KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Pollachi Main Road, Eachanari Post, Coimbatore - 641021
(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMMERCE (CA)

Semester VI

L	T	P	C
4	-	-	4

17CCP204 **VISUAL BASIC.NET**

Course Objective:

This course focuses on Module Coding, Controls, Strings and Databases

Learning Outcome:

- To enable the students to know the concepts of .net technologies.
- To create an about Intermediate language compiler.
- To create an about the application of web based software

Unit-I

Getting Started With VB.NET: The Integrated Development Environment-IDE Components- Environment Options. Visual Basic: The Language -Variables-Constants-Arrays – Variables as Objects-Flow Control Statements.

Unit-II

Writing and Using Procedures: Module Coding – Arguments. Working with Forms: Appearance of Forms- Loading and Showing Forms -Designing Menus. Multiple Document Interface

Unit- III

Basic Windows Controls: Textbox Control- ListBox, CheckedListBox-Scrollbar and TrackBar Controls. More Windows Control: The common Dialog Controls-The Rich TextBox Control.The TreeView and ListView Controls: Examining the Advanced Controls-The TreeView Control-The ListView Control-Content Page Holder

Unit-IV

Handling Strings, characters and Dates: Handling Strings and Characters – Handling Dates. Working with Folders and Files: Accessing Folders and Files – Accessing Files. Drawing and Painting with Visual Basic: Displaying Images – Drawing with GDI – Co-ordinate Transformation – Bitmaps.

Unit-V

Databases: Architecture and Basic Concepts: What is database? - Server Explorer – Structured Query Language – The Query Builder – Building database Application with ADO.Net: The Architecture of ADO.Net-Creating the dataset – Data Binding – Programming the Data Adapter Objects – The Command and Data Reader Object. Programming the ADO.Net objects: The Structure of the dataset – The DataForm Wizard – Transactions – Performing Update Operations.

Text Book

Evangelos Petroutsos (2012). "*Mastering Vb. Net".* 5nd Edition USA, SYBEX Inc.

Reference Books

1. Steven Holzner (2010). "*Vb.Net Programming Black Book".* USA, Dream Tech publications.

2. Bill Evjen, Scott Hanselman, Farhan Mohammed, Srinivasa Siva Kumar and Devin Rader (2012). "*Asp.Net 2.0".* USA, Wiley Publication.

3. Burrowss W.E and D. Langford (2010). "*Learning Programming using Visual Basic .Net".* New Delhi, McGraw Hill Edition.

4. Jeffrey R. Shapiro (2013). "*The Complete Reference Visual Basic.Ne"t.* New Delhi, Tata - McGraw-Hill Edition.

5. Richard Bowman (2012). "*Visual Basic.Net".* Canada, Hungry Minds Inc. Publication

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

LECTURE PLAN
DEPARTMENT OF COMPUTER SCIENCE

STAFF NAME: Dr.S.Hemalatha
SUBJECT NAME: Visual Basic.Net SUB.CODE:17CCP204
SEMESTER: II CLASS: I M.Com (CA)

S.No	Lecture Duration Period	Topics to be Covered	Support Material/Page Nos
		UNIT-I	
1	1	**Introduction, Getting Started with VB.Net**	T1:1-12,
2	1	**Integrated Development Environment-** Start Page, Project Types	T1:4-12
3	1	**IDE Component-** IDE Menu, ToolBox Window,Solution Explorer	T1:19-24,W1
4	1	Properties Window, Output Window, Command Window, Task List Window, Environment Options	T1:24-30
5	1	**Visual basic** : The language Variables, Declaring Variables, Types of Variables	T1:80-105 W2
6	1	Converting Variable Types, User-Defined Data Types, Examining Variable Types, Why Declare Variables?,	T1:107-110
7	1	A Variable's Scope, The Lifetime of a Variable, **Constants**	T1:110-119
8	1	**Arrays-** Declaring, initializing arrays, Array limits,	T1:122-124
9	1	Multidimensional arrays, dynamic arrays, arrays of arrays	T1:128-130,W2
10	1	**Variables as Objects-** What is an object?, Formatting numbers, date	T1:130-135,W2
11		**Flow Control Statements-** Test structures, loops	T1:136-140 W3
12	1	Nested control, exit statements. Recapitulation and Discussion of Important Questions	T1:140-148 W3
Total No Of Periods Planned For Unit 1 :12			

		UNIT-II	
1.	1	**Writing and Using Procedures Module Coding,** subroutines, functions. Calling functions and subroutines	T1-151-160
2.	1	**Arguments-** Argument-Passing mechanisms	T1-161-164
3.	1	Event Handler Arguments, Passing an Unknown number of arguments, Named Arguments	T1-165-170,R1
4.	1	More types of function return values,	T1-170-171,W4
5.	1	Overloading functions	T1-171-176
6.	1	**Working with Forms, Appearance of Forms,** Properties	T1-186-187,
7.	1	Placing controls, setting the tab order	T1-187-193
8.	1	VB.NET at work Anchoring And Docking, Form event's	T1-193-203,R1
9.	1	**Loading and Showing Forms:** startup, controlling forms	T1-207-219,
10.	1	**Designing Menus-** menu editor, menu item object properties, manipulating menus at runtime, iterating a menu's items	T1-219-231, W6
11.	1	MDI applications-basics, building an MDI, built-in capabilities	T1-837-840,R1
12	1	Accessing child forms, ending an MDI applications, A scrollable picture box Recapitulation and Discussion of Important Questions	T1-840-865, W5
Total No Of Periods Planned For Unit II :12			
		UNIT-III	
1.	1	**Basic Windows Controls, Textbox Control,** Basic properties, Text-manipulation properties,	T1-241-248 W7
2.	1	Text-selection properties and methods, undoing edits	T1-248-251,
3.	1	VB.NET at work, capturing keystrokes	T1-251-263
4.	1	**List Box, CheckedListBox,** properties,	T1-263-266,

		items collection, Scrollbar and Track Bar Controls	T1-279-287,R1
5.	1	**The common Dialog Controls-using dialog controls-** color dialog , font dialog, Open and save as dialog boxes, print dialog box	T1-292-294, T1-297-304 W8
6.	1	**The Rich Textbox Control-** RTF language, properties, Methods, advanced editing features.	T1-305-310
7.	1	VB.NET at work, searching, More Windows Control**,**	T1-266-273, T1-289-291 W7
8.	1	Cutting and pasting, searching in a rich textbox control, formatting URLs, VB.NET at work	T1-315-326,W3
9.	1	**The Tree View and List View Controls, Examining the advanced controls**	T1-741-747,R1
10.	1	**The TreeView Control-** adding new items at design time and runtime, assigning images to nodes, scanning the tree view control	T1-754-768, W9
11.	1	**The List View Control-** columns collection, listitem object, items collection, subitems collection	T1-768-782
12.	1	Recapitulation and Discussion of Important Questions	
Total No Of Periods Planned For Unit III :12			
		UNIT-IV	
1.	1	**Handling Strings and Characters**	T1- 530-534
2.	1	String Builder	T1-534-544 W10
3.	1	**Handling Dates-** Date Time, Time Span	T1- 552-567,R1
4.	1	**Working with folders and files- Accessing Folders and Files,** Directory class, File Class	T1- 570-578
5.	1	Directory Info Class, File Info Class, Path Class	T1-584-587
6.	1	**Accessing Files-** File Stream, Stream Writer, Stream Reader Object	T1- 594-607
7.	1	Sending Data to a file, Binary Writer, Binary Reader Object	T1-600-607

8.	1	**Drawing and painting with vb-** Displaying Images**,** image object, exchanging images through the clipboard	T1- 620-630,W11
9.	1	**Drawing with GDI+-** Basic Drawing Object	T1- 632-633 W12
10.	1	Drawing Shapes, Drawing Methods Gradients, Clipping	T1-642-647 T1-661-665
11.	1	**Coordinate Transformations-** Specifying Transformations	T1 – 668-675
12.	1	**Bitmaps-** Specifying Colors, Defining Colors, Processing Bitmaps **Recapitulation and Discussion of Important Questions**	T1- 681-697 W13
Total No Of Periods Planned For Unit IV:12			
		UNIT-V	
1.	1	**Databases: architecture and basic concepts, What is Database?-** Relational Databases, Exploring the Northwind Databases, Understanding Relations	T1 -870-878,R1
2.	1	**The server Explorer-** Working with Tables, Relationships, Indices and Constraints	T1 – 878-889
3.	1	**Structured Query Language-** Executing SQL Statements, Selection Queries, Calculated Fields, SQL Joins, Action Queries	T -889-906,W14
4.	1	**The Query Builder-** Limiting the Selection, Parameterized Queries, Calculated Columns, Specifying Left, Right and Inner Joins	T1 – 906-913 W15
5.	1	**The Architecture of ADO.Net, Creating a Dataset** **Data Binding-** Binding Complex Controls	T1-913-914 T1– 928-941
6.	1	**Programming the Data Adapter Object, The Command and Data Reader Objects**	T1 -942-950
7.	1	**Programming the ado.net objects** **The Structure of a Dataset,** The Data Form Wizard, Transactions	T1 – 964-982,R1
8.	1	**Performing Update Operations:** A Data Row's Versions, A Data Row's	T1 – 983-991

		States, Updating Tables Manually	
9.	1	Recapitulation and Discussion of important Questions	
10.	1	Discussion of Previous ESE Question Papers.	
11.	1	Discussion of Previous ESE Question Papers.	
12.	1	Discussion of Previous ESE Question Papers.	
Total No of Periods planned for Unit V: 12			
Total Planned Hours:-60			

Text Book

Evangelos Petroutsos (2012). "*Mastering Vb. Net".* 5nd Edition USA, SYBEX Inc.

Reference Books

1. Steven Holzner (2010). "*Vb.Net Programming Black Book".* USA, Dream Tech publications.

2. Bill Evjen, Scott Hanselman, Farhan Mohammed, Srinivasa Siva Kumar and Devin Rader (2012). "*Asp.Net 2.0".* USA, Wiley Publication.

3. Burrowss W.E and D. Langford (2010). "*Learning Programming using Visual Basic .Net".* New Delhi, McGraw Hill Edition.

4. Jeffrey R. Shapiro (2013). "*The Complete Reference Visual Basic.Ne"t.* New Delhi, Tata -McGraw-Hill Edition.

5. Richard Bowman (2012). "*Visual Basic.Net".* Canada, Hungry Minds Inc. Publication

Websites

1. W1: http://visualbasic.w3computing.com/vb2008/1/vb-2008-ide-components.php
2. W2: http://www.tutorialspoint.com/vb.net/vb.net_variables.htm
3. W3: http://visualbasic.w3computing.com/vb2008/3/vb-control-flow-statements-decision-statements.php
4. W4: http://www.dotnetperls.com/multiple-return-values-vbnet
5. W5: https://msdn.microsoft.com/en-us/library/7aw8zc76(v=vs.110).aspx
6. W6: http://visualbasic.w3computing.com/vb2008/5/vb-menus-menu-editor.php
7. W7: http://www.vb6.us/tutorials/button-label-textbox-common-controls
8. W8: http://visualbasic.w3computing.com/vb2008/4/vb-common-dialog-controls.php

UNIT – I

SYLLABUS

Getting Started With VB.NET: The Integrated Development Environment-IDE Components-Environment Options. Visual Basic: The Language -Variables-Constants-Arrays – Variables as Objects-Flow Control Statements.

GETTING STARTED WITH VB.NET

Integrated Development Environment

The Start Page

When we run the Visual Basic Setup program, it allows us to place the program items in an existing program group or create a new program group and new program items for Visual Basic in Windows.

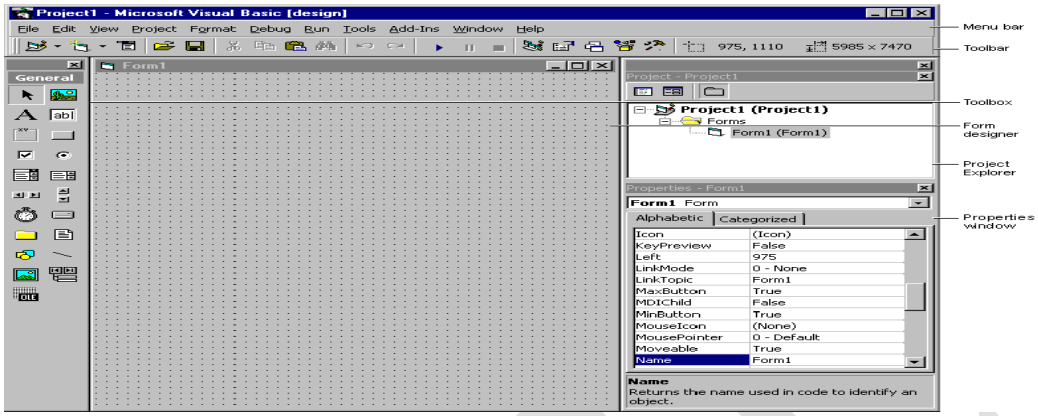
To start Visual Basic from Windows

1. Click **Start** on the Task bar.
2. Select Programs, Visual Studio and then Microsoft Visual Basic 6.0.–or–
   Click **Start** on the Task bar.
   Select **Programs**.
   Use the **Windows Explorer** to find the Visual Basic executable file.
3. Double-click the Visual Basic icon.

We can also create a shortcut to Visual Basic, and double-click the shortcut.

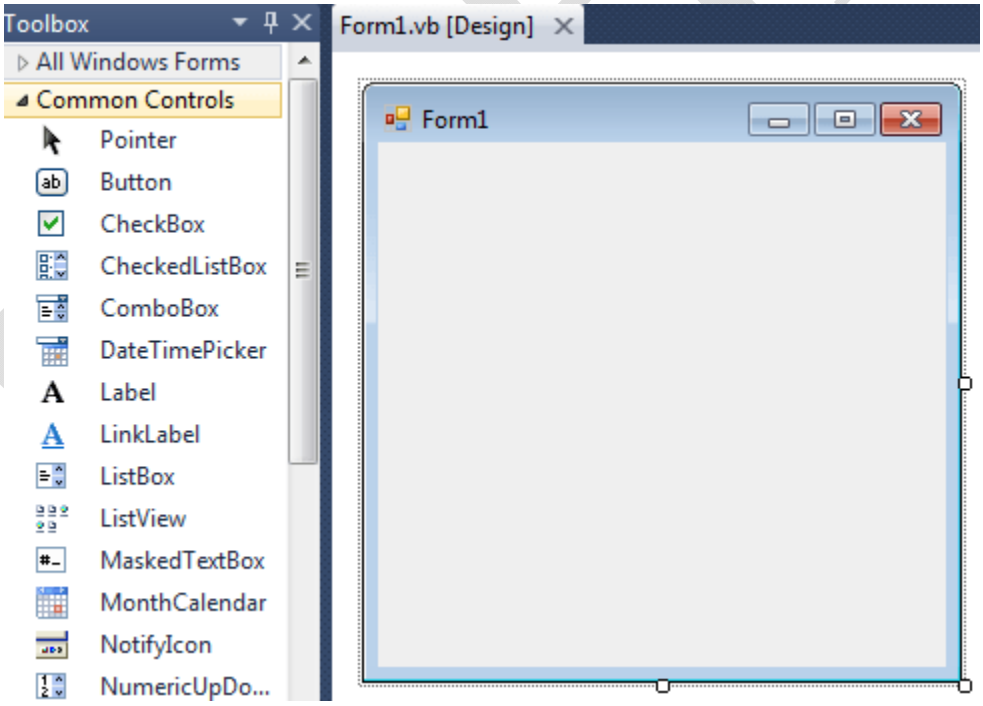
When we first start Visual Basic, we see the interface of the integrated development environment, as shown in Figure 2.1.

Figure: The Visual Basic Integrated Development Environment



Using the Windows Form Designer

Figure: The Windows Forms Toolbox of the Visual Studio IDE



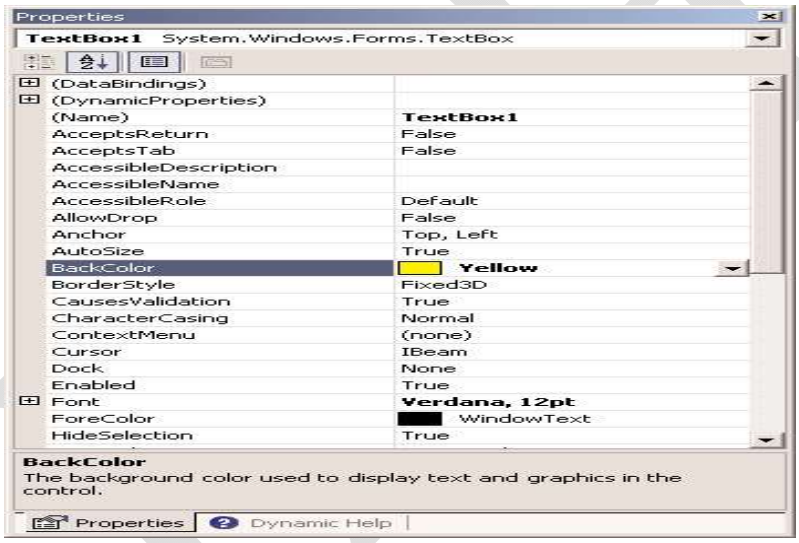
The above picture shows how is the default Form look like. At the top of the form there is a title bar which displays the forms title. Form1 is the default name; you can change the

name to your convenience. The title bar also includes the control box, which holds the minimize, maximize, and close buttons.

Control Properties

The control's properties will be displayed in the Properties window (Figure). This window, at the far left edge of the IDE, displays the properties of the selected control on the form. If the Properties window is not visible, select View ->Properties Window, or press F4. If no control is selected, the properties of the selected item in the Solution Explorer will be displayed. Place another TextBox control on the form. The new control will be placed almost on top of the previous one. Reposition the two controls on the form with the mouse. Then right-click one of them and, from the context menu, select Properties.

Figure - The properties of a TextBox control



In the Properties window, also known as the Property Browser, we see the properties that determine the appearance of the control, and in some cases, its function. Locate the TextBox control's Text property and set it to "My TextBox Control" by entering the string (without the quotes) into the box next to property name. Select the current setting, which is TextBox1, and type a new string. The control's Text property is the string that appears in the control.

Then locate its BackColor property and select it with the mouse. A button with an arrow will appear next to the current setting of the property. Click this button and we will see a dialog box with three tabs (Custom, Web, and System), as shown in Figure. On this dialog box, we can select the color, from any of the three tabs, that will fill the control's background. Set the control's background color to yellow and notice that the control's appearance will change on the form.

Figure - Setting a color property in the Properties dialog box

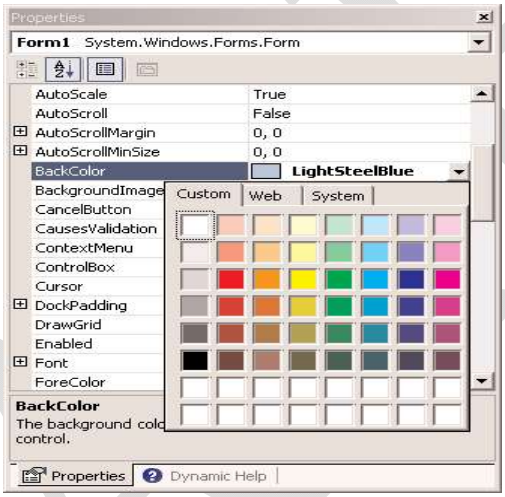
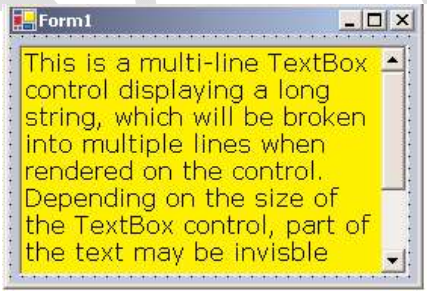
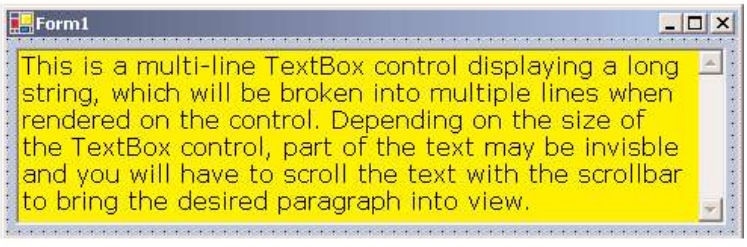


Figure - The appearance of a TextBox control displaying multiple text lines





Project Types

All the project types supported by Visual Studio are displayed on the New Project dialog box, and they're the following:

- **Class library** A class library is a basic code-building component, which has no visible interface and adds specific functionality to your project. Simply put, a class is a collection of functions that will be used in other projects beyond the current one.
- **Windows control library** A Windows control (or simply *control*), such as a TextBox or Button, is a basic element of the user interface. If the controls that come with Visual Basic (the ones that appear in the Toolbox by default) don't provide the functionality you need, you can build your own custom controls.
- **Console application** A Console application is an application with a very limited user interface.
- This type of application displays its output on a Command Prompt window and receives input from the same window.
- **Windows service** A Windows service is a new name for the old NT services, and they're long running applications that don't have a visible interface. These services can be started automatically when the computer is turned on, paused, and restarted. An application that monitors and reacts to changes in the file system is a prime candidate for implementing as a Windows service.
- **ASP.NET Web application** Web applications are among the most exciting new features of
- Visual Studio. A Web application is an app that resides on a Web server and services requests made through a browser. An online bookstore, for example, is a Web application. The application that runs on the Web server must accept requests made by a client (a remote computer with a browser) and return its responses to the requests in the form of HTML pages.

- **ASP.NET Web service** A Web service is not the equivalent of a Windows service. A Web service is a program that resides on a Web server and services requests, just like a Web application, but it doesn't return an HTML page. Instead, it returns the result of a calculation or a database lookup. Requests to Web services are usually made by another server, which is responsible for processing the data

- **Web control library** Just as you can build custom Windows controls to use with your Windows forms, you can create custom Web controls to use with your Web pages.

The IDE Components

The IDE of Visual Studio.NET contains numerous components, and it will take you a while to explore them. It's practically impossible to explain what each tool, each window, and each menu does.

The IDE Menu - The IDE main menu provides the following commands, which lead to submenus. Notice that most menus can also be displayed as toolbars. Also, not all options are available at all times. The options that cannot possibly apply to the current state of the IDE are either invisible or disabled. The Edit menu is a typical example.

File Menu - The File menu contains commands for opening and saving projects, or project items, as well as the commands for adding new or existing items to the current project.

Edit Menu -The Edit menu contains the usual editing commands. Among the commands of the Edit menu are the advanced command and the IntelliSense command.

Advanced Submenu - The more interesting options of the Edit -> advanced submenu are the following. Notice that the advanced submenu is invisible while you design a form visually and appears when you switch to the code editor.

View White Space - Space characters (necessary to indent lines of code and make it easy to read) are replaced by periods.

Word Wrap - When a code line's length exceeds the length of the code window, it's automatically wrapped.

Comment Selection/Uncomment Selection - Comments are lines you insert between your code's statements to document your application. Sometimes, we want to disable a few lines from our code, but not delete them (because we want to be able to restore them).

IntelliSense Submenu - The Edit -> IntelliSense menu item leads to a submenu with four options, which are described next. IntelliSense is a feature of the editor (and of other Microsoft applications) that displays as much information as possible, whenever possible.

List Members - When this option is on, the editor lists all the members (properties, methods,

events, and argument list) in a drop-down list.

TextBox1.

A list with the members of the TextBox control will appear Select the Text property and then type the equal sign, followed by a string in quotes like the following:

TextBox1.Text = "Your User Name"

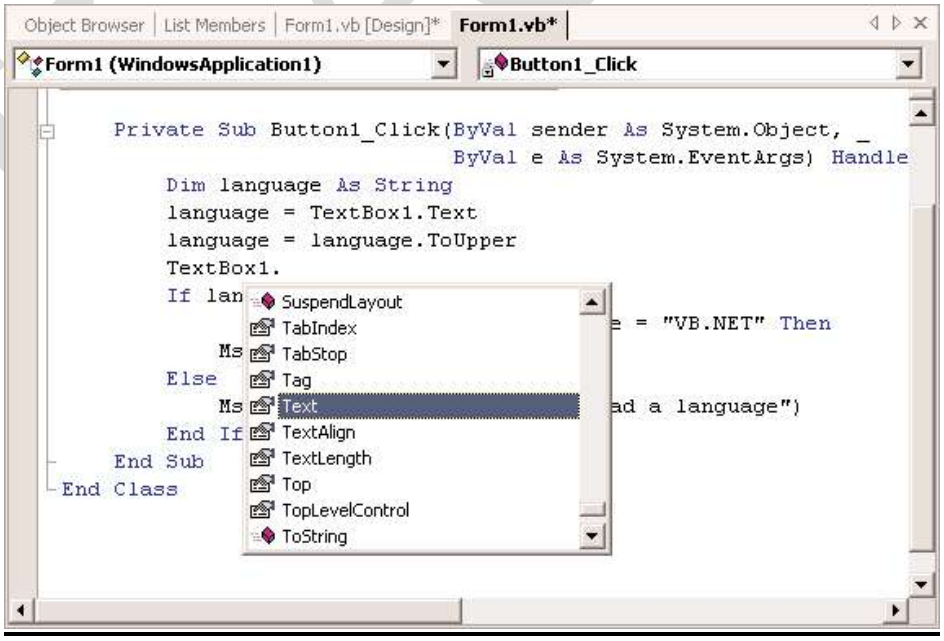
If you select a property that can accept a limited number of settings, you will see the names of the appropriate constants in a drop-down list. If you enter the following statement:

TextBox1.TextAlign =

you will see the constants you can assign to the property (as shown in Figure), they are the values HorizontalAlignment.Center, HorizontalAlignment.Right, and HorizontalAlignment.Left).

Parameter Info - While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow toolti

Figure - Viewing the members of a control in an IntelliSense dropdown list



Quick Info - This is another IntelliSense feature that displays information about commands and functions. When you type the opening parenthesis following the name of a function, for example, the function's arguments will be displayed in a tooltip box (a yellow horizontal box).

View Menu - This menu contains commands to display any toolbar or window of the IDE. You have already seen the Toolbars menu (earlier, under "Starting a New Project"). The Other Windows command leads to submenu with the names of some standard windows, including the Output and Command windows.

The Output window is the console of the application. The compiler's messages, for example, are displayed in the Output window. The Command window allows you to enter and execute statements. When you debug an application, you can stop it and enter VB statements in the Command window.

Project Menu - This menu contains commands for adding items to the current project (an item can be a form, a file, a component, even another project). The last option in this menu is the Set As StartUp Project command, which lets you specify which of the projects in a multiproject solution is the startup project (the one that will run when you press F5).

Build Menu - The Build menu contains commands for building (compiling) your project. The two basic commands in this menu are the Build and Rebuild All commands. The Build command compiles (builds the executable) of the entire solution, but it doesn't compile any components of the project that haven't changed since the last build. The Rebuild All command does the same, but it clears any existing files and builds the solution from scratch.

Debug Menu – This menu contains commands to start or end an application, as well as the basic debugging tools

Data Menu - This menu contains commands you will use with projects that access data.

Format Menu - The Format menu, which is visible only while you design a Windows or Web form, contains commands for aligning the controls on the form.

Tools Menu - This menu contains a list of tools, and most of them apply to C++. The Macros command of the Tools menu leads to a submenu with commands for creating macros. Just as you can create macros in an Office application to simplify many tasks, you can create macros to automate many of the repetitive tasks you perform in the IDE. I'm not going to discuss

macros in this book, but once you familiarize yourself with the environment, you should look up the topic of writing macros in the documentation.

Window Menu -This is the typical Window menu of any Windows application. In addition to the list of open windows, it also contains the Hide command, which hides all Toolboxes and devotes the entire window of the IDE to the code editor or the Form Designer. The Toolboxes don't disappear completely. They're all retracted, and you can see their tabs on the left and right edges of the IDE window. To expand a Toolbox, just hover the mouse pointer over the corresponding tab.

Help Menu -This menu contains the various help options. The Dynamic Help command opens the Dynamic

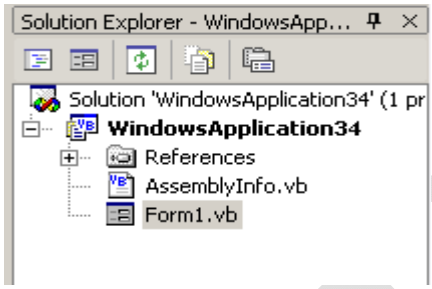
Help window, which is populated with topics that apply to the current operation. The Index command opens the Index window, where you can enter a topic and get help on the specific topic.

The Toolbox Window - Here you will find all the controls you can use to build your application's interface. The Toolbox window is usually retracted, and you must move the pointer over it to view the Toolbox. This window contains these tabs:

- Crystal Reports
- Data
- XML Schema
- Dialog Editor
- Web Forms
- Components
- Windows Forms
- HTML
- Clipboard Ring
- General

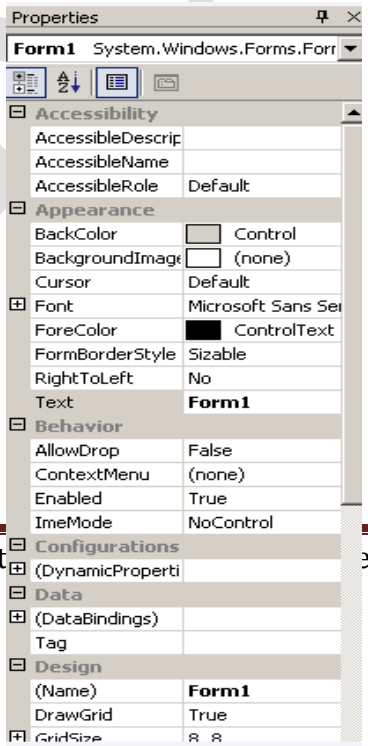
Solution Explorer Window

The Solution Explorer window gives an overview of the solution we are working with and lists all the files in the project. An image of the Solution Explorer window is shown on the right.



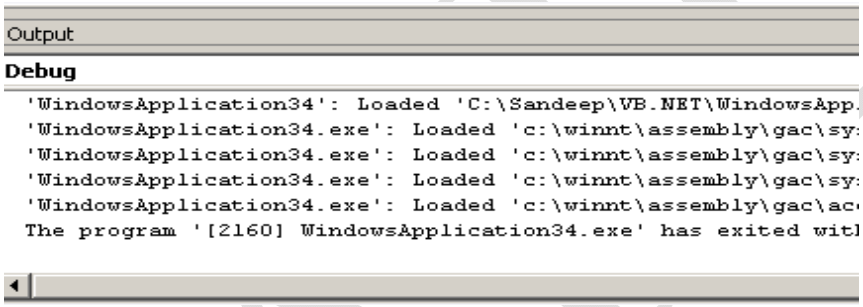
Properties Window

The properties window allows us to set properties for various objects at design time. For example, if you want to change the font, font size, backcolor, name, text that appears on a button, textbox etc, you can do that in this window. Below is the image of properties window. You can view the properties window by selecting View->Properties Window from the main menu or by pressing F4 on the keyboard.



Output Window

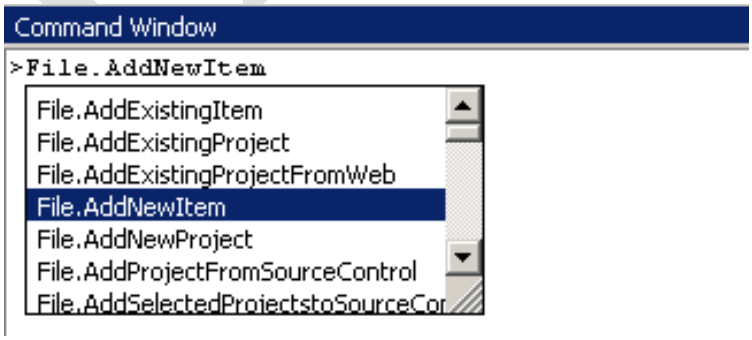
The output window as you can see in the image below displays the results of building and running applications. When a project is compiled the result of compilation, Build



succeeded or failed are displayed in the output window

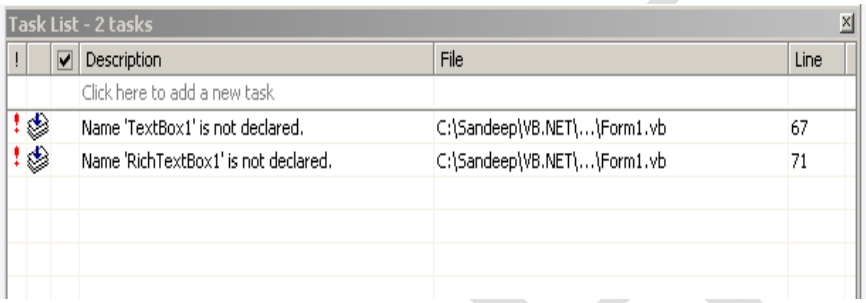
Command Window

The command window in the image below is a useful window. Using this window we can add new item to the project, add new project and so on. You can view the command window by selecting View->Other Windows -> Command Window from the main menu. The command window in the image displays all possible commands with File.



Task List Window

The task list window displays all the tasks that VB .NET assumes we still have to finish. You can view the task list window by selecting View->Show tasks->All or View->Other Windows->Task List from the main menu. The image below shows that. As you can see from the image, the task list displayed "TextBox1 not declared", "RichTextBox1 not declared". The reason for that message is, there were no controls on the form and attempts where made to write code for a textbox and a rich textbox. Task list also displays syntax errors and other errors you normally encounter during coding.



Environment Options

Open the Tools menu and select Options (the last item in the menu). The Options dialog box will appear where you can set all the options regarding the environment. Figure shows the options for the font of the various items of the IDE. Here you can set the font for various categories of items, like the Text Editor, the dialogs and toolboxes, and so on. Select an item in the Show Settings For list and then set the font for this item in the box below.

