]");
}
```

```
void CDAOSet::DoFieldExchange(CDaoFieldExchange* pFX)
{
    //{AFX_FIELD_MAP(CDAOSet)
    pFX->SetFieldType(CDaoFieldExchange::outputColumn);
    DFX_Text(pFX, _T("[LastName]"), m_LastName);
    DFX_Text(pFX, _T("[FirstName]"), m_FirstName);
    DFX_Double(pFX, _T("[Age]"), m_Age);
    DFX_Text(pFX, _T("[Name]"), m_Name);
    DFX_Double(pFX, _T("[MaxAge]"), m_MaxAge);
    //}AFX_FIELD_MAP
}
```

To do its work, DoFieldExchange makes use of DFX_ functions. These functions are the DAO cousins of the RFX_ functions used for ODBC field exchange. The set of DFX_ functions available for use in DoFieldExchange is summarized in the following table.

Function Name	Field Type	ODBC SQL Type
DFX_Binary	CByteArray	DAO_BYTES
DFX_Bool	BOOL	DAO_BOOL
DFX_Byte	BYTE	DAO_BYTES
DFX_Currency	COleCurrency	DAO_CURRENCY
DFX_DateTime	COleDateTime	DAO_DATE
DFX_Double	double	DAO_R8
DFX_Long	long	DAO_I4
DFX_LongBinary	CLongBinary	DAO_BYTES
DFX_Short	short	DAO_I2
DFX_Single	float	DAO_R4
DFX_Text	CString	DAO_CHAR, DAO_WCHAR

Note: It is recommended that applications not use the DFX_LongBinary function but use DFX_Binary instead. DFX_LongBinary is provided for compatibility with ODBC.

The new member variables and functions in our application's view and document classes are a simple business. The document class, CDAODoc, contains a new member variable of type CDAOSet, m_aDAOSet. Very obviously, this variable represents the recordset that the document is associated with.

The view class contains a pointer of type CDAOSet (m_pSet); in the default implementation, this pointer is set to point to the document object's m_aDAOSet member. The view class also has a new member function, OnGetRecordset, which in the default implementation simply returns m_pSet.

Although you can recompile the ADAO application at this time, as Figure 13.11 illustrates, it is not yet a very useful application. We need to customize its dialog, and we also need to add the necessary operations that will restrict the rows selected to rows that we wish to see.

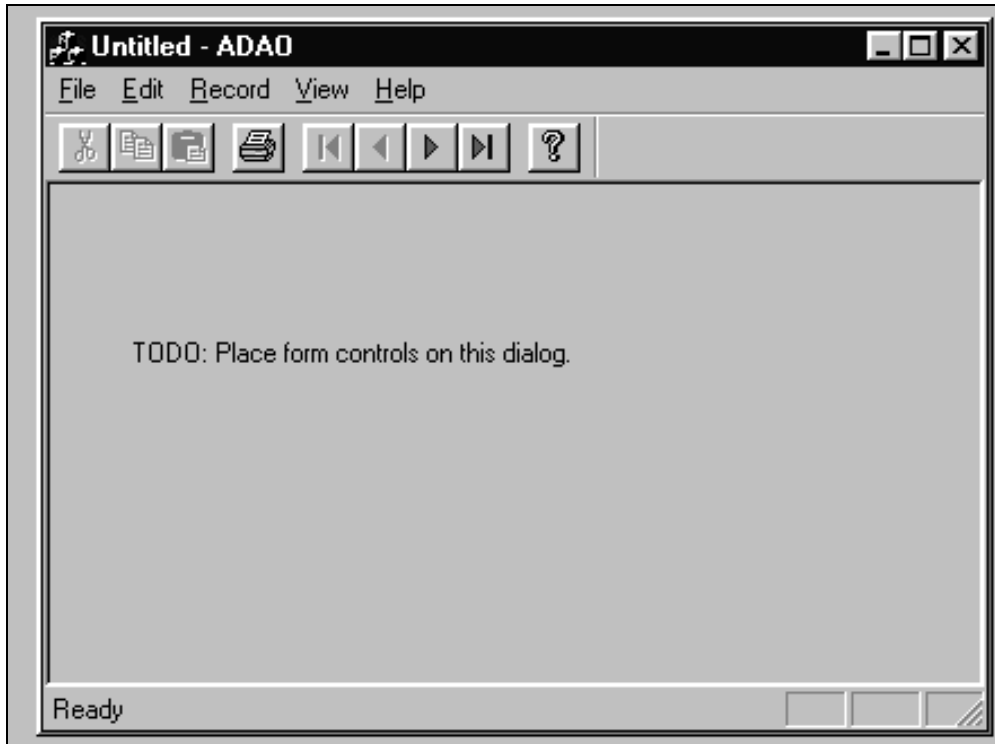


Figure 13.11: Running the ADAO application skeleton

Customizing the Application

The first step in customizing the ADAO application is changing its main dialog. Open the IDD_ADAO_FORM dialog for editing, remove the default "TODO" static control, and add static controls and edit controls as shown in Figure 13.12.

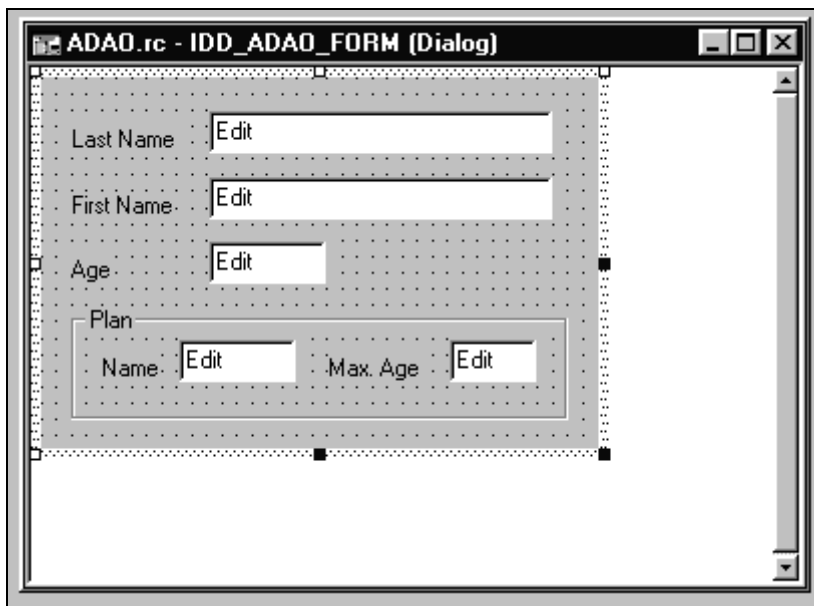


Figure 13.12: Customizing the ADAO dialog

Name the five edit fields IDC_LASTNAME, IDC_FIRSTNAME, IDC_AGE, IDC_NAME, and IDC_MAXAGE as appropriate. Before dismissing this dialog, you can also use the ClassWizard to identify dialog fields with corresponding recordset member variables.

To do so, hold down the Control key, and double-click on the IDC_LASTNAME edit field. The ClassWizard Add Member Variable dialog appears, with ClassWizard's guess as to the name of the member variable (Figure 13.13). ClassWizard derives its guess by looking at the static fields in the dialog.

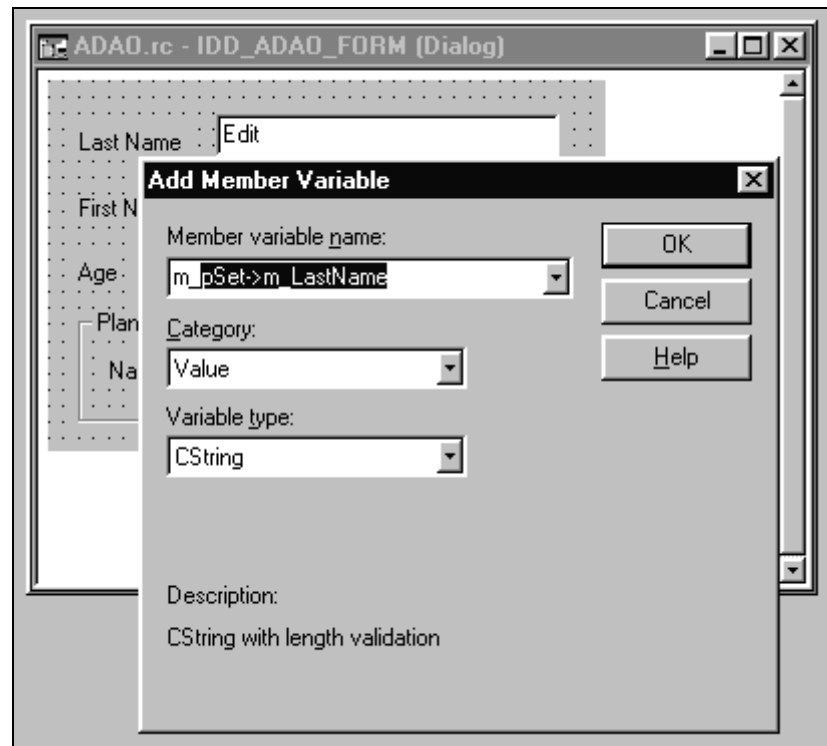


Figure 13.13: Adding a recordset member variable

ClassWizard's guess is appropriate for the first three fields in our dialog; however, for the plan name and plan maximum age fields, it is necessary to manually change ClassWizard's selection. This can be done by selecting the proper `m_pSet` member variable from the drop-down list in the Add Member Variable dialog.

After you specified the member variables for all five edit fields, you can dismiss the dialog. However, we are not done yet; we have not yet specified anywhere the selection criteria that would make our application display only the rows representing valid plans for each employee.

To change the selection criteria, open the `CADAOSet::GetDefaultSQL` function for editing. The default implementation of this function simply returns the table names that form the recordset. What we wish to do is add additional criteria that would restrict the selections from the tables to only those rows that we wish to see.

In SQL, our desired selection can be expressed in the form of a SELECT statement:

```
SELECT Employees.LastName, Employees.FirstName, Employees.Age,  
       Plans.Name, Plans.MaxAge  
FROM Employees, Plans  
WHERE Employees.Age < Plans.MaxAge  
ORDER BY Employees.LastName, Employees.FirstName, Plans.Name
```


Indeed, one way to specify our selection would be to change `GetDefaultSQL` to return a string representing the above SQL `SELECT` statement. However, there is another way; and that is to utilize the member variables of the `CDaoRecordset` class.

In particular, `CDaoRecordset` offers two member variables, one of which corresponds to the SQL `WHERE` clause, the other, to the SQL `ORDER_BY` clause. Our new version of `CADAOSet::GetDefaultSQL` utilizes these member variables to create the desired selection of rows.

Updated version of `CADAOSet::GetDefaultSQL`.