Figure - The Fonts and Colors options

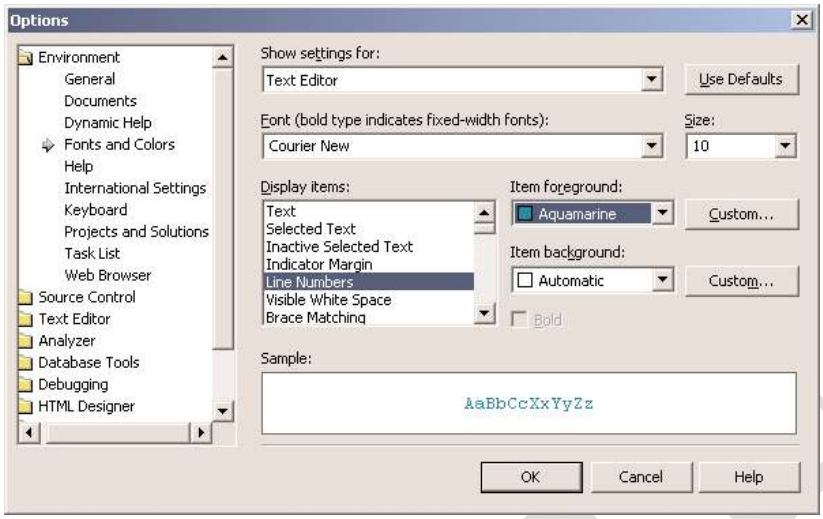
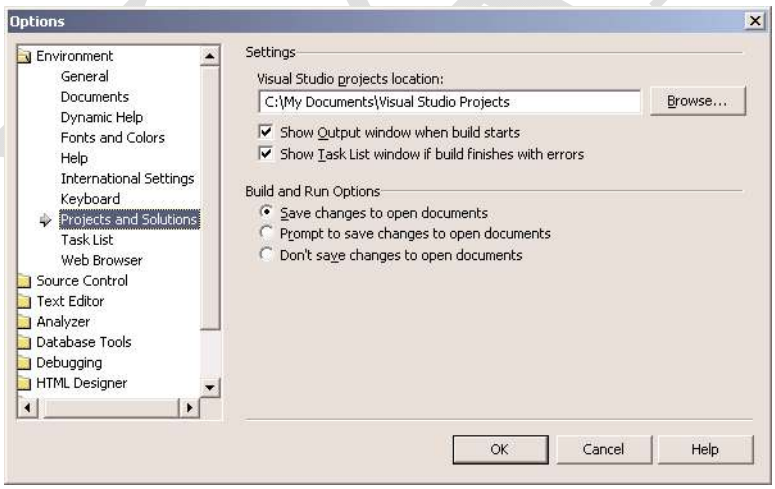


Figure shows the Projects and Solutions options. The top box is the default location for new projects. The three radio buttons in the lower half of the dialog box determine when the changes to the project are saved. By default, changes are saved when you run a project. If you activate the last option, then you must save your project from time to time with the File -> Save All command.

Figure -The Projects and Solutions options



VISUAL BASIC: THE LANGUAGE

Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in VB.Net has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Declaring Variables

To declare a variable, use the Dim statement followed by the variable's name, the As keyword, and its type, as follows:

```
Dim meters As Integer
Dim greetings As String
```

The first variable, meters, will store integers, such as 3 or 1,002; the second variable, greetings, will store text. You can declare multiple variables of the same or different type in the same line, as follows:

```
Dim Qty As Integer, Amount As Decimal, CardNum As String
```

If you want to declare multiple variables of the same type, you need not repeat the type. Just separate all the variables of the same type with commas and set the type of the last variable:

```
Dim Length, Width, Height As Integer, Volume, Area As Double
```

This statement declares three Integer variables and two Double variables. Double variables hold fractional values (or floating-point values, as they're usually called) that are similar to the Single data type, except that they can represent non-integer values with greater accuracy.

Variable-Naming Conventions

When declaring variables, you should be aware of a few naming conventions. A variable's name

- Must begin with a letter, followed by more letters or digits.
- Can't contain embedded periods or other special punctuation symbols. The only special character that can appear in a variable's name is the underscore character.
- Mustn't exceed 255 characters.

- Must be unique within its scope. This means that you can't have two identically named variables in the same subroutine, but you can have a variable named counter in many different subroutines.

Variable names in VB 2008 are case-insensitive: myAge, myage, and MYAGE all refer to the same variable in your code. Actually, as you enter variable names, the editor converts their casing so that they match their declaration.

Variable Initialization

The general form of initialization is:

```
variable_name = value;
```

for example,

```
Dim pi As Double
pi = 3.14159
```

You can initialize a variable at the time of declaration as follows:

```
Dim StudentID As Integer = 100
Dim StudentName As String = "Bill Smith"
```

Example

Try the following example which makes use of various types of variables:

```
Module variablesNdataypes
  Sub Main()
    Dim a As Short
    Dim b As Integer
    Dim c As Double
    a = 10
    b = 20
    c = a + b
    Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c)
    Console.ReadLine()
  End Sub
```

```
End Module
```

When the above code is compiled and executed, it produces the following result:

```
a = 10, b = 20, c = 30
```

Types of Variables

Visual Basic recognizes the following five categories of variables:

- Numeric
- String
- Boolean
- Date
- Object

Numeric variables

You'd expect that programming languages would use the same data type for numbers. After all, a number is a number. But this couldn't be further from the truth. All programming languages provide a variety of numeric data types, including the following:

- Integers (there are several integer data types)
- Decimals
- Single, or floating-point numbers with limited precision
- Double, or floating-point numbers with extreme precision

Integer variable

There are three types of variables for storing integers, and they differ only in the range of numbers each can represent. As you understand, the more bytes a type takes, the larger values it can hold. The type of Integer variable you'll use depends on the task at hand. You should choose the type that can represent the largest values you anticipate will come up in your calculations. You can go for the Long type, to be safe, but Long variables are four times as large as Short variables, and it takes the computer longer to process them.

Single and Double Precision numbers

The names Single and Double come from single-precision and double-precision numbers. Double-precision numbers are stored internally with greater accuracy than single-

precision numbers. In scientific calculations, you need all the precision you can get; in those cases, you should use the Double data type.

The result of the operation 1 / 3 is 0.333333. . . (an infinite number of digits 3). You could fill 256MB of RAM with 3 digits, and the result would still be truncated. Here's a simple example that demonstrates the effects of truncation:

In a button's Click event handler, declare two variables as follows:

```
Dim a As Single, b As Double
```

Then enter the following statements:

```
a=1/3
Debug.WriteLine(a)
```

Run the application, and you should get the following result in the Output window:

```
.3333333
```

There are seven digits to the right of the decimal point. Break the application by pressing Ctrl+Break and append the following lines to the end of the previous code segment:

```
a=a*100000
Debug.WriteLine(a)
```

This time, the following value will be printed in the Output window:

```
33333.34
```

The result is not as accurate as you might have expected initially — it isn't even rounded properly. If you divide a by 100,000, the result will be

```
0.3333334
```

The Decimal Data Type

Variables of the Decimal type are stored internally as integers in 16 bytes and are scaled by a power of 10. The scaling power determines the number of decimal digits to the right of the floating point, and it's an integer value from 0 to 28. When the scaling power is 0, the value is multiplied by 100, or 1, and it's represented without decimal digits. When the scaling power is 28, the value is divided by 1028 (which is 1 followed by 28 zeros — an enormous value), and it's represented with 28 decimal digits.

328.558 * 12.4051

First, you must turn them into integers. You must remember that the first number has three decimal digits, and the second number has four decimal digits. The result of the multiplication will have seven decimal digits. So you can multiply the following integer values:

328558 * 124051

and then treat the last seven digits of the result as decimals. Use the Windows Calculator (in the Scientific view) to calculate the previous product. The result is 40,757,948,458. The actual value after taking into consideration the decimal digits is 4,075.7948458. This is how the compiler manipulates the Decimal data type. Insert the following lines in a button's Click event handler and execute the program:

```
Dim a As Decimal=328.558D
Dim b As Decimal=12.4051D
Dim c As Decimal
c=a*b
Debug.WriteLine(c.ToString)
```

The D character at the end of the two numeric values specifies that the numbers should be converted into Decimal values. By default, every value with a fractional part is treated as a Double value. Assigning a Double value to a Decimal variable will produce an error if the strict option is on, so we must specify explicitly that the two values should be converted to the Decimal type. The D character at the end of the value is called a type character. Table 2.2 lists all of them.

Infinity and other Oddities

VB.NET can represent two very special values, which may not be numeric values themselves but are produced by numeric calculations:**NaN** (not a number) and Infinity. If your calculations produce NaN or Infinity, you should confirm the data and repeat the calculations, or give up. For all practical purposes, neither NaN nor Infinity can be used in everyday business calculations.

Not a Number (NaN)

```
Dim dbl Var As Double=999
```

Then divide this value by zero:

Dim infVa ras Double

infVar = dblVar / 0

and display the variable's value:

MsgBox(infVar)

result=largeVar/smallVar

MsgBox(result)

The result will be Infinity. If you reverse the operands (that is, you divide the very small by the very large variable), the result will be zero. It's not exactly zero, but the Double data type can't accurately represent numeric values that are very, very close to zero.

To divide zero by zero, set up two variables as follows:

Dim var1, var2 As Double

Dim result As Double

var1=0

var2=0

result=var1/var2

MsgBox(result)

If you execute these statements, the result will be NaN. Any calculations that involve the result variable will yield NaN as a result. The following statements will produce a NaN value:

result=result+result

result=10/result

result=result+1E299

MsgBox(result)

If you make var2 a very small number, such as 1E-299, the result will be zero. If you make var1 a very small number, the result will be Infinity.

For most practical purposes, Infinity is handled just like NaN. They're both numbers that shouldn't occur in business applications (unless you're projecting the national deficit in the next 50 years), and when they do, it means that you must double-check your code or your data.

The Byte Data Type

None of the previous numeric types is stored in a single byte. In some situations, however, data are stored as bytes, and you must be able to access individual bytes. The Byte data type holds an integer in the range of 0 to 255. Bytes are frequently used to access binary files, image and sound files, and so on. Note that you no longer use bytes to access individual characters. Unicode characters are stored in two bytes.

To declare a variable as a Byte, use the following statement:

```
Dim n As Byte
```

The variable n can be used in numeric calculations too, but you must be careful not to assign the result to another Byte variable if its value might exceed the range of the Byte type. If the variables A and B are initialized as follows:

```
Dim A As Byte, B As Byte
A=233
B = 50
```

the following statement will produce an overflow exception:

```
Debug.WriteLine(A + B)
```

The same will happen if you attempt to assign this value to a Byte variable with the following statement:

```
B = A + B
```

The result (283) can't be stored in a single byte. Visual Basic generates the correct answer, but it can't store it into a Byte variable.

Boolean variable

The Boolean data type stores True/False values. Boolean variables are, in essence, integers that take the value −1 (for True) and 0 (for False). Actually, any nonzero value is considered True. Boolean variables are declared as

```
Dim failure As Boolean
```

and they are initialized to False. Boolean variables are used in testing conditions, such as the following:

```
Dim failure As Boolean=False
' other statements ...
```

If failure Then MsgBox(”Couldn't complete the operation”)

They are also combined with the logical operators And, Or, Not, and Xor. The Not operator toggles the value of a Boolean variable. The following statement is a toggle:

running = Not running

If the variable running is True, it's reset to False, and vice versa. This statement is a shorter way of coding the following:

```
Dim running As Boolean
If running=True Then
running=False
Else
running=True
End If
```

String variable

The String data type stores only text, and string variables are declared as follows:

```
Dim anyText As String
Dim a String As String
aString = "Now is the time for all good men to come " &
" to the aid of their country"
aString=""
aString = "There are approximately 25,000 words in this chapter"
aString = "25,000"
```

The second assignment creates an empty string, and the last one creates a string that just happens to contain numeric digits, which are also characters. The difference between these two variables is that they hold different values:

```
Dim a Number As Integer=25000
Dim aString As String = "25,000"
```

The aString variable holds the characters 2, 5, comma, 0, 0, and 0; and aNumber holds a single numeric value. However, you can use the variable aString in numeric calculations, and the

variable aNumber in string operations. VB will perform the necessary conversions as long as the strict option is off.

Character Variable

Character variables store a single Unicode character in two bytes. In effect, characters are Unsigned Short integers (UInt16); you can use the CChar() function to convert integers to characters and use the CInt() function to convert characters to their equivalent integer values.

To declare a Character variable, use the Char keyword:

```
Dim char1, char2 As Char
```

You can initialize a Character variable by assigning either a character or a string to it. In the latter case, only the first character of the string is assigned to the variable. The following statements will print the characters a and A to the Output window:

```
Dim char1 As Char = "a", char2 As Char = "ABC"
Debug.WriteLine(char1)
Debug.WriteLine(char2)
```

These statements will work only if the Strict option is off. If it's on, the values assigned to the char1 and char2 variables will be marked in error. To fix the error that prevents the compilation of the code, change the Dim statement as follows:

```
Dim char1 As Char = "a"c, char2 As Char = "A"c
```

When the Strict option is on, you can't assign a string to a Char variable and expect that only the first character of the string will be used.

The Integer values that correspond to the English characters are the ANSI (American National Standards Institute) codes of the equivalent characters. The following statement will print the value 65:

```
Debug.WriteLine(Convert.ToInt32("a"))
```

If you convert the Greek character alpha (α) to an integer, its value is 945. The Unicode value of the famous character π is 960.

Date variable

Date and time values are stored internally in a special format, but you don't need to know the exact format. They are double-precision numbers: the integer part represents the date, and the fractional part represents the time. A variable declared as Date with a statement like the following can store both date and time values:

```
Dim expiration As Date
```

The following are all valid assignments:

```
expiration=#01/01/2008#
expiration=#8/27/20086:29:11PM#
expiration="July2,2008"
expiration = Today()
```

By the way, the Today() function returns the current date and time, while the Now() function returns the current date. You can also retrieve the current date by calling the Today property of the Date data type: Date.Today.

```
Dimd1,d2 As Date
d1=Now
d2 = #1/1/2004#Debug.WriteLine(d1 - d2)
```

The value of the TimeSpan object represents an interval of 638 days, 8 hours, 49 minutes, and 51.497 seconds.

Data Type Identifier

Finally, you can omit the As clause of the Dim statement, yet create typed variables, with the variable declaration characters, or data type identifiers. These characters are special symbols that you append to the variable name to denote the variable's type. To create a string variable, you can use this statement:

```
Dim myText$
```

The dollar sign signifies a string variable. Notice that the name of the variable includes the dollar sign — it's myText$, not myText. To create a variable of a particular type, use one of the data declaration characters shown in the following table. (Not all data types have their own identifiers.)

Table 2.3 - Data Type Definition Characters

Symbol	Data Type	Example
$	String	A$, messageText$
%	Integer (Int32)	counter%, var%
&	Long (Int64)	population&, colorValue&
!	Single	distance!
#	Double	ExactDistance
@	Decimal	Balance@

Using type identifiers doesn't help to produce the cleanest and easiest-to-read code.

The Strict and Explicit options

The Visual Basic compiler provides three options that determine how it handles variables:

- The Explicit option indicates whether you will declare all variables.
- The Strict option indicates whether all variables will be of a specific type.
- The Infer option indicates whether the compiler should determine the type of a variable from its value.

To change the default behavior, you must insert the following statement at the beginning of the file:

Option Explicit Off

The Option Explicit statement must appear at the very beginning of the file. This setting affects the code in the current module, not in all files of your project or solution. You can turn on the Strict (as well as the Explicit) option for an entire solution. Open the solution's properties dialog box (right-click the solution's name in Solution Explorer and select Properties), select the Compile tab, and set the Strict and Explicit options accordingly, as shown in Figure

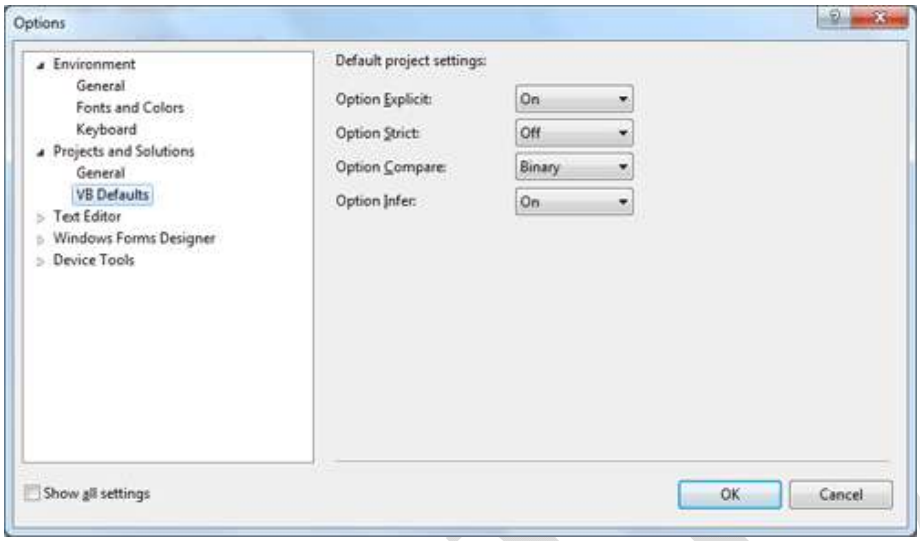


Figure - Setting the variable-related options in the Visual Studio Options dialog box

The Strict option requires that variables are declared with a specific type. In other words, the Strict option disallows the use of generic variables that can store any data type.

The default value of the Explicit statement is On. This is also the recommended value, and you should not make a habit of changing this setting. In the section "Reasons for Decalring Variables" later in this chapter, you will see an example of the pitfalls you'll avoid by declaring your variables. By setting the Explicit option to Off, you're telling VB that you intend to use variables without declaring them. As a consequence, VB can't make any assumption about the variable's type, so it uses a generic type of variable that can hold any type of information. These variables are called Object variables, and they're equivalent to the old variants.

While the option Explicit is set to Off, every time Visual Basic runs into an undeclared variable name, it creates a new variable on the spot and uses it. The new variable's type is Object, the generic data type that can accommodate all other data types. Using a new variable in your code is equivalent to declaring it without type. Visual Basic adjusts its type according to the value you assign to it. Create two variables, var1 and var2, by referencing them in your code with statements like the following ones:

Option Strict On

If you attempt to execute any of the last two code segments while the Strict option is on, the compiler will underline a segment of the statement to indicate an error. If you rest the

pointer over the underlined segment of the code, the following error message will appear in a tip box:

Option strict disallows implicit conversions from String to Double

(or whatever type of conversion is implied by the statement).

When the Strict option is set to On, the compiler doesn't disallow all implicit conversions between data types. For example, it will allow you to assign the value of an integer to a Long, but not the opposite. The Long value might exceed the range of values that can be represented by an Integer variable.

Object Variables

Variants — variables without a fixed data type— were the bread and butter of VB programmers up to version 6. Variants are the opposite of strictly typed variables: They can store all types of values, from a single character to an object. If you're starting with VB 2008, you should use strictly typed variables. However, variants are a major part of the history of VB, and most applications out there (the ones you may be called to maintain) use them. I will discuss variants briefly in this section and show you what was so good (and bad) about them.

Variants, or object variables, were the most flexible data types because they could accommodate all other types. A variable declared as Object (or a variable that hasn't been declared at all) is handled by Visual Basic according to the variable's current contents. If you assign an integer value to an object variable, Visual Basic treats it as an integer. If you assign a string to an object variable, Visual Basic treats it as a string. Variants can also hold different data types in the course of the same program. Visual Basic performs the necessary conversions for you.

To declare a variant, you can turn off the Strict option and use the Dim statement without specifying a type, as follows:

```
Dim myVar
```

If you don't want to turn off the Strict option (which isn't recommended, anyway), you can declare the variable with the Object data type:

```
Dim myVar As Object
```

Every time your code references a new variable, Visual Basic will create an object variable. For example, if the variable validKey hasn't been declared, when Visual Basic runs into the following line, it will create a new object variable and assign the value 002-6abbgd to it:

validKey = "002-6abbgd"

You can use object variables in both numeric and string calculations. Suppose that the variable modemSpeed has been declared as Object with one of the following statements:

Dim modemSpeed ' with Option Strict = Off

Dim modemSpeed As Object ' with Option Strict = On

and later in your code you assign the following value to it:

modemSpeed = "28.8"

The modemSpeed variable is a string variable that you can use in statements such as the following:

MsgBox "We suggest a " & modemSpeed & " modem."

This statement displays the following message:

"We suggest a 28.8 modem."

Converting Variable Types

In many situations, you will need to convert variables from one type into another. Table 2.4 shows the methods of the Convert class that perform data-type conversions.

In addition to the methods of the Convert class, you can still use the data-conversion functions of VB (CInt() to convert a numeric value to an Integer, CDbl() to convert a numeric value to a Double, CSng() to convert a numeric value to a Single, and so on), which you can look up in the documentation. If you're writing new applications in VB 2008, use the new Convert class to convert between data types.

To convert the variable initialized as the following

Dim A As Integer

to a Double, use the ToDouble method of the Convert class:

Dim B As Double

B = Convert.ToDouble(A)

Suppose that you have declared two integers, as follows:

```
Dim A As Integer, B As Integer
A=23
B = 7
```

The result of the operation A / B will be a Double value. The following statement

```
Debug.Write(A / B)
```

displays the value 3.28571428571429. The result is a Double value, which provides the greatest possible accuracy. If you attempt to assign the result to a variable that hasn't been declared as Double, and the Strict option is on, then VB 2008 will generate an error message. No other data type can accept this value without loss of accuracy. To store the result to a Single variable, you must convert it explicitly with a statement like the following:

```
Convert.ToSingle(A / B)
```

You can also use the DirectCast() function to convert a variable or expression from one type to another. The DirectCast() function is identical to the CType() function. Let's say the variable A has been declared as String and holds the value 34.56. The following statement converts the value of the A variable to a Decimal value and uses it in a calculation:

```
Dim A As String="34.56"
Dim B As Double
B = DirectCast(A, Double) / 1.14
```

The conversion is necessary only if the strict option is on, but it's a good practice to perform your conversions explicitly. The following section explains what might happen if your code relies on implicit conversions.

Table - The Data-Type Conversion Methods of the Convert Class

Method	Converts Its Argument To
ToBoolean	Boolean
ToByte	Byte
ToChar	Unicode character

ToDateTime	Date
ToDecimal	Decimal
ToDouble	Double
ToInt16	Short Integer (2-byte integer, Int16)
ToInt32	Integer (4-byte integer, Int32)
ToInt64	Long (8-byte integer, Int64)
ToSByte	Signed Byte
CShort	Short (2-byte integer, Int16)
ToSingle	Single
ToString	String
ToUInt16	Unsigned Integer (2-byte integer, Int16)
ToUInt32	Unsigned Integer (4-byte integer, Int32)
ToUInt64	Unsigned Long (8-byte integer, Int64)

User Defined Data Types

You can create custom data types that are made up of multiple values using structures. A VB structure allows you to combine multiple values of the basic data types and handle them as a whole.

For example, each check in a check tutorial-balancing application is stored in a separate structure (or record), as shown in Figure 2.3. When you recall a given check, you need all the information stored in the structure.

Record Structure

Check Number	Check Date	Check Amount	Check Paid To

Array Of Records

275	04/12/01	104.25	Gas Co.
276	04/12/01	48.76	Books
277	04/14/01	200.00	VISA
278	04/21/01	430.00	Rent

Figure - Pictorial representation of a structure

To define a structure in VB 2008, use the Structure statement, which has the following syntax:

```
Structure structureName
Dim variable1 As varType
Dim variable2 As varType
...
Dim variable As varType
End Structure
```

Where, varType can be any of the data types supported by the CLR. The Dim statement can be replaced by the Private or Public access modifiers. For structures, Dim is equivalent to Public.

After this declaration, you have in essence created a new data type that you can use in your application. structureName can be used anywhere you'd use any of the base types (Integers, Doubles, and so on). You can declare variables of this type and manipulate them as you manipulate all other variables (with a little extra typing). The declaration for the CheckRecord structure shown in Figure 2.3 is as follows:

```
Structure CheckRecord
Dim CheckNumber As Integer
Dim CheckDate As Date
Dim CheckAmount As Single
Dim CheckPaidTo As String
End Structure
```

This declaration must appear outside any procedure; you can't declare a Structure in a subroutine or function. Once declared, The CheckRecord structure becomes a new data type for your application.

To declare variables of this new type, use a statement such as this one:

Dim check1 As CheckRecord, check2 As CheckRecord

To assign a value to one of these variables, you must separately assign a value to each one of its components (they are called fields), which can be accessed by combining the name of the variable and the name of a field, separated by a period, as follows:

check1.CheckNumber = 275

Actually, as soon as you type the period following the variable's name, a list of all members to the CheckRecord structure will appear. Notice that the structure supports a few members on its own.

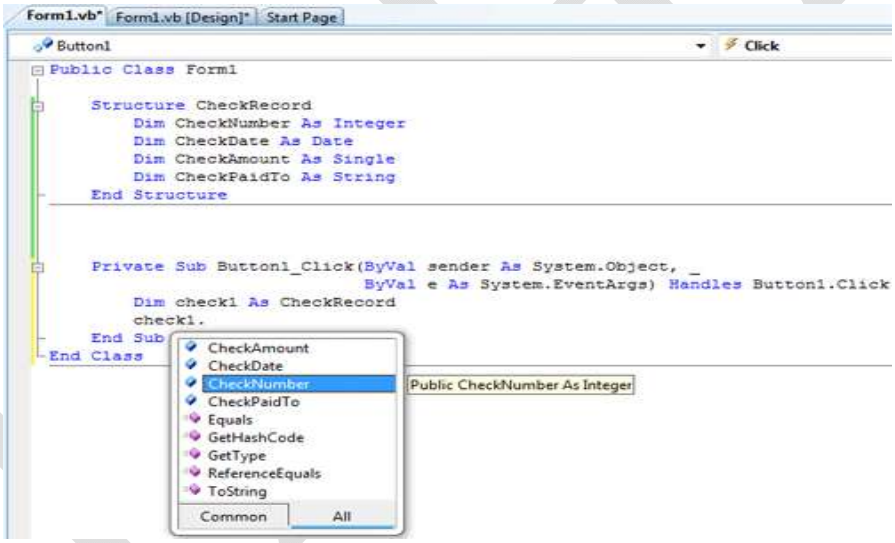


Figure - Variables of custom types expose their members as properties

You didn't write any code for the Equals, GetType, and ToString members, but they're standard members of any Structure object, and you can use them in your code. Both the GetType and ToString methods will return a string like ProjectName.FormName + CheckRecord. You can provide your own implementation of the ToString method, which will return a more meaningful string:

```
Public Overrides Function ToString() As String
Return "CHECK # " & CheckNumber & " FOR " &
CheckAmount.ToString("C")
End Function
```

As you understand, structures are a lot like objects that expose their fields as properties and then expose a few members of their own. The following statements initialize a CheckRecord variable:

```
check2.CheckNumber=275
check2.CheckDate=#09/12/2008#
check2.CheckAmount=104.25
check2.CheckPaidTo = "Gas Co."
```

You can also create arrays of structures with a declaration such as the following (arrays are discussed later in this chapter):

```
Dim Checks(100) As CheckRecord
```

Each element in this array is a CheckRecord structure and it holds all the fields of a given check. To access the fields of the third element of the array, use the following notation:

```
Checks(2).CheckNumber=275
Checks(2).CheckDate=#09/12/2008#
Checks(2).CheckAmount=104.25
Checks(2).CheckPaidTo = "Gas Co."
```

Examining the Variable Types

- **IsNumeric()**

Returns True if its argument is a number (Short, Integer, Long, Single, Double, Decimal). Use this function to determine whether a variable holds a numeric value before passing it to a procedure that expects a numeric value or before processing it as a number. The following statements keep prompting the user with an InputBox for a numeric value. The user must enter a numeric value or click the Cancel button to exit. As long as the user enters non-numeric values, the Input box keeps popping up and prompting for a numeric value:

```
Dim str Age As String= ""
Dim Age As Integer
While NotIsNumeric(strAge)
strAge=InputBox("Please enter your age")
EndWhile
Age = Convert.ToInt16(strAge)
```

The variable strAge is initialized to a non-numeric value so that the While. . .End While loop will be executed at least once

- **IsDate()**

Returns True if its argument is a valid date (or time). The following expressions return True because they all represent valid dates:

```
IsDate(#10/12/2010#)
IsDate("10/12/2010")
IsDate("October 12, 2010")
```

If the date expression includes the day name, as in the following expression, the IsDate() function will return False:

```
IsDate("Sat. October 12, 2010") ' FALSE
```

- **IsArray()**

Returns True if its argument is an array.

A Variable's Scope

In addition to its type, a variable also has a scope. The scope (or visibility) of a variable is the section of the application that can see and manipulate the variable. If a variable is declared within a procedure, only the code in the specific procedure has access to that variable; this variable doesn't exist for the rest of the application. When the variable's scope is limited to a procedure, it's called local.

Suppose that you're coding the Click event of a button to calculate the sum of all even numbers in the range 0 to 100. One possible implementation is shown in Listing 2.4.

Listing: Summing Even Numbers

```
Private Sub Button1_Click(ByValsenderAsObject,_ByVale As System.EventArguments)
Handles Button1.Click
Dim I As Integer
Dim Sum As Integer
For i=0 to100 Step2
Sum=Sum+i
Next
MsgBox "The sum is " & Sum.ToString
End Sub
```

Listing: Variable Scoped in ItsOwn Block

```
Private SubButton1_Click(ByValsenderAsObject,_ByVale As System.EventArguments)
Handles Button1.Click
Dim i, Sum As Integer
For i=0 to100 Step2
Dim sqrValue As Integer
sqrValue=i*i
Sum=Sum+sqrValue
Next
MsgBox "The sum of the squares is " & Sum
End Sub
```

Constants

Some variables don't change value during the execution of a program. These variables are constants that appear many times in your code. For instance, if your program does math calculations, the value of pi (3.14159. . .) might appear many times. Instead of typing the value 3.14159 over and over again, you can define a constant, name it pi, and use the name of the constant in your code. The statement

circumference = 2 * pi * radius

is much easier to understand than the equivalent

circumference = 2 * 3.14159 * radius

You could declare pi as a variable, but constants are preferred for two reasons:

Constants don't change value. This is a safety feature. After a constant has been declared, you can't change its value in subsequent statements, so you can be sure that the value specified in the constant's declaration will take effect in the entire program.

Constants are processed faster than variables. When the program is running, the values of constants don't have to be looked up. The compiler substitutes constant names with their values, and the program executes faster.

```
' The following statements declare constants.
Const maxval As Long = 4999
Public Const message As String = "HELLO"
Private Const piValue As Double = 3.1415
```

Example

The following example demonstrates declaration and use of a constant value:

```
Module constantsNenum
  Sub Main()
    Const PI = 3.14149
    Dim radius, area As Single
    radius = 7
    area = PI * radius * radius
    Console.WriteLine("Area = " & Str(area))
    Console.ReadKey()
  End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
Area = 153.933
```

Print and Display Constants in VB.Net

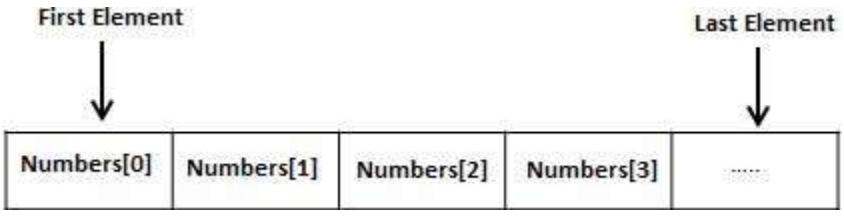
VB.Net provides the following print and display constants:

Constant	Description
vbCrLf	Carriage return/linefeed character combination.
vbCr	Carriage returns character.
vbLf	Linefeed character.
vbNewLine	Newline character.
vbNullChar	Null character.
vbNullString	Not the same as a zero-length string (""); used for calling external procedures.
vbObjectError	Error number. User-defined error numbers should be greater than this value. For example: Err.Raise(Number) = vbObjectError + 1000
vbTab	Tab character.
vbBack	Backspace character.

Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Creating Arrays in VB.Net

To declare an array in VB.Net, you use the Dim statement. For example,

```
Dim intData(30)      ' an array of 31 elements
Dim strData(20) As String   ' an array of 21 strings
Dim twoDarray(10, 20) As Integer     'a two dimensional array of integers
Dim ranges(10, 100)          'a two dimensional array
```

You can also initialize the array elements while declaring the array. For example,

```
Dim intData() As Integer = {12, 16, 20, 24, 28, 32}
Dim names() As String = {"Karthik", "Sandhya","Shivangi", "Ashwitha", "Somnath"}
Dim miscData() As Object = {"Hello World", 12d, 16ui, "A"c}
```

Initializing Arrays

Just as you can initialize variables in the same line in which you declare them, you can initialize arrays, too, with the following constructor (an array initializer, as it's called):

Dim arrayname() As type = {entry0, entry1, ... entryN}

Here's an example that initializes an array of strings:

Dim Names() As String = {"Joe Doe", "Peter Smack"}

This statement is equivalent to the following statements, which declare an array with two elements and then set their values:

Dim Names(1) As String

Names(0)="JoeDoe"

Names(1) = "Peter Smack"

Array Limits

The first element of an array has index 0. The number that appears in parentheses in the Dim statement is one fewer than the array's total capacity and is the array's upper limit (or upper bound). The index of the last element of an array (its upper bound) is given by the method GetUpperBound, which accepts as an argument the dimension of the array and returns the upper bound for this dimension.

The arrays we examined so far are one-dimensional and the argument to be passed to the GetUpperBound method is the value 0. The total number of elements in the array is given by the method GetLength, which also accepts a dimension as an argument. The upper bound of the following array is 19, and the capacity of the array is 20 elements:

```
Dim Names(19) As Integer
```

The first element is Names(0), and the last is Names(19). If you execute the following statements, the highlighted values will appear in the Output window:

```
Debug.WriteLine(Names.GetLowerBound(0))
0
Debug.WriteLine(Names.GetUpperBound(0))
19
```

To assign a value to the first and last element of the Names array, use the following statements:

```
Names(0)="Firstentry"
Names(19) = "Last entry"
```

If you want to iterate through the array's elements, use a loop like the following one:

```
Dim I As Integer, myArray(19) As Integer
For i=0TomyArray.GetUpperBound(0)
myArray(i)=i*1000
Next
```

The actual number of elements in an array is given by the expression myArray.GetUpperBound(0) + 1. You can also use the array's Length property to retrieve the count of elements. The following statement will print the number of elements in the array myArray in the Output window:

```
Debug.WriteLine(myArray.Length)
```

Dynamic Arrays

Dynamic arrays are arrays that can be dimensioned and re-dimensioned as par the need of the program. You can declare a dynamic array using the **ReDim** statement.

Syntax for ReDim statement:

```
ReDim [Preserve] arrayname(subscripts)
```

Where,

The **Preserve** keyword helps to preserve the data in an existing array, when you resize it.

arrayname is the name of the array to re-dimension.

subscripts specifies the new dimension.

```
Module arrayApl
  Sub Main()
    Dim marks() As Integer
    ReDim marks(2)
    marks(0) = 85
    marks(1) = 75
    marks(2) = 90
    ReDim Preserve marks(10)
    marks(3) = 80
    marks(4) = 76
    marks(5) = 92
    marks(6) = 99
    marks(7) = 79
    marks(8) = 75
    For i = 0 To 10
      Console.WriteLine(i & vbTab & marks(i))
    Next i
    Console.ReadKey()
  End Sub
End Module
```

Multi-Dimensional Arrays

VB.Net allows multidimensional arrays. Multidimensional arrays are also called rectangular arrays.

You can declare a 2-dimensional array of strings as:

```
Dim twoDStringArray(10, 20) As String
```

or, a 3-dimensional array of Integer variables:

```
Dim threeDIntArray(10, 10, 10) As Integer
```

The following program demonstrates creating and using a 2-dimensional array:

```
Module arrayApl
  Sub Main()
    ' an array with 5 rows and 2 columns
    Dim a(,) As Integer = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}}
    Dim i, j As Integer

    ' output each array element's value '
    For i = 0 To 4
      For j = 0 To 1
        Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))
      Next j
    Next i
    Console.ReadKey()
  End Sub
End Module
```

Reinitializing Arrays

We can change the size of an array after creating them. The ReDim statement assigns a completely new array object to the specified array variable. You use ReDim statement to change the number of elements in an array. The following lines of code demonstrate that. This code reinitializes the Test array declared above.

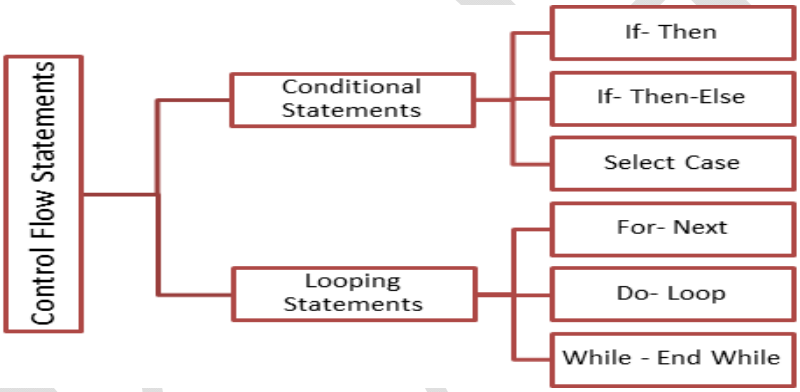
```
Dim Test(10) As Integer
ReDim Test(25) As Integer
'Reinitializing the array
```

When using the Redim statement all the data contained in the array is lost. If you want to preserve existing data when reinitializing an array then you should use the Preserve keyword which looks like this:

```
Dim Test() as Integer={1,3,5}
'declares an array an initializes it with three members
ReDim Preserve Test(25)
'resizes the array and retains the the data in elements 0 to 2
```

Control Flow statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false



Decision Statements

Applications need a mechanism to test conditions and take a different course of action depending on the outcome of the test. Visual Basic provides three such decision, or conditional, statements:

- If. . .Then
- If. . .Then. . .Else
- Select Case

Loop Statements

Loop statements allow you to execute one or more lines of code repetitively. Many tasks consist of operations that must be repeated over and over again, and loop statements are an

important part of any programming language. Visual Basic supports the following loop statements:

- For. . .Next
- Do. . .Loop
- While. . .End While

Decision Statements

1) If Then Statement

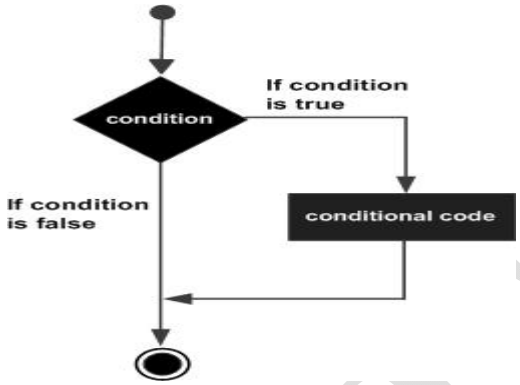
If Then statement is a control structure which executes a set of code only when the given condition is true.

Syntax:

If [Condition] Then
[Statements]

In the above syntax when the **Condition** is true then the **Statements** after **Then** are executed.

Flow Diagram:



Example:

```
Private Sub Button1_Click_1(ByVal sender As System.Object, ByVal e As system.EventArgs)
Handles Button1.Click
  If  Val(TextBox1.Text) > 25 Then
     TextBox2.Text = "Eligible"
  End If
```

Description:

In the above If Then example the button click event is used to check if the age got using **TextBox1** is greater than **25**, if true a message is displayed in **TextBox2**

2) If Then Else Statement

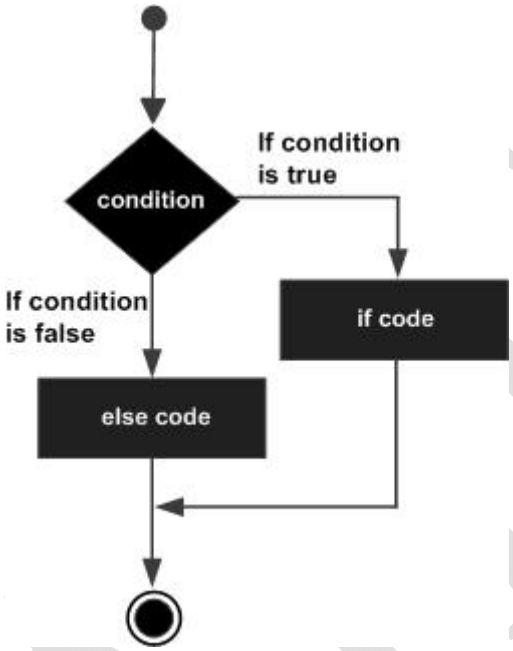
If Then Else statement is a control structure which executes different set of code statements when the given condition is true or false.

Syntax:

```
If [Condition] Then
  [Statements]
Else
  [Statements]
```

In the above syntax when the **Condition** is true, the **Statements** after **Then** are executed.If the condition is false then the statements after the **Else** part is executed.

Flow Diagram:



Example:

```
Private Sub Button1_Click (ByVal sender As System.Object,  ByVal e As System.EventArgs)
Handles Button1.Click
  If  Val(TextBox1.Text) >= 40 Then
    MsgBox("GRADUATED")
      Else
    MsgBox("NOT GRADUATED")
  End If
End Sub
```

Description:

In the above If Then Else example the marks are entered in **TextBox1**.When a button is clicked a message **GRADUATED** is displayed if the condition (>40) is true and **NOT GRADUATED** if it is false.

3) Nested If Then Else Statement

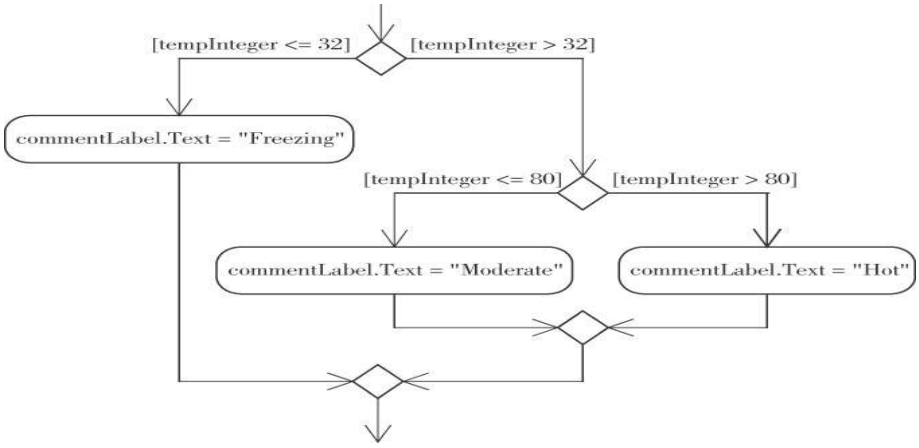
Nested If..Then..Else statement is used to check multiple conditions using if then else statements nested inside one another.

Syntax:

```
If [Condition] Then
  If [Condition] Then
    [Statements]
  Else
    [Statements]
Else
  [Statements]
```

In the above syntax when the **Condition** of the first if then else is true, the second if then else is executed to check another two conditions. If false the statements under the Else part of the first statement is executed.

Flow Diagram



Example:

```
 Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click
   If  Val(TextBox1.Text) >= 40 Then
```

```
    If  Val(TextBox1.Text) >= 60 Then
      MsgBox("You have FIRST Class")
    Else
      MsgBox("You have SECOND Class")
    End If
  Else
    MsgBox("Check your Average marks entered")
  End If
End Sub
```

Description:

In the above nested if then else statement example first the average mark is checked if it is more than 40, if true the second if then else control is used check for first or second class. If the first condition is false the statements under the else part is executed.

4) Select Case Statement

Select case statement is used when the expected results for a condition can be known previously so that different set of operations can be done based on each condition.

Syntax:

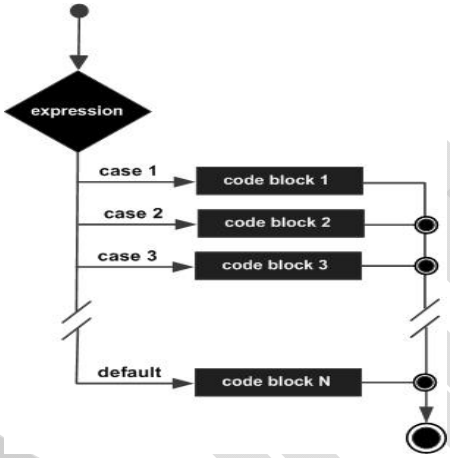
```
Select Case Expression
  Case Expression1
    Statement1
  Case Expression2
    Statement2
  Case Expressionn
    Statementn
    ...
  Case Else
```

Statement

End Select

In the above syntax, the value of the **Expression** is checked with **Expression1..n** to check if the condition is true. If none of the conditions are matched the statements under the **Case Else** is executed.

Flow Diagram:



Example:

```
Private Sub Button1_Click(ByVal sender As System.Object,ByVal e As System.EventArgs)
Handles Button1.Click
  Dim c As String
  c = TextBox1.Text
  Select c
  Case "Red"
    MsgBox("Color code of Red is::#FF0000")
  Case "Green"
    MsgBox("Color code of Green is::#808000")
```

```
  Case "Blue"
    MsgBox("Color code of Blue is:: #0000FF")
  Case Else
    MsgBox("Enter correct choice")
  End Select
End Sub
```

Description:

In the above example based on the color input in **TextBox1**, the color code for RGB colors are displayed, if the color is different then the statement under **Case Else** is executed. Thus we can easily execute the select case statement.

Loop Statements

1) Do While Loop Statement

Do While Loop Statement is used to execute a set of statements only if the condition is satisfied. But the loop gets executed once for a false condition once before exiting the loop. This is also known as **Entry Controlled loop**.

Syntax:

```
  Do While [Condition]
    [Statements]
  Loop
```

In the above syntax the **Statements** are executed till the **Condition** remains true.

Example:

```
  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    Dim a As Integer
    a = 1
```

```
Do While a < 100
  a = a * 2
  MsgBox("Product is::" & a)
Loop
End Sub
```

Description:

In the above Do While Loop example the loop is continued after the value 64 to display 128 which is false according to the given condition and then the loop exits.

2) Do Loop While Statement

Do Loop While Statement executes a set of statements and checks the condition, this is repeated until the condition is true. .It is also known as an **Exit Control** loop

Syntax:

```
Do
  [Statements]
Loop While [Condition]
```

In the above syntax the **Statements** are executed first then the **Condition** is checked to find if it is true.

Example:

```
Private Sub Form1_Load(ByVal sender As System.Object,  ByVal e As System.EventArgs)
Handles MyBase.Load
  Dim cnt As Integer
    Do
   cnt = 10
     MsgBox("Value of cnt is::" & cnt)
  Loop While cnt <= 9
End Sub
```

Description:

In the above Do Loop While example, a message is displayed with a value 10 only after which the condition is checked, since it is not satisfied the loop exits.

3) For Next Loop Statement

For Next Loop Statement executes a set of statements repeatedly in a loop for the given initial, final value range with the specified step by step increment or decrement value.

Syntax:

```
For counter = start To end [Step]
  [Statement]
Next [counter]
```

In the above syntax the **Counter** is range of values specified using the **Start** ,**End** parameters. The **Step** specifies step increment or decrement value of the counter for which the statements are executed.

Example:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    Dim i As Integer
    Dim j As Integer
    j = 0
    For i = 1 To 10 Step 1
      j = j + 1
      MsgBox("Value of j is::" & j)
    Next i
End Sub
```

Description:

In the above For Next Loop example the counter value of i is set to be in the range of 1 to 10 and is incremented by 1. The value of j is increased by 1 for 10 times as the loop is repeated.

Nested Control Structures

You can place, or nest, control structures inside other control structures (such as an If. . .Then block within a For. . .Next loop). Control structures in Visual Basic can be nested in as many levels as you want. The editor automatically indents the bodies of nested decision and loop structures to make the program easier to read.

When you nest control structures, you must make sure that they open and close within the same structure. In other words, you can't start a For. . .Next loop in an If statement and close the loop after the corresponding End If. The following code segment demonstrates how to nest several flow-control statements. (The curly brackets denote that regular statements should appear in their place and will not compile, of course.)

```
For a=1 To 100
{statements}
If a=99 Then
{statements}
EndIf
While b<a
{statements}
If total<=0 Then
{statements}
EndIf
EndWhile
For c=1 to a
{statements}
Next c
Next a
```

Listing: Simple Nested If Statements

```
Dim Income As Decimal
Income=Convert.ToDecimal(InputBox("Enteryourincome"))
If Income >0 Then
If Income>12000 Then
MsgBox"You will pay taxes this year"
Else
MsgBox"You won't pay any taxes this year"
End If
Else
MsgBox"Bummer"
End If
```

The Exit Statement

The Exit statement allows you to exit prematurely from a block of statements in a control structure, from a loop, or even from a procedure. Suppose that you have a For. . .Next loop that calculates the square root of a series of numbers. Because the square root of negative numbers can't be calculated (the Math.Sqrt method will generate a runtime error

```
For i=0 ToUBound(nArray)
If nArray(i)<0 Then
MsgBox("Can'tcompletecalculations"&vbCrLf&_
"Item"& i.ToString & "isnegative!"
Exit For
EndIf
nArray(i)=Math.Sqrt(nArray(i))
Next
```

If a negative element is found in this loop, the program exits the loop and continues with the statement following the Next statement.

There are similar Exit statements for the Do loop (Exit Do), the While loop (Exit While), the Select statement (Exit Select), and for functions and subroutines (Exit Function and Exit Sub). If the previous loop was part of a function, you might want to display an error and exit not only the loop, but also the function itself by using the Exit Function statement.

POSSIBLE QUESTIONS

PART A (1 Mark)

(Online Examinations)

PART B (6 Marks)

1. Explain about variables with example.
2. Discuss in detail about flow control structures with example
3. Explain about various data types with example.
4. Write a vb.net program to calculate the Simple interest and Compound Interest.
5. Briefly explain about the IDE environment.
6. Explain about with example.

   i) While loop ii) Do while loop iii) if….else stmt iv) else if stmt
7. Discuss in detail about Integrated Development Components.
8. Write a vb.net program to calculate the factorial of n numbers.
9. Briefly explain about arrays and conditional statements with examples
10. Briefly explain about nested if statement with example

PART A (1 Mark) – Unit 1

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
1	.Net is a technology developed by _____ company	Microsoft	Sun Microsystems	IBM	Apple computers	Microsoft
2	_____is also known as the "execution engine" of .NET.	CLR	CTS	MSIL	WPF	CLR
3	Code that targets the Common Language Runtime is known as _____	Distributed Code	Managed Code	Legacy code	Native Code	Managed Code
4	The _____ defines the minimum standards that .NET language compilers must conform source code compiled by a .NET compiler	CLS	CTS	CLR	MSIL	CLS

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
5	VB.Net is a _____programming paradigm.	Procedural	Structured	Object Oriented	Monolithic	Object Oriented
6	Data members of a class are by default _____	public	private	static	volatile	private
7	Member functions of a class are by default _____	public	private	static	volatile	public
8	IDE stands for _____	Internet Design Environment	Integrated Development Environment	Internet Distributed Environment	Interface Design Environment	Integrated Development Environment
9	The final compiled version of a Project is ____	Form	Software	Components	Files	Components

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
10	_____ is a collection of files that can be compiled to create a distributed component	Form	Software	Components	Project	Project
11	_____ is a collection of projects and files that composed an application or component	Solution	Software	Forms	Project	Solution
12	_____menu contains commands for opening and saving projects	File	Edit	View	Project	File
13	Every object has a distinct set of attributes known as	members	datas	properties	methods	properties
14	The property that must be set first for any new object is the _____	Name	Colour	Size	Binding	Name

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
15	Objects that can be placed on a form are called _____	Pictures	Tools	Buttons	Controls	Controls
16	Controls that do not have physical appearance are called _____	invisible-at-runtime-controls	visible-at-runtime-controls	virtual controls	physical controls	invisible-at-runtime-controls
17	The Design window appears_____ by default.	Auto-Hidden	Docked	Floating	Closed	Docked
18	_____ windows appears attached to the side, top or bottom of the work area or to some other window	Auto-Hidden	Docked	Floating	Closed	Docked
19	_____ can be distributed to other people/computer and do not require Visual Basic to run	Files	Forms	Projects	Components	Components

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
20	_____ is a programming structure that encapsulates data and functionality as a single unit.	Class	Object	Collection	methods	Object
21	_____window gives an overview of the solution we are working with and lists all the files in the project.	Solution Explorer	Properties window	Explorer	methods	Solution Explorer
22	_____window allows us to set properties for various objects at design time	Explorer	Solution Explorer	Properties window	tootlbox	Properties window
23	_____window as you can see in the image below displays the results of building and running applications	Command window	output window	Task window	both a&b	output window
24	When we type the period(dot) after the object name a small dropdown list containing all the properties and methods related to that object appears. This feature is called _	IntelliSense	OnlineHelp	QuickMenu	DropHelp	IntelliSense

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
25	A property that returns an object is called	Collection	subroutine	Object Oriented	Object Property	Object Property
26	____window displays all the tasks that VB .NET	Task window	output window	command window	Object Property	Task window
27	_____is nothing but a name given to a storage area that our programs can manipulate.	Variable	variable declaration	variable initialization	initialization	Variable
28	To declare a variable, use the _____statement followed by the variable's name, the As keyword, and its type,	Dim	integer	String	Dim as	Dim
29	The data type of the variable is defined by using the -------- clause	in	where	as	is	as

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
30	A composite data type is of ---------- types	3	4	5	2	2
31	Size of integer data type is ---------- bits	8	16	24	32	32
32	Which of the given data types used to represent integer numbers	int	character	byte	precision	int
33	------- is the operator used for string concatenation	Cat	Str	^	&	&
34	The order in which the operators in an expression are evaluated is known as ----------	operator precedence	operator overloading	associatively of operators	operator evaluation	operator precedence

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
35	And , Or , Not, Xor are called _____ operators	Boolean	Relational	comparision	String	Boolean
36	In Select Case _____ Case is used to define codes that executes, if the expression does not evaluate to any of the Case statement	Default	Otherwise	Else	False	Else
37	This data type can be used for currency values	Currency	Dollar	Object	Decimal	Decimal
38	_____ function is used to retrieve only the month part of the date	DateDiff()	DatePart()	DateInterval()	Date.Month()	DatePart()
39	Which function returns the system's current date and time	DateTime.Now	DateTime.Today	DateTime.System	DateTime.Current	DateTime.Now

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
40	What statement is used to close a loop started with For statement?	Close	End For	Loop	Next	Next
41	What statement is used to terminate a Do..Loop without evaluating the test expression?	End Do	Loop	Exit	Exit Do	Exit Do
42	--------- method is create a new String object with the same content	CopyTo()	Copy()	Format()	Compare()	Copy()
43	The ----- function returns an array of String containing the substrings delimited by the given System.Char array.	Length()	Length()	Split()	Format()	Split()
44	The ---------- function remeove an item from a specified position	Add	Insert()	RemoveAt	Remove	Remove

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
45	The String data type comes from the ------- class	System.String	System	System.Forms	System.Array	System.String
46	The String is ----------	locatable	mutable	immutable	notable	immutable
47	The --------- function in String Class will insert a String in a specified index in the String instance.	Length()	Insert()	Length()	Format()	Insert()
48	Which of the following when turned on do not allow to use any variable without proper declaration?	Option Restrict	Option Explicit	Option Implicit	Option All	Option Explicit
49	Which of the following methods can be used to add items to an ArrayList class?	Insert method	collection method	top method	Add method	Add method

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
50	Parameters to methods in VB.NET are declared by default as ---------	ByVal	ByRef	Val	Ref	ByVal
51	Which of the following does not denote a arithmetic operator allowed in VB.Net?	Mod	/	*	~	~
52	Which of the following denote the method used for compatible type conversions?	TypeCov()	Type()	CTyp()	CType()	CType()
53	Which of the following does not denote a data type in VB.Net?	Boolean	Float	Decimal	Byte	Float
54	The format used for Date is ----------	{0:D}	{0:T}	{0:DD}	{0:Dy}	{0:D}

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
55	The format used for Time is --------	{0:D}	{0:T}	{0:TT}	{0:TTY}	{0:T}
56	_____ is an alternative to If…Then….Else.	select…case	case…select	select	case	select…case
57	Do Loop While Statement executes a set of statements and checks the condition, this is repeated until the condition is true. .It is also known as ____	Exit control	Entry control	control	entry exit control	Exit control
58	_____ is the value range of integer	-32767 to 32768	-32768 to 32767	32767 to -32768	32768 to -32767	-32768 to 32767
59	_____ are used for storing values temporarily.	character	constant	variable	module	variable

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
60	_____ is the value range of byte	0 to 255	1 to 255	0 to 266	1 to 266	0 to 255

UNIT II

SYLLABUS

Writing and Using Procedures: Module Coding – Arguments. Working with Forms: Appearance of Forms- Loading and Showing Forms -Designing Menus. Multiple Document Interface

WRITING AND USING PROCEDURE

Procedures are also used for implementing repeated tasks, such as frequently used calculations. The two types of procedures supported by Visual Basic-***subroutines*** and ***functions***

MODULAR CODING

The idea of breaking a large application into smaller, more manageable sections is not new to computing. Few tasks, programming or otherwise, can be managed as a whole. The event handlers are just one example of breaking a large application into smaller tasks. Some event handlers may require a lot of code.

Subroutines

A subroutine is a block of statements that carries out a well-defined task. The block of statements is placed within a set of Sub. . .End Sub statements and can be invoked by name.

The following subroutine displays the current date in a message box and can be called by its name, ShowDate():

```
Sub ShowDate()
MsgBox(Now().ToShortDateString)
End Sub
```

Most procedures also accept and act upon arguments. The ShowDate() subroutine displays the current date in a message box. If you want to display any other date, you have to implement it differently and add an argument to the subroutine:

```
Sub ShowDate(ByVal birthDate As Date)
MsgBox(birthDate.ToShortDateString)
End Sub
```

birthDate is a variable that holds the date to be displayed; its type is Date. The ByVal keyword means that the subroutine sees a copy of the variable, not the variable itself. What this means practically is that the subroutine can't change the value of the variable passed by the calling application. To display the current date in a message box, you must call the ShowDate() subroutine as follows from within your program:

ShowDate() -To display any other date with the second implementation of the subroutine, use a statement like the following:

```
Dim myBirthDate = #2/9/1960#
ShowDate(myBirthDate)
```

Or, you can pass the value to be displayed directly without the use of an intermediate variable: ShowDate(#2/9/1960#)

Functions

A function is similar to a subroutine, but a function returns a result. Because they return values, functions — like variables — have types. The value you pass back to the calling program from a function is called the return value, and its type must match the type of the function. Functions accept arguments, just like subroutines. The statements that make up a function are placed in a set of Function. . .End Function statement

A procedure is a group of statements that together perform a task, when called. After the procedure is executed, the control returns to the statement calling the procedure. VB.Net has two types of procedures:

- ✓ Functions
- ✓ Sub procedures or Subs

Functions return a value, where Subs do not return a value.

Defining a Function

The Function statement is used to declare the name, parameter and the body of a function. The syntax for the Function statement is:

```
[Modifiers] Function FunctionName [(ParameterList)] As ReturnType
   [Statements]
End Function
```

Where,

- ✓ ***Modifiers***: specifiy the access level of the function; possible values are: Public, Private, Protected, Friend, Protected Friend and information regarding overloading, overriding, sharing, and shadowing.
- ✓ ***FunctionName***: indicates the name of the function
- ✓ ***ParameterList***: specifies the list of the parameters
- ✓ ***ReturnType***: specifies the data type of the variable the function returns

Example

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two.

```
Function FindMax(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
  ' local variable declaration */
  Dim result As Integer
  If (num1 > num2) Then
     result = num1
  Else
     result = num2
  End If
  FindMax = result
End Function
```

Function Returning a Value

In VB.Net a function can return a value to the calling code in two ways:

- ✓ By using the return statement
- ✓ By assigning the value to the function name

The following example demonstrates using the *FindMax* function:

```
Module myfunctions
  Function FindMax(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
    ' local variable declaration */
    Dim result As Integer
    If (num1 > num2) Then
       result = num1
    Else
       result = num2
    End If
    FindMax = result
  End Function
  Sub Main()
    Dim a As Integer = 100
    Dim b As Integer = 200
    Dim res As Integer
    res = FindMax(a, b)
    Console.WriteLine("Max value is : {0}", res)
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
Max value is : 200
```

More Types of Function Return Values

1) Functions returning Structures

Suppose you need a function that returns a customer's savings and checking account balances. So far, you've learned that you can return two or more values from a function by supplying arguments with the ByRef keyword. A more elegant method is to create a custom data type (a structure) and write a function that returns a variable of this type.

Here's a simple example of a function that returns a custom data type. This example outlines the steps you must repeat every time you want to create functions that return custom data types:

1. Create a new project and insert the declarations of a custom data type in the declarations section of the form:

```
Structure CustBalance
    Dim SavingsBalance As Decimal
    Dim CheckingBalance As Decimal
End Structure
```

2. Implement the function that returns a value of the custom type. In the function's body, you must declare a variable of the type returned by the function and assign the proper values to its fields. The following function assigns random values to the fields CheckingBalance and SavingsBalance. Then assign the variable to the function's name, as shown next:

```
Function GetCustBalance(ID As Long) As CustBalance
    Dim tBalance As CustBalance
    tBalance.CheckingBalance = CDec(1000 + 4000 * rnd())
    tBalance.SavingsBalance = CDec(1000 + 15000 * rnd())
    Return(tBalance)
End Function
```

3. Place a button on the form from which you want to call the function. Declare a variable of the same type and assign to it the function's return value. The example that follows prints the savings and checking balances in the Output window:

```
Private Sub Button1 Click(...) Handles Button1.Click
```

```
Dim balance As CustBalance
balance = GetCustBalance(1)
Debug.WriteLine(balance.CheckingBalance)
Debug.WriteLine(balance.SavingsBalance)
End Sub
```

The code shown in this section belongs to the Structures sample project. Create this project from scratch, perhaps by using your own custom data type, to explore its structure and experiment with functions that return custom data types.

2) Function Returning Arrays

In addition to returning custom data types, VB 2008 functions can also return arrays. This is an interesting possibility that allows you to write functions that return not only multiple values, but also an unknown number of values.

In this section, we'll write the Statistics() function, similar to the CalculateStatistics() function you saw a little earlier in this chapter. The Statistics() function returns the statistics in an array. Moreover, it returns not only the average and the standard deviation, but the minimum and maximum values in the data set as well. One way to declare a function that calculates all the statistics is as follows:

```
Function Statistics(ByRef DataArray() As Double) As Double()
```

This function accepts an array with the data values and returns an array of Doubles. To implement a function that returns an array, you must do the following:

1. Specify a type for the function's return value and add a pair of parentheses after the type's name. Don't specify the dimensions of the array to be returned here; the array will be declared formally in the function.
2. In the function's code, declare an array of the same type and specify its dimensions. If the function should return four values, use a declaration like this one:

```
Dim Results(3) As Double
```

The Results array, which will be used to store the results, must be of the same type as the function— its name can be anything.

3. To return the Results array, simply use it as an argument to the Return statement:

   Return(Results)

4. In the calling procedure, you must declare an array of the same type without dimensions:

   Dim Statistics() As Double

5. Finally, you must call the function and assign its return value to this array:

   Stats() = Statistics(DataSet())

Here, DataSet is an array with the values whose basic statistics will be calculated by the Statistics() function. Your code can then retrieve each element of the array with an index value as usual.

ARGUMENTS

Subroutines and functions aren't entirely isolated from the rest of the application. Most procedures accept arguments from the calling program. Recall that an argument is a value you pass to the procedure and on which the procedure usually acts. This is how subroutines and functions communicate with the rest of the application.

Subroutines and functions may accept any number of arguments, and you must supply a value for each argument of the procedure when you call it. Some of the arguments may be optional, which means you can omit them; you will see shortly how to handle optional arguments.

The custom function Min(), for instance, accepts two numbers and returns the smaller one:

```
Function Min(ByVal a As Single, ByVal b As Single) As Single
Min = IIf(a < b, a, b)
End Function
```

IIf() is a built-in function that evaluates the first argument, which is a logical expression. If the expression is True, the IIf() function returns the second argument. If the expression is False, the function returns the third argument.

To call the Min() custom function, use a few statements like the following:

```
Dim val1 As Single = 33.001
Dim val2 As Single = 33.0011
```

```
Dim smallerVal as Single
smallerVal = Min(val1, val2)
Debug.Write("The smaller value is " & smallerVal)
```

If you execute these statements (place them in a button's Click event handler), you will see the following in the Immediate window:

```
The smaller value is 33.001
```

If you attempt to call the same function with two Double values, with a statement like the following, you will see the value 3.33 in the Immediate window:

```
Debug.WriteLine(Min(3.33000000111, 3.33000000222))
```

The compiler converted the two values from Double to Single data type and returned one of them.

Interesting things will happen if you attempt to use the Min() function with the Strict option turned on. Insert the statement Option Strict On at the very beginning of the file, or set Option Strict to On in the Compile tab of the project's Properties pages. The editor will underline the statement that implements the Min() function: the IIf() function. The IIf() function accepts two Object variables as arguments, and returns one of them as its result. The Strict option prevents the compiler from converting an Object to a numeric variable. To use the IIf() function with the Strict option, you must change its implementation as follows:

```
Function Min(ByVal a As Object, ByVal b As Object) As Object
Min = IIf(Val(a) < Val(b), a, b)
End Function
```

Argument Passing Mechanisms

One of the most important topics in implementing your own procedures is the mechanism used to pass arguments. The examples so far have used the default mechanism: passing arguments by value. The other mechanism is passing them by reference. Although most programmers use the default mechanism, it's important to know the difference between the two mechanisms and when to use each.

- ✓ Passing arguments By Value
- ✓ Passing arguments by Reference

- ✓ Returning Multiple Values
- ✓ Passing Objects as Arguments

Passing arguments by value

This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter. The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument.

VB.Net, you declare the reference parameters using the **ByVal** keyword. The following example demonstrates the concept:

```
Module paramByval
  Sub swap(ByVal x As Integer, ByVal y As Integer)
    Dim temp As Integer
    temp = x ' save the value of x
    x = y    ' put y into x
    y = temp 'put temp into y
  End Sub
  Sub Main()
    ' local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    Console.WriteLine("Before swap, value of a : {0}", a)
    Console.WriteLine("Before swap, value of b : {0}", b)
    ' calling a function to swap the values '
    swap(a, b)
    Console.WriteLine("After swap, value of a : {0}", a)
    Console.WriteLine("After swap, value of b : {0}", b)
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

Passing Parameters by Reference

A reference parameter is a reference to a memory location of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

In VB.Net, you declare the reference parameters using the **ByRef** keyword. The following example demonstrates this:

```
Module paramByref
  Sub swap(ByRef x As Integer, ByRef y As Integer)
    Dim temp As Integer
    temp = x ' save the value of x
    x = y    ' put y into x
    y = temp 'put temp into y
  End Sub
  Sub Main()
    ' local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    Console.WriteLine("Before swap, value of a : {0}", a)
    Console.WriteLine("Before swap, value of b : {0}", b)
```

```
    ' calling a function to swap the values '
    swap(a, b)
    Console.WriteLine("After swap, value of a : {0}", a)
    Console.WriteLine("After swap, value of b : {0}", b)
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Returning Multiple Values

If you want to write a function that returns more than a single result, you will most likely pass additional arguments by reference and set their values from within the function's code. The CalculateStatistics() function, calculates the basic statistics of a data set. The values of the data set are stored in an array, which is passed to the function by reference. The CalculateStatistics() function must return two values: the average and standard deviation of the data set. Here's the declaration of the CalculateStatistics() function:

```
Function CalculateStatistics(ByRef Data() As Double, ByRef Avg As Double, ByRef StDev As
Double) As Integer
```

The function returns an integer, which is the number of values in the data set. The two important values calculated by the function are returned in the Avg and StDev arguments:

```
Function CalculateStatistics(ByRef Data() As Double, ByRef Avg As Double, ByRef StDev As
Double) As Integer
Dim i As Integer, sum As Double, sumSqr As Double, points As Integer
points = Data.Length
For i = 0 To points - 1
```

```
sum = sum + Data(i)
sumSqr = sumSqr + Data(i) ^ 2
Next
Avg = sum / points
StDev = System.Math.Sqrt(sumSqr / points - Avg ^ 2)
Return(points)
End Function
```

To call the CalculateStatistics() function from within your code, set up an array of Doubles and declare two variables that will hold the average and standard deviation of the data set:

```
Dim Values(99) As Double
' Statements to populate the data set
Dim average, deviation As Double
Dim points As Integer
points = Stats(Values, average, deviation)
Debug.WriteLine points & " values processed."
Debug.WriteLine "The average is " & average & " and"
Debug.WriteLine "the standard deviation is " & deviation
```

Using ByRef arguments is the simplest method for a function to return multiple values. However, the definition of your functions might become cluttered, especially if youwant to returnmore than a few values. Another problem with this technique is that it's not clear whether an argument must be set before calling the function. As you will see shortly, it is possible for a function to return an array or a custom structure with fields for any number of values.

Passing Objects as Arguments

When you pass objects as arguments, they're passed by reference, even if you have specified the ByVal keyword. The procedure can access and modify the members of the object passed as an argument, and the new value will be visible in the procedure that made the call.

The following code segment demonstrates this. The object is an ArrayList, which is an enhanced form of an array. The ArrayList is discussed in detail later in the tutorial, but to follow this example all you need to know is that the Add method adds new items to the ArrayList, and

you can access individual items with an index value, similar to an array's elements. In the Click event handler of a Button control, create a new instance of the ArrayList object and call the PopulateList() subroutine to populate the list. Even if the ArrayList object is passed to the subroutine by value, the subroutine has access to its items:

```
Private Sub Button1 Click(ByVal sender As System.Object, ByVal e As system.EventArgs)
Handles Button1.Click
Dim aList As New ArrayList()
PopulateList(aList)
Debug.WriteLine(aList(0).ToString)
Debug.WriteLine(aList(1).ToString)
Debug.WriteLine(aList(2).ToString)
End Sub