```
CString CADAOSet::GetDefaultSQL()
{
    m_strFilter = "[Employees].[Age] < [Plans].[MaxAge]";
    m_strSort =
        "[Employees].[LastName],[Employees].[FirstName],[Plans].[Name]";
    return _T("[Employees],[Plans]");
}
```

This completes our work on the ADAO project. Recompiling and running the application (Figure 13.14) shows that it indeed behaves as expected, displaying benefit plans that employees qualify for, ordered by the name of the employee and the name of the benefit package.

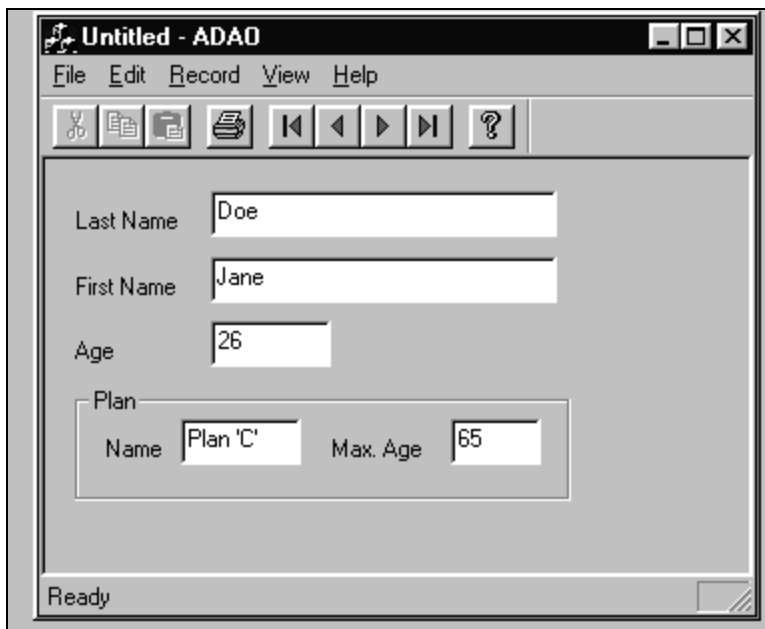


Figure 13.14: Running the ADAO application

DAO Classes

Although the ADAO application demonstrates how a simple DAO program can be created, it fails to demonstrate many DAO features. To remedy this deficiency, in the remainder of this chapter we take a brief tour of MFC DAO classes and their capabilities.

The set of DAO classes offered by MFC is shown in Figure 13.15. In addition to the `CDAORecordset` class that we have encountered while constructing ADAO, there are four other major classes and two helper classes related to DAO. Still, this is a significant improvement over the multitude of raw DAO objects (Figure 13.15).

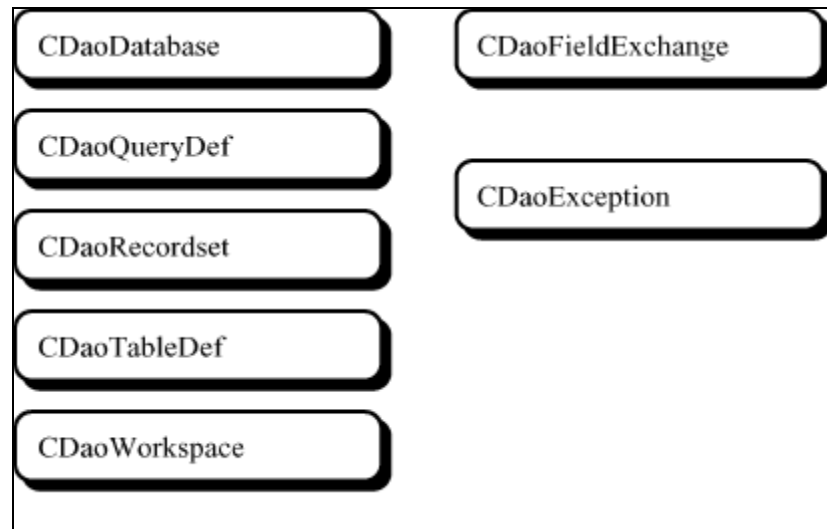


Figure 13.15: DAO Classes

Before we review the role and features of each of these classes one by one, this section presents a brief overview of how DAO works. For this, it might be helpful to take another look at Figure 1, at the beginning of this chapter.

All DAO objects are derived from the `DBEngine` object; furthermore, all database objects are derived from DAO workspace objects. However, unless you need to manipulate secure databases, you typically do not need to reference either of these; instead, a default workspace object is assumed for all transactions.

The database and recordset objects very obviously represent databases and selection sets (tables, recordsets, or dynasets) in those databases.

Query definition (QueryDef) objects are used to execute specific SQL queries against a database. Query definitions are normally used in conjunction with recordsets to access data in a database using a specific query.

Table definition (TableDef) objects represent the structure of tables in the database. Through table definition objects, it is possible to create new tables, and to modify the structure and characteristics of existing tables.

There are several other DAO object types. These object types (Field objects, Parameter objects, Index objects, User objects, Group objects, and Error objects) are not represented by specific MFC classes. Instead, DAO objects of this type are accessed through the other DAO MFC classes as appropriate.

The CDAORecordset Class

`CDAORecordset` objects represent recordsets. A recordset can represent records in a table, a dynaset, and a snapshot. A table-type recordset is updatable and represents the records in a single table. A dynaset-type recordset represents records from one or more tables as a result of a query; dynaset records are also updatable. A snapshot, on the other hand, can also contain fields from one or more tables but these fields are not updatable; the snapshot is a static copy of records used to find data or generate reports.

A recordset is created by calling the `CDaoRecordset::Open` member function. The three forms of this function enable you to create a recordset using an SQL query string, a `CDaoTableDef` object, or a `CDaoQueryDef` object.

The `CDaoRecordset` class offers a large number of member functions. Perhaps the most important among these are recordset navigation functions and data update functions. The navigation functions include `Find`, `FindFirst`, `FindLast`, `FindNext`, and `FindPrev`; and `Move`, `MoveFirst`, `MoveLast`, `MoveNext`, and `MovePrev`. Data update functions include `AddNew`, `CancelUpdate`, `Delete`, `Edit`, and `Update`.

Other navigation-related functions include `GetAbsolutePosition`, `Getguidemark`, `GetPercentPosition`, and `SetAbsolutePosition`, `Setguidemark`, and `SetPercentPosition`.

The `CDaoRecordset` class offers a variety of attribute functions to set and retrieve recordset attributes. For example, the `CanUpdate` function can be used to determine whether a recordset is updatable; the `SetCurrentIndex` function can be used to set the current index on a table-type recordset.

Normally, you use the `CDaoRecordset` class by deriving your own recordset class from it, adding member variables that represent fields, and overriding the `DoFieldExchange` member function to facilitate the exchange of data between the database and the member variables. However, several member functions exist that provide an alternative. These include `GetFieldValue` and `SetFieldValue`, which enable you to directly access the value of a field by name. This method is referred to as dynamic binding, as opposed to the static binding accomplished through `DoFieldExchange`.

Other recordset operations can be used to control the locally maintained cache of records and to manipulate recordset indexes.

The CDaoDatabase Class

The `CDaoDatabase` class represents a connection to a database. A connection is created by calling `CDaoDatabase::Open` and terminated by calling `CDaoDatabase::Close`. A new database can be created by calling `CDaoDatabase::Create`.

The `CDaoDatabase` class offers a series of attribute member functions; for example, the `GetName` member function can be used to retrieve the name of the database, or the `IsOpen` member function can be used to determine if the connection represented by the `CDaoDatabase` object is open.

Other member functions can be used to manipulate the collections of table definition and query definition objects that are defined for this database. In particular, you can use the `DeleteTableDef` member function to delete not only a DAO `TableDef` object but also the underlying table and all its data from the database.

The CDaoWorkspace Class

The `CDaoWorkspace` class represents database sessions. Typically, you do not need to create objects of type `CDaoWorkspace`, unless you wish to utilize specific functionality available through this class or to access password-protected databases.

A DAO workspace can be created by calling `CDaoWorkspace::Create`. Arguments to this function specify the name of the workspace, the user name, and password. An existing workspace object can be opened by calling `CDaoWorkspace::Open`; by passing a `NULL` parameter to this function, you can explicitly open the default workspace.

Several member functions exist that manipulate databases and the database engine itself. For example, you can compact or repair a database by calling the CompactDatabase or RepairDatabase member functions. Other functions can be used to manipulate user names, passwords, and other database attributes.

The CDaoQueryDef Class

The CDaoQueryDef class represents query definitions. To create a new query definition, use the CQueryDef::Create member function; to access a query definition that was saved into a database, use CQueryDef::Open. A newly created query can be added to the database by calling the CQueryDef::Append member function.

CQueryDef objects can be used in conjunction with CRecordSet objects to retrieve data from the database. CQueryDef objects can also be used directly; to execute an action query that modifies the data in the database, use the CQueryDef::Execute member function.

Other CQueryDef member functions can be used to set and retrieve query definition attributes and to manipulate query fields and parameters.

The CDaoTableDef Class

The CDaoTableDef class represents table definitions. A table definition describes the structure and attributes of a table in a database.

You can open an existing table definition in a database by calling CDaoTableDef::Open. A new table definition can be created by calling CDaoTableDef::Create. To add a table corresponding to a new definition to the database, call the Append member function.