Sub PopulateList(ByVal list As ArrayList)
list.Add("1")
list.Add("2")
list.Add("3")
End Sub
```

The same is true for arrays and all other collections. Even if you specify the ByVal keyword, they're passed by reference.

Passing unknown number of Arguments

VB 2008 supports the ParamArray keyword, which allows you to pass a variable number of arguments to a procedure.

Let's look at an example. Suppose that you want to populate a ListBox control with elements. To add an item to the ListBox control, you call the Add method of its Items collection as follows:

```
ListBox1.Items.Add("new item")
```

This statement adds the string new item to the ListBox1 control. If you frequently add multiple items to a ListBox control from within your code, you can write a subroutine that performs this task. The following subroutine adds a variable number of arguments to the ListBox1 control:

```
Sub AddNamesToList(ByVal ParamArray NamesArray() As Object)
Dim x As Object
For Each x In NamesArray
ListBox1.Items.Add(x)
Next x
End Sub
```

This subroutine's argument is an array prefixed with the keyword ParamArray, which holds all the parameters passed to the subroutine. If the parameter array holds items of the same type, you can declare the array to be of the specific type (string, integer, and so on). To add items to the list, call the AddNamesToList() subroutine as follows:

```
AddNamesToList("Robert", "Manny", "Renee", "Charles", "Madonna")
```

If you want to know the number of arguments actually passed to the procedure, use the Length property of the parameter array. The number of arguments passed to the AddNamesToList() subroutine is given by the following expression:

```
NamesArray.Length
```

The following loop goes through all the elements of the NamesArray and adds them to the list:

```
Dim i As Integer
For i = 0 to NamesArray.GetUpperBound(0)
ListBox1.Items.Add(NamesArray(i))
Next i
```

VB arrays are zero-based (the index of the first item is 0), and the GetUpperBound method returns the index of the last item in the array.

A procedure that accepts multiple arguments relies on the order of the arguments. To omit some of the arguments, you must use the corresponding comma. Let's say you want to call such a

procedure and specify the first, third, and fourth arguments. The procedure must be called as follows:

ProcName(arg1, , arg3, arg4)

The arguments to similar procedures are usually of equal stature, and their order doesn't make any difference. A function that calculates the mean or other basic statistics of a set of numbers, or a subroutine that populates a ListBox or ComboBox control, are prime candidates for implementing this technique. If the procedure accepts a variable number of arguments that aren't equal in stature, you should consider the technique described in the following section. If the function accepts a parameter array, this must the last argument in the list, and none of the other parameters can be optional.

Param Arrays

At times, while declaring a function or sub procedure you are not sure of the number of arguments passed as a parameter. VB.Net param arrays (or parameter arrays) come into help at these times.

The following example demonstrates this:

```
Module myparamfunc
  Function AddElements(ParamArray arr As Integer()) As Integer
    Dim sum As Integer = 0
    Dim i As Integer = 0
    For Each i In arr
      sum += i
    Next i
    Return sum
  End Function
  Sub Main()
    Dim sum As Integer
    sum = AddElements(512, 720, 250, 567, 889)
    Console.WriteLine("The sum is: {0}", sum)
    Console.ReadLine()
```

```
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
The sum is: 2938
```

Named Arguments

The main limitation of the argument-passing mechanism, though, is the order of the arguments. By default, Visual Basic matches the values passed to a procedure to the declared arguments by their order.

This limitation is lifted by Visual Basic's capability to specify named arguments. With named arguments, you can supply arguments in any order because they are recognized by name and not by their order in the list of the procedure's arguments. Suppose you've written a function that expects three arguments: a name, an address, and an email address:

Function Contact(Name As String, Address As String, EMail As String)

When calling this function, you must supply three strings that correspond to the arguments Name, Address, and EMail, in that order. However, there's a safer way to call this function: Supply the arguments in any order by their names. Instead of calling the Contact() function as follows:

Contact("Peter Evans", "2020 Palm Ave., Santa Barbara, CA 90000", _
"PeterEvans@example.com")

you can call it this way:

Contact(Address:="2020 Palm Ave., Santa Barbara, CA 90000", _
EMail:="PeterEvans@example.com", Name:="Peter Evans")

The := operator assigns values to the named arguments. Because the arguments are passed by name, you can supply them in any order.

To test this technique, enter the following function declaration in a form's code:

Function Contact(ByVal Name As String, ByVal Address As String, _ByVal EMail As String) As String
Debug.WriteLine(Name)
Debug.WriteLine(Address)

```
Debug.WriteLine(EMail)
Return ("OK")
End Function
```

Then call the Contact() function from within a button's Click event with the following statement:

```
Debug.WriteLine( Contact(Address:="2020 Palm Ave., Santa Barbara, CA 90000", _
Name:="Peter Evans", EMail:="PeterEvans@example.com"))
```

You'll see the following in the Immediate window:

```
Peter Evans
2020 Palm Ave., Santa Barbara, CA 90000
PeterEvans@example.com
OK
```

The function knows which value corresponds to which argument and can process them the same way that it processes positional arguments. Notice that the function's definition is the same, whether you call it with positional or named arguments. The difference is in how you call the function and not how you declare it.

Named arguments make code safer and easier to read, but because they require a lot of typing, most programmers don't use them. Besides, when IntelliSense is on, you can see the definition of the function as you enter the arguments, and this minimizes the chances of swapping two values by mistake.

Named Visual Basic Arguments

Some obvious ways to write readable code include the use of program comments in your code -- no matter what the language you are using to develop your program, all major languages provide for comments. Something else that can make your Visual Basic more readable is the use of Named Arguments.

This is illustrated by executing the Visual Basic MsgBox Function to display a Windows Message Box. The Visual Basic MsgBox function has one required argument (Prompt), and four optional arguments (Buttons, Title, HelpFile and Context).

MsgBox "I love Visual Basic"

By default, this code will display a Message Box with a single command button captioned OK, with the text "I love Visual Basic", and the Visual Basic Project name displayed in the Title Bar of the Message Box.

Suppose I'm not happy with the default Title in the Message Box, and I decide I want to customize it. Doing this is easy-all I need to do is supply the Title argument to the MsgBox function. However, since Title is the third argument, I either need to supply the second argument -- Buttons, which is by default presumed to be the value vbOKOnly -- or provide a 'comma placeholder', like this.

MsgBox "I love Visual Basic",, "SearchVB.Com"

Notice the two commas back-to-back, with no value in-between. This is the 'comma placeholder' and is how we tell VB that although we have a value for the third argument, we have no explicit value for the second argument.

When we execute this code, we'll see a Message Box that reads "I love Visual Basic", and that has "SearchVB.Com" for its Title Bar.

Named Arguments can make passing optional arguments easier-and make your code infinitely easier to read and modify. For instance, the code we wrote above can be re-written the following way using Named Arguments.

MsgBox Prompt:="I love Visual Basic", Title:="SearchVB.Com"

With Named Arguments, we specify the name of the argument, followed by a colon and equals sign (:=), then the value for the argument. By using Named Arguments, we don't need to

provide a 'comma placeholder' for the second argument Buttons. Since we are naming the argument, VB knows that 'SearchVB.Com' is the value for the Optional Argument 'Title'. And since we name the arguments, being able to read and understand the code in the future is much easier.

Overloading Functions

Function overloading, means that you can have multiple implementations of the same function, each with a different set of arguments and possibly a different return value. Yet all overloaded functions share the same name.

The Next method of the System.Random class returns an integer value from –2,147,483,648 to 2,147,483,647. (This is the range of values that can be represented by the Integer data type.) We should also be able to generate random numbers in a limited range of integer values. To emulate the throw of a die, we want a random value in the range from 1 to 6, whereas for a roulette game we want an integer random value in the range from 0 to 36. You can specify an upper limit for the random number with an optional integer argument. The following statement will return a random integer in the range from 0 to 99:

```
randomInt = rnd.Next(100)
```

You can also specify both the lower and upper limits of the random number's range. The following statement will return a random integer in the range from 1,000 to 1,999:

```
randomInt = rnd.Next(1000, 2000)
```

The same method behaves differently based on the arguments we supply. The behavior of the method depends either on the type of the arguments, the number of the arguments, or both. As you will see, there's no single function that alters its behavior based on its arguments. There are as many different implementations of the same function as there are argument combinations. All the functions share the same name, so they appear to the user as a single multifaceted function. These functions are overloaded, and you'll see how they're implemented in the following section.

Let's return to the Min() function we implemented earlier in this chapter. The initial implementation of the Min() function is shown next:

```
Function Min(ByVal a As Double, ByVal b As Double) As Double
Min = IIf(a < b, a, b)
End Function
```

To write a Min() function that can handle both numeric and string values, you must, in essence, write two Min() functions. All Min() functions must be prefixed with the Overloads keyword. The following statements show two different implementations of the same function:

```
Overloads Function Min(ByVal a As Double, ByVal b As Double) As Double
Min = Convert.ToDouble(IIf(a < b, a, b))
End Function
Overloads Function Min(ByVal a As String, ByVal b As String) As String
Min = Convert.ToString(IIf(a < b, a, b))
End Function
```

We need a third overloaded form of the same function to compare dates. If you call the Min() function, passing as an argument two dates, as in the following statement, the Min() function will compare them as strings and return (incorrectly) the first date.

```
Debug.WriteLine(Min(#1/1/2009#, #3/4/2008#))
```

This statement is not even valid when the Strict option is on, so you clearly need another overloaded form of the function that accepts two dates as arguments, as shown here:

```
Overloads Function Min(ByVal a As Date, ByVal b As Date) As Date
Min = IIf(a < b, a, b)
End Function
```

If you now call the Min() function with the dates #1/1/2009# and #3/4/2008#, the function will return the second date, which is chronologically smaller than the first.

Event-Handler Arguments

Events are basically a user action like key press, clicks, mouse movements etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur.

Clicking on a button, or entering some text in a text box, or clicking on a menu item all are examples of events. An event is an action that calls a function or may cause another event.

Event handlers are functions that tell how to respond to an event.

VB.Net is an event-driven language. There are mainly two types of events:

- ✓ Mouse events
- ✓ Keyboard events

Handling Mouse Events

Mouse events occur with mouse movements in forms and controls. Following are the various mouse events related with a Control class:

- ✓ **MouseDown** - it occurs when a mouse button is pressed
- ✓ **MouseEnter** - it occurs when the mouse pointer enters the control
- ✓ **MouseHover** - it occurs when the mouse pointer hovers over the control
- ✓ **MouseLeave** - it occurs when the mouse pointer leaves the control
- ✓ **MouseMove** - it occurs when the mouse pointer moves over the control
- ✓ **MouseUp** - it occurs when the mouse pointer is over the control and the mouse button is released
- ✓ **MouseWheel** - it occurs when the mouse wheel moves and the control has focus

The event handlers of the mouse events get an argument of type **MouseEventArgs**.

The MouseEventArgs object is used for handling mouse events. It has the following properties:

- ✓ **Buttons** - indicates the mouse button pressed
- ✓ **Clicks** - indicates the number of clicks
- ✓ **Delta** - indicates the number of detents the mouse wheel rotated
- ✓ **X** - indicates the x-coordinate of mouse click
- ✓ **Y** - indicates the y-coordinate of mouse click

Handling Keyboard Events

Following are the various keyboard events related with a Control class:

- ✓ **KeyDown** - occurs when a key is pressed down and the control has focus
- ✓ **KeyPress** - occurs when a key is pressed and the control has focus
- ✓ **KeyUp** - occurs when a key is released while the control has focus

✓

The event handlers of the KeyDown and KeyUp events get an argument of type **KeyEventArgs**. This object has the following properties:

- ✓ **Alt** - it indicates whether the ALT key is pressed/p>
- ✓ **Control** - it indicates whether the CTRL key is pressed
- ✓ **Handled** - it indicates whether the event is handled
- ✓ **KeyCode** - stores the keyboard code for the event
- ✓ **KeyData** - stores the keyboard data for the event
- ✓ **KeyValue** - stores the keyboard value for the event
- ✓ **Modifiers** - it indicates which modifier keys (Ctrl, Shift, and/or Alt) are pressed
- ✓ **Shift** - it indicates if the Shift key is pressed

The event handlers of the KeyDown and KeyUp events get an argument of type **KeyEventArgs**. This object has the following properties:

- ✓ **Handled** - indicates if the KeyPress event is handled
- ✓ **KeyChar** - stores the character corresponding to the key pressed

WORKING WITH FORMS

In Visual Basic, the *form* is the container for all the controls that make up the user interface. When a Visual Basic application is executing, each window it displays on the desktop is a form. In previous chapters, we concentrated on placing the elements of the user interface on forms, setting their properties, and adding code behind selected events. Now, we'll look at forms themselves and at a few related topics, such as menus (forms are the only objects that can have menus attached), how to design forms that can be automatically resized, and how to access the controls of one form from within another form's code. The form is the top-level object in a Visual Basic application, and every application starts with the form.

The forms that constitute the visible interface of your application are called *Windows forms;* this term includes both the regular forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a specific piece of data or to display critical information. A *dialog box* is a form with a small number of controls, no menus, and usually an OK and a Cancel button to close it. These are also called Modal Forms and the regular forms are non-Modal.

APPEARANCE OF FORMS

Applications are made up of one or more forms (usually more than one), and the forms are what users see. You should craft your forms carefully, make them functional, and keep them simple and intuitive. You already know how to place controls on the form, but there's more to designing forms than populating them with controls. The main characteristic of a form is the title bar on which the form's caption is displayed.

Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands shown in Table 2.1 On the right end of the title bar are three buttons: Minimize, Maximize, and Close. Clicking these buttons performs the associated function. When a form is maximized, the Maximize button is replaced by the Restore button. When clicked, this button resets the form to the size and position before it was maximized. The Restore button is then replaced by the Maximize button

Commands of the Control Menu of the Form

Command	Effect
Restore	Restores a maximized form to the size it was before it was maximized;

	available only if the form has been maximized.
Move	Lets the user move the form around with the arrow keys.
Size	Lets the user resize the form with the arrow keys.
Minimize	Minimizes the form.
Maximize	Maximizes the form.
Close	Closes the current form

Properties of the Form Object

You're familiar with the appearance of forms, even if you haven't programmed in the Windows environment in the past; you have seen nearly all types of windows in the applications you're using every day. The floating toolbars used by many graphics applications, for example, are actually forms with a narrow title bar. The dialog boxes that display critical information or prompt you to select the file to be opened are also forms. You can duplicate the look of any window or dialog box through the following properties of the Form object.

AcceptButton, CancelButton

These two properties let you specify the default Accept and Cancel buttons. The Accept button is the one that's automatically activated when you press Enter, no matter which control has the focus at the time, and is usually the button with the OK caption. Likewise, the Cancel button is the one that's automatically activated when you hit the Esc key and is usually the button with the Cancel caption. To specify the Accept and Cancel buttons on a form, locate the AcceptButton and CancelButton properties of the form and select the corresponding controls from a drop-down list, which contains the names of all the buttons on the form. For more information on these two properties, see the section "Forms versus Dialog Boxes in VB.NET," later in this chapter.

AutoScaleMode

This property determines how the control is scaled, and its value is a member of the AutoScale-Mode enumeration: None (automatic scaling is disabled), Font (the controls on the form are scaled relative to the size of their font), Dpi, which stands for dots per inch (the controls on the form are scaled relative to the display resolution), and Inherit (the controls are scaled

according to the AutoScaleMode property of their parent class). The default value is Font; if you change the form's font size, the controls on it are scaled to the new font size.

AutoScroll

The **AutoScroll property** is a True/False value that indicates whether scroll bars will be automatically attached to the form if the form is resized to a point that not all its controls are visible. Use this property to design large forms without having to worry about the resolution of the monitor on which they'll be displayed. The AutoScroll property is used in conjunction with two other properties, **AutoScrollMargin** and **AutoScrollMinSize**. Note that the AutoScroll property applies to a few controls as well, including the Panel and SplitContainer controls. For example, you can create a form with a fixed and a scrolling pane by placing two Panel controls on it and setting the AutoScroll property of one of them (the Panel you want to scroll) to True.

AutoScrollPosition

This property is available from within your code only (you can't set this property at design time), and it indicates the number of pixels that the form was scrolled up or down. Its initial value is zero, and it assumes a value when the user scrolls the form (provided that the form's AutoScroll property is True). Use this property to find out the visible controls from within your code, or scroll the form programmatically to bring a specific control into view.

AutoScrollMargin

This is a margin, expressed in pixels, that's added around all the controls on the form. If the form is smaller than the rectangle that encloses all the controls adjusted by the margin, the appropriate scroll bar(s) will be displayed automatically.

AutoScrollMinSize

This property lets you specify the minimum size of the form before the scroll bars are attached. If your form contains graphics that you want to be visible at all times, set the Width and Height members of the AutoScrollMinSize property to the dimensions of the graphics. (Of course, the graphics won't be visible at all times, but the scroll bars indicate that there's more to the form than can fit in the current window.) Notice that this isn't the form's minimum size; users can make the form even smaller. To specify a minimum size for the form, use the MinimumSize property, described later in this section.

FormBorderStyle

The FormBorderStyle property determines the style of the form's border; its value is one of the FormBorderStyle enumeration's members, which are shown in Table 2.3. You can make the form's title bar disappear altogether by setting the form's FormBorderStyle property to FixedToolWindow, the ControlBox property to False, and the Text property (the form's caption) to an empty string

Tabel 2.3 - The FormBorderStyle Enumeration

Value	Effect
None	A borderless window that can't be resized. This setting is rarely used.
Sizable	(default) A resizable window that's used for displaying regular forms.
Fixed3D	A window with a fixed visible border, ‘‘raised" relative to the main area. Unlike the None setting, this setting allows users to minimize and close the window.
FixedDialog	A fixed window used to implement dialog boxes.
FixedSingle	A fixed window with a single-line border.
FixedToolWindow	A fixed window with a Close button only. It looks like a toolbar displayed by drawing and imaging applications.
SizableToolWindow	Same as the FixedToolWindow, but is resizable. In addition, its caption font is smaller than the usual.

ControlBox

This property is also True by default. Set it to False to hide the control box icon and disable the Control menu. Although the Control menu is rarely used, Windows applications don't disable it. When the ControlBox property is False, the three buttons on the title bar are also disabled. If you set the Text property to an empty string, the title bar disappears altogether.

MinimizeBox, MaximizeBox

These two properties, which specify whether the Minimize and Maximize buttons will appear on the form's title bar, are True by default. Set them to False to hide the corresponding buttons on the form's title bar.

MinimumSize, MaximumSize

These two properties read or set the minimum and maximum size of a form. When users resize the form at runtime, the form won't become any smaller than the dimensions specified by the MinimumSize property and no larger than the dimensions specified by the MaximumSize property. The MinimumSize property is a Size object, and you can set it with a statement like the following:

```
Me.MinimumSize = New Size(400, 300)
```

Or you can set the width and height separately:

```
Me.MinimumSize.Width = 400
Me.MinimumSize.Height = 300
```

The MinimumSize.Height property includes the height of the form's title bar; you should take that into consideration. If the minimum usable size of the form is 400 × 300, use the following statement to set the MinimumSize property:

```
Me.MinimumSize = New Size(400, 300 + SystemInformation.CaptionHeight)
```

The default value of both properties is (0, 0), which means that no minimum or maximum size is imposed on the form, and the user can resize it as desired.

KeyPreview

This property enables the form to capture all keystrokes before they're passed to the control that has the focus. Normally, when you press a key, the KeyPress event of the control with the focus is triggered (as well as the KeyUp and KeyDown events), and you can handle the

keystroke from within the control's appropriate handler. In most cases, you let the control handle the keystroke and don't write any form code for that.

SizeGripStyle

This property gets or sets the style of the sizing handle to display in the bottom-right corner of the form. You can set it to a member of the SizeGripStyle enumeration: Auto (the size grip is displayed as needed), Show (the size grip is displayed at all times), or Hide (the size grip is not displayed, but users can still resize the form with the mouse).

StartPosition, Location

The StartPosition property, which determines the initial position of the form when it's first displayed, can be set to one of the members of the FormStartPosition enumeration: Center-Parent (the form is centered in the area of its parent form), CenterScreen (the form is centered on the monitor), Manual (the position of the form is determined by the Location property), WindowsDefaultLocation (the form is positioned at the Windows default location), and WindowsDefaultBound (the form's location and bounds are determined by Windows defaults). The Location property allows you to set the form's initial position at design time or to change the form's location at runtime.

TopMost

This property is a True/False value that lets you specify whether the form will remain on top of all other forms in your application. Its default property is False, and you should change it only on rare occasions. Some dialog boxes, such as the Find & Replace dialog box of any text-processing application, are always visible, even when they don't have the focus.

Size

Use the Size property to set the form's size at design time or at runtime. Normally, the form's width and height are controlled by the user at runtime. This property is usually set from within the form's Resize event handler to maintain a reasonable aspect ratio when the user resizes the form. The Form object also exposes the Width and Height properties for controlling its size.

Placing Controls on Forms

The first step in designing your application's interface is, of course, the analysis and careful planning of the basic operations you want to provide through your interface. The second

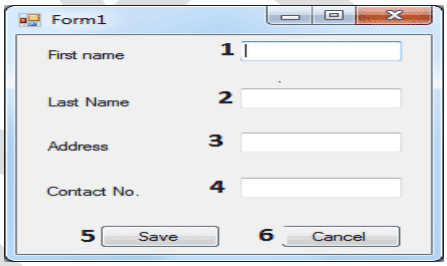
step is to design the forms. Designing a form means placing Windows controls on it, setting the controls' properties, and then writing code to handle the events of interest.

To place controls on your form, you select them in the Toolbox and then draw, on the form, the rectangle in which the control will be enclosed. Or you can double-click the control's icon to place an instance of the control on the form. All controls have a default size, and you can resize the control on the form by using the mouse.

Setting the TabIndex Property

Another important issue in form design is the tab order of the controls on the form. As you know, pressing the Tab key at runtime takes you to the next control on the form. The order of the controls is the order in which they were placed on the form, but this is never what we want. When you design the application, you can specify in which order the controls receive the focus (the tab order, as it is known) with the help of the TabIndex property. Each control has its own TabIndex setting, which is an integer value. When the Tab key is pressed, the focus is moved to the control whose tab order immediately follows the tab order of the current control. The values of the TabIndex properties of the various controls on the form need not be consecutive.

To specify the tab order of the various controls, you can set their TabIndex property in the Properties window or you can choose the Tab Order command from the View menu. The tab order of each control will be displayed on the corresponding control, as shown in Figure 5.3.



Setting the Tab order by using the TabIndex property of the form

To set the tab order of the controls, click each control in the order in which you want them to receive the focus. You must click all of them in the desired order, starting with the first control in the tab order. Each control's index in the tab order appears in the upper-left corner of the control. When you're finished, choose the Tab Order command from the View menu again to hide these numbers. As you place controls on the form, don't forget to lock them, so that you won't

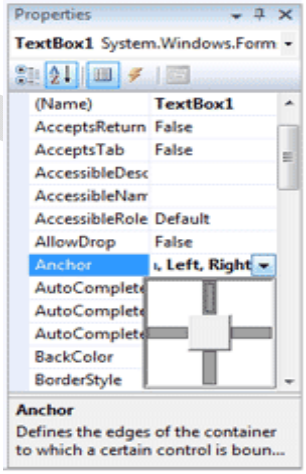
move them around by mistake as you work with other controls. You can lock the controls in their places either by setting each control's Locked property to True or by locking all the controls on the form at once via the Format > Lock Controls command.

Anchoring and Docking Controls

Anchoring Controls

The Anchor property lets you attach one or more edges of the control to corresponding edges of the form. The anchored edges of the control maintain the same distance from the corresponding edges of the form.

Place a TextBox control on a new form, set its MultiLine property to True, and then open the control's Anchor property in the Properties window. You will see a rectangle within a larger rectangle and four pegs that connect the small control to the sides of the larger box (see Figure 5.5). The large box is the form, and the small one is the control. The four pegs are the anchors, which can be either white or gray. The gray anchors denote a fixed distance between the control and the form. By default, the control is placed at a fixed distance from the top-left corner of the form. When the form is resized, the control retains its size and its distance from the top-left corner of the form.

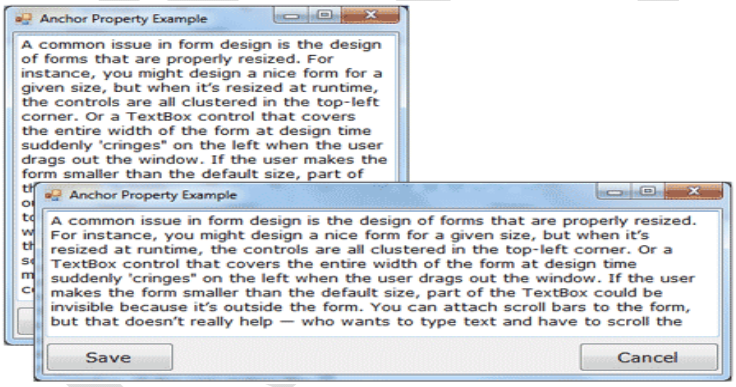


The settings of the Anchor property

We want our TextBox control to fill the width of the form, be aligned to the top of the form, and leave some space for a few buttons at the bottom. We also want our form to maintain this arrangement, regardless of its size. Make the TextBox control as wide as the form (allowing, perhaps, a margin of a few pixels on either side). Then place a couple of buttons at the bottom of the form and make the TextBox control tall enough that it stops above the buttons. This is the form of the Anchor property example project.

Now open the TextBox control's Anchor property and make all four anchors gray by clicking them. This action tells the Form Designer to resize the control accordingly at runtime, so that the distances between the sides of the control and the corresponding sides of the form are the same as those you set at design time. Select each button on the form and set their Anchor properties in the Properties window: Anchor the left button to the left and bottom of the form, and the right button to the right and bottom of the form.

Resize the form at design time without running the project, and you'll see that all the controls are resized and rearranged on the form at all times. Figure 5.6 shows the Anchor project's main form in two different sizes.



Use the Anchor property of the various controls to design forms that can be resized gracefully at runtime.

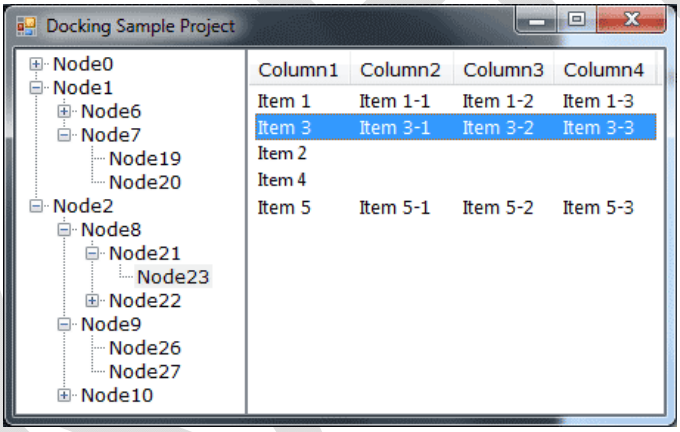
Yet, there's a small problem: If you make the form very narrow, there will be no room for both buttons across the form's width. The simplest way to fix this problem is to impose a minimum size for the form. To do so, you must first decide the form's minimum width and height and then set the MinimumSize property to these values. You can also use the AutoScroll properties, but it's not recommended that you add scroll bars to a small form like ours.

Docking Controls

In addition to the Anchor property, most controls provide the Dock property, which determines how a control will dock on the form. The default value of this property is None.

Create a new form, place a multiline TextBox control on it, and then open the control's Dock property. The various rectangular shapes are the settings of the property. If you click the middle rectangle, the control will be docked over the entire form: It will expand and shrink both horizontally and vertically to cover the entire form. This setting is appropriate for simple forms that contain a single control, usually a TextBox, and sometimes a menu. Try it out.

Let's create a more complicated form with two controls (see the Docking sample project). The form shown in Figure 5.7 contains a TreeView control on the left and a ListView control on the right. The two controls display folder and file data on an interface that's very similar to that of Windows Explorer. The TreeView control displays the directory structure, and the ListView control displays the selected folder's files.



Setting the Dock property of the controls to Fill so the form at runtime will be filled with controls even when it is re-sized

Place a TreeView control on the left side of the form and a ListView control on the right side of the form. Then dock the TreeView to the left and the ListView to the right. If you run the application now, as you resize the form, the two controls remain docked to the two sides of the form — but their sizes don't change. If you make the form wider, there will be a gap between the two controls. If you make the form narrower, one of the controls will overlap the other.

End the application, return to the Form Designer, select the ListView control, and set its Dock property to Fill. This time, the ListView will change size to take up all the space to the right of the TreeView. The ListView control will attempt to fill the form, but it won't take up the space of another control that has been docked already.

Form Events

The Form object triggers several events. The most important are Activated, Deactivate, Form-Closing, Resize, and Paint.

The Activated and Deactivate Events

When more than one form is displayed, the user can switch from one to the other by using the mouse or by pressing Alt+Tab. Each time a form is activated, the Activated event takes place. Likewise, when a form is activated, the previously active form receives the Deactivate event. Insert in these two event handlers the code you want to execute when a form is activated (set certain control properties, for example) and when a form loses the focus or is deactivated. These two events are the form's equivalents of the Enter and Leave events of the various controls. Notice an inconsistency in the names of the two events: the Activated event takes place after the form has been activated, whereas the Deactivate event takes place right before the form is deactivated.

The FormClosing and FormClosed Events

The FormClosing event is fired when the user closes the form by clicking its Close button. If the application must terminate because Windows is shutting down, the same event will be fired as well. Users don't always quit applications in an orderly manner, and a professional application should behave gracefully under all circumstances. The same code you execute in the application's Exit command must also be executed from within the closing event.

Listing: Cancelling the Closing of a Form

```
Public Sub Form1 FormClosing(...) Handles Me.FormClosing
Dim reply As MsgBoxResult
reply = MsgBox("Document has been edited. " &
"OK to terminate application, Cancel to " &
"return to your document.", MsgBoxStyle.OKCancel)
If reply = MsgBoxResult.Cancel Then
```

```
e.Cancel = True
End If
End Sub
```

The e argument of the FormClosing event provides the CloseReason property, which reports how the form is closing. Its value is one of the following members of the CloseReason enumeration: FormOwnerClosing, MdiFormClosing, None, TaskManagerClosing, WindowsShutDown. The names of the members are self-descriptive, and you can query the CloseReason property to determine how the window is closing.

The FormClosed event fires after the form has been closed. You can find out the action that caused the form to be closed through the e.CloseReason property, but it's too late to cancel the closing of the form.

The Resize, ResizeBegin, and ResizeEnd Events

The Resize event is fired every time the user resizes the form by using the mouse. With previous versions of VB, programmers had to insert quite a bit of code in the Resize event's handler to resize the controls and possibly rearrange them on the form.With the Anchor and Dock properties, much of this overhead can be passed to the form itself. If you want the two sides of the form to maintain a fixed ratio, however, you have to resize one of the dimensions from within the Resize event handler

```
Private Form1 Resize(...) Handles Me.Resize
Me.Width = (0.75 * Me.Height)
End Sub
```

The Resize event is fired continuously while the form is being resized. If youwant to keep track of the initial form's size and perform all the calculations after the user has finished resizing the form, you can use the ResizeBegin and ResizeEnd events, which are fired at the beginning and after the end of a resize operation, respectively. Store the form's width and height to two global variables in the ResizeBegin event and use these two variables in the ResizeEnd event handler.

The Scroll Event

The Scroll event is fired by forms that have their AutoScroll property set to True when the user scrolls the form. The second argument of the Scroll event handler exposes the OldValue and NewValue properties, which are the displacements of the form before and after the scroll operation. This event can be used to keep a specific control in view when the form's contents are scrolled.

The AutoScroll property is handy for large forms, but it has a serious drawback: It scrolls the entire form. In most cases, we want to keep certain controls in view at all times. Instead of a scrollable form, you can create forms with scrollable sections by exploiting the AutoScroll properties of the Panel and/or the SplitContainer controls. You can also reposition certain controls from within the form's Scroll event handler. Let's say you have placed a few controls on a Panel container and you want to keep this Panel at the top of a scrolling form. The following statements in the form's Scroll event handler reposition the Panel at the top of the form every time the user scrolls the form:

```
Private Sub Form1 Scroll(...) Handles Me.Scroll
Panel1.Top = Panel1.Top + (e.NewValue - e.OldValue)
End Sub
```

The Paint Event

This event takes place every time the form must be refreshed, and we use its handler to execute code for any custom drawing on the form. When you switch to another form that partially or totally overlaps the current one and then switch back to the first form, the Paint event will be fired to notify your application that it must redraw the form. The form will refresh its controls automatically, but any custom drawing on the form won't be refreshed automatically.

LOADING AND SHOWING FORMS

One of the operations you'll have to perform with multi-form applications is to load and manipulate forms from within other forms' code. For example, you may wish to display a second form to prompt the user for data specific to an application. You must explicitly load the second form, read the information entered by the user, and then close the form. Or, you may wish to

maintain two forms open at once and let the user switch between them.. To show Form2 when an action takes place on Form1, first declare a variable that references Form2:

```
Dim frm As New Form2
```

This declaration must appear in Form1 and must be placed outside any procedure. (If you place it in a procedure's code, then every time the procedure is executed, a new reference to Form2 will be created. This means that the user can display the same form multiple times.

Then, to invoke Form2 from within Form1, execute the following statement:

```
frm.Show
```

This statement will bring up Form2 and usually appears in a button's or menu item's Click event handler. At this point, the two forms don't communicate with one another. However, they're both on the desktop and you can switch between them. There's no mechanism to move information from Form2 back to Form1, and neither form can access the other's controls or variables. The Show method opens Form2 in a modaless manner. The two forms are equal in stature on the desktop, and the user can switch between them. You can also display the second form in a modal manner, which means that users won't be able to return to the form from which they invoked it.

While a modal form is open, it remains on top of the desktop and you can't move the focus to the any other form of the same application (but you can switch to another application). To open a modal form, use the statement

```
frm.ShowDialog
```

The modal form is, in effect, a dialog box, like the Open File dialog box. You must first select a file on this form and click the Open button, or click the Cancel button, to close the dialog box and return to the form from which the dialog box was invoked.

The Startup Form

A typical application has more than a single form. When an application starts, the main form is loaded. You can control which form is initially loaded by setting the startup object in the Project Properties window. To open this, right-click the project's name in the Solution Explorer

and select Properties. In the project's Property Pages, select the Startup Object from the drop-down list.

You can also start an application with a subroutine without loading a form. This subroutine must be called Main() and must be placed in a Module. Right-click the project's name in the Solution Explorer window and select the Add Item command. When the dialog box appears, select a Module. Name it StartUp (or anything you like; you can keep the default name Module1) and then insert the Main() subroutine in the module. The Main() subroutine usually contains initialization code and ends with a statement that displays one of the project's forms; to display the AuxiliaryForm object from within the Main() subroutine, use the following statements:

```
Module StartUpModule
  Sub Main()

  System.Windows.Forms.Application.Run(New _ AuxiliaryForm())
  End Sub
End Module
```

Then, you must open the Project Properties dialog box and specify that the project's startup object is the subroutine Main(). When you run the application, the form you specified in the Run method will be loaded.

Controlling One Form from within Another

Loading and displaying a form from within another form's code is fairly trivial. In some situations, this is all the interaction you need between forms. Each form is designed to operate independently of the others, but they can communicate via public variables (see, "Private & Public Variables"). In most situations, however, you need to control one form from within another's code. Controlling the form means accessing its controls and setting or reading values from within another form's code.

Example:

TextPad is a text editor that consists of the main form and an auxiliary form for the Find & Replace operation. All other operations on the text are performed with the commands of the

menu you see on the main form. When the user wants to search for and/or replace a string, the program displays another form on which they specify the text to find, the type of search, and so on. When the user clicks one of the Find & Replace form's buttons, the corresponding code must access the text on the main form of the application and search for a word or replace a string with another. The Find & Replace dialog box not only interacts with the TextBox control on the main form, it also remains visible at all times while it's open, even if it doesn't have the focus, because its TopMost property was set to True. In the Properties window, you can specify which form is to be displayed when the application starts.

Forms Vs Dialog Boxes

A dialog box is simply a modal form. When we display forms as dialog boxes, we change the border of the forms to the setting FixedDialog and invoke them with the ShowDialog method. Modeless forms are more difficult to program, because the user may switch among them at any time. Not only that, but the two forms that are open at once must interact with one another. When the user acts on one of the forms, this may necessitate some changes in the other, and you'll see shortly how this is done.

DESIGNING MENUS

The MenuStrip class is the foundation of menus functionality in Windows Forms. If you have worked with menus in .NET 1.0 and 2.0, you must be familiar with the MainMenu control. In .NET 3.5 and 4.0, the MainMenu control is replaced with the MenuStrip control.

Menu Editor

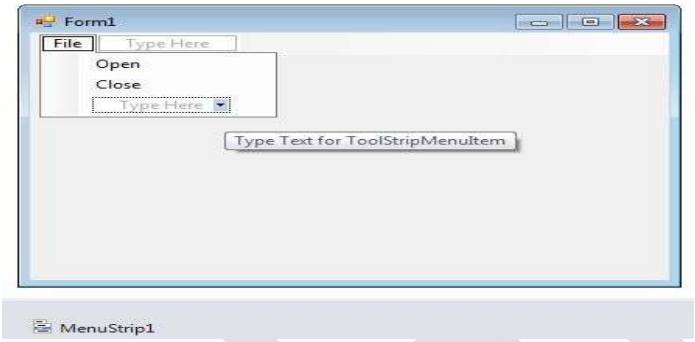
Menus can be attached only to forms, and they're implemented through the MenuStrip control. The items that make up the menu are ToolStripMenuItem objects. As you will see, the MenuStrip control and ToolStripMenuItem objects give you absolute control over the structure and appearance of the menus of your application. The MenuStrip control is a variation of the Strip control, which is the base of menus, toolbars, and status bars.

We can create a MenuStrip control using a Forms designer at design-time or using the MenuStrip class in code at run-time or dynamically. To create a MenuStrip control at design-time, you simply drag and drop a MenuStrip control from Toolbox to a Form in Visual Studio. After you drag and drop a MenuStrip on a Form, the MenuStrip1 is added to the Form and looks like

Figure below. Once a MenuStrip is on the Form, you can add menu items and set its properties and events.

Creating a MenuStrip control at run-time is merely a work of creating an instance of MenuStrip class, set its properties and adds MenuStrip class to the Form controls.

First step to create a dynamic MenuStrip is to create an instance of MenuStrip class. The following code snippet creates a MenuStrip control object.



VB.NET Code:

```
Dim MainMenu As New MenuStrip()
```

In the next step, you may set properties of a MenuStrip control. The following code snippet sets background color, foreground color, Text, Name, and Font properties of a MenuStrip.

```
MainMenu.BackColor = Color.OrangeRed
MainMenu.ForeColor = Color.Black
MainMenu.Text = "File Menu"
MainMenu.Font = New Font("Georgia", 16)
```

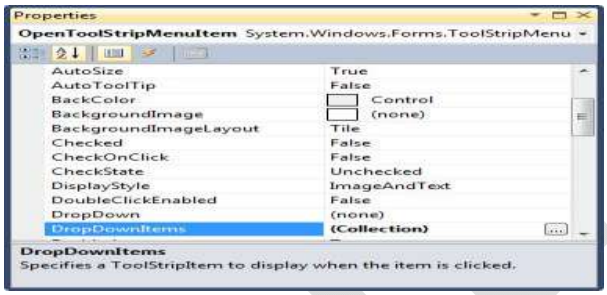
Once the MenuStrip control is ready with its properties, the next step is to add the MenuStrip to a Form. To do so, first we set MainMenuStrip property and then use Form.Controls.Add method that adds MenuStrip control to the Form controls and displays on the Form based on the location and size of the control. The following code snippet adds a MenuStrip control to the current Form.

```
Me.MainMenuStrip = MainMenu
Controls.Add(MainMenu)
```

Setting MenuStrip Properties

After you place a MenuStrip control on a Form, the next step is to set properties. The easiest way to set properties is from the Properties Window. You can open Properties window by pressing F4 or right click on a control and select Properties menu item.

The Properties window looks like Figure below.



Name

Name property represents a unique name of a MenuStrip control. It is used to access the control in the code. The following code snippet sets and gets the name and text of a MenuStrip control.

```
MainMenu.Name = "MailMenu"
```

Positioning a MenuStrip

The Dock property is used to set the position of a MenuStrip. It is of type DockStyle that can have values Top, Bottom, Left, Right, and Fill. The following code snippet sets Location, Width, and Height properties of a MenuStrip control.

```
MainMenu.Dock = DockStyle.Left
```

Font

Font property represents the font of text of a MenuStrip control. If you click on the Font property in Properties window, you will see Font name, size and other font options. The following code snippet sets Font property at run-time.

```
MainMenu.Font = new Font("Georgia", 16)
```

Background and Foreground

BackColor and ForeColor properties are used to set background and foreground color of a MenuStrip respectively. If you click on these properties in Properties window, the Color Dialog pops up.

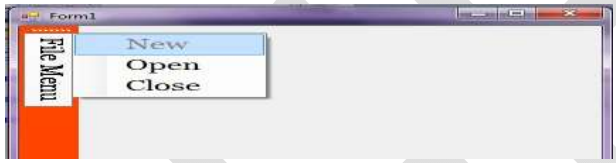
Alternatively, you can set background and foreground colors at run-time. The following code snippet sets

BackColor and ForeColor properties.

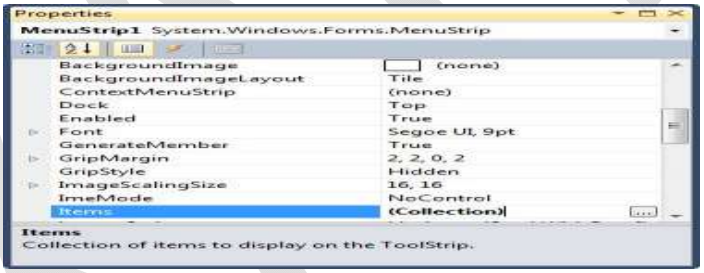
MainMenu.BackColor = System.Drawing.Color.OrangeRed

MainMenu.ForeColor = System.Drawing.Color.Black

Then the MenuStrip looks like Figure below.



MenuStrip Items A Menu control is nothing without menu items. The Items property is used to add and work with items in a MenuStrip. We can add items to a MenuStrip at design-time from Properties Window by clicking on Items Collection as you can see in Figure below.



When you click on the Collections, the String Collection Editor window will pop up where you can type strings. Each line added to this collection will become a MenuStrip item. (See the Figure below.)

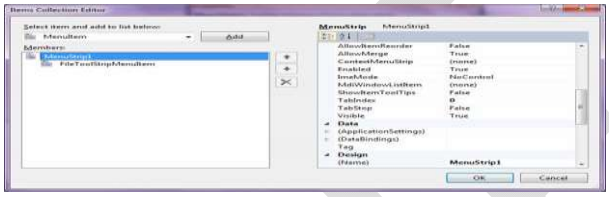
A ToolStripMenuItem represents a menu items. The following code snippet creates a menu item and sets its properties.

```
Dim FileMenu As New ToolStripMenuItem("File")
FileMenu.BackColor = Color.OrangeRed
FileMenu.ForeColor = Color.Black
```

```
FileMenu.Text = "File Menu"
FileMenu.Font = New Font("Georgia", 16)

FileMenu.TextAlign = ContentAlignment.BottomRight
FileMenu.TextDirection =  ToolStripTextDirection.Vertical90
FileMenu.ToolTipText = "Click Me"
```

Figure showing Menu Item Collection

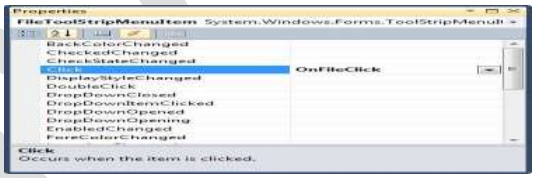


Once a menu item is created, we can add it to the main menu by using MenuStrip.Items.Add method. The following code snippet adds FileMenu item to the MainMenu.

```
MainMenu.Items.Add(FileMenu)
```

Adding Menu Item Click Event Handler

The main purpose of a menu item is to add a click event handler and write code that we need to execute on the menu item click event handler. For example, on File >> New menu item click event handler, we may want to create a new file. To add an event handler, you go to Events window and double click on Click and other as you can see in Figure below.



We can also define and implement an event handler dynamically. The following code snippet defines and implements these events and their respective event handlers.

```
Dim NewMenuItem As New ToolStripMenuItem("New", Nothing, New
EventHandler(AddressOf NewMenuItemClick))
```

```
Private Sub NewMenuItemClick(ByVal sender As Object, ByVal e As EventArgs)
MessageBox.Show("New menu item clicked!")
End Sub
```

Manipulating Menu's at Runtime

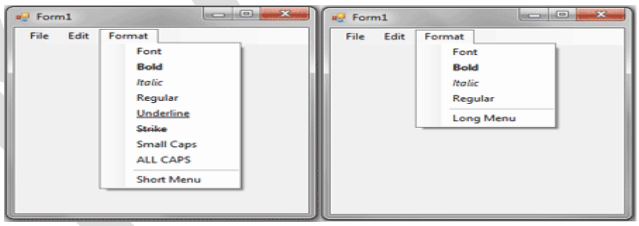
Dynamic menus change at runtime to display more or fewer commands, depending on the current status of the program. This section explores two techniques for implementing dynamic menus:

- Creating short and long versions of the same menu
- Adding and removing menu commands at runtime

Creating Short and Long Menus

A common technique in menu design is to create long and short versions of a menu. If a menu contains many commands, and most of the time only a few of them are needed, you can create one menu with all the commands and another with the most common ones. The first menu is the long one, and the second is the short one. The last command in the long menu should be Short Menu, and when selected, it should display the short version. The last command in the short menu should be Long Menu, and it should display the long version.

Figure shows a long and a short version of the same menu for the example the LongMenu Example. The short version omits infrequently used commands and is easier to handle.



The two versions of the Format menu of the LongMenu application

To implement the LongMenu command, start a new project and create a menu with the options shown in Figure. Listing is the code that shows/hides the long menu in the MenuSize command's Click event.

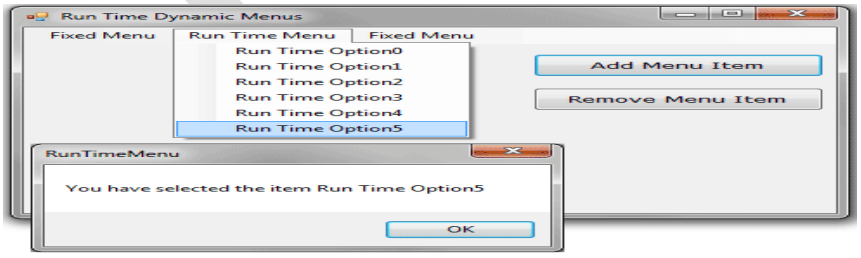
Listing :TheMenuSizeMenu Item's Click Event

```
Private Sub mnuMenuSize_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles mnuSize.Click
If mnuSize.Text = "Short Menu" Then
mnuSize.Text = "Long Menu"
mnuUnderline.Visible = False
mnuStrike.Visible = False
mnuSmallCaps.Visible = False
mnuAllCaps.Visible = False
Else
mnuSize.Text = "Short Menu"
mnuUnderline.Visible = True
mnuStrike.Visible = True
mnuSmallCaps.Visible = True
mnuAllCaps.Visible = True
End If
End Sub
```

The subroutine in Listing 5.11 doesn't do much. It simply toggles the Visible property of certain menu commands and changes the command's caption to Short Menu or Long Menu, depending on the menu's current status.

Adding and Removing Commands at Runtime

The RunTimeMenu project (Figure 5.18) demonstrates how to add items to and remove items from a menu at runtime. The main menu of the application's form contains the Run Time Menu submenu, which is initially empty.



Adding and removing menu items at runtime

The two buttons on the form add commands to and remove commands from the Run Time Menu. Each new command is appended at the end of the menu, and the commands are removed from the bottom of the menu first (the most recently added commands are removed first). To change this order and display the most recent command at the beginning of the menu, use the Insert method instead of the Add method to insert the new item. Listing shows the code behind the two buttons that add and remove menu items.

Listing :Adding and RemovingMenu Items at Runtime

```
Private Sub bttnAddItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles bttnAddItem.Click
Dim Item As New ToolStripMenuItem
Item.Text = "Run Time Option" &
RunTimeMenuToolStripMenuItem.DropDownItems.Count.ToString
RunTimeMenuToolStripMenuItem.DropDownItems.Add(Item)
AddHandler Item.Click, New System.EventHandler(AddressOf OptionClick)
End Sub
Private Sub bttnRemoveItem_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles bttnRemoveItem.Click
If RunTimeMenuToolStripMenuItem.DropDownItems.Count > 0 Then
Dim mItem As ToolStripItem
Dim items As Integer = RunTimeMenuToolStripMenuItem.DropDownItems.Count
mItem = RunTimeMenuToolStripMenuItem.DropDownItems(items - 1)
RunTimeMenuToolStripMenuItem.DropDownItems.Remove(mItem)
' To remove a menu item other than the last one, use the following statement:
'
' RunTimeMenuToolStripMenuItem.DropDownItems.RemoveAt(position)
'
' WHERE position IS THE INDEX OF THE ITEM TO BE REMOVED IN THE DROP DOWN
MENU
End If
```

```
End Sub
```

The Remove button's code uses the Remove method to remove the last item in the menu by its index, after making sure the menu contains at least one item. The Add button adds a new item, sets its caption to Run Time Option n, where n is the item's order in the menu. In addition, it assigns an event handler to the new item's Click event. This event handler is the same for all the items added at runtime; it's the OptionClick() subroutine.

All the runtime options invoke the same event handler — it would be quite cumbersome to come up with a separate event handler for different items. In the single event handler, you can examine the name of the ToolStripMenuItem object that invoked the event handler and act accordingly. The OptionClick() subroutine used in Listing displays the name of the menu item that invoked it. It doesn't do anything, but it shows you how to figure out which item of the Run Time Menu was clicked.

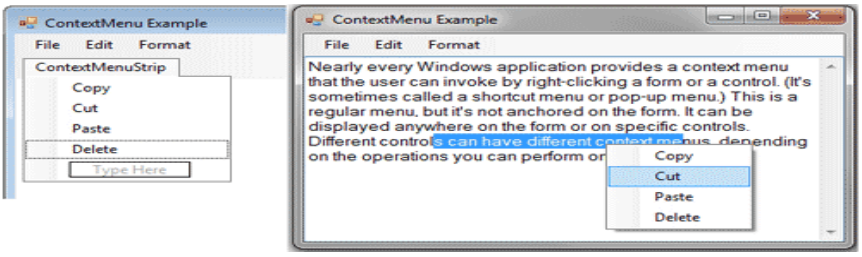
Listing: Programming DynamicMenu Items

```
Private Sub OptionClick(ByVal sender As Object, ByVal e As EventArgs)
Dim itemClicked As New ToolStripMenuItem
itemClicked = CType(sender, ToolStripMenuItem)
MsgBox("You have selected the item" & itemClicked.Text)
End Sub
```

Creating Context Menus

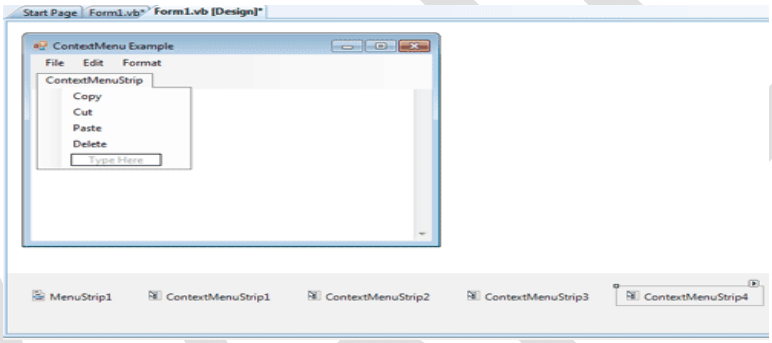
To create a context menu, place a ContextMenuStrip control on your form. The new context menu will appear on the form just like a regular menu, but it won't be displayed there at runtime. You can create as many context menus as you need by placing multiple instances of the ContextMenuStrip control on your form and adding the appropriate commands to each one. To associate a context menu with a control on your form, set the control's ContextMenuStrip property to the name of the corresponding context menu.

Designing a context menu is identical to designing a regular menu. The only difference is that the first command in the menu is always ContextMenuStrip and it's not displayed along with the menu. Figure shows a context menu at design time and how the same menu is displayed at runtime.



A context menu at design time (left) and at runtime (right)

You can create as many context menus as you want on a form. Each control has a ContextMenu property, which you can set to any of the existing ContextMenuStrip controls. Select the control (In Figure it is the TextBox control) for which you want to specify a context menu and locate the ContextMenu property in the Properties window. Expand the drop-down list and select the name of the desired context menu.



Created ContextMenuStrip controls at the bottom of the Designer

To edit one of the context menus on a form, select the appropriate ContextMenuStrip control at the bottom of the Designer as shown in Figure .The corresponding context menu will appear on the form's menu bar, as if it were a regular form menu. This is temporary, however, and the only menu that appears on the form's menu bar at runtime is the one that corresponds to the MenuStrip control (and there can be only one of them on each form).

Iterating a Menu's Items

The last menu-related topic in this chapter demonstrates how to iterate through all the items of a menu structure, including their submenus, at any depth. The main menu of an application can be accessed by the expression Me.MenuStrip1 (assuming that you're using the default names). This is a reference to the top-level commands of the menu, which appear in the form's menu bar. Each command, in turn, is represented by a ToolStripMenuItem object. All the

items under a menu command form a ToolStripMenuItems collection, which you can scan to retrieve the individual commands.

The first command in a menu is accessed with the expression Me.MenuStrip1.Items(0); this is the File command in a typical application. The expression Me.MenuStrip1.Items(1) is the second command on the same level as the File command (typically, the Edit menu).

To access the items under the first menu, use the DropDownItems collection of the top command. The first command in the File menu can be accessed by this expression:

Me.MenuStrip1.Items(0).DropDownItems(0)

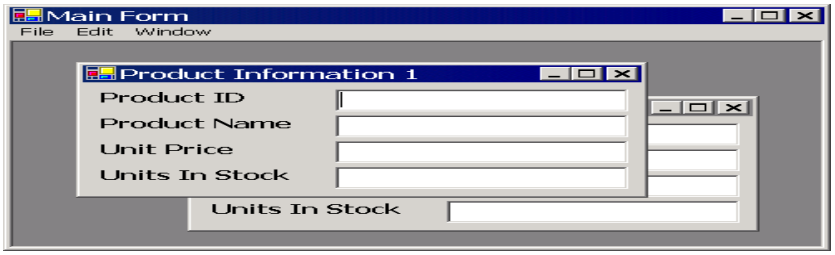
The same items can be accessed by name as well, and this is how you should manipulate the menu items from within your code. In unusual situations, or if you're using dynamic menus to which you add and subtract commands at runtime, you'll have to access the menu items through the DropDownItems collection.

MULTIPLE DOCUMENT INTERFACE

MDI Overview

This session introduces the concept of Multiple Document Interface (MDI) and to create menus within an MDI application. You will learn to create an MDI application in Microsoft Visual Studio .NET and learn why you might want to use this type of interface. You will learn about child forms that are contained within the MDI application, and learn to create shortcut, or context-sensitive, menus.

MDI is a popular interface because it allows you to have multiple documents (or forms) open in one application. Examples of MDI applications include Microsoft Word, Microsoft Excel, Microsoft PowerPoint®, and even the Visual Studio integrated development environment itself. Each application consists of one (or more) parent windows, each containing an MDI client area—the area where the child forms (or documents) will be displayed. Code you write displays as many instances of each of the child forms that you want displayed, and each child form can only be displayed within the confines of the parent window—this means you can't drag the child forms outside the MDI container. Figure shows a basic MDI application in use.



Using MDI – multiple windows contained within the parent area

Single Document Interface

MDI is only one of several possible paradigms for creating a user interface. You can also create applications that display just a single form. They're easier to create, in fact. Those applications are called Single Document Interface (SDI) applications. Microsoft Windows® Notepad is an SDI application, and you can only open a single document at a time. (If you want multiple documents open, you simply run Notepad multiple times.) You are under no obligation to create your applications using the MDI paradigm. Even if you have multiple forms in your project, you can simply have each one as a stand-alone form, not contained by any parent form.

Uses of MDI

MDI are used most often in applications where the user might like to have multiple forms or documents open concurrently. Word processing applications (like Microsoft Word), spreadsheet applications (like Microsoft Excel), and project manager applications (like Microsoft Project) are all good candidates for MDI applications. MDI is also handy when you have a large application, and you want to provide a simple mechanism for closing all the child forms when the user exits the application

Creating an MDI Parent Form

To create an MDI parent form, you can simply take one of your existing forms and set its IsMDIContainer property to **True**. This form will now be able to contain other forms as child forms. You may have one or many container forms within your application.

Tip Note the difference here between Visual Studio .NET and Microsoft Visual Basic® 6.0 behavior. In Visual Basic 6.0, you could only have a single MDI parent form per application, and you had to use the **Project** menu to add that one special form. In Visual Studio .NET, you can turn any form into an MDI parent form by simply modifying a property, and you can have as many MDI parent forms as you require within the same project.

You may have as many different child forms (the forms that remain contained within the parent form) as you want in your project. A child form is nothing more than a regular form for which you dynamically set the **MdiParent** property to refer to the MDI container form.

Run-time Features of MDI Child Forms

At run time, the MDI parent form and the MDI child forms take on special features:

- All child forms are displayed within the MDI parent's *client* area. The client area is the area below the MDI parent's title bar, any menus, and any tool bars.
- Child forms can be moved and sized only within the MDI parent's client area.
- Child forms can be minimized and their icon will be displayed within the parent's client area.
- Child forms can be maximized within the parent's client area and the caption of the child form is appended to the caption of the MDI form.
- Windows automatically gives child forms that have their **FormBorderStyle** property set to a sizable border a default size. This size is based on the size of the MDI parent's client area. You can override this by setting the **FormBorderStyle** property of the child form to any of the fixed type of borders.
- Child forms cannot be displayed modally.

Create an MDI Project

In this section, you will walk through the steps of creating a simple MDI application using Visual Studio .NET. To do this, you will create a new form that will be the MDI parent form. You will add some menus to this new form, and then you will load the product form from a menu as a child form.

Create the MDI Parent Form

To create the MDI parent form

1. Open Visual Studio .NET
2. Create a new Windows application project.
3. Set the name of the project to **MDI.sln**.
4. Rename the form that is created automatically to **frmMain.vb**.
5. With the frmMain selected, set the form's **IsMdiContainer** property to **True**.

6. Set the **WindowState** property to **Maximized**.

Now we have created an MDI parent form.

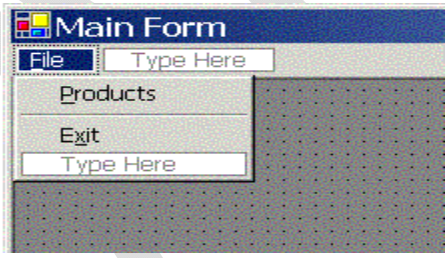
Creating Menus in MDI Main Form

Your main form will require menus so that you can perform actions such as opening child forms, copying and pasting data, and arranging windows. Visual Studio .NET includes a new menu designer that makes creating & modifying menus easy.

To add menus to your MDI parent form

1. Double-click the MenuStrip tool in the Toolbox window to add a new object named MenuStrip1 to the form tray.
2. At the top of the MDI parent form, click the box with **Type Here** in it and type **&File**.
3. Press **Enter** to move to the next menu item and type **&Products**.
4. Press **Enter** to move to the next menu item and type a hyphen (**-**).
5. Press **Enter** and type **E&xit**.

You have now created the first drop-down menu on your main form. You should have something that looks like Figure.



The menu designer allows you to type your menu structure in a WYSIWYG fashion

To the right of the **File** menu and at the same level, you'll see another small box with the text, **Type Here**. Click it and type the following menu items by pressing **Enter** after each one.

- &Edit
  - Cu&t
  - &Copy
  - &Paste

Once more to the right of the Edit menu and at the same level, add the following menu items in the same manner.

- &Window
    - &Cascade
    - Tile &Horizontal
    - Tile &Vertical
    - &Arrange Icons

Creating Names for Each Menu

After creating all the menu items, you'll need to set the **Name** property for each. (Because you'll refer to the name of each menu item from any code you write concerning that menu item, it's important to choose a name you can understand from within your code.) Instead of clicking each menu item one at a time and then moving over to the Properties window to set the **Name** property, Visual Studio provides a shortcut: Right-click an item in the menu, then select **Edit Names** from the context menu..

Use the following names for your menu items:

- mnuFile
    - mnuFProducts
    - mnuFExit
- mnuEdit
    - mnuECut
    - mnuECopy
    - mnuEPaste
- mnuWindow
    - mnuWCasade
    - mnuWHorizontal
    - mnuWVertical
    - mnuWArrange

Test out your application: Press **F5** and you should see your main MDI window appear with your menu system in place.

Display a Child Form

To add the code that displays the child form, frmProducts, make sure the main form is open in Design view, and on the **File** menu, double-click **Products**. Visual Studio .NET will create the stub of the menu item's Click event handler for you. Modify the procedure so that it looks like the following:

```
Private Sub mnuFProducts_Click( _
 ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles_  mnuFProducts.Click
   Dim frm As New frmProducts()
   frm.MdiParent = Me
   frm.Show()
End Sub
```

This code declares a variable, frm, which refers to a new instance of the frmProducts form in the sample project. Then, you set the **MdiParent** property of the new form, indicating that its parent should be the current form (using the **Me** keyword). Finally, the code calls the **Show** method of the child form, making it appear on the screen.

Child Menus in MDI Applications

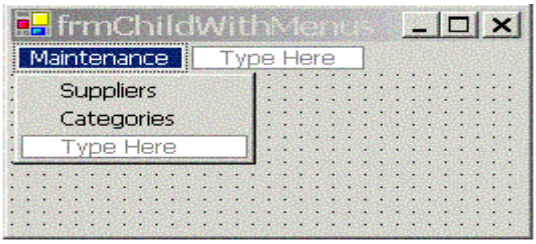
In Visual Studio .NET, however, you can control how the menus interact, using the **MergeOrder** and **MergeType** properties of the individual menu items.

The **MergeOrder** property controls the relative position of the menu item when its menu structure gets merged with the parent form's menus. The default value for this property is 0, indicating that this menu item will be added at the end of the existing menu items. The **MergeType** property controls how the menu item behaves when it has the same merge order as another menu item being merged. Table shows a list of the possible values you can assign to the **MergeType** property.

The MergeType property allows you to specify what happens when menu items merge

Value	Description
Add	The MenuItem is added to the collection of existing MenuItem objects in a
MergeItems	All submenu items of this MenuItem are merged with those of existing MenuItem
Remove	The MenuItem is not included in a merged
Replace	The MenuItem replaces an existing MenuItem at the same position in a merged

By default, a menu item's **MergeOrder** property is set to 0. The **MergeType** property is set to **Add** by default. This means that if you create a child form with a menu on it, the menu will be added at the end of the main menu. Consider Figure 3, which shows a child form called from the parent form's main menu. This form has a **Maintenance** menu on it (and the parent form does not). All of the items on the parent's main menu have their **MergeOrder** properties set to 0 and this menu's **MergeOrder** property is set to 0, so this menu will be added at the end of the main menu on the MDI parent form.



A child form that has menus will by default be added to the end of the main menu

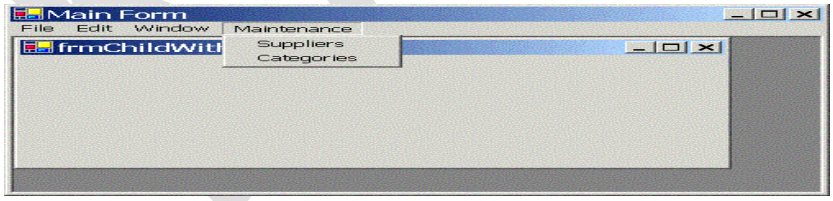
To create the form in Figure 3

1. On the **Project** menu, click **Add Windows Form**.
2. Set the new form's name to frmChildWithMenus.vb.
3. Add a MenuStrip control to this form.
4. Set the Name property for the MenuStrip control to **mnuMainMaint**.
5. Add the following menus as shown in Table 2.

Windows Form menus

Menu	Name
&Maintenance	mnuMaint
&Suppliers	mnuMSuppliers
&Categories	mnuMCategories

If you were to call this form exactly like you did the Products form in the previous section you will see that your main form looks like Figure 4. You can see that by default, the menu is added to the end of this form.



Menus are added to the end of the main menu by default

Call this form by adding a new menu item under the **File** menu:

1. Open frmMain.vb in Design view.
2. Click on the separator after the **Products** menu item and press the **Insert** key to add a new menu item.
3. Type **Child form with Menus** as the text of this new menu item.

4. Set the Name property of this new menu item to **mnuFChild**.
5. Double click this new menu item and modify its Click event handler so that it looks like this:

```
Private Sub mnuFChildMenus_Click( _
 ByVal sender As System.Object, _
 ByVal e As System.EventArgs) _
 Handles mnuFChildMenus.Click
   Dim frm As New frmChildWithMenus()
   frm.MdiParent = Me
   frm.Show()
End Sub
```

Note: If you wish to merge the **Maintenance** menu in between the **Edit** and **Window** menus, you could set the **MergeOrder** property on the **Edit** menu item to 1, and the **MergeOrder** property on the **Window** menu to a 2. Then on the **Maintenance** menu item on frmChildWithMenus, set the **MergeOrder** property to 1 and leave the **MergeType** with its default value, **Add**. Taking these steps will add the **Maintenance** menu after the menu on the main form with the same **MergeOrder** number as it has (that is, after the **Edit** menu, but before the **Window** menu).

Working with MDI Child Forms

If you have multiple child forms open, you may want to have them arrange themselves, much as you can do in Word or Excel, choosing options under the **Window** menu. Table lists the available options when arranging child windows.

Choose one of these values when arranging child windows

Menu Item	Enumerated Value
Tile Horizontal	MdlLayout.TileHorizontal
Tile Vertical	MdiLayout.TileVertical
Cascade	MdiLayout.Cascade
Arrange Icons	MdiLayout.ArrangeIcons

Add some menus to your main form for each of these options:

1. Open **frmMain.vb** in Design view.
2. On the **Window menu,** double-click **Cascade**.
3. For the Cascade menu item, modify the Click event handler so that it looks like the following:

```
Private Sub mnuWCascade_Click( _
 ByVal sender As System.Object, _
 ByVal e As System.EventArgs) _
 Handles mnuWCascade.Click
   Me.LayoutMdi(MdiLayout.Cascade)
End Sub
```

On the **Window** menu, double-click each menu item and add the appropriate code.

Tracking Child Windows

Visual Basic .NET will keep track of all child forms that you create, and it's easy to create a window list menu to manage the child windows. If you wish to see a list of all of the child forms and be able to give a specific child form focus, follow these steps:

1. Load frmMain in Design view.
2. Select frmMain's Window menu.
3. In the Properties window, set the **MdiList** to **True**.
4. Run the project, open a couple of Products forms, and then click the Window drop-down menu. You should see each instance of the Product form that you opened displayed in the window list.

Ending an MDI Application

In most cases, ending an application with the End statement isn't necessarily the most user-friendly approach. Before you end an application, you must always offer your users a chance to save their work. Ideally, you should maintain a *True/False* variable whose value is set every time the user edits the open document terminating an MDI application with the End statement is unacceptable. First, you need a mechanism to detect whether a document needs to be saved or not. In a text-processing application, you can examine the Modified property of the TextBox control.

Insert the proper code in the Close command's event handler to detect whether the document being closed contains unsaved data and prompt the user accordingly. When the user clicks the child form's Close button, the child form's Closing event is fired, this time by the child form. Finally, when the MDI form is closed, each of the child forms receives the Closing event. In addition, the MDI form's Closing event is also fired. Normally, there's no reason to program this event. As long as you handle the Closing event of the child form, no data will be lost. In the Closing event, you can cancel the operation of closing a document, or the MDI form itself, by settings the e.Cancel property to True.

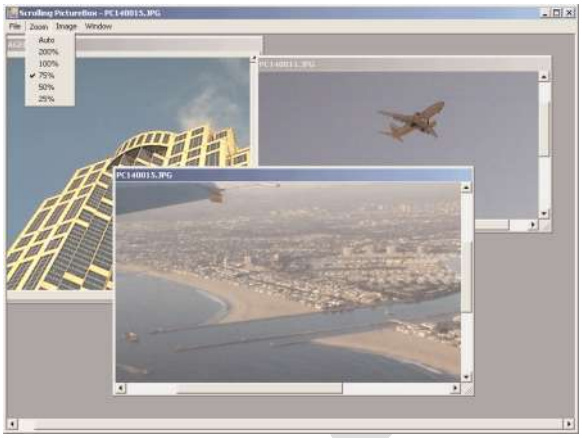
To close the active child form, execute the following statements (they must appear in the Close command's Click event handler):