Fields can be created and deleted by calling the CreateField and DeleteField member functions. Indexes for the table can be created or deleted by calling CreateIndex and DeleteIndex. Other member functions can be used to set or retrieve various table attributes; for example, GetFieldCount returns the number of fields in the table, and SetValidationRule can be used to assign a validation rule to a field.

Miscellaneous DAO Classes

In addition to the five fundamental DAO classes, DAO operations make use of two additional classes: CDaoFieldExchange and CDaoException.

CDaoFieldExchange is used in calls to CDaoRecordset::DoFieldExchange. An object of type CDaoFieldExchange defines the field that is affected by the field exchange operation and provides other parameters that characterize the field exchange.

All DAO classes utilize exception objects of type CDaoException to report errors.

Check Your Progress 1

True or False:

1. Data Access Object technology is based on OLE.
2. CADAODoc is a document class

13.2.2 ODBC

ODBC, or Open Database Connectivity, represents a vendor-independent mechanism for accessing data in a variety of data sources.

ODBC drivers are available for many different types of data sources. You can use ODBC to retrieve data from text files, dBase tables, Excel spreadsheets, SQL Server databases, and many other sources.

Many ODBC drivers are redistributable. You can package your application for installation with the appropriate ODBC drivers and software for driver installation and management.

At the heart of ODBC is its capability to execute SQL (Structured Query Language) statements against data sources. The MFC Library provides extensive support for ODBC applications. A series of classes exists encapsulating ODBC databases, tables, and records. The AppWizard supports the creation of ODBC applications, and further support for ODBC is provided by ClassWizard.

ODBC in Action

This section presents a review of some of the fundamental concepts of ODBC that we need to cover before we can begin an attempt to create an ODBC application.

The ODBC Setup Applet

Invoked through the Control Panel or as a stand-alone application, the ODBC setup applet is used to register data sources.

What exactly is a data source? That depends on the driver. In the case of a driver such as the SQL Server driver, the data source can be a database on a server. In the case of a driver such as the Microsoft Access or Microsoft Excel drivers, the database is a file (an MDB or XLS file). In the case of the Microsoft Text driver, the database is a disk directory that contains text files, which serve as tables in the database from the driver's perspective.

To add a data source, invoke the ODBC setup applet and select the Add button. In the resulting dialog (Figure 13.16), pick a driver and click OK.

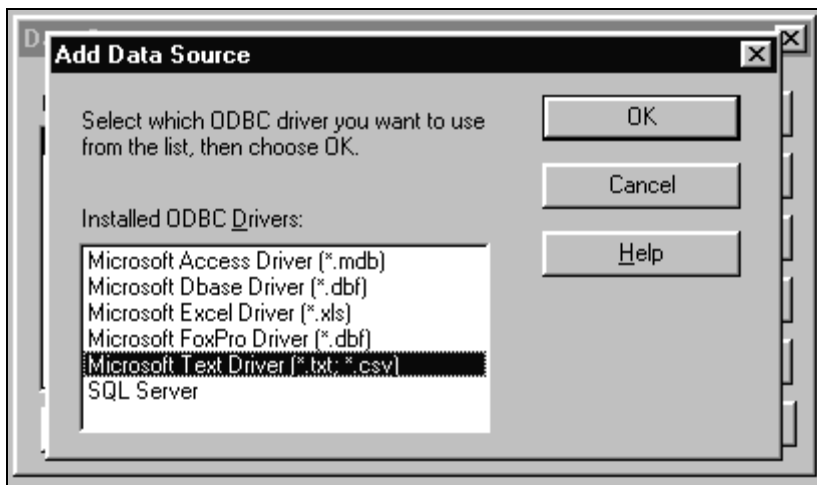


Figure 13.16: ODBC Add Data Source dialog

Next, a driver-specific dialog is displayed (Figure 13.17), where you can select the database and adjust the desired characteristics of the driver.

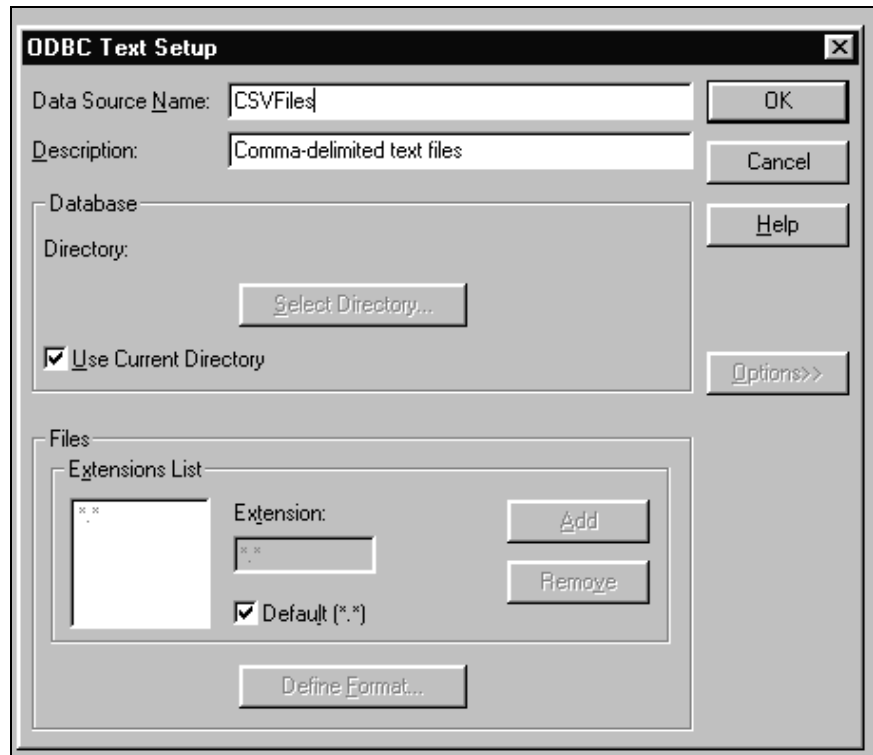


Figure 13.17: ODBC driver setup (Microsoft Text driver)

The ODBC setup applet's main dialog (Figure 13.18) lists all installed data sources. You can add or delete data sources, or you can modify the setup of existing data sources using this dialog.

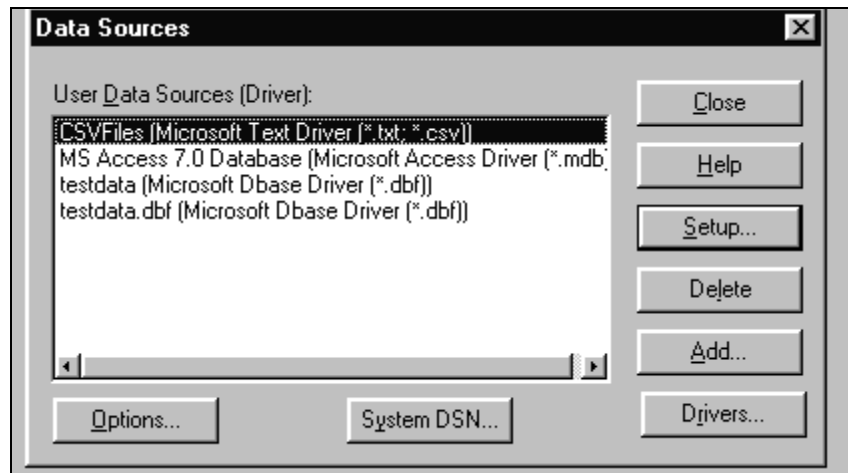


Figure 13.18: ODBC Setup Applet

ODBC API Concepts

Applications that use ODBC rely on ODBC drivers for data access. Drivers can be single-tier or multiple-tier. Single-tier drivers process ODBC calls and SQL statements. Multiple-tier drivers process ODBC calls and pass SQL statements to the data source (potentially a server residing elsewhere on the network).

The ODBC standard defines three conformance levels. The Core API includes those fundamental ODBC calls that are required to access a data source and execute SQL commands. The Level 1 API contains a set of additional calls used to retrieve

information about data sources and the driver itself. The Level 2 API contains additional calls, such as calls that operate using parameter and result arrays. As some drivers may not support Level 2 calls (although most support Level 1), it is important to know whether a particular command is available or not; ODBC references clearly mark each command with the API level that it conforms to.

With respect to the SQL grammar, ODBC defines a core grammar and two variants: a minimum SQL grammar and an extended grammar.

Note that ODBC is not equivalent to Embedded SQL. Embedded SQL uses SQL statements in source programs written in another language. Such a hybrid program is processed by a precompiler before it is passed to the compiler of the host programming language.

In contrast, ODBC interprets SQL statements at run-time. The host program does not need to be recompiled to execute different SQL statements, nor is it necessary to compile separate versions of a host program for different data sources.

An ODBC application has to perform a series of steps to connect to a data source before it can execute SQL statements. These steps are illustrated in Figure 13.19.

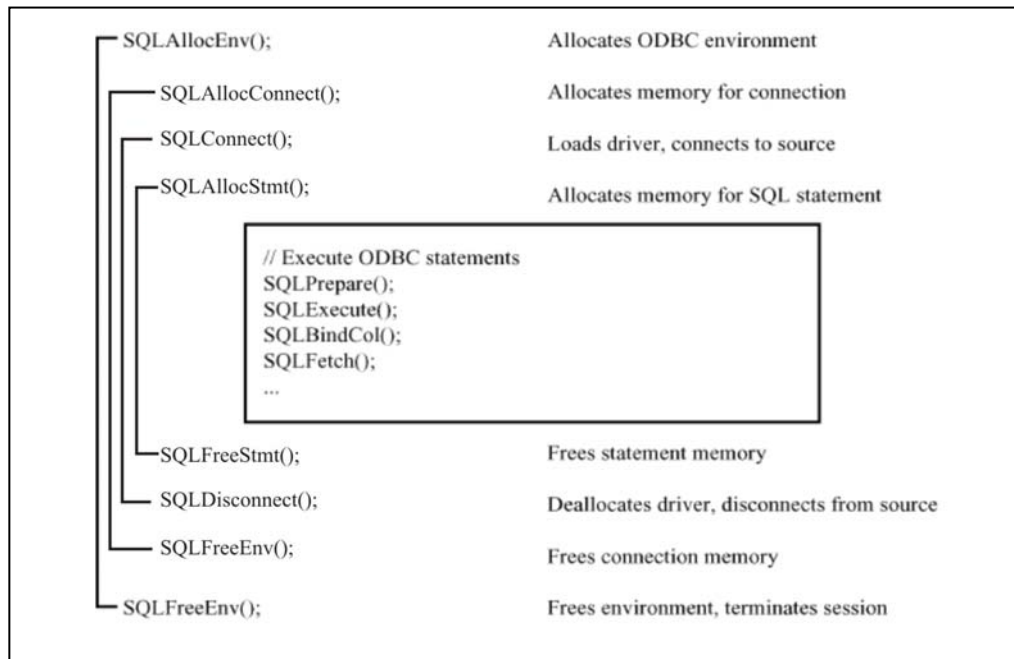


Figure 13.19: Typical set of ODBC calls

The first of the calls in Figure 16, `SQLAllocEnv`, allocates an ODBC environment. In effect, this call initializes the ODBC library and returns an environment handle of type `SQLENVH`.

The second call, `SQLAllocConnect`, allocates memory for a connection. The handle that is returned by this function, of type `SQLHDBC`, is used in subsequent ODBC function calls to refer to a specific connection. One application can maintain several open connections.

The third call, `SQLConnect`, establishes a connection by loading the driver and connecting to the data source. This call has alternatives; for example, the `SQLDriverConnect` call can be utilized to connect to data sources that are not set up via the ODBC setup applet.

Memory for an SQL statement is allocated through a call to `SQLAllocStmt`. By allocating memory for statements in a separate step, ODBC offers a mechanism whereas statements can be constructed, used, and reused before the memory allocated for them is released.

After these four calls, a typical ODBC application performs a series of calls to execute SQL statements against a database. It can use `SQLPrepare` to prepare (compile) an SQL statement for execution and `SQLExecute` to actually execute it. It can use a variety of calls to bind variables to statements and to retrieve the results of a statement.

When its work is finished, the application should free the ODBC resources it has allocated. The statement handle is freed by calling `SQLFreeStmt`. The connection is terminated by calling `SQLDisconnect`; the memory allocated for the collection is released by a call to `SQLFreeConnect`. Finally, the ODBC environment is released by calling `SQLFreeEnv`.

ODBC in MFC Applications

The use of ODBC is greatly simplified by the Microsoft Foundation Classes Library. Simple applications that access tables through ODBC can be created with only a few mouse clicks using the AppWizard and ClassWizard. Several MFC classes exist that support accessing databases and recordsets.

Our discussion of ODBC-related features in the MFC Library starts with the construction of a simple example.

Setting Up a Data Source

Before an MFC ODBC application can be constructed using AppWizard, it is necessary to identify a data source on which the application will operate. The data source must be identified and set up through the ODBC setup applet.

The data source used in our example application is a text file. To access this file, we need the Microsoft Text ODBC driver. (If you did not install this driver when you set up Visual C++, rerun the Visual C++ setup program.)

The data file, `ages.txt`, will contain a set of records with first names, last names, and ages. The first row in the file will be used as a header row. The file will be a comma-separated file, with the following contents:

```
LastName,FirstName,Age  
  
Doe,John,29  
  
Doe,Jane,26  
  
Smith,Joe,44  
  
Brown,Joseph,27
```

After creating this file, we must identify the data source through the 32-bit ODBC setup applet. Invoke this applet and click on the Add button; select the Microsoft Text Driver in the dialog shown in Figure 13.20.

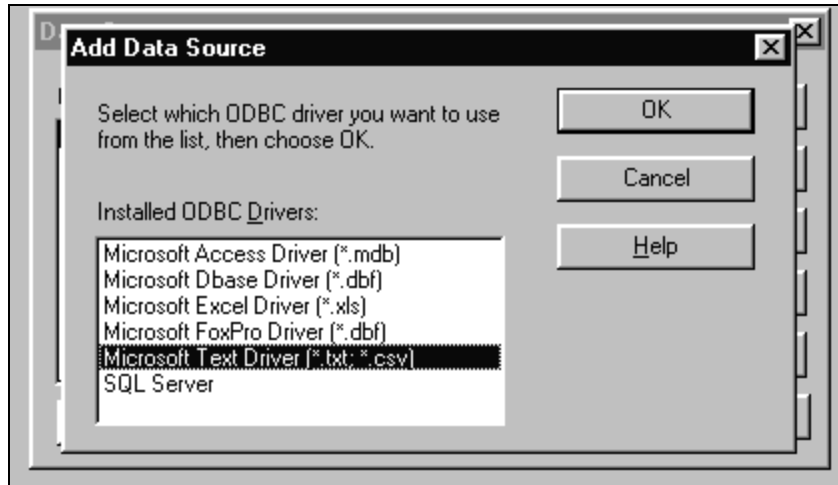


Figure 13.20: Adding a text data source

Clicking on this dialog's OK button invokes the ODBC Text Setup dialog (Figure 13.21), which is a dialog specific to the selected driver. The Microsoft Text driver views disk directories as databases and individual text files as tables in the database. The driver can be set up to use either the current directory or a specific directory as the data source.

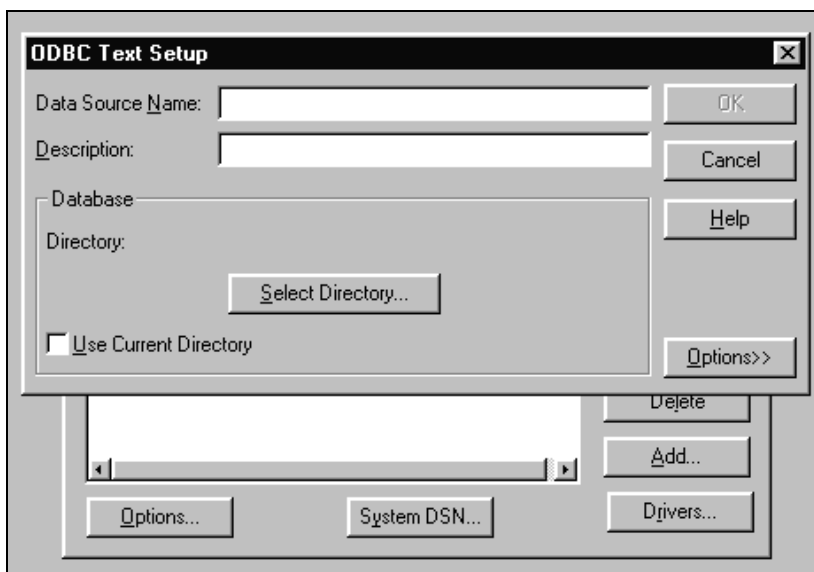


Figure 13.21: ODBC Text Setup

If you select a specific directory, the driver enables, through the Options extension of its dialog, setting up individual tables (text files). For example, it is specified that g:\amfc as the directory where the new application will be placed and created ages.txt in that directory. After specifying this directory name by clicking on the Select Directory button, the Define Format button became active in the ODBC Text Setup dialog (Figure 13.22).

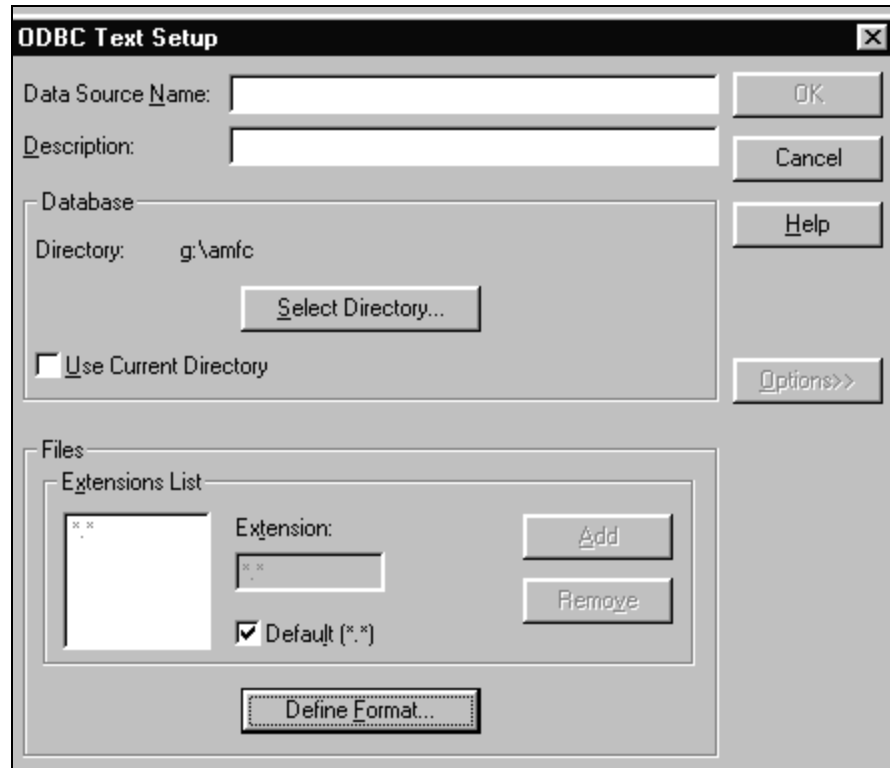


Figure 13.22: ODBC Text Setup options

Clicking on the Define Format button brings up yet another dialog (Figure 13.23) where the format of individual tables (text files) can be specified. In the case of the ages.txt table, setting the Column Name Header check box enables the Guess button to work correctly and retrieve the names of fields and correctly guess their type.

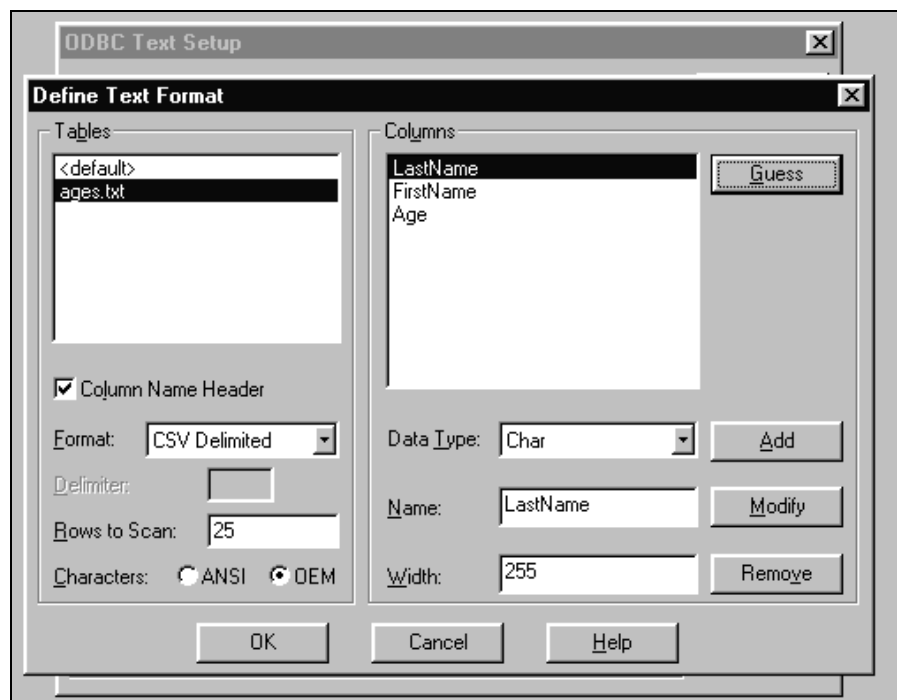


Figure 13.23: Defining the format of a text table

Dismiss this dialog by clicking on the OK button. When the ODBC Text Setup dialog reappears, add a name to this data source. We decided to name this data source "CSV Files in AMFC." Dismiss this dialog, too, by clicking on its OK button, and dismiss the Data Sources dialog by clicking on its Close button.

At this point, a look at the amfc directory where the ages.txt file resides reveals that the ODBC setup applet created another file, one named schema.ini. This file, shown in the following listing, contains information on the characteristics of the ODBC data source that we just specified.

The schema.ini file created by the ODBC setup applet.

```
[ages.txt]

ColNameHeader=True

Format=CSVDelimited

MaxScanRows=25

CharacterSet=OEM

Col1=LASTNAME Char Width 255

Col2=FIRSTNAME Char Width 255

Col3=AGE Integer
```

Now that our data source has been set up and identified, we can turn to the AppWizard to construct a skeleton for our application.

Check Your Progress 2

Write the full form of the following abbreviations:

1. DAO
2. OLE DB
3. ODBC
4. SQL

13.3 LET US SUM UP

MFC provides classes for using the open database connectivity (ODBC) API to interface with ODBC datasources, as well as classes for working with object linking and embedding databases (OLE DB) and Data Access Objects (DAO) to work with desktop databases. Data Access Objects enable you to access and manipulate databases through the Microsoft Jet database engine. Data Access Object technology is based on OLE. Visual C++ provides extensive support for building DAO applications through the AppWizard. ODBC, or Open Database Connectivity, represents a vendor-independent mechanism for accessing data in a variety of data sources. At the heart of ODBC is its capability to execute SQL (Structured Query Language) statements against data sources.

13.4 LESSON END ACTIVITY

Create your own SDI and MDI applications for reading and writing files to the disk drive.

13.5 KEYWORDS

Data Access Object: It enables you to access and manipulate databases through the Microsoft Jet database engine.

Recordset: It can be represented by records in a table, a dynaset, and a snapshot.

Table-type Recordset: It is updatable and represents the records in a single table.

Dynaset-type Recordset: It represents records from one or more tables as a result of a query; dynaset records are also updatable.

Snapshot-type Recordset: It contains fields from one or more tables but these fields are not updatable; the snapshot is a static copy of records used to find data or generate reports.

Open Database Connectivity: It represents a vendor-independent mechanism for accessing data in a variety of data sources.

13.6 QUESTIONS FOR DISCUSSION

1. Do you prefer to use the ODBC interface instead of the Data Access Objects? Explain your answer.
2. How can you add different record sets in an MDI application?
3. What sequence of functions do you need to call to add a new record to a record set?
4. What does ODBC stand for? What view class should you use with an ODBC application?
5. Build an MDI application using a custom document type. Save and restore the document.

Check Your Progress: Model Answers

CYP 1

1. True
2. True

CYP 2

1. Data Access Object
2. Object linking and embedding databases
3. Open database connectivity
4. Structured Query Language

13.7 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up*, 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth, *Visual C++6 Unleashed*, Second edition, Techmedia.

LESSON

14

THREAD-BASED MULTITASKING

CONTENTS

- 14.0 Aims and Objectives
- 14.1 Introduction
- 14.2 Thread-based Multitasking
- 14.3 Let us Sum up
- 14.4 Lesson End Activity
- 14.5 Keywords
- 14.6 Questions for Discussion
- 14.7 Suggested Readings

14.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- Concept of Multithreading
- How to create and how to use thread

14.1 INTRODUCTION

Sometimes it is useful to arrange for two or more processes to work together to accomplish one goal. One situation where this is beneficial is where the computer's hardware offers multiple processors. In the old days this meant two sockets on the motherboard each populated with an expensive Xeon chip. Thanks to advances in VLSI integration, these two processor chips can now fit in a single package. Examples are Intel's "Core Duo" and AMD's "Athlon 64 X2". If you want to keep two microprocessors busy working on a single goal, you basically have two choices: Either design your program to use multiple processes (which usually means multiple programs), or design your program to use multiple threads.

14.2 THREAD-BASED MULTITASKING

Multithreading is a specialized form of multitasking. In general, there are two types of multitasking: process-based and thread-based.

A thread is a dispatchable unit of executable code. The name comes from the concept of a "thread of execution." In a thread-based multitasking environment, all processes have at least one thread, but they can have more. This means that a single program can perform two or more tasks concurrently. For instance, a text editor can be formatting text at the same time that it is printing, as long as these two actions are being performed by two separate threads. The differences between process-based and thread-based multitasking can be summarized like this: Process-based multitasking

handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

The true concurrent execution is possible only in a multiple-CPU system in which each process or thread has unrestricted access to a CPU. For single CPU systems, which constitute the vast majority of systems in use today, only the appearance of simultaneous execution is achieved. In a single CPU system, each process or thread receives a portion of the CPU's time, with the amount of time determined by several factors, including the priority of the process or thread. Although truly concurrent execution does not exist on most computers, when writing multithreaded programs, you should assume that it does. This is because you can't know the precise order in which separate threads will be executed, or if they will execute in the same sequence twice. Thus, it's best to program as if true concurrent execution is the case.

Before you start writing the code, you must think twice on threading. First you have to identify the tasks that can be done by threads. Second you think or innovate a way to communicate between threads. Finally you start writing the code. For example an application receives some data on the serial port, computer the data and than plots some graphical means. For this application to run properly, you will have to create 3 threads:

- one for communication (working thread)
- one for computation (working thread)
- one for user interface (user thread)

Now that you have created the 3 threads, it is time for you to start thinking of communication part. The first thread implements serial communication and it will transmit to the second thread (the computational one) the received and unpacked data. The second thread will do the computation and will send the results to the user-interface thread that will plot the data. Imagine that no thread had been created. The computation will be something like $10 \exp 30$. This will slow down or even kill your application. Or imagine that some problems may occur with the serial (e.g. waiting several seconds to receive some data that will not show up). This will also block your application.

Check Your Progress 1

True or False:

1. Multithreading is a specialized form of multitasking.
2. Thread-based multitasking deals with the concurrent execution of pieces of the different program.

How to create a thread

Right click the class view folder that contains your program and choose new class.

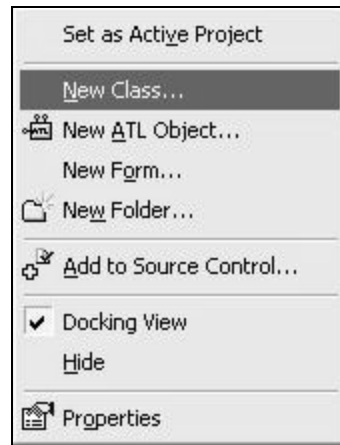


Figure 14.1: Context menu of the class view folder

Choose the Base class to be CWinThread and name your class CThreadEx1.

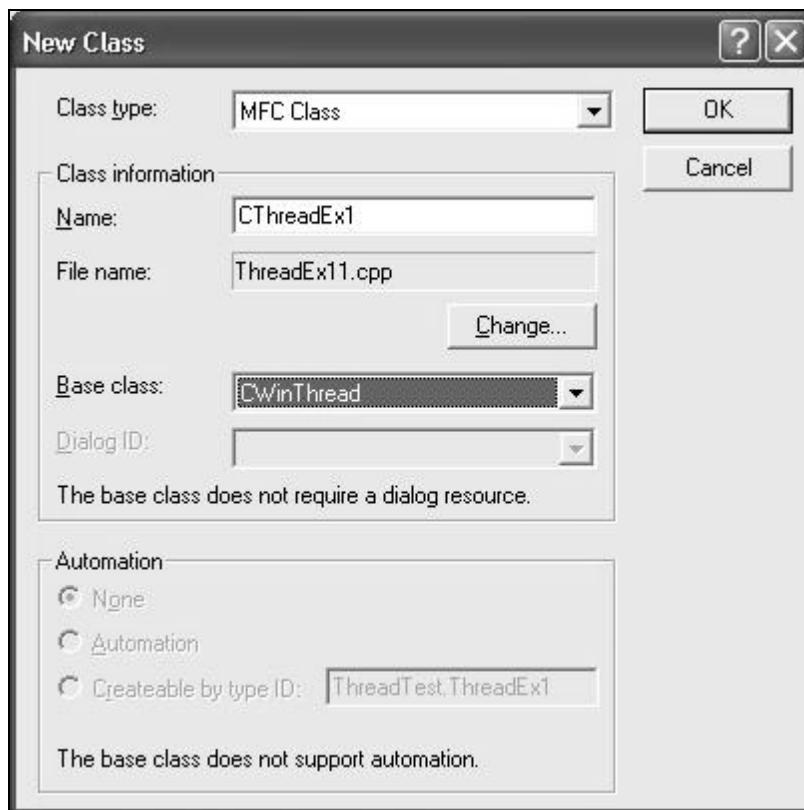


Figure 14.2: configuration screen of ClassView

Hitting OK will bring you to the next configuration in your ClassView

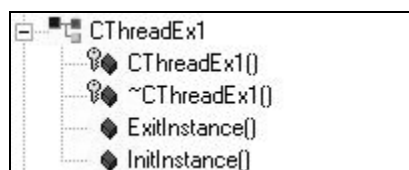


Figure 14.3: Functions for new class

The class has now the constructor, the destructor and two methods: `InitInstance()` and `ExitInstance()`. In the `InitInstance` do the initialization and in the `ExitInstance` do the cleanup. NOTE! The Constructor is protected! Move the constructor to the public zone, this way you can create thread objects in the `OnCreate` method of your application.

How to use a thread

You will add two more methods: `OnIdle` (use the ClassWizard to add this) an `IsIdleMessage`. The last one you will insert manually. In the header define virtual `BOOL IsIdleMessage(MSG* pMsg)`; and the public overrides will look like this:

```
public:
    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CThreadEx1)
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    virtual BOOL OnIdle(LONG lCount);
    virtual BOOL IsIdleMessage(MSG* pMsg);
    //
}}
AFX_VIRTUAL
```

This will be the body of `IsIdleMessage` for the moment:

```
BOOL CThreadEx1::IsIdleMessage(MSG* pMsg)
{
    return CWinThread::IsIdleMessage(pMsg);
}
```

The communication to a thread it is done by messages. The thread receives messages in a queue and pops the messages one by one. The messages are then unpacked, understood, acknowledged and, of course, executed. When is ready to be processed, every message in the queue will be extracted and this event will occur in `IsIdleMessage`. Here you will have a `pMsg` pointer to a `MSG` structure that contains your message. When no messages are in the queue, `OnIdle` will occur. There are two type of messages that you could send to a thread: blocking and unblocking. The blocking messages type are posted using the `SendThreadMessage(...)`. This means that when you use this function the execution of your program will stop at the `SendThreadMessage()` until the thread will execute the message. The unblocking messages type are posted using `PostThreadMessage(...)`. The message will be posted to the queue and the execution of the program will continue asynchronous.

Check Your Progress 2

Fill in the blanks:

- (a) The communication to a thread it is done by _____.
- (b) The blocking messages type are posted using the _____.
- (c) The unblocking messages type are posted using _____.

14.3 LET US SUM UP

Multithreading is a specialized form of multitasking. In general, there are two types of multitasking: process-based and thread-based. In a thread-based multitasking environment, all processes have at least one thread, but they can have more. This means that a single program can perform two or more tasks concurrently.

14.4 LESSON END ACTIVITY

Discuss the importance of multithreading.

14.5 KEYWORDS

Thread: It is a dispatchable unit of executable code.

Multithreading: It is a specialized form of multitasking where all processes have at least one thread, but they can have more.

14.6 QUESTIONS FOR DISCUSSION

1. What is thread?
2. What is multithreading?
3. Compare and contrast between multitasking and multithreading.
4. How multithreading is created?
5. How multithreading is used?

Check Your Progress: Model Answers

CYP 1

1. True
2. False

CYP 2

- (a) Messages
- (b) SendThreadMessage(...)
- (c) PostThreadMessage(...)

14.7 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up* 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth, *Visual C++6 Unleashed*, Second edition, Techmedia.

LESSON

15

WIZARD

CONTENTS

- 15.0 Aims and Objectives
- 15.1 Introduction
- 15.2 Visual C++ APPWIZARD
- 15.3 Class Wizard
- 15.4 Let us Sum up
- 15.5 Lesson End Activity
- 15.6 Keywords
- 15.7 Questions for Discussion
- 15.8 Suggested Readings

15.0 AIMS AND OBJECTIVES

At the conclusion of this lesson you should be able to understand:

- An overview of application wizard
- Concept of ClassWizard

15.1 INTRODUCTION

The AppWizard asks you a series of questions about what type of application you are building and what features and functionality you need. It uses this information to create a shell of an application that you can immediately compile and run. This shell provides you with the basic infrastructure that you need to build your application around. Let's discuss the AppWizard.

15.2 VISUAL C++ APPWIZARD

One of the great strengths of the Visual C++ development system is its ability to create highly functional application skeletons through the AppWizard tool. The AppWizard can be used to generate skeleton Windows applications with a variety of OLE, database, windowing, help, and other options. The AppWizard can also be used to create Windows DLLs; in addition, specialized AppWizards exist for creating OLE controls and custom AppWizards.

Creating Windows Projects

When you select the New command from the Developer Studio's File menu, you are presented with the choice of creating a new text file, a new project workspace, or a variety of new resource file components (Figure 15.1).

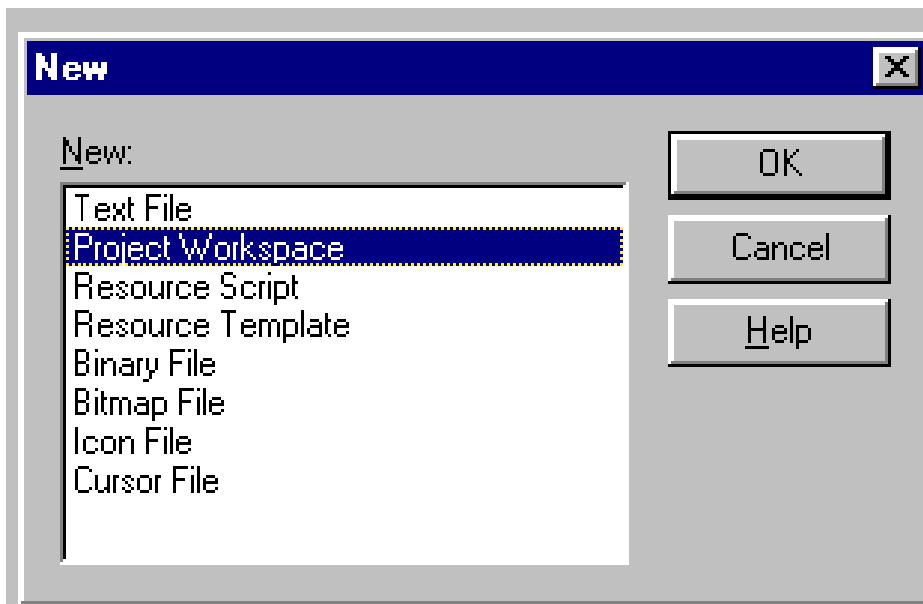


Figure 15.1: The New dialog

If you select Project Workspace in the New dialog, another dialog, the New Project Workspace dialog, appears (Figure 15.2). In this dialog, you can choose from a variety of workspace options.

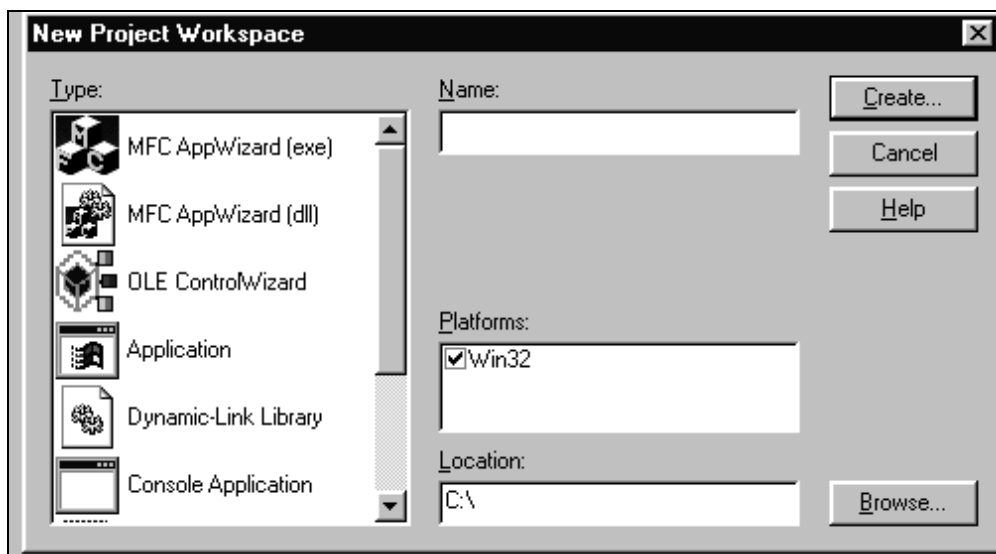


Figure 15.2: The New Project Workspace dialog

The first two choices listed in this dialog represent AppWizard-generated skeletons for Windows executable programs and for Windows DLLs. Next, we examine these options in detail. Later in this chapter, we take another look at the other project workspace types and how they can be used with new or existing projects.

Types of Windows Applications Created Through AppWizard

If you select MFC AppWizard (exe) as the type of your new project, you are presented with step one of a multistep wizard process (Figure 15.3). In this first step, you can decide the basic characteristic of your new application: whether it will have a single-document-based, multiple-document-based, or dialog-based user interface.

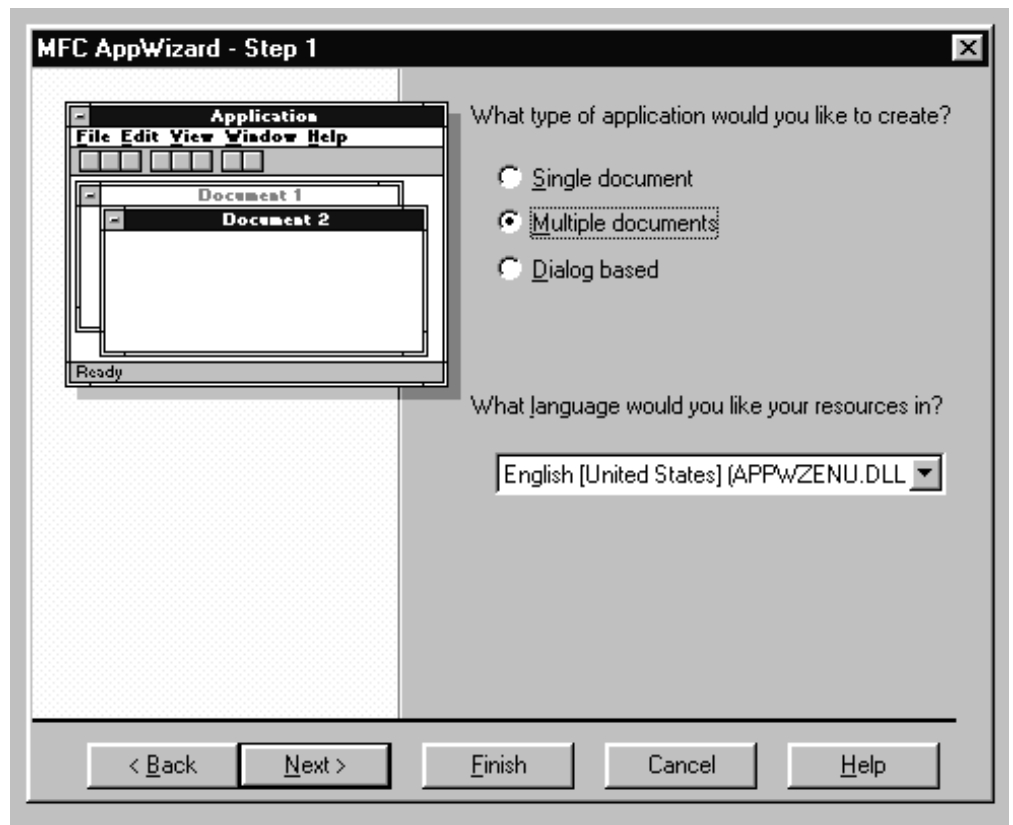


Figure 15.3: MFC AppWizard: Selecting your project's type

A single-document-based application can present only one file to the user at any given time. A good example for such an application is the Windows Notepad.