```
Private Sub FileExit_Click(ByVal sender As System.Object, _ByVal e As System.EventArgs)
Handles FileExit.Click
Me.Close()
End Sub
```

The Close method invokes the Closing event of the child form.

A Scrollable PictureBox

The scrollable PictureBox isn't a new control; it's not even a PictureBox with its own scroll bars. It's a child form filled with a PictureBox control. The size of the PictureBox is determined by the user at runtime, but if it gets smaller than the size of the image, the scroll bars will be attached automatically. This is a feature of the Form object, and child forms support it, because they inherit the Windows.Forms.Form class. Figure shows a child form with an image and the appropriate scroll bars attached to it. From a user's point of view, it looks just like a PictureBox with scroll bars.



Using an MDI form to simulate a scrolling PictureBox control

POSSIBLE QUESTIONS

PART A (1 Mark)

(Online Examinations)

PART B (6 Marks)

1) Explain in detail about Argument Passing Mechanisms.
2) Write about Forms Vs DialogBoxes.
3) Write a brief notes about overloading functions. Give Example.
4) Compare and contrast the subroutines and functions. Give Example for each.
5) How will you Manipulating Menu's at Runtime. Explain in detail
6) Write a program to implement calculator
7) Explain in detail about Argument Passing Mechanism with example.
8) Illustrate the usage of message box.
9) Explain on Loading and showing forms
10) Describe the properties and methods of Text box.

PART A (1 Mark) – Unit I1

S.no	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
1	The _____ statement first execute the statement and then test the condition after each execution	do….while	while….do	select….case	while	do….while
2	_____ structure executes the statements until the condition is satisfied	do…loop	do..loop until	do while…loop	do until	do..loop until
3	do…loop until is ---------- loop	finite	infinite	long	small	infinite
4	_____ function retrieves only date	for…next	next…for	exit for	exit do	for…next

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
5	A _____ loop can be terminated by an exit for statement	for…next	next…for	exit for	for exit	for…next
6	do….while loop is terminated using _____ statement	exit for	for exit	exit do	do exit	exit do
7	A sequence of variables by the same name can be referred using _____	arrays	modules	sub-routines	functions	arrays
8	_____operator in VB is used for string concatenation	&	*	+	|	&
9	Code in VB is stored in the form of _____	modules	subroutines	variables	standards	modules

S.no..	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
10	Procedures that returns a value are called ----------	subroutines	sub units	parameters	functions	functions
11	A calling procedure passes data to the parameters by way of ----------	objects	arguments	strings	numbers	arguments
12	Which of the following can be called by value?	Class	Module	Assembly	Function	Function
13	Functions are especially useful for taking one or more pieces of data called	modules	arguments	procedures	programs	arguments
14	In the function 'Public Function name(ByVal str as String) As Integer' the return type is ----------	Void	String	Integer	Any	Integer

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
15	A calling procedure passes data to the parameters by way of ----------	objects	arguments	strings	numbers	arguments
16	To create a procedure as an entry point in code, you must name the procedure ---------	Main	Sub	Entry	Start	Main
17	Which of the following can be called by value?	Class	Module	Assembly	Function	Function
18	The ----- event happens when the mouse pointer hovers over the form/control	MouseWheel	MouseUp	MouseDown	MouseHover	MouseHover
19	the ____occurs when a mouse button is pressed	MouseDown	MouseUp	MouseWheel	mouse double click	MouseDown

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
20	------ specifies number of times the mouse button is pressed and released	Button	Click	Delta	X	Click
21	If the number of items exceed the value that can be displayed, _____ bars will automatically appear on the control	icon	option button	command button	scroll bars	scroll bars
22	_____ provides easy navigation through a list of items or a large amount of information	scroll bar	command button	tool bar	tool box	scroll bar
23	_____is the container for all the controls that make up the user interface.	Form	form window	tool window	toolbar	Form
24	The forms that constitute the visible interface of your application are called_____	forms	Windows forms	Form window	toolbar	Windows forms

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
25	_____ToolWindow, but is resizable. In addition, its caption font is smaller than the usual.	Sizable	non sizable	both a & b	resizable	Sizable
26	The _____ is an one example of breaking a large application into smaller tasks.	Event Handler	Coding	both a & b	module	Event Handler
27	_____specify the access level of the function	Function name	Modifiers	Parameters	Return type	Modifiers
28	In event-driven programming an event is generated by:	a user's action.	the program itself.	None	Both a & b	Both a & b
29	Which helps the user are not sure of the number of arguments passed as a parameter while declaring a function or sub procedure	Named Arrays	Param Arrays	Unknown arrays	Arrays	Param Arrays

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
30	In _____ arguments, the user can supply arguments in any order to the functions.	Named arguments	Param arguments	List arguments	Value arguments	Named arguments
31	Multiple implementations of the same function is called ____	poly overloading	overloading function	Override function	Project	overloading function
32	The user action like key press, clicks, mouse movements are called __	Handlers	Triggers	Events	Methods	Events
33	when a mouse button is pressed _____ event will fired	Mouse Enter	Mouse Up	Mouse down	MouseHover	Mouse down
34	_____ event is fired when a key is released while the control has focus	Key Up	Key press	Key Down	Key Enter	Key Up

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
35	___ allows the user to open the menu by pressing the Alt key and a letter	Access key	shortcut key	accept key	keys	Access key
36	_____ is a value you pass to the procedure and on which the procedure usually acts	constant	function	subroutine	arguments	arguments
37	_____ is a segment of the code that is executed each time an external condition triggers the event	Event Handler	function	Coding	built-in function	Event Handler
38	A _____ is a block of statements that carries out a well-defined task, those statements are placed between Sub …. End Sub.	Modular	subroutines	Function	Events	subroutines

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
39	Which property of the key events returns the keyboard value for the key that was pressed	Key Data	Key Value	Key Code	key	Key Value
40	The _____ concept helps the user can supply arguments in any order	named arguments	arguments value	order arguments	order value	named arguments
41	_____ means the user can have multiple implementation of the same function, each with a different set of arguments and a different return type	Overloading	overriding	multiple function	procedures	Overloading
42	WindowState property is _____ by default	Normal	Maximized	Minimuzed	functions	Normal

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
43	Procedures that returns a value are called _____	subroutines	sub units	parameters	functions	functions
44	Which property has to be set to minimize maximize ot restore a form in code?	Windows Applications	WindowState	FormBorderStyle	WindowSize	WindowState
45	Which property is used to specify the tab order of the various controls	tab order	Accept return	Control Box	Auto tab	tab order
46	The tab order command will appear in which menu	File	Format	View	Edit	View
47	Which property is used to not move the controls around the forms.	Control	Top	Locked	autotab	Locked

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
48	_____ refers to the position a control has relative to the edge of the form	Anchor	Dock	Key stokes	Key preview	Anchor
49	_____ refers to how much space the control to take up on the form	Anchor	Dock	Key stokes	Key preview	Dock
50	The _____ event takes place every time the form must be refreshed	Resize	Paint	Close	refersh	Paint
51	The default value of FormBorderStyle property is	FixedSingle	FixedToolWindow	Sizable	SizableToolWindow	Sizable
52	The _____ property determines the initial position of the form when its first displayed	initial position	Start position	sizedripstyle	Fiexed3D	Start position

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
53	the _____ value position the form at the default location and size determined by windows	WindowsDefault Location	WindowsDefault Bounds	Fixed Dialog	Fixed 3D	WindowsDefault Bounds
54	To attach the scroll bar automatically to the form, which property to set true.	Auto Scale	Auto scroll	Auto scroll bar	Auto accept	Auto scroll
55	Without the _____ the form cannot be repositioned by the user	Minimize / Maximize button	Border	Title bar	Control Menu	Title bar
56	_____ method does not simply hides the form, but destroy it completely	Close()	Hide()	Distroy()	Remove()	Close()
57	The simplest method for two forms to communicate with each other is via ____ variables	Private	Common	public	form	public

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
58	How many parent form will be in MDI	2	0	1	many	1
59	The _____ property is one that automatically activated when you press Enter	Accept button	Cancel button	Control box	Border style	Accept button
60	When the user pass an argument _____, the procedure sees only a copy of the argument	By val	By ref	By values	By references	By val

UNIT – III

SYLLABUS

Basic Windows Controls: Textbox Control- ListBox, CheckedListBox-Scrollbar and TrackBar Controls. More Windows Control: The common Dialog Controls-The Rich TextBox Control.The TreeView and ListView Controls: Examining the Advanced Controls-The TreeView Control-The ListView Control-Content Page Holder

BASIC WINDOWS CONTROLS

The TextBox Control

The TextBox control is the primary mechanism for displaying and entering text. It is a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.

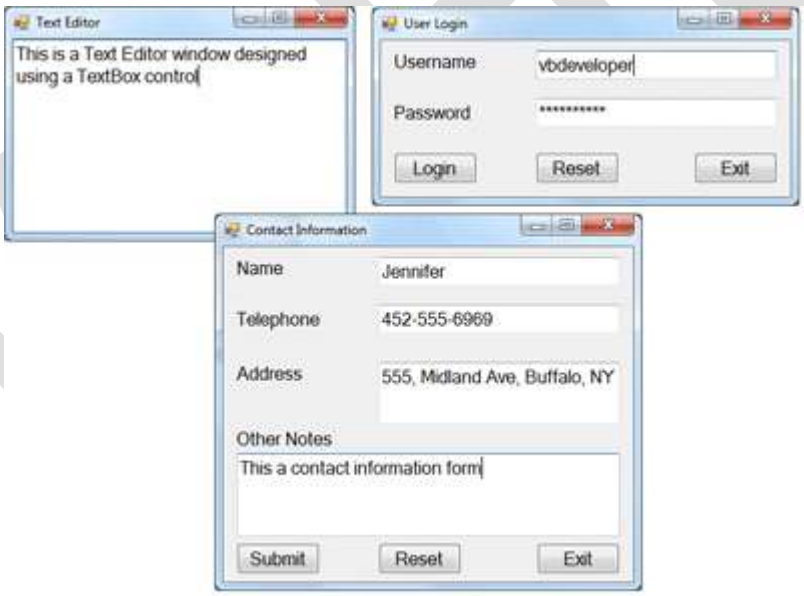


Figure - TextBox Examples

Basic Properties of the TextBox Control

Let's start with the properties that specify the appearance and, to some degree, the functionality of the TextBox control; these properties are usually set at design time through the Propertieswindow.

TextAlign

This property sets (or returns) the alignment of the text on the control, and its value is a member of the HorizontalAlignment enumeration: Left, Right, or Center.

MultiLine

This property determines whether the TextBox control will hold a single line or multiple lines of text. Every time you place a TextBox control on your form, it's sized for a single line of text and you can change its width only. To change this behavior, set the MultiLine property to True. When creating multiline TextBoxes, you will most likely have to set one or more of the MaxLength, ScrollBars, and WordWrap properties in the Properties window.

MaxLength

This property determines the number of characters that the TextBox control will accept. Its default value is 32,767, which was the maximum number of characters the VB 6 version of the control could hold. Set this property to zero, so that the text can have any length, up to the control's capacity limit — 2,147,483,647 characters, to be exact.

ScrollBars

This property lets you specify the scroll bars you want to attach to the TextBox if the text exceeds the control's dimensions. Single-line text boxes can't have a scroll bar attached, even if the text exceeds the width of the control. Multiline text boxes can have a horizontal or a vertical scroll bar, or both.

WordWrap

This property determines whether the text is wrapped automatically when it reaches the right edge of the control. The default value of this property is True. If the control has a horizontal scroll bar, however, you can enter very long lines of text.

AcceptsReturn, AcceptsTab

These two properties specify how the TextBox control reacts to the Return (Enter) and Tab keys. The Enter key activates the default button on the form, if there is one. The default button is usually an OK button that can be activated with the Enter key, even if it doesn't have the focus.

The default value of the AcceptsReturn property is True, so pressing Enter creates a new line on the control. If you set it to False, users can still create new lines in the TextBox control, but they'll have to press Ctrl+Enter.

Likewise, the AcceptsTab property determines how the control reacts to the Tab key.Normally, the Tab key takes you to the next control in the Tab order, and we generally avoid changing the default setting of the AcceptsTab property.

CharacterCasing

This property tells the control to change the casing of the characters as they're entered by the user. Its default value is Normal, and characters are displayed as typed. You can set it to Upper or Lower to convert the characters to upper- or lowercase automatically.

PasswordChar

This property turns the characters typed into any character you specify. If you don't want to display the actual characters typed by the user (when entering a password, for instance), use this property to define the character to appear in place of each character the user types.

The default value of this property is an empty string, which tells the control to display the characters as entered. If you set this value to an asterisk (*), for example, the user sees an asterisk in the place of every character typed. This property doesn't affect the control's Text property, which contains the actual characters. If the **PasswordChar property of the TextBox control** is set to any character, the user can't copy or cut the text on the control.

ReadOnly, Locked

If you want to display text on a TextBox control but prevent users from editing it (such as for an agreement or a contract they must read, software installation instructions, and so on), you can set the ReadOnly property to True.When ReadOnly is set to True, you can put text on the control from within your code, and users can view it, yet they can't edit it.

Text-Manipulation Properties

Most of the properties for manipulating text in a TextBox control are available at runtime only. This section presents a breakdown of each property.

Text

The most important property of the TextBox control is the Text property, which holds the control's text. You can set this property at design time to display some text on the control

initially. Notice that there are two methods of setting the Text property at design time. For single-line TextBox controls, set the Text property to a short string, as usual. For multiline TextBox controls, open the Lines property and enter the text in the String Collection Editor window, which will appear.

```
Dim strLen As Integer = TextBox1.Text.Length
```

The IndexOf method of the String class will locate a specific string in the control's text. The following statement returns the location of the first occurrence of the string Visual in the text:

```
Dim location As Integer
location = TextBox1.Text.IndexOf("Visual")
```

For more information on locating strings in a TextBox control, see the section "VB 2008 The TextPad Project" later in this chapter, where we'll build a text editor with search-and-replace capabilities. For a detailed discussion of the String class, see Chapter, "Handling Strings, Characters, and Dates."

To store the control's contents in a file, use a statement such as the following:

```
StrWriter.Write(TextBox1.Text)
```

Similarly, you can read the contents of a text file into a TextBox control by using a statement such as the following:

```
TextBox1.Text = StrReader.ReadToEnd
```

Listing 6.1: Locating All Instances of a String in a TextBox

```
Dim startIndex = -1
startIndex = TextBox1.Text.IndexOf("Basic", startIndex + 1)
While startIndex > 0
Console.WriteLine "String found at " & startIndex
startIndex = TextBox1.Text.IndexOf("Basic", startIndex + 1)
End While
```

The following statement appends a string to the existing text on the control:

```
TextBox1.Text = TextBox1.Text & newString
```

To append a string to a TextBox control, use the following statement:

```
TextBox1.AppendText(newString)
TextBox1.AppendText(newString & vbCrLf)
```

Lines

In addition to the Text property, you can access the text on the control by using the Lines property. The Lines property is a string array, and each element holds a paragraph of text. The first paragraph is stored in the element Lines(0), the second paragraph in the element Lines(1), and so on. You can iterate through the text lines with a loop such as the following:

```
Dim iLine As Integer
For iLine = 0 To TextBox1.Lines.GetUpperBound(0) - 1
{ process string TextBox1.Lines(iLine) }
Next
```

READONLY, LOCKED

If you want to display text on a TextBox control but prevent users from editing it (an agreement or a contract they must read, software installation instructions, and so on), you can set the ReadOnly property to True. When ReadOnly is set to True, you can put text on the control from within your code, and users can view it, yet they can't edit it

PASSWORDCHAR

Available at design time, this property turns the characters typed into any character you specify. If you don't want to display the actual characters typed by the user (when entering a password, for instance), use this property to define the character to appear in place of each character the user types.

The default value of this property is an empty string, which tells the control to display the characters as entered. If you set this value to an asterisk (*), for example, the user sees an asterisk in the place of every character typed.

Text-Selection Properties

The TextBox control provides three properties for manipulating the text selected by the user: SelectedText, SelectionStart, and SelectionLength. Users can select a range of text with a click-and-drag operation, and the selected text will appear in reverse color. You can access the selected text from within your code through the SelectedText property, and its location in the control's text through the SelectionStart and SelectionLength properties.

SelectedText

This property returns the selected text, enabling you to manipulate the current selection from within your code. For example, you can replace the selection by assigning a new value to the SelectedText property. To convert the selected text to uppercase, use the ToUpper method of the String class:

```
TextBox1.SelectedText = TextBox1.SelectedText.ToUpper
```

SelectionStart, SelectionLength

Use these two properties to read the text selected by the user on the control, or to select text from within your code. The SelectionStart property returns or sets the position of the first character of the selected text, somewhat like placing the cursor at a specific location in the text and selecting text by dragging the mouse. The SelectionLength property returns or sets the length of the selected text.

```
Dim seekString As String = "Visual"
Dim strLocation As Long
strLocation = TextBox1.Text.IndexOf(seekString)
If strLocation > 0 Then
TextBox1.SelectionStart = strLocation
TextBox1.SelectionLength = seekString.Length
End If
TextBox1.ScrollToCaret()
```

HideSelection

The selected text in the TextBox does not remain highlighted when the user moves to another control or form; to change this default behavior, set the HideSelection property to False. Use this property to keep the selected text highlighted, even if another form or a dialog box, such as a Find & Replace dialog box, has the focus. Its default value is True, which means that the text doesn't remain highlighted when the TextBox loses the focus.

Locating the Cursor Position in the Control

The SelectionStart and SelectionLength properties always have a value even if no text is selected on the control. In this case, SelectionLength is 0, and SelectionStart is the current

position of the pointer in the text. If you want to insert some text at the pointer's location, simply assign it to the SelectedText property, even if no text is selected on the control.

Text-Selection Methods

addition to properties, the TextBox control exposes two methods for selecting text. You can select some text by using the Select method, whose syntax is shown next:

TextBox1.Select(start, length)

The Select method is equivalent to setting the SelectionStart and SelectionLength properties. To select the characters 100 through 105 on the control, call the Select method, passing the values 99 and 6 as arguments:

TextBox1.Select(99, 6)

TextBox1.Select(3, 4)

If you insert a line break every third character and the text becomes the following, the same statement will select the characters DE only:

ABC

DEF

GHI

In reality, it has also selected the two characters that separate the first two lines, but special characters aren't displayed and can't be highlighted. The length of the selection, however, is 4. A variation of the Select method is the SelectAll method, which selects all the text on the control.

Undoing Edits - CanUndo property

An interesting feature of the TextBox control is that it can automatically undo the most recent edit operation. To undo an operation from within your code, you must first examine the value of the CanUndo property. If it's True, the control can undo the operation; then you can call the Undo method to undo the most recent edit.

The ListBox, CheckedBox, and ComboBox Controls

The ListBox, CheckedListBox, and ComboBox controls present lists of choices, from which the user can select one or more. The ListBox control occupies a user-specified amount of space on the form and is populated with a list of items. If the list of items is longer than can fit on the control, a vertical scroll bar appears automatically.

The CheckedListBox control is a variation of the ListBox control. It's identical to the ListBox control, but a check box appears in front of each item. The user can select any number of items by selecting the check boxes in front of them. As you know, you can also select multiple items from a ListBox control by pressing the Shift and Ctrl keys.

The ComboBox control also contains multiple items but typically occupies less space on the screen. The ComboBox control is an expandable ListBox control: The user can expand it to make a selection, and collapse it after the selection is made. The real advantage of the ComboBox control, however, is that the user can enter new information in the ComboBox, rather than being forced to select from the items listed.

Basic Properties The ListBox, CheckedListBox, and ComboBox Controls

In this section, you'll find the properties that determine the functionality of the three controls. These properties are usually set at design time, but you can change their setting from within your application's code.

IntegralHeight

This property is a Boolean value (True/False) that indicates whether the control's height will be adjusted to avoid the partial display of the last item. When set to True, the control's actual height changes in multiples of the height of a single line, so only an integer number of rows are displayed at all times.

Items

The Items property is a collection that holds the control's items. At design time, you can populate this list through the String Collection Editor window. At runtime, you can access and manipulate the items through the methods and properties of the Items collection, which are described shortly.

MultiColumn

A ListBox control can display its items in multiple columns if you set its MultiColumn property to True. The problem with multicolumn ListBoxes is that you can't specify the column in which each item will appear. ListBoxes with many items and their MultiColumn property set to True expand horizontally, not vertically. A horizontal scroll bar will be attached to a multicolumn ListBox, so that users can bring any column into view. This property does not apply to the ComboBox control.

SelectionMode

This property, which applies to the ListBox and CheckedListBox controls only, determines how the user can select the list's items. The possible values of this property—members of the SelectionMode enumeration— are shown in Table 4.3.

Table - The SelectionMode Enumeration

Value	Description
None	No selection at all is allowed.
One	(Default) Only a single item can be selected.
MultiSimple	Simple multiple selection: A mouse click (or pressing the spacebar) selects or deselects an item in the list. You must click all the items you want to select.
MultiExtended	Extended multiple selection: Press Shift and click the mouse (or press one of the arrow keys) to expand the selection. This process highlights all the items between the previously selected item and the current selection. Press Ctrl and click the mouse to select or deselect single items in the list.

Sorted

When this property is True, the items remain sorted at all times. The default is False, because it takes longer to insert new items in their proper location. This property's value can be set at design time as well as runtime.

Text

The Text property returns the selected text on the control. Although you can set the Text property for the ComboBox control at design time, this property is available only at runtime for the other two controls. Notice that the items need not be strings.

The Items Collection

To manipulate a ListBox control from within your application, you should be able to do the following:

- Add items to the list
- Remove items from the list
- Access individual items in the list

If you add a Color object and a Rectangle object to the Items collection with the following statements:

```
ListBox1.Items.Add(New Font("Verdana", 12, FontStyle.Bold)
ListBox1.Items.Add(New Rectangle(0, 0, 100, 100))
```

then the following strings appear on the first two lines of the control:

```
[Font: Name=Verdana, Size=12, Units=3, GdiCharSet=1, gdiVerticalFont=False]
{X=0, Y=0, Width=100, Height=100}
```

However, you can access the members of the two objects because the ListBox stores objects, not their descriptions.

```
Debug.WriteLine(ListBox1.Items.Item(1).Width)
100
If ListBox1.Items.Item(0).GetType Is GetType(Rectangle) Then
Debug.WriteLine(CType(ListBox1.Items.Item(0), Rectangle).Width)
End If
```

The Add Method

To add items to the list, use the Items.Add or Items.Insert method. The syntax of the Add method is as follows:

```
ListBox1.Items.Add(item)
```

The following loop adds the elements of the array words to a ListBox control, one at a time:

```
Dim words(100) As String
{ statements to populate array }
Dim i As Integer
For i = 0 To 99
ListBox1.Items.Add(words(i))
Next
```

Similarly, you can iterate through all the items on the control by using a loop such as the following:

```
Dim i As Integer
For i = 0 To ListBox1.Items.Count - 1
{ statements to process item ListBox1.Items(i) }
Next
```

You can also use the For Each . . . Next statement to iterate through the Items collection, as shown here:

```
Dim itm As Object
For Each itm In ListBox1.Items
{ process the current item, represented by the itm variable }
Next
```

The Insert Method

To insert an item at a specific location, use the Insert method, whose syntax is as follows:

```
ListBox1.Items.Insert(index, item)
```

The Clear Method

The Clear method removes all the items from the control. Its syntax is quite simple:

```
List1.Items.Clear
```

The Count Property

This is the number of items in the list. If you want to access all the items with a For . . . Next loop, the loop's counter must go from 0 to ListBox.Items.Count - 1, as shown in the example of the Add method.

The CopyTo Method

The CopyTo method of the Items collection retrieves all the items from a ListBox control and stores them in the array passed to the method as an argument. The syntax of the CopyTo method is

```
ListBox.CopyTo(destination, index)
```

The Remove and RemoveAt Method

To remove an item from the list, you must first find its position (index) in the list, and all the Remove method passing the position as argument:

```
ListBox1.Items.Remove(index)
```

The index parameter is the order of the item to be removed, and this time it's not optional. The following statement removes the item at the top of the list:

```
ListBox1.Remove(0)
```

If the control contains strings, pass the string to be removed. If the same string appears multiple times on the control, only the first instance will be removed. If the control contains object, pass a variable that references the item you want to remove.

You can also remove an item by specifying its position (reference) in the list via the RemoveAt method, which accepts as argument the position of the item to be removed:

```
ListBox1.Items.RemoveAt(index)
```

The index parameter is the order of the item to be removed, and the first item's order is 0.

The Contains Method

The Contains method of the Items collection — not to be confused with the control's Contains method — accepts an object as an argument and returns a True/False value that indicates whether the collection contains this object. Use the Contains method to avoid the insertion of identical objects into the ListBox control. The following statements add a string to the Items collection, only if the string isn't already part of the collection:

```
Dim itm As String = "Remote Computing"
If Not ListBox1.Items.Contains(itm) Then
ListBox1.Items.Add(itm)
End If
```

Searching:

Two of the most useful methods of the ListBox control are the FindString and FindStringExact methods, which allow you to quickly locate any item in the list. The FindString method locates a string that partially matches the one you're searching for; FindStringExact finds an exact match. If you're searching for Man, and the control contains a name such as Mansfield, FindStringmatches the item, but FindStringExact does not.

Both the FindString and FindStringExact methods perform case-insensitive searches. If you're searching for visual, and the list contains the item Visual, both methods will locate it. Their syntax is the same:

```
itemIndex = ListBox1.FindString(searchStr As String)
```

where searchStr is the string you're searching for. An alternative form of both methods allows you to specify the order of the item at which the search will begin:

itemIndex = ListBox1.FindString(searchStr As String, startIndex As Integer)

The startIndex argument allows you to specify the beginning of the search, but you can't specify where the search will end.

The ListBoxSearch Application

The application you'll build in this section (seen in Figure 4.5) populates a list with a large number of items and then locates any string you specify. Click the button Populate List to populate the ListBox control with 10,000 random strings. This process will take a few seconds and will populate the control with different random strings every time. Then, you can enter a string in the TextBox control at the bottom of the form.

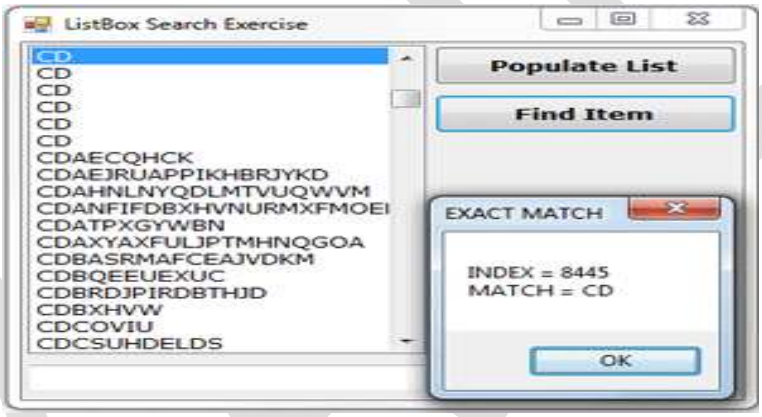


Figure - ListBox Control Search example

Listing: Searching the List

```
Private Sub TextBox1 TextChanged(...) Handles TextBox1.TextChanged
Dim srchWord As String = TextBox1.Text.Trim
If srchWord.Length = 0 Then Exit Sub
Dim wordIndex As Integer
wordIndex = ListBox1.FindStringExact(srchWord)
If wordIndex >= 0 Then
ListBox1.TopIndex = wordIndex
ListBox1.SelectedIndex = wordIndex
Else
```

```
wordIndex = ListBox1.FindString(srchWord)
If wordIndex >= 0 Then
ListBox1.TopIndex = wordIndex
ListBox1.SelectedIndex = wordIndex
Else
Debug.WriteLine("Item " & srchWord &
" is not in the list")
End If
End If
End Sub
```

The ComboBox Control

The ComboBox control is similar to the ListBox control in the sense that it contains multiple items and the user may select one, but it typically occupies less space onscreen. The ComboBox is practically an expandable ListBox control, which can grow when the user wants to make a selection and retract after the selection is made. Normally, the ComboBox control displays one line with the selected item, as this control doesn't allow multiple item selection. The essential difference, however, between ComboBox and ListBox controls is that the ComboBox allows the user to specify items that don't exist in the list.

Table - Styles of the ComboBox Control

Value	Effect
DropDown	(Default) The control is made up of a drop-down list, which is visible at all times, and a text box. The user can select an item from the list or type a new one in the text box.
DropDownList	This style is a drop-down list from which the user can select one of its items but can't enter a new one. The control displays a single item, and the list is expanded as needed.
Simple	The control includes a text box and a list that doesn't drop down.

	The user can select from the list or type in the text box.

The DropDown and Simple ComboBox controls allow the user to select an item from the list or enter a new one in the edit box of the control. Moreover, they're collapsed by default and they display a single item, unless the user expands the list of items to make a selection. The DropDownList ComboBox is similar to a ListBox control in the sense that it restricts the user to selecting an item (the user cannot enter a new one).

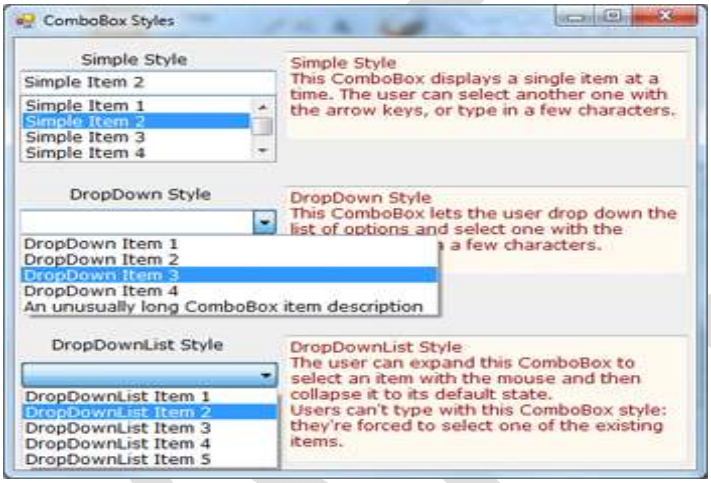


Figure VB.NET ComboBox control's Simple style, DropDown style and DropDownList style.

Adding Items to the ComboBox Control

Although the ComboBox control allows users to enter text in the control's edit box, it doesn't provide a simple mechanism for adding new items at runtime. Let's say you provide a ComboBox with city names. Users can type the first few characters and quickly locate the desired item.

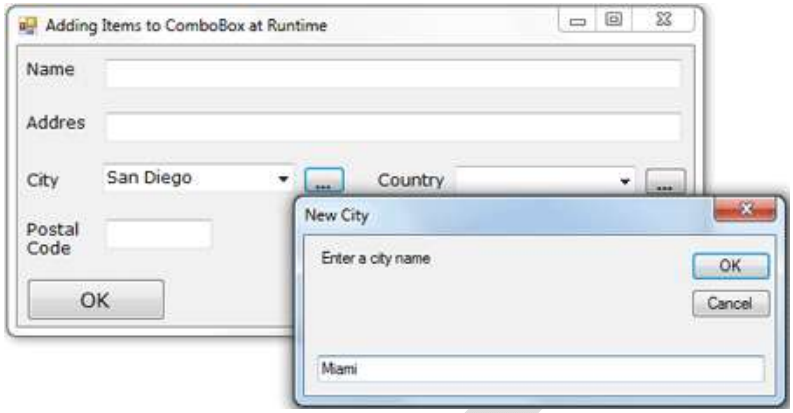


Figure - Adding items to ComboBox control at runtime - VB.NET

VB.NET ComboBox Control Example

The ellipsis button next to the City ComboBox control prompts the user for the new item via the InputBox() function. Then it searches the Items collection of the control via the FindString method, and if the new item isn't found, it's added to the control. Then the code selects the new item in the list. To do so, it sets the control's SelectedIndex property to the value returned by the Items.Add method, or the value returned by the FindString method, depending on whether the item was located or added to the list. Listing 4.14 shows the code behind the ellipsis button.

Listing : Adding a New Item to the ComboBox Control at Runtime

```
Private Sub Button1 Click(...) Button1.Click
Dim itm As String
itm = InputBox("Enter new item", "New Item")
If itm.Trim <> "" Then AddElement(itm)
End Sub
```

The AddElement() subroutine, which accepts a string as an argument and adds it to the control, is shown in Listing 4.15. If the item doesn't exist in the control, it's added to the Items collection. If the item is a member of the Items collection, it's selected. As you will see, the same subroutine will be used by the second method for adding items to the control at runtime.

Listing: The AddElement() Subroutine

```
Sub AddElement(ByVal newItem As String)
Dim idx As Integer
```

```
If ComboBox1.FindString(newItem) > 0 Then
idx = ComboBox1.FindString(newItem)
Else
idx = ComboBox1.Items.Add(newItem)
End If
ComboBox1.SelectedIndex = idx
End Sub
```

You can also add new items at runtime by adding the same code in the control's LostFocus event handler:

```
Private Sub ComboBox1 LostFocus(...) Handles ComboBox1.LostFocus
Dim newItem As String = ComboBox1.Text
AddElement(newItem)
End Sub
```

The ScrollBar and TrackBar Controls

The ScrollBar and TrackBar controls let the user specify a magnitude by scrolling a selector between its minimum and maximum values. In some situations, the user doesn't know in advance the exact value of the quantity to specify (in which case, a text box would suffice), so your application must provide a more-flexible mechanism for specifying a value, along with some type of visual feedback.

The vertical scroll bar that lets a user move up and down a long document is a typical example of the use of the ScrollBar control. The scroll bar and visual feedback are the prime mechanisms for repositioning the view in a long document or in a large picture thatwon't fit entirely in its window.

The TrackBar control is similar to the ScrollBar control, but it doesn't cover a continuous range of values. The TrackBar control has a fixed number of tick marks, which the developer can label. Users can place the slider's indicator to he desired value.Whereas the ScrollBar control relies on some visual feedback outside the control to help the user position the

indicator to the desired value, the TrackBar control forces the user to select from a range of valid values.

The ScrollBar Control

There's no ScrollBar control per se in the Toolbox; instead, there are two versions of it: the HScroll-Bar and VScrollBar controls. They differ only in their orientation, but because they share the same members, I will refer to both controls collectively as ScrollBar controls. Actually, both controls inherit from the ScrollBar control, which is an abstract control: It can be used to implement vertical and horizontal scroll bars, but it can't be used directly on a form. Moreover, the HScrollBar and VScrollBar controls are not displayed in the Common Controls tab of the Toolbox. You have to open the All Windows Forms tab to locate these two controls.

Minimum - The control's minimum value. The default value is 0, but because this is an Integer value, you can set it to negative values as well.

Maximum - The control's maximum value. The default value is 100, but you can set it to any value that you can represent with the Integer data type.

Value - The control's current value, specified by the indicator's position.

The ScrollBar Control Colors Exercise

Figure 4.8 shows the main form of the Colors sample project, which lets the user specify a color by manipulating the value of its basic colors (red, green, and blue) through scroll bars. Each basic color is controlled by a scroll bar and has a minimum value of 0 and a maximum value of 255. If you aren't familiar with color definition in the Windows environment, see the section "Specifying Colors" in Chapter, "Manipulating Images and Bitmaps."

Figure - When the ScrollBar is moved the corresponding color is displayed

The ScrollBar Control's Events

The user can change the ScrollBar control's value in three ways: by clicking the two arrows at its ends, by clicking the area between the indicator and the arrows, and by dragging the indicator with the mouse. You can monitor the changes of the ScrollBar's value from within your code by using two events: ValueChanged and Scroll. Both events are fired every time the indicator's position is changed. If you change the control's value from within your code, only the ValueChanged event will be fired.

The Scroll event can be fired in response to many different actions, such as the scrolling of the indicator with the mouse, a click on one of the two buttons at the ends of the scroll bars, and so on. If you want to know the action that caused this event, you can examine the Type property of the second argument of the event handler. The settings of the e.Type property are members of the ScrollEventType enumeration (LargeDecrement, SmallIncrement, Track, and so on).

Handling the Events in the Colors Application

The Colors application demonstrates how to program the two events of the ScrollBar control. The two PictureBox controls display the color designed with the three scroll bars. The left PictureBox is colored from within the Scroll event, whereas the other one is colored from within the ValueChanged event. Both events are fired as the user scrolls the scrollbar's indicator, but in the Scroll event handler of the three scroll bars, the code examines the value of the e.Type property and reacts to it only if the event was fired because the scrolling of the indicator has ended. For all other actions, the event handler doesn't update the color of the left PictureBox.

Listing: Programming the ScrollBar Control's Scroll Event

```
Private Sub redBar Scroll(...) Handles redBar.Scroll
If e.Type = ScrollEventType.EndScroll Then
ColorBox1()
lblRed.Text = "RED " & redBar.Value.ToString("###")
End If
End Sub
Private Sub redBar ValueChanged(...) Handles redBar.ValueChanged
ColorBox2()
End Sub
```

The ColorBox1() and ColorBox2() subroutines update the color of the two PictureBox controls by setting their background colors. You can open the Colors project in Visual Studio and examine the code of these two routines.

The TrackBar Control

The TrackBar control is similar to the ScrollBar control, but it lacks the granularity of ScrollBar. Suppose that you want the user of an application to supply a value in a specific range, such as the speed of a moving object. Moreover, you don't want to allow extreme precision; you need only a few settings, as shown in the examples in this page. The user can set the control's value by sliding the indicator or by clicking on either side of the indicator.

Granularity is how specific youwant to be inmeasuring. Inmeasuring distances between towns, a granularity of amile is quite adequate. In measuring (or specifying) the dimensions of a building, the granularity could be on the order of a foot or an inch. The TrackBar control lets you set the type of granularity that's necessary for your application.

Similar to the ScrollBar control, SmallChange and LargeChange properties are available. SmallChange is the smallest increment by which the Slider value can change. The user can change the slider by the SmallChange value only by sliding the indicator. (Unlike the ScrollBar control, there are no arrows at the two ends of the Slider control.) To change the Slider's value by LargeChange, the user can click on either side of the indicator.

The TrackBar Control Inches Exercise

The Figure demonstrates a typical use of the TrackBar control. The form in the figure is an element of a program's user interface that lets the user specify a distance between 0 and 10 inches in increments of 0.2 inches. As the user slides the indicator, the current value is displayed on a Label control below the TrackBar. If you open the Inches application, you'll notice that there are more stops than there are tick marks on the control. This is made possible with the TickFrequency property, which determines the frequency of the visible tick marks.

Figure - A typical use of TrackBar control in VB.NET - The Inches Application

The properties of the TrackBar control in the Inches application are as follows:

Minimum = 0

Maximum = 50

SmallChange = 1

LargeChange = 5

TickFrequency = 5

The TrackBar needs to cover a range of 10 inches in increments of 0.2 inches. If you set the SmallChange property to 1, you have to set LargeChange to 5. Moreover, the TickFrequency is set to 5, so there will be a total of five divisions in every inch. The numbers below the tick marks were placed there with properly aligned Label controls.

```
Private Sub TrackBar1 ValueChanged(...)Handles TrackBar1.ValueChanged
lblInches.Text = "Length in inches = " &
```

```
Format(TrackBar1.Value / 5, "#.00")
End Sub
```

The Label controls below the tick marks can also be used to set the value of the control. Every time you click one of the labels, the following statement sets the TrackBar control's value. Notice that all the Label controls' Click events are handled by a common handler:

```
Private Sub Label Click(...) Handles Label1.Click, Label9.Click
TrackBar1.Value = sender.text * 5
End Sub
```

Common Dialog Controls

The common dialog controls are invisible at runtime, and they're not placed on your forms, because they're implemented as modal dialog boxes and they're displayed as needed. You simply add them to the project by double-clicking their icons in the Toolbox; a new icon appears in the components tray of the form, just below the Form Designer. The common dialog controls in the Toolbox are the following:

- **OpenFileDialog** - Lets users select a file to open. It also allows the selection of multiple files for applications that must process many files at once.
- **SaveFileDialog** - Lets users select or specify the path of a file in which the current document will be saved.
- **ColorDialog** - Lets users select a color from a list of predefined colors or specify custom colors. FontDialog Lets users select a typeface and style to be applied to the current text selection. The Font dialog box has an Apply button, which you can intercept from within your code and use to apply the currently selected font to the text without closing the dialog box.

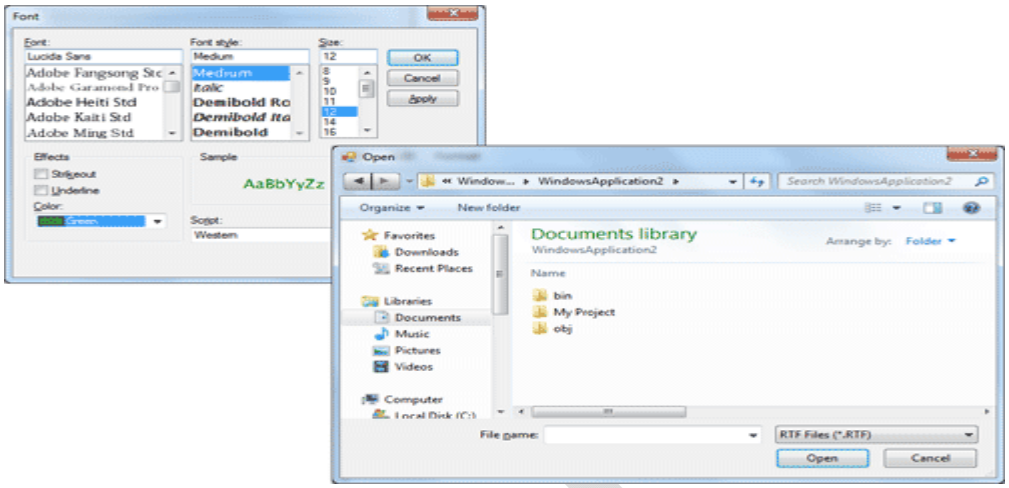


Figure - Common Font and Open dialog controls

There are three more common dialog controls: the PrintDialog, PrintPreviewDialog, and PageSetupDialog controls. These controls are discussed in detail in Chapter, "Printing with Visual Basic 2008," in the context of VB's printing capabilities.

Using the Common Dialog Controls

To display any of the common dialog boxes from within your application, you must first add an instance of the appropriate control to your project. Then you must set some basic properties of the control through the Properties window. Most applications set the control's properties from within the code because common dialogs interact closely with the application. When you call the Color common dialog, for example, you should preselect a color from within your application and make it the default selection on the control. When prompting the user for the color of the text, the default selection should be the current setting of the control's ForeColor property. Likewise, the Save dialog box must suggest a filename when it first pops up (or the file's extension, at least).

Here is the sequence of statements used to invoke the Open common dialog and retrieve the selected filename:

```
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
fileName = OpenFileDialog1.FileName
' Statements to open the selected file
End If
```

The ShowDialog method returns a value indicating how the dialog box was closed. You should read this value from within your code and ignore the settings of the dialog box if the operation was cancelled.

The variable fileName in the preceding code segment is the full pathname of the file selected by the user. You can also set the FileName property to a filename, which will be displayed when the Open dialog box is first opened:

```
OpenFileDialog1.FileName = "C:\WorkFiles\Documents\Document1.doc"
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
fileName = OpenFileDialog1.FileName
' Statements to open the selected file
End If
```

Similarly, you can invoke the Color dialog box and read the value of the selected color by using the following statements:

```
ColorDialog1.Color = TextBox1.BackColor
If ColorDialog1.ShowDialog = DialogResult.OK Then
TextBox1.BackColor = ColorDialog1.Color
End If
```

The ShowDialog method is common to all controls. The Title property is also common to all controls and it's the string displayed in the title bar of the dialog box. The default title is the name of the dialog box (for example, Open, Color, and so on), but you can adjust it from within your code with a statement such as the following:

```
ColorDialog1.Title = "Select Drawing Color"
```

Color Dialog Box Control

The Color dialog box, shown in Figure 4.11, is one of the simplest dialog boxes. Its Color property returns the color selected by the user or sets the initially selected color when the user opens the dialog box.

The following statements set the initial color of the ColorDialog control, display the dialog box, and then use the color selected in the control to fill the form. First, place a ColorDialog control in the form and then insert the following statements in a button's Click event handler:

```
Private Sub Button1 Click(...) Handles Button1.Click
ColorDialog1.Color = Me.BackColor
If ColorDialog1.ShowDialog =
Windows.Forms.DialogResult.OK Then
Me.BackColor = ColorDialog1.Color
End If
End Sub
```

The following sections discuss the basic properties of the ColorDialog control.

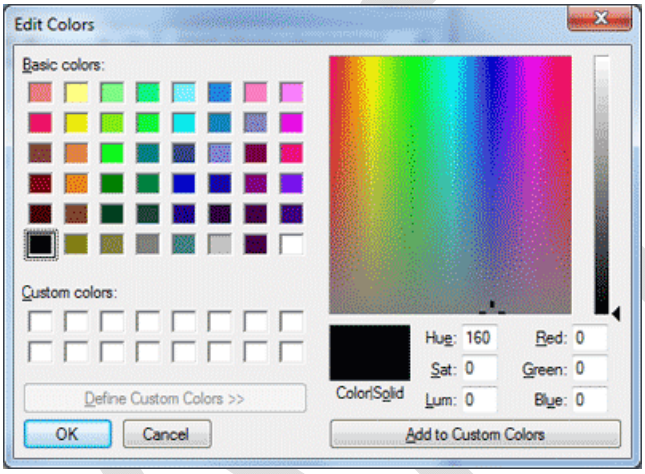


Figure - The Color Dialog Box

AllowFullOpen

Set this property to True if you want users to be able to open the dialog box and define their own custom colors, like the one shown in Figure 8.2. The AllowFullOpen property doesn't open the custom section of the dialog box; it simply enables the Define Custom Colors button in the dialog box. Otherwise, this button is disabled.

AnyColor

This property is a Boolean value that determines whether the dialog box displays all available colors in the set of basic colors.

Color

This is the color specified on the control. You can set it to a color value before showing the dialog box to suggest a reasonable selection. On return, read the value of the same property to find out which color was picked by the user in the control:

```
ColorDialog1.Color = Me.BackColor
If ColorDialog1.ShowDialog = DialogResult.OK Then
Me.BackColor = ColorDialog1.Color
End If
```

CustomColors

This property indicates the set of custom colors that will be shown in the dialog box. The Color dialog box has a section called Custom Colors, in which you can display 16 additional custom colors. The CustomColors property is an array of integers that represent colors. To display three custom colors in the lower section of the Color dialog box, use a statement such as the following:

```
Dim colors() As Integer = {222663, 35453, 7888}
ColorDialog1.CustomColors = colors
```

You'd expect that the CustomColors property would be an array of Color values, but it's not. You can't create the array CustomColors with a statement such as this one:

```
Dim colors() As Color = {Color.Azure, Color.Navy, Color.Teal}
```

Because it's awkward to work with numeric values, you should convert color values to integer values by using a statement such as the following:

```
Color.Navy.ToArgb
```

The preceding statement returns an integer value that represents the color navy. This value, however, is negative because the first byte in the color value represents the transparency of the color. To get the value of the color, you must take the absolute value of the integer value returned by the previous expression. To create an array of integers that represent color values, use a statement such as the following:

```
Dim colors() As Integer = {Math.Abs(Color.Gray.ToArgb),
Math.Abs(Color.Navy.ToArgb), Math.Abs(Color.Teal.ToArgb)}
```

Now you can assign the colors array to the CustomColors property of the control, and the colors will appear in the Custom Colors section of the Color dialog box.

SolidColorOnly

This indicates whether the dialog box will restrict users to selecting solid colors only. This setting should be used with systems that can display only 256 colors. Although today few

systems can't display more than 256 colors, some interfaces are limited to this number. When you run an application through Remote Desktop, for example, only the solid colors are displayed correctly on the remote screen, regardless of the remote computer's graphics card (and that's for efficiency reasons).

Font Dialog Box Control

The Font dialog box, shown in Figure 4.12, lets the user review and select a font and then set its size and style. Optionally, users can also select the font's color and even apply the current settings to the selected text on a control of the form without closing the dialog box, by clicking the Apply button.

```
FontDialog1.Font = TextBox1.Font
If FontDialog1.ShowDialog = DialogResult.OK Then
TextBox1.Font = FontDialog1.Font
End If
```

Use the following properties to customize the Font dialog box before displaying it.

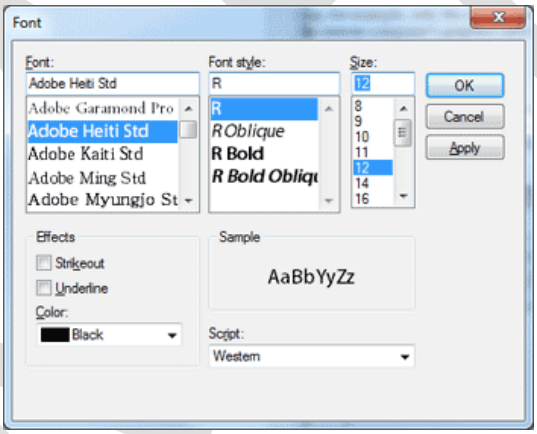


Figure - The Font Dialog Control

AllowScriptChange

This property is a Boolean value that indicates whether the Script combo box will be displayed in the Font dialog box. This combo box allows the user to change the current character set and select a non-Western language (such as Greek, Hebrew, Cyrillic, and so on).

AllowVerticalFonts

This property is a Boolean value that indicates whether the dialog box allows the display and selection of both vertical and horizontal fonts. Its default value is False, which displays only horizontal fonts.

Color, ShowColor

The Color property sets or returns the selected font color. To enable users to select a color for the font, you must also set the ShowColor property to True.

FixedPitchOnly

This property is a Boolean value that indicates whether the dialog box allows only the selection of fixed-pitch fonts. Its default value is False, which means that all fonts (fixed- and variable-pitch fonts) are displayed in the Font dialog box. Fixed-pitch fonts, or monospaced fonts, consist of characters of equal widths that are sometimes used to display columns of numeric values so that the digits are aligned vertically.

Font

This property is a Font object. You can set it to the preselected font before displaying the dialog box and assign it to a Font property upon return. You've already seen how to preselect a font and how to apply the selected font to a control from within your application.
You can also create a new Font object and assign it to the control's Font property. Upon return, the TextBox control's Font property is set to the selected font:

```
Dim newFont As Font("Verdana", 12, FontStyle.Underline)
FontDialog1.Font = newFont
If FontDialog1.ShowDialog() = DialogResult.OK Then
TextBox1.ForeColor = FontDialog1.Color
End If
```

FontMustExist

This property is a Boolean value that indicates whether the dialog box forces the selection of an existing font. If the user enters a font name that doesn't correspond to a name in the list of available fonts, a warning is displayed. Its default value is True, and there's no reason to change it.

MaxSize, MinSize

These two properties are integers that determine the minimum and maximum point size the user can specify in the Font dialog box. Use these two properties to prevent the selection of extremely large or extremely small font sizes, because these fonts might throw off a well-balanced interface (text will overflow in labels, for example).

ShowApply

This property is a Boolean value that indicates whether the dialog box provides an Apply button. Its default value is False, so the Apply button isn't normally displayed. If you set this property to True, you must also program the control's Apply event — the changes aren't applied automatically to any of the controls in the current form.

The following statements display the Font dialog box with the Apply button:

```
Private Sub Button2 Click(...) Handles Button2.Click
FontDialog1.Font = TextBox1.Font
FontDialog1.ShowApply = True
If FontDialog1.ShowDialog = DialogResult.OK Then
TextBox1.Font = FontDialog1.Font
End If
End Sub
```

The FontDialog control raises the Apply event every time the user clicks the Apply button. In this event's handler, you must read the currently selected font and use it in the form, so that users can preview the effect of their selection:

```
Private Sub FontDialog1 Apply(...) Handles FontDialog1.Apply
TextBox1.Font = FontDialog1.Font
End Sub
```

ShowEffects

This property is a Boolean value that indicates whether the dialog box allows the selection of special text effects, such as strikethrough and underline. The effects are returned to the application as attributes of the selected Font object, and you don't have to do anything special in your application.

Open Dialog Box and Save Dialog Box Controls

Open and Save As, the two most widely used common dialog boxes (see Figure 4.13), are implemented by the OpenFileDialog and SaveFileDialog controls. Nearly every application prompts users for filenames, and the .NET Framework provides two controls for this purpose. The two dialog boxes are nearly identical, and most of their properties are common, so we'll start with the properties that are common to both controls.

When either of the two controls is displayed, it rarely displays all the files in any given folder. Usually the files displayed are limited to the ones that the application recognizes so that users can easily spot the file they want. The Filter property limits the types of files that will appear in the Open or Save As dialog box.

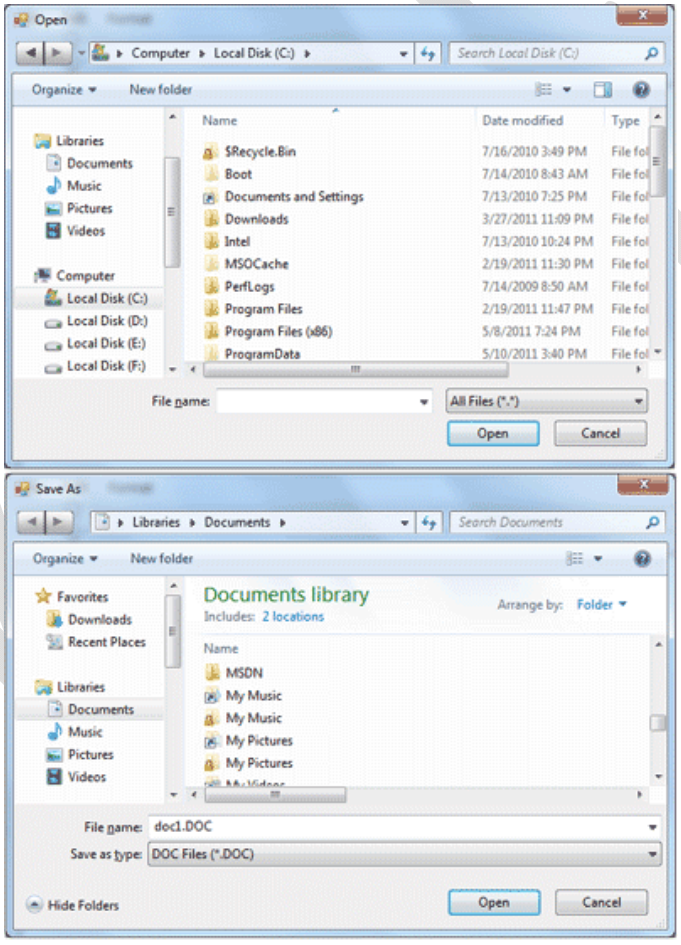


Figure - The OpenDialog and SaveDialog controls

The extension of the default file type for the application is described by the DefaultExtension property, and the list of the file types displayed in the Save As Type box is determined by the Filter property.

To prompt the user for a file to be opened, use the following statements. The Open dialog box displays the files with the extension .bin only.

```
OpenFileDialog1.DefaultExt = ".bin"
OpenFileDialog1.AddExtension = True
OpenFileDialog1.Filter = "Binary Files|*.bin"
If OpenFileDialog1.ShowDialog() =
Windows.Forms.DialogResult.OK Then
Debug.WriteLine(OpenFileDialog1.FileName)
End If
```

The following sections describe the properties of the OpenFileDialog and SaveFileDialog controls.

AddExtension

This property is a Boolean value that determines whether the dialog box automatically adds an extension to a filename if the user omits it. The extension added automatically is the one specified by the DefaultExtension property, which you must set before calling the ShowDialog method. This is the default extension of the files recognized by your application.

CheckFileExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user enters the name of a file that does not exist in the Open dialog box, or if the user enters the name of a file that exists in the Save dialog box.

CheckPathExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user specifies a path that does not exist, as part of the user-supplied filename.

DefaultExt

This property sets the default extension for the filenames specified on the control. Use this property to specify a default filename extension, such as .txt or .doc, so that when a file with no extension is specified by the user, the default extension is automatically appended to the filename. You must also set the AddExtension property to True. The default extension property starts with the period, and it's a string — for example, .bin.

DereferenceLinks

This property indicates whether the dialog box returns the location of the file referenced by the shortcut or the location of the shortcut itself. If you attempt to select a shortcut on your desktop when the DereferenceLinks property is set to False, the dialog box will return to your application a value such as C:\WINDOWS\SYSTEM32\lnkstub.exe, which is the name of the shortcut, not the name of the file represented by the shortcut. If you set the DereferenceLinks property to True, the dialog box will return the actual filename represented by the shortcut, which you can use in your code.

FileName

Use this property to retrieve the full path of the file selected by the user in the control. If you set this property to a filename before opening the dialog box, this value will be the proposed filename. The user can click OK to select this file or select another one in the control. The two controls provide another related property, the FileNames property, which returns an array of filenames. To find out how to allow the user to select multiple files, see the discussion of the MultipleFiles and FileNames properties in ‘‘VB 2008 at Work: Multiple File Selection" at the end of this section.

Filter

This property is used to specify the type(s) of files displayed in the dialog box. To display text files only, set the Filter property to Text files|*.txt. The pipe symbol separates the description of the files (what the user sees) from the actual extension (how the operating system distinguishes the various file types).

If you want to display multiple extensions, such as .BMP, .GIF, and .JPG, use a semicolon to separate extensions with the Filter property. Set the Filter property to the string Images|*.BMP; *.GIF;*.JPG to display all the files of these three types when the user selects Images in the Save As Type combo box, under the box with the filename.

Don't include spaces before or after the pipe symbol because these spaces will be displayed on the dialog box. In the Open dialog box of an image-processing application, you'll probably provide options for each image file type, as well as an option for all images:

```
OpenFileDialog1.Filter =
"Bitmaps|*.BMP|GIF Images|*.GIF|" &
"JPEG Images|*.JPG|All Images|*.BMP;*.GIF;*.JPG"
```

FilterIndex

When you specify more than one file type when using the Filter property of the Open dialog box, the first file type becomes the default. If you want to use a file type other than the first one, use the FilterIndex property to determine which file type will be displayed as the default when the Open dialog box is opened. The index of the first type is 1, and there's no reason to ever set this property to 1. If you use the Filter property value of the example in the preceding section and set the FilterIndex property to 2, the Open dialog box will display GIF files by default.

InitialDirectory

This property sets the initial folder whose files are displayed the first time that the Open and Save dialog boxes are opened. Use this property to display the files of the application's folder or to specify a folder in which the application stores its files by default. If you don't specify an initial folder, the dialog box will default to the last folder where the most recent file was opened or saved. It's also customary to set the initial folder to the application's path by using the following statement:

OpenFileDialog1.InitialDirectory = Application.ExecutablePath

The expression Application.ExecutablePath returns the path in which the application's executable file resides.

RestoreDirectory

Every time the Open and Save As dialog boxes are displayed, the current folder is the one that was selected by the user the last time the control was displayed. The RestoreDirectory property is a Boolean value that indicates whether the dialog box restores the current directory before closing. Its default value is False, which means that the initial directory is not restored automatically. The InitialDirectory property overrides the RestoreDirectory property.

The following four properties are properties of the OpenFileDialog control only: FileNames, MultiSelect, ReadOnlyChecked, and ShowReadOnly.

FileNames

If the Open dialog box allows the selection of multiple files (see the later section "VB 2008 at Work: Multiple File Selection"), the FileNames property contains the pathnames of all selected files. FileNames is a collection, and you can iterate through the filenames with an

enumerator. This property should be used only with the OpenFileDialog control, even though the SaveFileDialog control exposes a FileNames property.

MultiSelect

This property is a Boolean value that indicates whether the user can select multiple files in the dialog box. Its default value is False, and users can select a single file. When the MultiSelect property is True, the user can select multiple files, but they must all come from the same folder (you can't allow the selection of multiple files from different folders). This property is unique to the OpenFileDialog control.

ReadOnlyChecked, ShowReadOnly

The ReadOnlyChecked property is a Boolean value that indicates whether the Read-Only check box is selected when the dialog box first pops up (the user can clear this box to open a file in read/write mode). You can set this property to True only if the ShowReadOnly property is also set to True. The ShowReadOnly property is also a Boolean value that indicates whether the Read-Only check box is available..

The OpenFile and SaveFile Methods

The OpenFileDialog control exposes the OpenFile method, which allows you to quickly open the selected file. Likewise, the SaveFileDialog control exposes the SaveFile method, which allows you to quickly save a document to the selected file.

OpenDialog and SaveDialog controls example: Multiple File Selection

The Open dialog box allows the selection of multiple files. This feature can come in handy when you want to process files en masse. You can let the user select many files, usually of the same type, and then process them one at a time. Or, you might want to prompt the user to select multiple files to be moved or copied.

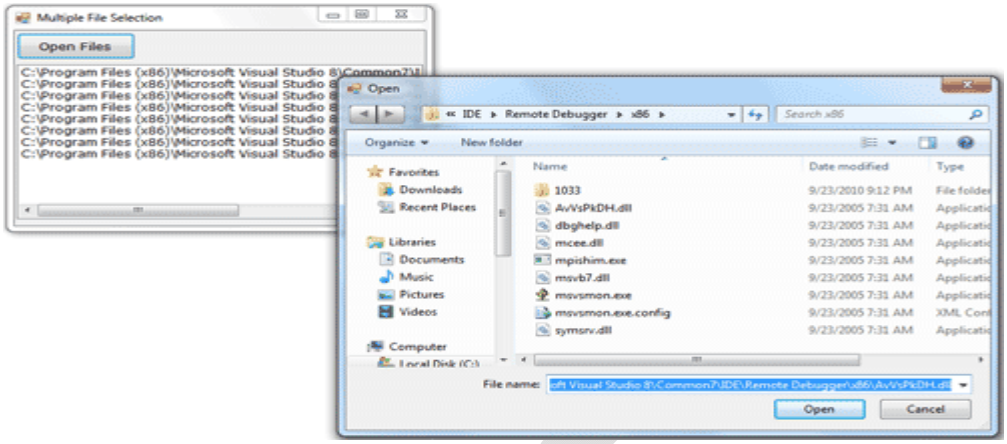


Figure - Selecting multiple files in an open dialog box - Visual Basic

The code behind the Open Files button is shown in Listing 4.17. In this example, I used the array's enumerator to iterate through the elements of the FileNames array. You can use any of the methods discussed in the section "Arrays in Visual basic 2008" to iterate through the array.

Listing: Processing Multiple Selected Files

```
Private Sub bttnFile Click(...) Handles bttnFile.Click
OpenFileDialog1.Multiselect = True
OpenFileDialog1.ShowDialog()
Dim filesEnum As IEnumerator
ListBox1.Items.Clear()
filesEnum = OpenFileDialog1.FileNames.GetEnumerator()
While filesEnum.MoveNext
ListBox1.Items.Add(filesEnum.Current)
End While
End Sub
```

Print Dialog Box Control

A PrintDialog control is used to open the Windows Print Dialog and let user select the printer, set printer and paper properties and print a file. A typical Open File Dialog looks like Figure 1 where you select a printer from available printers, set printer properties, set print range, number of pages and copies and so on. Clicking on OK button sends the document to the printer.

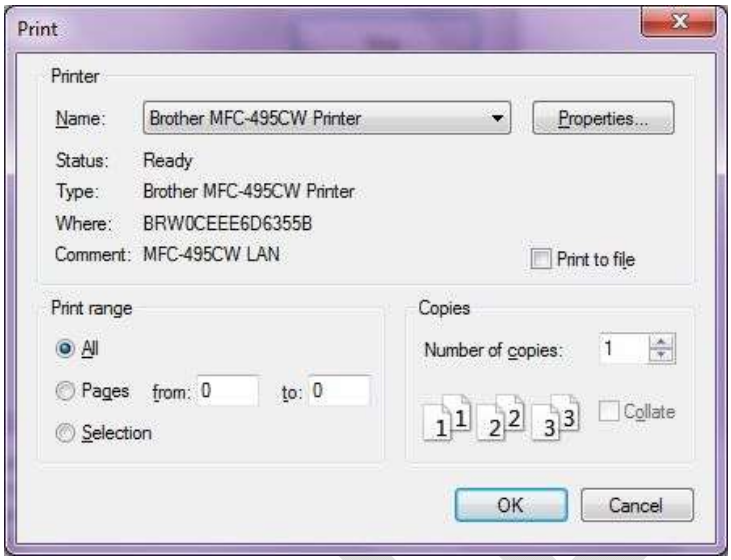


Figure – Print Dialog Control

Creating a PrintDialog

We can create a PrintDialog at design-time as well as at run-time.

Design-time

To create a PrintDialog control at design-time, you simply drag and drop a PrintDialog control from Toolbox to a Form in Visual Studio. After you drag and drop a PrintDialog on a Form, the PrintDialog looks like Figure 2.

Figure – design time

Run-time

Creating a PrintDialog control at run-time is simple. First step is to create an instance of PrintDialog class and then call the ShowDialog method. The following code snippet creates a PrintDialog control.

```
Dim PrintDialog1 As New PrintDialog()
PrintDialog1.ShowDialog()
```

Printing Documents

PrintDocument object represents a document to be printed. Once a PrintDocument is created, we can set the Document property of PrintDialog as this document. After that we can also set other properties. The following code snippet creates a PrintDialog and sends some text to a printer.