A multiple-document-based application, in contrast, can present several documents at once, each in its own child window. Many word processing applications, such as Microsoft's Word for Windows, are multiple-document-based applications.

A dialog-based application presents a single dialog as its user interface. These applications are used when all user interaction can take place through a single dialog template. An example for a dialog-based application is the Windows Character Map.

Note: AppWizard-generated dialog-based applications offer substantially fewer MFC features than MFC-based SDI or MDI applications. If you plan to use a document class, view class, or other MFC features, consider creating an SDI application based on the CFormView view class instead of creating a dialog-based application.

During this first AppWizard step, you can also specify the language of your application. Your language selection defines which standard MFC resource set will be included with your project.

You can also use the AppWizard to create a project that supports multiple languages. To do so, create the additional resource files (perhaps by rerunning AppWizard to create dummy projects in the desired language) and add the new resource files to your project. Create additional project configurations, including and excluding resource files as needed.

Document-Based MFC Applications

The creation of single- or multiple-document-based (SDI or MDI) applications through AppWizard are nearly identical procedures, consisting of the same AppWizard steps.

The project files created by AppWizard for SDI and MDI projects are somewhat different; in particular, for an MDI project, AppWizard generates an additional class, `CChildFrame`, that represents MDI child windows.

After you select one of these document-based options, clicking the Next button takes you to Step 2 of a six-step process. In this step (Figure 15.4), you must specify the level of database support your application will provide. Database support is in the form of MFC's Open Database Connectivity (ODBC) and Data Access Objects (DAO) classes.

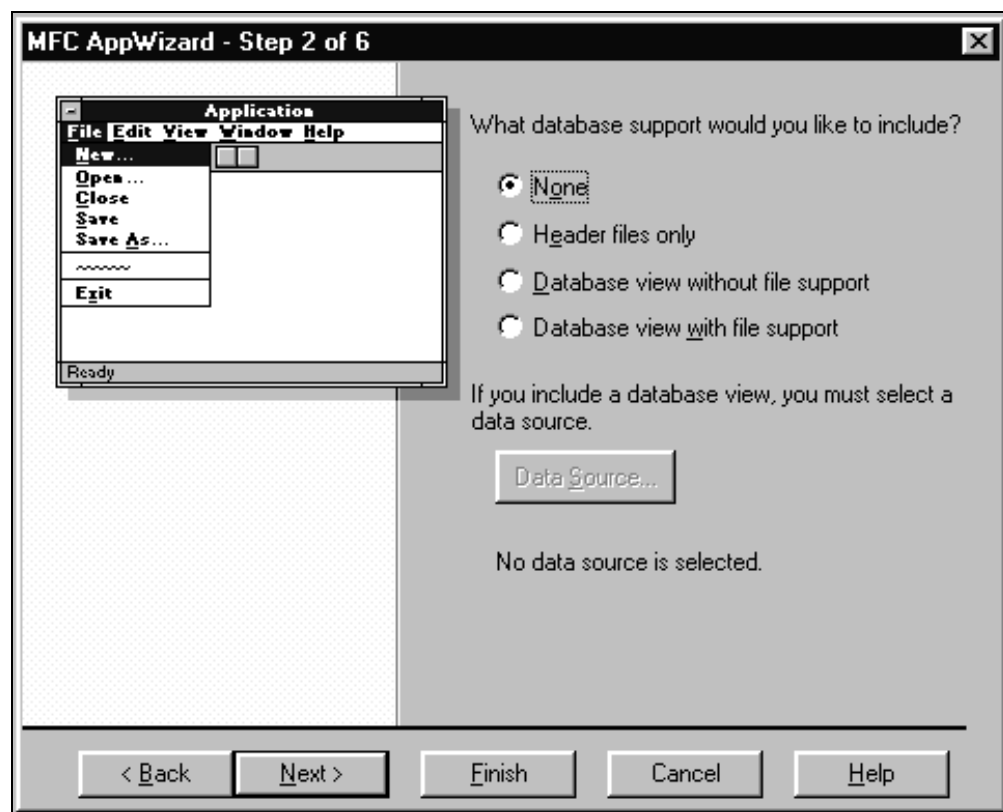


Figure 15.4: MFC AppWizard: Selecting database support

The meaning of the None option is obvious. The Header files only option creates a project with the necessary header files included, but otherwise the project files will be no different from files in a project with no database support.

The remaining two options represent significant additions to your project. First, your project's view class will be a class derived from `CRecordView` or `CDaoRecordView`; second, a new class, derived from `CRecordSet` or `CDaoRecordSet`, will be added to your project. The new view class is dialog template-based and represents the fields in a record; the record set class provides an internal representation for those fields and methods to access the underlying tables.

The difference between the Database view with file support and Database view without file support options in AppWizard Step 2 is simple. The former provides

menu and toolbar commands to load and save document files; the latter does not. Often, when creating a database application, your application's document class provides merely a transient representation of the database and does not need to be saved; in this case, use the Database view without file support option.

Regardless which of these two options you choose, before you proceed to AppWizard Step 3, you must also specify a data source. The data source is a table or set of tables in a database that can be accessed through the ODBC or DAO mechanism. To select a data source, click on the Data Source button.

AppWizard Step 3 (Figure 15.5) is about support for Object Linking and Embedding features. You can specify whether your application supports OLE compound document functionality as a server, container, mini-server, or container-server. You can also add OLE automation server and OLE control container support; the latter is important if you wish to use OLE controls in your application's dialogs.

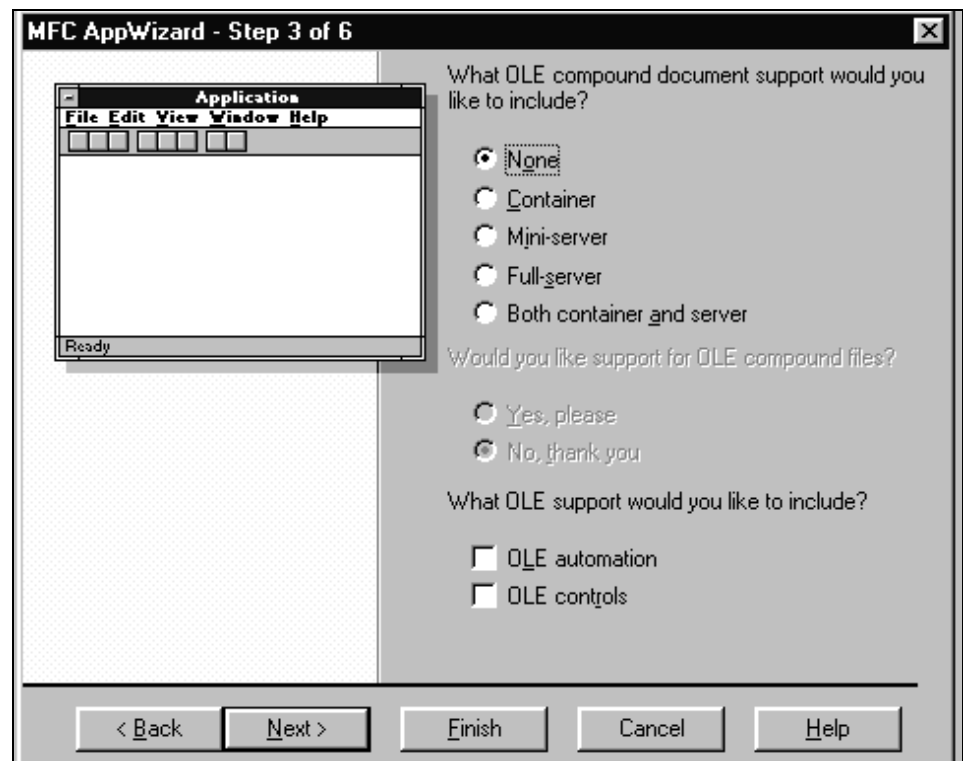


Figure 15.5: MFC AppWizard: OLE features

If you select OLE compound document support, you can also specify OLE compound file support.

Depending on which OLE features you select, the AppWizard may generate additional classes for your application. For OLE servers, these additional classes include `CInPlaceFrame` (representing the frame window during in-place editing), and a `COleServerItem`-derived class representing the server object in container applications. For OLE containers, a new `COleClientItem`-derived class is added, representing OLE server items in the container. In both cases, the base class of your application's document class is also modified; it is derived from `COleDocument` in the case of containers, and `COleServerDoc` in the case of servers or container-servers.

Step 4 of AppWizard (Figure 15.6) contains a variety of miscellaneous options. The checkboxes for toolbar support, status bar support, and support for 3-D controls require little explanation.

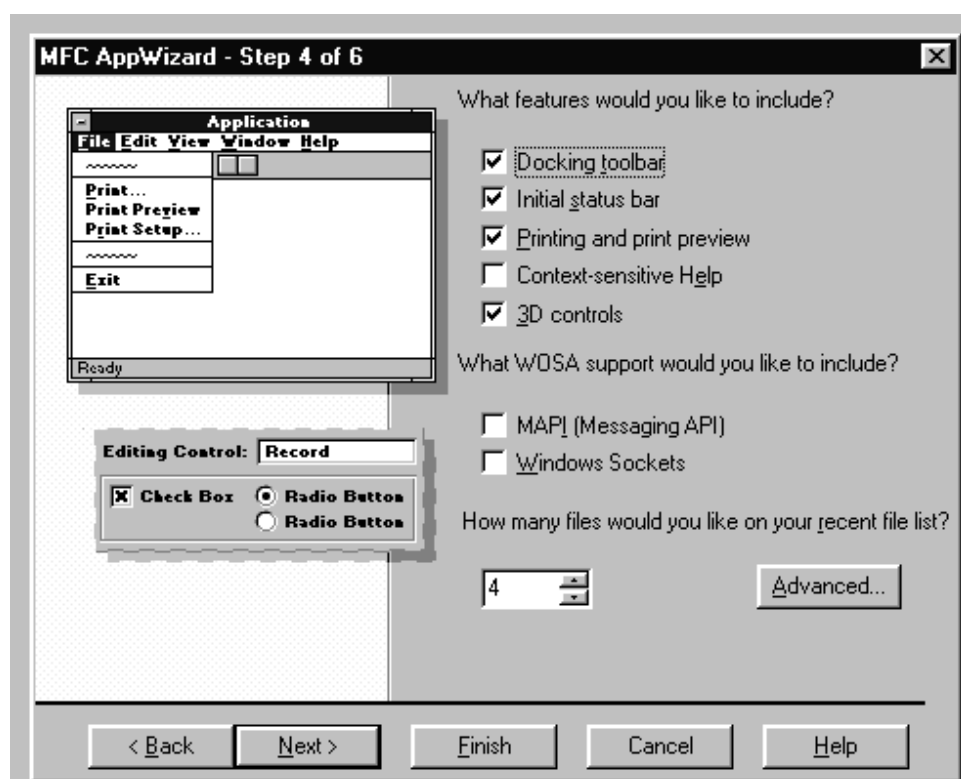


Figure 15.6: MFC AppWizard: Miscellaneous options in Step 4

If you clear the Printing and print preview checkbox (it is set by default), your application will not have a Print or Print Preview command in its File menu, nor will it provide support for print preview mode. It is generally a good idea to leave this box checked even if you do not wish to provide the printing commands in the File menu, as long as you intend to use the printing and print preview-related features of MFC.

Setting the Context-sensitive Help checkbox adds a skeletal help project file and help topic files to your application. It also adds the batch file `makehelp.bat` that can be used to regenerate your project's help file.

Adding Messaging API (MAPI) support to your application means two things: first, your application will be linked with the MAPI libraries; second, your application will have a Send menu item in its File menu. Often this is all you need to provide minimal MAPI support for compatibility with Windows 95 application requirements.

Setting the Windows Sockets checkbox adds WinSock libraries and header files to your project. However, you are responsible for adding any specific WinSock functionality.

Of particular interest in AppWizard Step 4 is the Advanced Options dialog, invoked when you click on the Advanced button. Through this dialog, you can specify a variety of additional options that affect your application's appearance and execution in subtle ways.

The Advanced Options dialog is a tabbed dialog. The first tab, Document Template Strings (Figure 15.7), enables you to specify several string values defining your application's default filename extension, filename filter, file type identifier, and more. These strings are combined and stored in your application's string table (in the resource file) under the identifier `IDR_MAINFRAME`. For example, the string corresponding to the settings shown in Figure 32 is this:

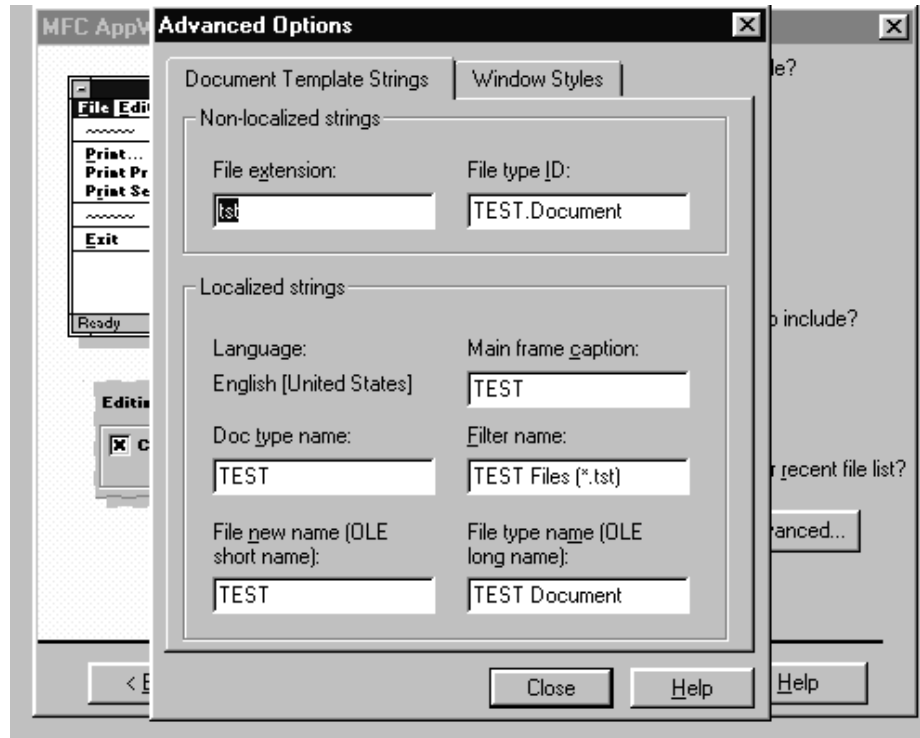


Figure 15.7: MFC AppWizard: Advanced document template string settings

The second tab in this dialog, Window Styles (Figure 15.8), can be used to specify the window styles for the application's frame window. You can also select split window support; when this checkbox is set, your application's views will have a splitter bar.

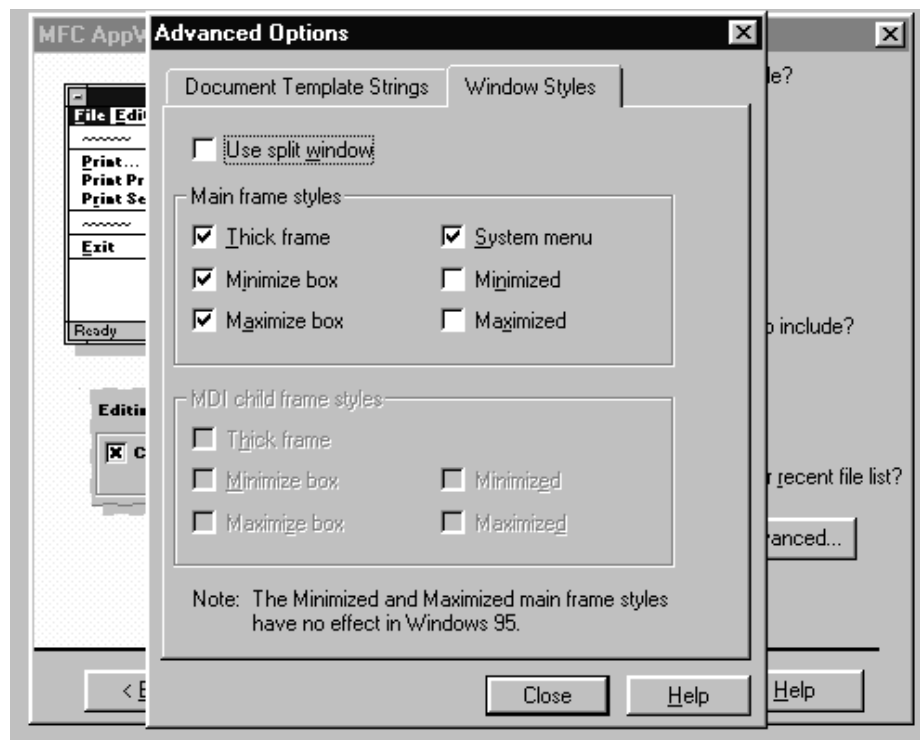


Figure 15.8: MFC AppWizard: Advanced window style settings

In this part of the Advanced Options dialog, you can also modify settings for child frame windows in MDI applications.

Step 5 of AppWizard (Figure 15.9) enables you to specify two simple options; whether the AppWizard-generated skeleton source should contain comments or not, and whether the MFC Library should be linked to your project as a static library or a DLL.

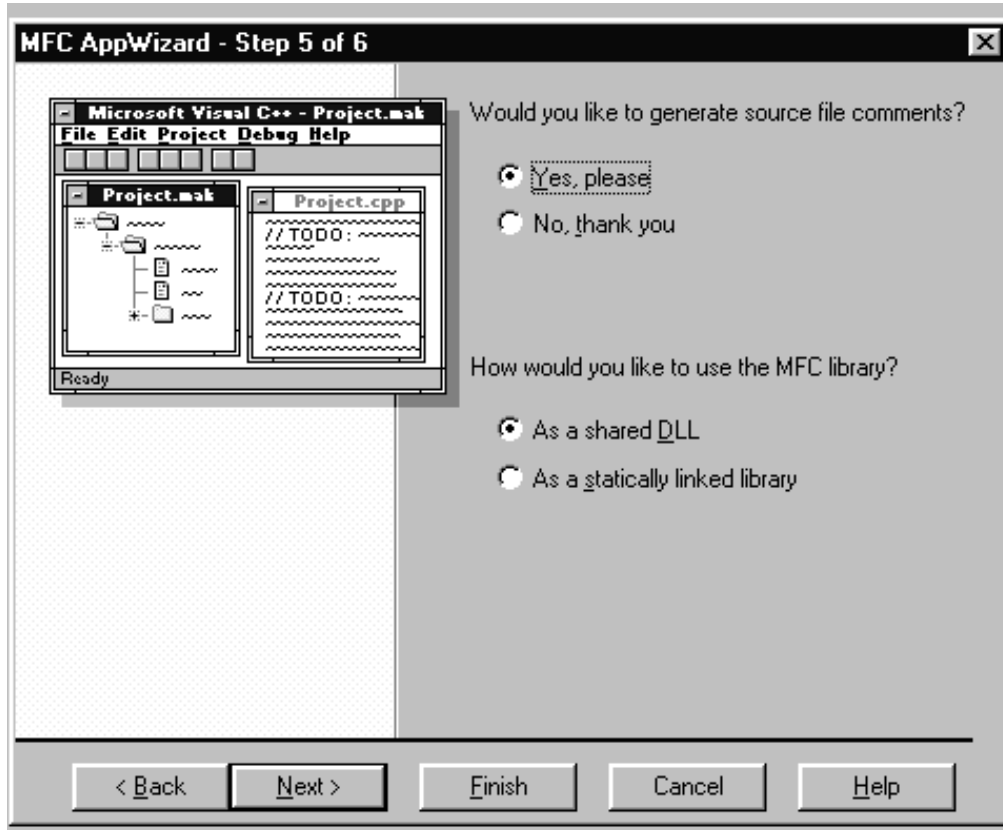


Figure 15.9: MFC AppWizard Step 5

Usually, specifying source file comments is a good idea. In addition to providing meaningful explanations in your code, the AppWizard also generates "TODO" comments, which indicate those points in your code that you need to complete or modify manually.

The final step in the AppWizard process, Step 6 (Figure 15.10), lists the classes that the AppWizard is about to create for your application. You can also modify some aspects of class creation at this stage. For example, you can modify the base class of your application's view class to be based on the CFormView or CScrollView class, instead of the default CView.

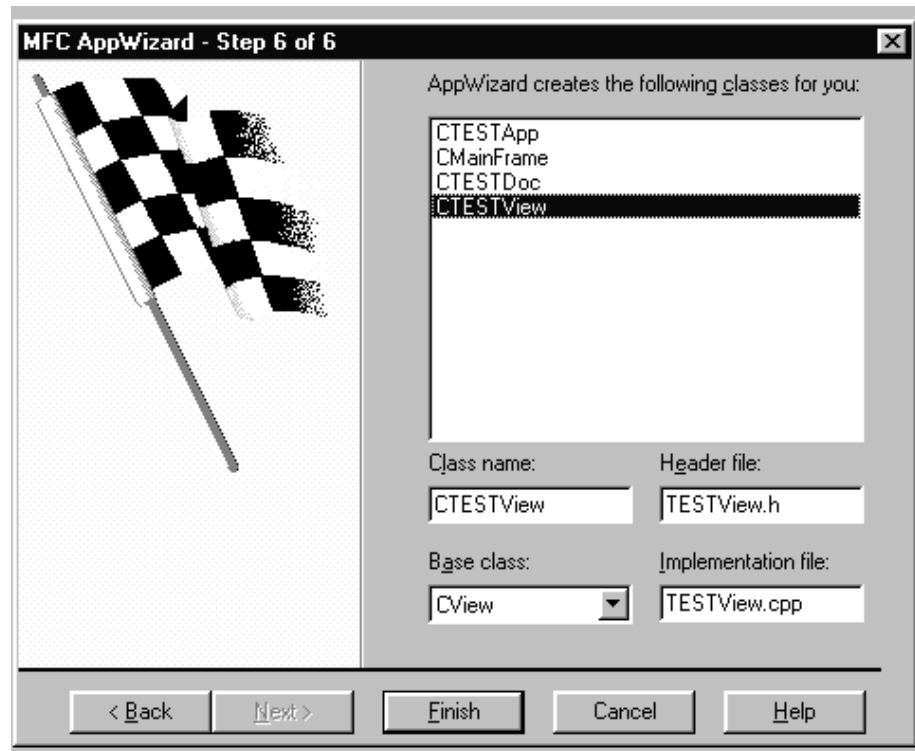


Figure 15.10: MFC AppWizard: Class overview

To complete the AppWizard process, click the Finish button. This displays the New Project Information dialog (incidentally, giving you one more chance to back out and cancel the AppWizard procedure); this dialog displays information about the skeleton project. This information is also saved in your project directory as your project's readme.txt file.

The classes generated for a basic single-document base application include a document class, a view class, a frame window class, and a CWinApp-derived class representing the application. The declaration and definition of a CDialog-derived class, representing your applications About dialog, will also be included in the implementation file of your application's CWinApp-derived class. Depending on what additional options you specified (MDI support, database support, OLE features), additional classes may be generated by AppWizard.

Check Your Progress 1

Fill in the blanks:

1. A _____ document-based application can present only one file to the user at any given time
2. The class _____ represents MDI child windows.

Dialog-Based MFC Applications

The AppWizard process for creating a dialog-based MFC application is much shorter than creating document-based applications. Selecting a dialog-based application in AppWizard Step 1 and clicking on the Next button takes you to Step 2 of a four-step process (Figure 15.11).

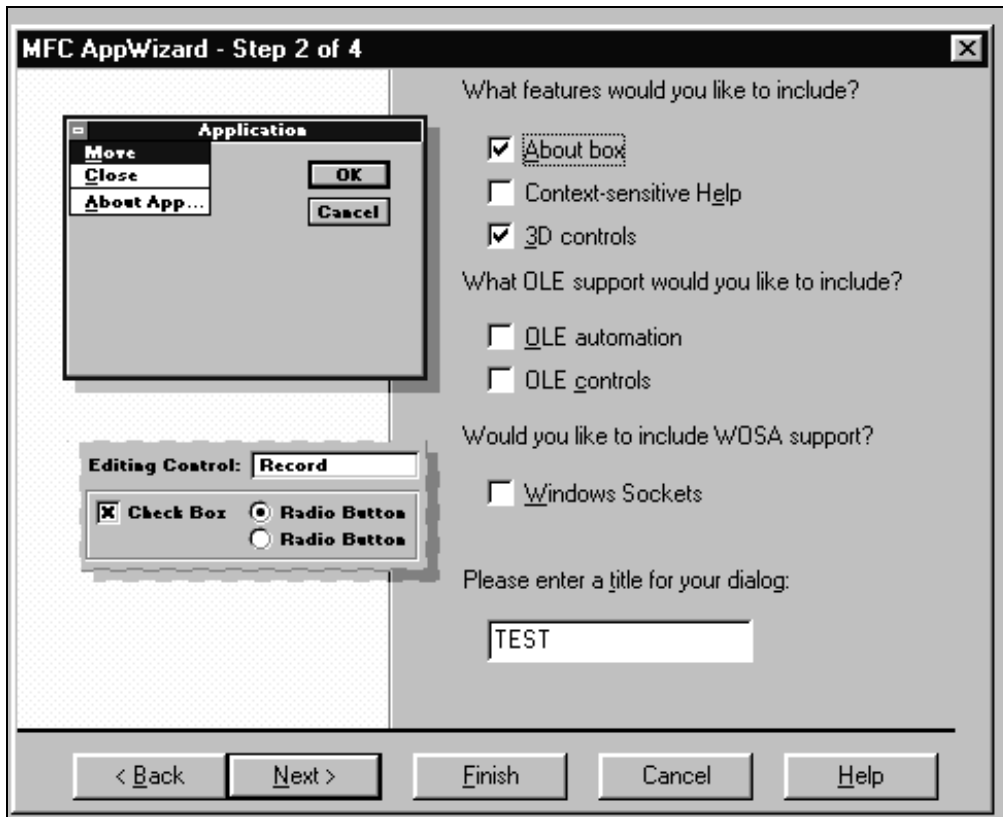


Figure 15.11: MFC AppWizard: Creating a dialog-based application

The About box option in this step adds an About command to the dialog's control menu; note that the dialog will not have a menu bar. The Context-sensitive Help option adds AppWizard-generated skeletal help support in the form of a help project file, topic files, and the makehelp.bat script for regenerating your application's help file. The 3-D controls option needs no explanation.

You can also specify OLE automation and OLE control support. Check the latter if you wish to use OLE controls in the dialog.

The Windows Sockets option adds WinSock header and library files to your project; however, it is up to you to implement any specific WinSock functionality.

You can also specify the title of your application; this text will be displayed in the title bar of the application's dialog box.

Step 3 of the AppWizard process for dialog-based applications is identical to Step 5 for document-based applications (see Figure 15.8). Similarly, Step 4 for dialog-based applications is the same as Step 6 for document-based ones; however, the list of generated classes is different (Figure 15.12). Only two classes are created: one representing the application object, the other representing the application's dialog.

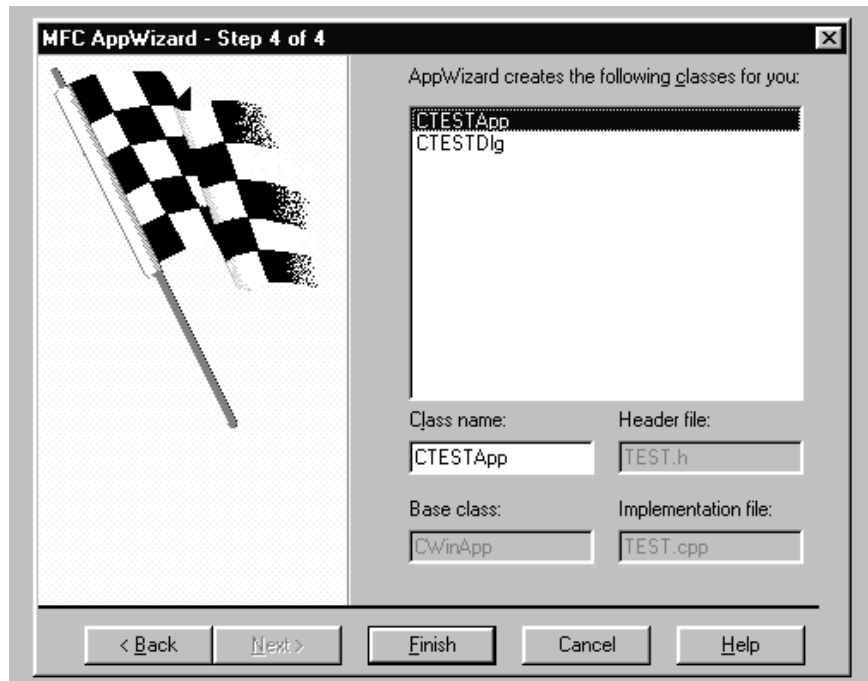


Figure 15.12: MFC AppWizard: Dialog-based application classes

In many cases, you may find that a dialog-based AppWizard-generated skeleton application lacks the features that you wish to see in your program. In such cases, consider using an SDI application based on the CFormView class.

Using AppWizard to Create MFC-Based DLLs

If you use the AppWizard to create an MFC-based DLL skeleton, you are presented with a one-step procedure where you can specify your DLL's characteristics (Figure 15.13).

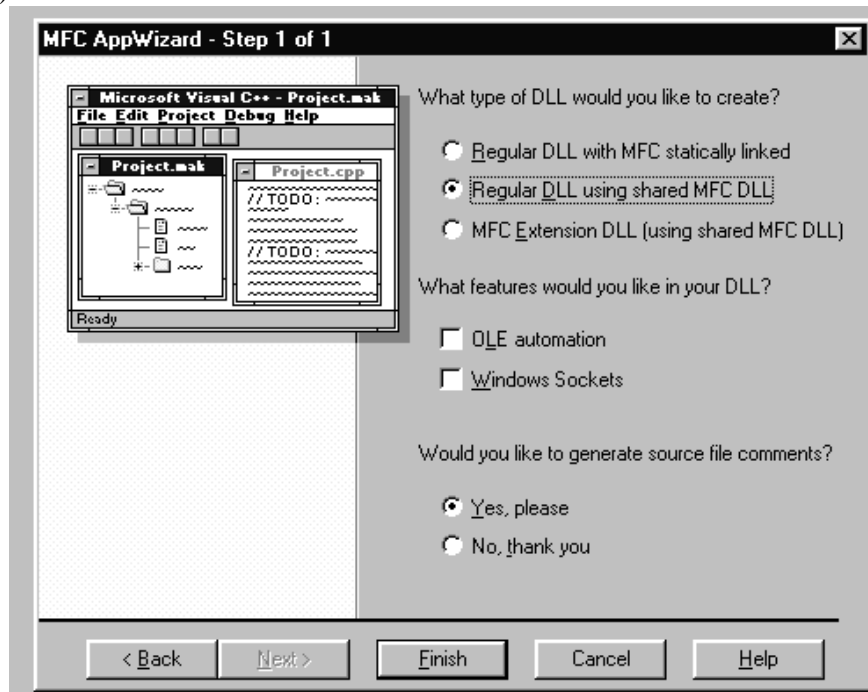


Figure 15.13: MFC AppWizard: Creating an MFC-based DLL

The first set of options is where you specify how the DLL should be linked with the MFC Library. Linking statically is the most expensive in terms of disk space and memory requirements. The least expensive is creating an MFC extension DLL; however, these DLLs can only be used with MFC applications. A regular DLL linked with the MFC DLL can, in turn, be called from any application.

By setting the OLE Automation checkbox, you enable support for using your DLL as an OLE Automation inproc server. (An inproc server is a server that executes in the process space of the client application, as opposed to using a remote procedure call mechanism for communicating with the client.)

Adding WinSock support enables compiling and linking with the WinSock header and library files; however, you must add your own implementation of any WinSock-specific functionality.

Finally, you can also specify if you wish to see AppWizard-generated source comments in your DLL.

Check Your Progress 2

Fill in the blanks:

1. An example for single-document-based application is _____.
2. An example for multiple-document-based application is _____.
3. An example for a dialog-based application is _____.

15.3 CLASS WIZARD

Next to the AppWizard, the ClassWizard represents the second most powerful tool in the Visual C++ development system. ClassWizard represents a unique way of looking at and managing certain types of classes; namely, classes that are command targets, from the MFC class CCmdTarget.

ClassWizard Features

The ClassWizard is typically invoked from the Developer Studio's View menu. It can also be invoked from many popup menus; for example, you can invoke the ClassWizard from the popup menu that appears during dialog editing. In such cases, the ClassWizard is invoked in a special mode. It usually appears with the relevant class (that is, the class associated with the dialog template you were editing) preselected. If such a class does not exist, the ClassWizard is invoked in class creation mode where a new class for the selected object (dialog) can be created.

The ClassWizard appears to the user as a large dialog with five tabs. Each of these tabs represents a special ClassWizard function. The Message Map tab presents options for defining and editing message handler functions. The Member Variables tab is where you assign member variables to dialog controls in classes that are associated with a dialog template. The OLE Automation tab offers a choice of OLE automation methods and properties of an OLE server application or DLL. The OLE Events tab is where you add events to an OLE control. Lastly, the Class Info tab presents a review of some of the overall characteristics of your class.

15.4 LET US SUM UP

One of the great strengths of the Visual C++ development system is its ability to create highly functional application skeletons through the AppWizard tool. The AppWizard can be used to generate skeleton Windows applications with a variety of

OLE, database, windowing, help, and other options. The AppWizard can also be used to create Windows DLLs; in addition, specialized AppWizards exist for creating OLE controls and custom AppWizards. Next to the AppWizard, the ClassWizard represents the second most powerful tool in the Visual C++ development system. ClassWizard represents a unique way of looking at and managing certain types of classes; namely, classes that are command targets, from the MFC class CCmdTarget.

15.5 LESSON END ACTIVITY

The Microsoft Foundation Classes (MFC) AppWizard or MFC Application Wizard in Visual C++ does not generate resources in sublanguages. Discuss.

15.6 KEYWORDS

AppWizard: It is a tool which can be used to generate skeleton Windows applications with a variety of OLE, database, windowing, help, and other options.

ClassWizard: It represents a unique way of looking at and managing certain types of classes; namely, classes that are command targets, from the MFC class CCmdTarget.

15.7 QUESTIONS FOR DISCUSSION

1. What is AppWizard?
2. What is its role in Microsoft VC++ project?
3. How AppWizard is related with Windows DLL?
4. What is ClassWizard?
5. Describe the features of ClassWizard.

Check Your Progress: Model Answers

CYP 1

1. single
2. CChildFrame

CYP 2

1. Windows NotePad
2. Microsoft's Word for Windows
3. Windows Character Map

15.8 SUGGESTED READINGS

Herbert Schildt, *MFC Programming from the Ground up* 2nd edition, Tata McGraw Hill.

MSDN Visual studio Library.

Mveller, *Visual C++ from the Ground up*, TMCH

Viktor Toth. *Visual C++6 Unleashed*, Second Edition. Techmedia.