```
Imports System.Drawing.Printing

Public Class Form1
    Private Sub PrintButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles PrintButton.Click
      Dim printDlg As New PrintDialog()
      Dim printDoc As  New PrintDocument()
      printDoc.DocumentName = "Print Document"
      printDlg.Document = printDoc
      printDlg.AllowSelection = True
      printDlg.AllowSomePages = True
       If (printDlg.ShowDialog() = DialogResult.OK) Then
        printDoc.Print()
      End If
    End Sub
End Class
```

The RichTextBox Control

The **RichTextBox** control is the core of a full-blown word processor. It provides all the functionality of a TextBox control; it can handle multiple typefaces, sizes, and attributes, and offers precise control over the margins of the text (see Figure 4.16). You can even place images in your text on a RichTextBox control (although you won't have the kind of control over the embedded images that you have with Microsoft Word).

The fundamental property of the **RichTextBox** control is its Rtf property. Similar to the Text property of the TextBox control, this property is the text displayed on the control. Unlike the Text property, however, which returns (or sets) the text of the control but doesn't contain formatting information, the Rtf property returns the text along with any formatting information.

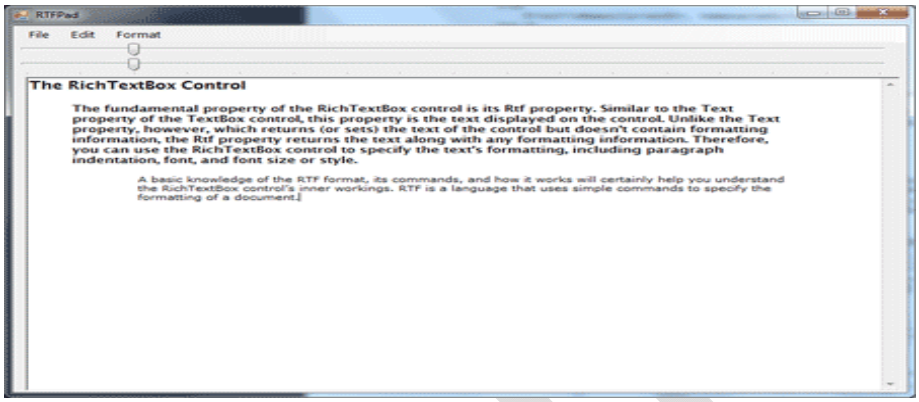


Figure - A word processor based on the functionality of the RichTextBox control

The RTF Language

A basic knowledge of the RTF format, its commands, and how it works will certainly help you understand the RichTextBox control's inner workings. RTF is a language that uses simple commands to specify the formatting of a document. These commands, or tags, are ASCII strings, such as \par (the tag that marks the beginning of a new paragraph) and \b (the tag that turns on the bold style). And this is where the value of the RTF format lies. RTF documents don't contain special characters and can be easily exchanged among different operating systems and computers, as long as there is an RTF-capable application to read the document. Let's look at an RTF document in action.

Open the WordPad application (choose *Start* > *Programs* > *Accessories* > *WordPad*) and enter a few lines of text (see Figure 4.17). Select a few words or sentences, and format them in different ways with any of WordPad's formatting commands. Then save the document in RTF format: Choose *File* > *Save As*, select Rich Text Format, and then save the file as Document.rtf. If you open this file with a text editor such as Notepad, you'll see the actual RTF code that produced the document. A section of the RTF file for the document shown in Figure 4.17 is shown in Listing 4.20.

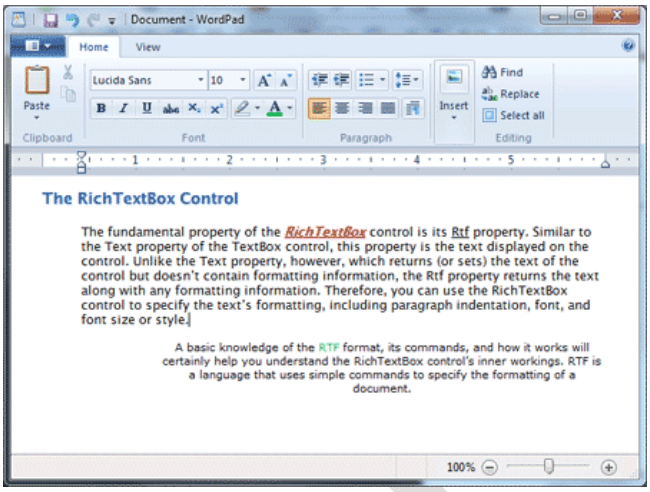


Figure - The formatting applied to the text by using WordPad's commands is stored along with the text in RTF format.

The RichTextBox's Properties

The **RichTextBox** control provides properties for manipulating the selected text on the control. The names of these properties start with the Selection or Selected prefix, and the most commonly used ones are shown in Table 4.5. Some of these properties are discussed in further detail in following sections.

SelectedText

The **SelectedText** property represents the selected text, whether it was selected by the user via the mouse or from within your code. To assign the selected text to a variable, use the following statement:

```
selText=RichTextbox1.SelectedText
```

You can also modify the selected text by assigning a new value to the **SelectedText** property. The following statement converts the selected text to uppercase:

```
RichTextbox1.SelectedText =
RichTextbox1.SelectedText.ToUpper
```

You can assign any string to the **SelectedText** property. If no text is selected at the time, the statement will insert the string at the location of the pointer.

Table - RichTextBox Properties for Manipulating Selected Text

Property	What It Manipulates
SelectedText	The selected text
SelectedRtf	The RTF code of the selected text
SelectionStart	The position of the selected text's first character
SelectionLength	The length of the selected text
SelectionFont	The font of the selected text
SelectionColor	The color of the selected text
SelectionBackColor	The background color of the selected text
SelectionAlignment	The alignment of the selected text
SelectionIndent, SelectionRightIndent, SelectionHangingIndent	The indentation of the selected text
RightMargin	The distance of the text's right margin from the left edge of the control
SelectionTabs	An array of integers that sets the tab stop positions in the control
SelectionBullet	Whether the selected text is bulleted
BulletIndent	The amount of bullet indent for the selected text

SelectionStart, SelectionLength

SelectionLength, report (or set) the position of the first selected character in the text and the length of the selection, respectively, regardless of the formatting of the selected text. One obvious use of these properties is to select (and highlight) some text on the control:

```
RichTextBox1.SelectionStart = 0
RichTextBox1.SelectionLength = 100
```

You can also use the **Select** method, which accepts as arguments the starting location and the length of the text to be selected.

SelectionAlignment

Use this property to read or change the alignment of one or more paragraphs. This property's value is one of the members of the HorizontalAlignment enumeration: Left, Right, and Center. Users don't have to select an entire paragraph to align it; just placing the pointer anywhere in the paragraph will do the trick, because you can't align part of the paragraph.

SelectionIndent, SelectionRightIndent, SelectionHangingIndent

These properties allow you to change the margins of individual paragraphs. The Selection Indent property sets (or returns) the amount of the text's indentation from the left edge of the control. The SelectionRightIndent property sets (or returns) the amount of the text's indentation from the right edge of the control. The SelectionHangingIndent property indicates the indentation of each paragraph's first line with respect to the following lines of the same paragraph. All three properties are expressed in pixels.

The SelectionHangingIndent property includes the current setting of the SelectionIndent property. If all the lines of a paragraph are aligned to the left, the SelectionIndent property can have any value (this is the distance of all lines from the left edge of the control), but the SelectionHangingIndent property must be zero. If the first line of the paragraph is shorter than the following lines, the SelectionHangingIndent has a negative value. Figure 4.18 shows several differently formatted paragraphs. The settings of the SelectionIndent and SelectionHangingIndent properties are determined by the two sliders at the top of the form.

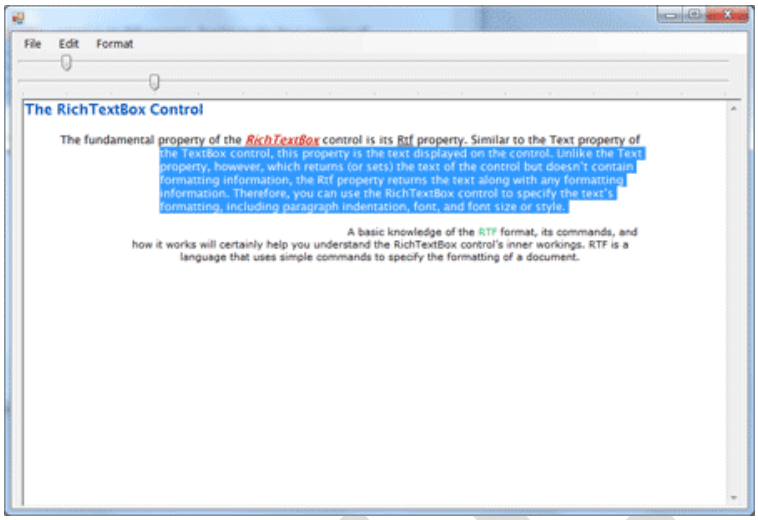


Figure - Various combinations of the **SelectionIndent** *and* **SelectionHangingIndent** *properties produce all possible paragraph styles.*

SelectionBullet, BulletIndent

You use these properties to create a list of bulleted items. If you set the **SelectionBullet** property to True, the selected paragraphs are formatted with a bullet style, similar to the **<ul>** tag in HTML. To create a list of bulleted items, select them from within your code and assign the value True to the **SelectionBullet** property. To change a list of bulleted items back to normal text, make the same property False.

The paragraphs formatted as bullets are also indented from the left by a small amount. To set the amount of the indentation, use the **BulletIndent** property, which is also expressed in pixels.

SelectionTabs

Use this property to set the tab stops in the RichTextBox control. The Selection tab should be set to an array of integer values, which are the absolute tab positions in pixels. Use this property to set up a RichTextBox control for displaying tab-delimited data.

Methods of the RichTextBox control

The first two methods of the RichTextBox control you need to know are **SaveFile** and **LoadFile**. The **SaveFile** method saves the contents of the control to a disk file, and the **LoadFile** method loads the control from a disk file.

SaveFile

The syntax of the SaveFile method is as follows:

RichTextBox1.SaveFile(path, filetype)

where path is the path of the file in which the current document will be saved. By default, the SaveFile method saves the document in RTF format and uses the .RTF extension. You can specify a different format by using the second optional argument, which can take on the value of one of the members of the RichTextBoxStreamType enumeration, described in Table 4.6.

Table - The RichTextBoxStreamType Enumeration

Format	Effect
PlainText	Stores the text on the control without any formatting
RichNoOLEObjs	Stores the text without any formatting and ignores any embedded OLE objects
RichText	Stores the text in RTF format (text with embedded RTF commands)
TextTextOLEObjs	Stores the text along with the embedded OLE objects
UnicodePlainText	Stores the text in Unicode format

LoadFile

Similarly, the **LoadFile** method loads a text or RTF file to the control. Its syntax is identical to the syntax of the SaveFile method:

RichTextBox1.LoadFile(path, filetype)

The filetype argument is optional and can have one of the values of the **RichTextBoxStreamType** enumeration. Saving and loading files to and from disk files is as simple as presenting a Save or Open common dialog to the user and then calling one of the **SaveFile** or **LoadFile** methods with the filename returned by the common dialog box.

Select, SelectAll

The **Select** method selects a section of the text on the control, similar to setting the **SelectionStart** and **SelectionLength** properties. The **Select** method accepts two arguments: the location of the first character to be selected and the length of the selection:

RichTextBox1.Select(start, length)

The SelectAll method accepts no arguments and it selects all the text on the control.

Editing Features in RichTextBox

The RichTextBox control provides all the text-editing features you'd expect to find in a text-editing application, similar to the TextBox control. Among its more-advanced features, the RichTextBox control provides the **AutoWordSelection** property, which controls how the control selects text. If it's True, the control selects a word at a time.

In addition to formatted text, the RichTextBox control can handle object linking and embedding (OLE) objects. You can insert images in the text by pasting them with the Paste method. The **Paste** method doesn't require any arguments; it simply inserts the contents of the Clipboard at the current location in the document.

CanUndo, CanRedo

These two properties are Boolean values you can read to find out whether there's an operation that can be undone or redone. If they're False, you must disable the corresponding menu command from within your code. The following statements disable the Undo command if there's no action to be undone at the time (**EditUndo** is the name of the Undo command on the Edit menu):

```
If RichTextBox1.CanUndo Then
EditUndo.Enabled = True
Else
EditUndo.Enabled = False
End If
```

UndoActionName, RedoActionName

These two properties return the name of the action that can be undone or redone. The most common value of both properties is **Typing**, which indicates that the Undo command will delete a number of characters. Another common value is Delete, whereas some operations are named Unknown. If you change the indentation of a paragraph on the control, this action's name is Unknown. Even when an action's name is Unknown, the action can be undone with the Undo method.

The following statement sets the caption of the Undo command to a string that indicates the action to be undone (Editor is the name of a RichTextBox control):

```
If Editor.CanUndo Then
EditUndo.Text = ”Undo ” & Editor.UndoActionName
End If
```

Undo, Redo

These two methods undo or redo an action. The Undo method cancels the effects of the last action of the user on the control. The Redo method redoes the most recent undo action. The Redo method does not repeat the last action; it applies to undo operations only.

Cutting and Pasting

To cut, copy, and paste text in the RichTextBox control, you can use the same techniques you use with the regular TextBox control. For example, you can replace the current selection by assigning a string to the **SelectedText** property. The RichTextBox, however, provides a few useful methods for performing these operations. The **Copy**, **Cut**, and **Paste** methods perform the corresponding operations. The **Cut** and **Copy** methods are straightforward and require no arguments. The **Paste** method accepts a single argument, which is the format of the data to be pasted. Because the data will come from the Clipboard, you can extract the format of the data in the Clipboard at the time and then call the **CanPaste** method to find out whether the control can handle this type of data. If so, you can then paste them in the control by using the **Paste** method.

This technique requires a bit of code because the Clipboard class doesn't return the format of the data in the Clipboard. You must call the following method of the Clipboard class to find out whether the data is of a specific type and then paste it on the control:

```
If Clipboard.GetDataObject.GetDataPresent(DataFormats.Text) Then
RichTextBox.Paste(DataFormats.Text)
End If
```

This is a very simple case because we know that the RichTextBox control can accept text. For a robust application, you must call the **GetDataPresent** method for each type of data your application should be able to handle. (You may not want to allow users to paste all types of data that the control can handle.) By the way, you can simplify the code with the help of the **ContainsText/ContainsImage** and **GetText/GetImage** methods of the **My.Application.Clipboard object**.

Searching in a RichTextBox Control

To locate a string in the text of the RichTextBox control, use the Find method. The Find method is quite flexible, as it allows you to specify the type of the search, whether it will locate entire words, and so on. The simplest form of this method accepts the search string as an argument and returns the location of the first instance of the word in the text. If the search argument isn't found, the method returns the value −1.

RichTextBox1.Find(string)

Another equally simple syntax of the Find method allows you to specify how the control will search for the string:

RichTextBox1.Find(string, searchMode)

The searchMode argument is a member of the RichTextBoxFinds enumeration, which is shown in Table 4.7.

Table - The RichTextBoxFinds Enumeration

Value	Effect
MatchCase	Performs a case-sensitive search.
NoHighlight	The text found will not be highlighted.
None	Locates instances of the specified string even if they're not whole words.
Reverse	The search starts at the end of the document.
WholeWord	Locates only instances of the specified string that are whole words.

Two more forms of the Find method allow you specify the range of the text in which the search will take place:

RichTextBox1.Find(string, start, searchMode)

RichTextBox1.Find(string, start, end, searchMode)

The arguments start and end are the starting and ending locations of the search (use them to search for a string within a specified range only). If you omit the end argument, the search will start at the location specified by the start argument and will extend to the end of the text.

Tree View and List View Controls

The TreeView control implements a data structure known as a tree. A tree is the most appropriate structure for storing hierarchical information. The organizational chart of a company, for example, is a tree structure. Every person reports to another person above him or her, all the way to the president or CEO. Figure 4.21 depicts a possible organization of continents, countries, and cities as a tree. Every city belongs to a country, and every country to a continent. In the same way, every computer file belongs to a folder that may belong to an even bigger folder, and so on up to the drive level. You can't draw large tree structures on paper, but it's possible to create a similar structure in the computer's memory without size limitations.

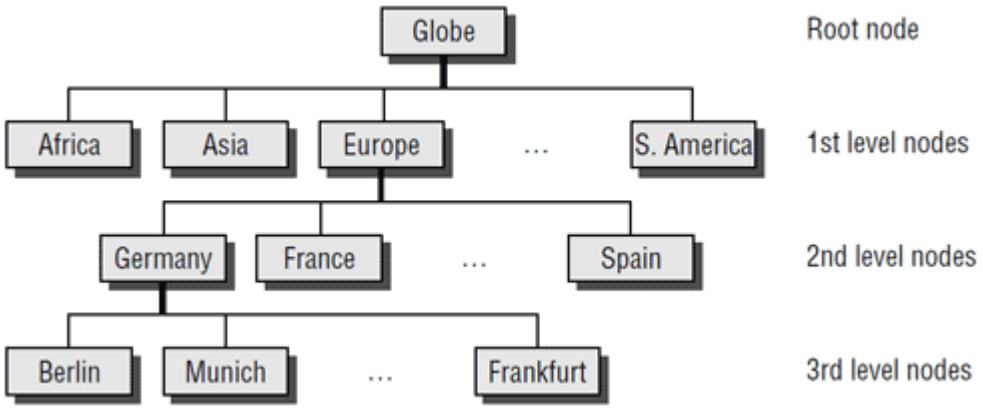


Figure - The World View as Tree

Note: ***The items displayed on a TreeView control are just strings***. Moreover, the TreeView control doesn't require that the items be unique. You can have identically named nodes in the same branch — as unlikely as this might be for a real application. There's no property that makes a node unique in the tree structure or even in its own branch.

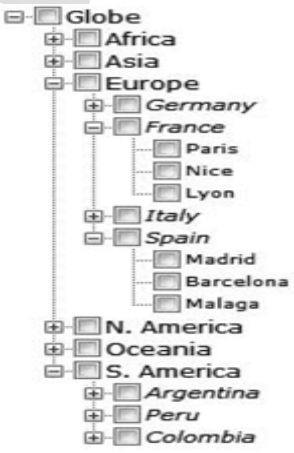


Figure - The tree implemented with a TreeView control

The tree structure is ideal for data with parent-child relations (relations that can be described as belongs to or owns). The continents-countries-cities data is a typical example. The folder structure on a hard disk is another typical example. Any given folder is the child of another folder or the root folder.

The ListView control implements a simpler structure, known as a list. A list's items aren't structured in a hierarchy; they are all on the same level and can be traversed serially, one after the other. You can also think of the list as a multidimensional array, but the list offersmore features. A list item can have subitems and can be sorted according to any column. For example, you can set up a list of customer names (the list's items) and assign a number of subitems to each customer: a contact, an address, a phone number, and so on. Or you can set up a list of files with their attributes as subitems. Figure 4.23 shows a Windows folder mapped on a ListView control. Each file is an item, and its attributes are the subitems. As you already know, you can sort this list by filename, size, file type, and so on. All you have to do is click the header of the corresponding column.

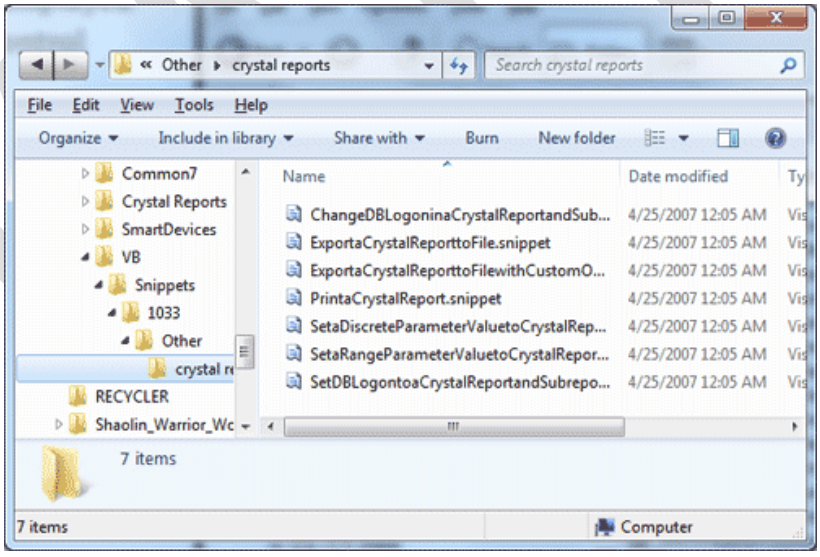


Figure - A folder's files displayed in a ListView control (Details view)

The ListView control is a glorified ListBox control. If all you need is a control to store sorted objects, use a ListBox control. If you want more features, such as storing multiple items per row, sorting them in different ways, or locating them based on any subitem's value, you must consider the ListView control. You can also look at the ListView control as a view-only grid.

The TreeView and ListView controls are commonly used along with the ImageList control. The ImageList control is a simple control for storing images so they can be retrieved quickly and used at runtime. You populate the ImageList control with the images you want to use on your interface, usually at design time, and then you recall them by an index value at runtime. Before we get into the details of the TreeView and ListView controls, a quick overview of the ImageList control is in order.

The ImageList Control

The ImageList is a simple control that stores images used by other controls at runtime. For example, a TreeView control can use icons to identify its nodes. The simplest and quickest method of preparing these images is to create an ImageList control and add to it all the icons you need for decorating the TreeView control's nodes. The ImageList control maintains a series of bitmaps in memory that the TreeView control can access quickly at runtime. Keep in mind that the ImageList control can't be used on its own and remains invisible at runtime.

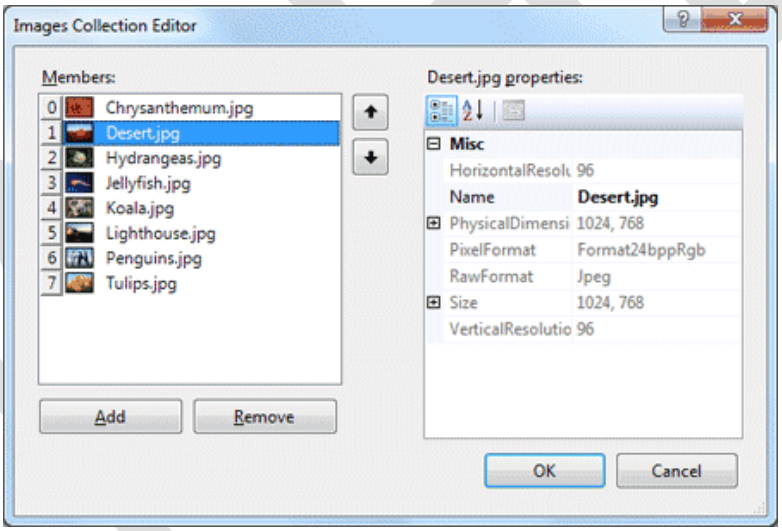


Figure - The Images Collection Editor of ImageList Control

The other method of adding images to an ImageList control is to call the Add method of the Images collection, which contains all the images stored in the control. To add an image at runtime, you must first create an Image object with the image (or icon) you want to add to the control and then call the Add method as follows:

ImageList1.Images.Add(image)

where image is an Image object with the desired image. You will usually call this method as follows:

ImageList1.Images.Add(Image.FromFile(path))

where - path is the full path of the file with the image.

The Images collection of the ImageList control is a collection of Image objects, not the files in which the pictures are stored. This means that the image files need not reside on the computer on which the application will be executed, as long as they have been added to the collection at design time.

TreeView Control

Let's start our discussion of TreeView control with a few simple properties that you can set at design time. To experiment with the properties discussed in this section, open the TreeView Example project. The project's main form is shown in Figure. After setting some properties (they are discussed next), run the project and click the Populate button to populate the control. After that, you can click the other buttons to see the effect of the various property settings on the control.

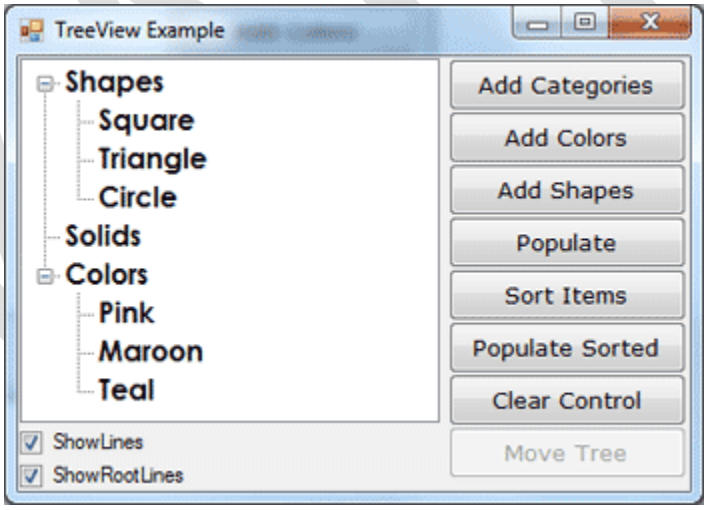


Figure - The TreeView Example project demonstrates the basic properties and methods of the TreeView control.

Here are the basic properties that determine the appearance of the control:

- **ShowCheckBoxes -** If this property is True, a check box appears in front of each node. If the control displays check boxes, you can select multiple nodes; otherwise, you're limited to a single selection.

- **FullRowSelect** - This True/False value determines whether a node will be selected even if the user clicks outside the node's caption.
- **HideSelection** - This property determines whether the selected node will remain highlighted when the focus is moved to another control. By default, the selected node doesn't remain highlighted when the control loses the focus.
- **HotTracking** - This property is another True/False value that determines whether nodes are highlighted as the pointer hovers over them.When it's True, the TreeView control behaves like a web document with the nodes acting as hyperlinks — they turn blue while the pointer hovers over them. Use the NodeMouseHover event to detect when the pointer hovers over a node.
- **Indent** - This property specifies the indentation level in pixels. The same indentation applies to all levels of the tree—each level is indented by the same number of pixels with respect to its parent level.
- **PathSeparator** - A node's full name is made up of the names of its parent nodes, separated by a backslash. To use a different separator, set this property to the desired symbol.
- **ShowLines** - The ShowLines property is a True/False value that determines whether the control's nodes will be connected to its parent items with lines. These lines help users visualize the hierarchy of nodes, and it's customary to display them.
- **ShowPlusMinus** - The ShowPlusMinus property is a True/False value that determines whether the plus/minus button is shown next to the nodes that have children. The plus button is displayed when the node is collapsed, and it causes the node to expand when clicked. Likewise, the minus sign is displayed when the node is expanded, and it causes the node to collapse when clicked. Users can also expand the current node by pressing the left-arrow button and collapse it with the right-arrow button.
- **ShowRootLines** - This is another True/False property that determines whether there will be lines between each node and root of the tree view. Experiment with the ShowLines and ShowRootLines properties to find out how they affect the appearance of the control.

- **Sorted** - This property determines whether the items in the control will be automatically sorted. The control sorts each level of nodes separately. In our Globe example, it will sort the continents, then the countries within each continent, and then the cities within each country.

Adding New Items at Design Time

Let's look now at the process of populating the TreeView control. Adding an initial collection of nodes to a TreeView control at design time is trivial. Locate the Nodes property in the Properties window, and you'll see that its value is Collection. To add items, click the ellipsis button, and the TreeNode Editor dialog box will appear, as shown in Figure 4.26. To add a root item, just click the Add Root button. The new item will be named Node0 by default. You can change its caption by selecting the item in the list and setting its Text property accordingly. You can also change the node's Name property, as well as the node's appearance by using the NodeFont, FontColor, and ForeColor properties.

To specify an image for the node, set the control's ImageList property to the name of an ImageList control that contains the appropriate images, and then set either the node's ImageKey property to the name of the image, or the node's ImageIndex property to the index of the desired image in the ImageList control. If you want to display a different image when the control is selected, set the **SelectedImageKey** or the **SelectedImageIndex** property accordingly.

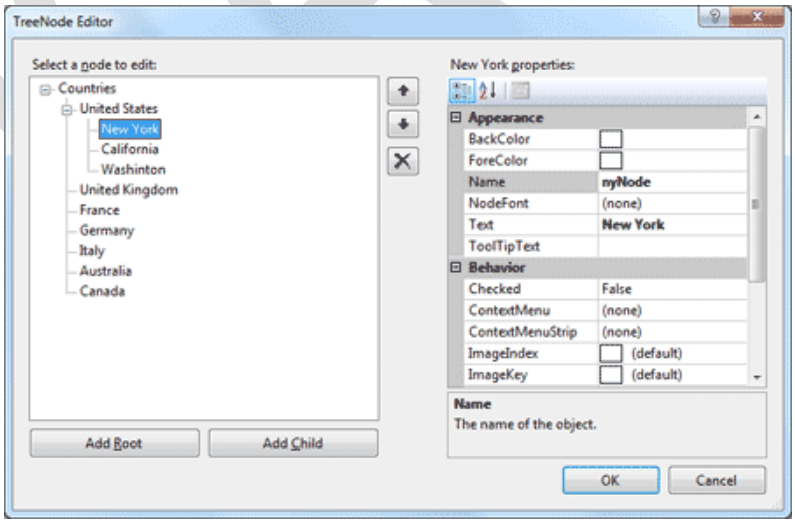


Figure - The TreeNode Editor dialog box

Click the Add Root button first. A new node is added automatically to the list of nodes, and it is named Node0. Select it with the mouse, and its properties appear in the right pane of the TreeNode Editor window. Here you can change the node's Text property to Countries. You can specify the appearance of each node by setting its font and fore/background colors.

Adding New Items at Runtime

Adding items to the control at runtime is a bit more involved. All the nodes belong to the control's Nodes collection, which is made up of TreeNode objects. To access the **Nodes** collection, use the following expression, where TreeView1 is the control's name and **Nodes** is a collection of TreeNode objects:

TreeView1.Nodes

This expression returns a collection of TreeNode objects and exposes the proper members for accessing and manipulating the individual nodes. The control's **Nodes** property is the collection of all root nodes.

To access the first node, use the expression **TreeView.Nodes(0)** (this is the Globe node in our example). The Text property returns the node's value, which is a string. **TreeView1.Nodes(0).Text** is the caption of the root node on the control. The caption of the second node on the same level is **TreeView1.Nodes(1).Text**, and so on.

The following statements print the strings shown highlighted below them (these strings are not part of the statements; they're the output that the statements produce):

```
Debug.WriteLine(TreeView1.Nodes(0).Text)
Countries
Debug.WriteLine(TreeView1.Nodes(0).Nodes(0).Text)
UnitedStates
Debug.WriteLine(TreeView1.Nodes(0).Nodes(0).Nodes(1).Text)
New York
```

Let's take a closer look at these expressions. **TreeView1.Nodes(0)** is the first root node, the Countries node. Under this node, there is a collection of nodes, the **TreeView1.Nodes(0).Nodes** collection. Each node in this collection is a country name. The first node in this collection is United States, and you can access it with the expression **TreeView1.Nodes(0).Nodes(0)**. If you want to change the appearance of the node United States,

type a period after the preceding expression to access its properties (the NodeFont property to set its font, the ForeColor property to set it color, the ImageIndex property, and so on). Likewise, this node has its own Nodes collection, which contains the states under the specific country.

Adding New Nodes

The Add method adds a new node to the Nodes collection. The Addmethod accepts as an argument a string or a TreeNode object. The simplest form of the Add method is

newNode = Nodes.Add(nodeCaption)

where **nodeCaption** is a string that will be displayed on the control. Another form of the Add method allows you to add a TreeNode object directly (**nodeObj** is a properly initialized TreeNode variable):

newNode = Nodes.Add(nodeObj)

To use this form of the method, you must first declare and initialize a TreeNode object:

```
Dim nodeObj As New TreeNode
nodeObj.Text = "Tree Node"
nodeObj.ForeColor = Color.BlueViolet
TreeView1.Nodes.Add(nodeObj)
```

The last overloaded form of the Add method allows you to specify the index in the current Nodes collection, where the node will be added:

newNode = Nodes.Add(index, nodeObj)

The nodeObj TreeNode object must be initialized as usual. To add a child node to the root node, use a statement such as the following:

TreeView1.Nodes(0).Nodes.Add("United States")

To add a state under United States, use a statement such as the following:

TreeView1.Nodes(0).Nodes(1).Nodes.Add("New York")

The expressions can get quite lengthy. The proper way to add child items to a node is to create a TreeNode variable that represents the parent node, under which the child nodes will be added. Let's say that the CountryNode variable in the following example represents the node United States:

```
Dim CountryNode As TreeNode
CountryNode = TreeView1.Nodes(0).Nodes(2)
```

Then you can add child nodes to the ContinentNode node:

```
CountryNode.Nodes.Add("New York")
CountryNode.Nodes.Add("California")
```

To add yet another level of nodes, the city nodes, create a new variable that represents a specific state. The Add method actually returns a TreeNode object that represents the newly added node, so you can add a state and a few cities by using statements such as the following:

```
Dim StateNode As TreeNode
StateNode = CountryNode.Nodes.Add("New York")
StateNode.Nodes.Add("Alberny")
StateNode.Nodes.Add("Amsterdam")
StateNode.Nodes.Add("Auburn")
```

Then you can continue adding states under another country as follows:

```
StateNode = CountryNode.Nodes.Add("United Kingdom")
StateNode.Nodes.Add("London")
StateNode.Nodes.Add("Manchester")
```

The Nodes Collection Members

The Nodes collection exposes the usual members of a collection. The Count property returns the number of nodes in the Nodes collection. Again, this is not the total number of nodes in the control, just the number of nodes in the current Nodes collection. The expression

```
TreeView1.Nodes.Count
```

returns the number of all nodes in the first level of the control. In the case of the Countries example, it returns the value 1. The expression

```
TreeView1.Nodes(0).Nodes.Count
```

returns the number of countries in the Countries example. Again, you can simplify this expression by using an intermediate TreeNode object:

```
Dim Countries As TreeNode
Countries = TreeView1.Nodes(0)
Debug.WriteLine("There are "& Countries.Nodes.Count.ToString & _
" countries on the control")
```

The **Clear** method removes all the child nodes from the current node. If you apply this method to the control's root node, it will clear the control. To remove all the cities under the California node, use a statement such as the following:

```
TreeView1.Nodes(0).Nodes(2).Nodes(1).Nodes.Clear
```

This example assumes that the third node under Countries corresponds to United States, and the second node under United Sates corresponds to California.

The Item property retrieves a node specified by an index value. The expression **Nodes.Item(1)** is equivalent to the expression **Nodes(1)**. Finally, the Remove method removes a node from the Nodes collection. Its syntax is

```
Nodes.Remove(index)
```

Where - index is the order of the node in the current Nodes collection. To remove the selected node, call the Remove method on the **SelectedNode** property without arguments:

```
TreeView1.SelectedNode.Remove
```

Or you can apply the Remove method to a TreeNode object that represents the node you want to remove:

```
Dim Node As TreeNode
Node = TreeView1.Nodes(0).Nodes(5)
Node.Remove
```

Basic Nodes Properties

There are a few properties you will find extremely handy as you program the TreeView control. The ***IsVisible*** property is a True/False value indicating whether the node to which it's applied is visible. To bring an invisible node into view, call its **EnsureVisible** method:

```
If Not TreeView1.SelectedNode.IsVisible Then
TreeView1.EnsureVisible
End If
```

How can the selected node be invisible? It can, if you select it from within your code in a search operation. The ***IsSelected*** property returns True if the specified node is selected, while the ***IsExpanded*** property returns True if the specified node is expanded. You can toggle a node's state by calling its Toggle method. You can also expand or collapse a node by calling its

Expand or Collapse method, respectively. Finally, you can collapse or expand all nodes by calling the CollapseAll or ExpandAll method of the TreeView control.

Scanning the Tree View control

The TreeViewScan Example, whose main form is shown in Figure 4.28, demonstrates the process of scanning the nodes of a TreeView control. The form contains a TreeView control on the left, which is populated with the same data as the Globe project, and a ListBox control on the right, in which the tree's nodes are listed. Child nodes in the ListBox control are indented according to the level to which they belong.

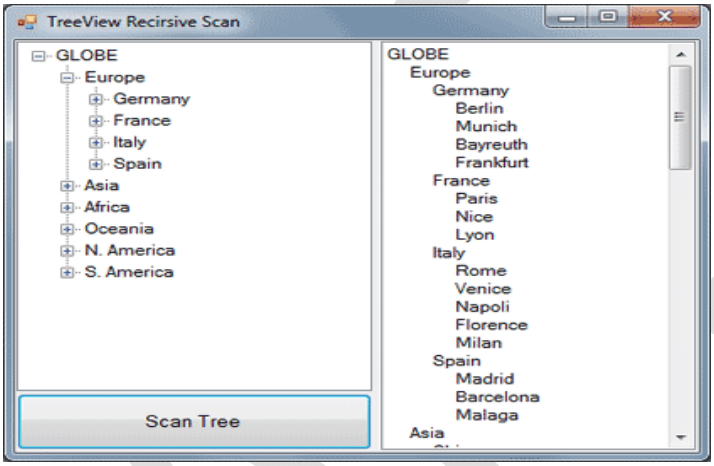


Figure - TreeView Scan Example

Recursive Scanning of the Nodes Collection

To scan the nodes of the TreeView1 control, start at the top node of the control by using the following statement:

ScanNode(GlobeTree.Nodes(0))

This is the code behind the Scan Tree button, and it doesn't get any simpler. It calls the ScanNode() subroutine to scan the child nodes of a specific node, which is passed to the subroutine as an argument. GlobeTree.Nodes(0) is the root node. By passing the root node to the ScanNode() subroutine, we're in effect asking it to scan the entire tree.

This example assumes that the TreeView control contains a single root node and that all other nodes are under the root node. If your control contains multiple root nodes, then you must set up a small loop and call the ScanNode() subroutine once for each root node:

```
For Each node In GlobeTree.Nodes
ScanNode(node)
Next
```

Let's look now at the ScanNode() subroutine shown in Listing

Listing: Scanning a Tree Recursively

```
Sub ScanNode(ByVal node As TreeNode)
Dim thisNode As TreeNode
Static indentationLevel As Integer
Application.DoEvents()
ListBox1.Items.Add(Space(indentationLevel) & node.Text)
If node.Nodes.Count > 0 Then
indentationLevel += 5
For Each thisNode In node.Nodes
ScanNode(thisNode)
Next
indentationLevel -= 5
End If
End Sub
```

The ListView Control

The ListView control is similar to the ListBox control except that it can display its items in many forms, along with any number of subitems for each item. To use the ListView control in your project, place an instance of the control on a form and then set its basic properties, which are described in the following list.

View and Arrange - Two properties determine how the various items will be displayed on the control: the View property, which determines the general appearance of the items, and the Arrange property, which determines the alignment of the items on the control's surface. The View property can have one of the values shown in Table

Table: Settings of the View Property of VB.NET ListView Control

Setting	Description
LargeIcon (Default)	Each item is represented by an icon and a caption below the icon.
SmallIcon	Each item is represented by a small icon and a caption that appears to the right of the icon.
List	Each item is represented by a caption.
Details	Each item is displayed in a column with its subitems in adjacent columns.
Tile	Each item is displayed with an icon and its subitems to the right of the icon. This view is available only on Windows XP and Windows Server 2003.

The Arrange property can have one of the settings shown in Table 4.9.

Table: Settings of the Arrange Property of VB.NET ListView Control

Setting	Description
Default	When an item is moved on the control, the item remains where it is dropped.
Left	Items are aligned to the left side of the control.
SnapToGrid	Items are aligned to an invisible grid on the control. When the user moves an item, the item moves to the closest grid point on the control.
Top	Items are aligned to the top of the control.

HeaderStyle - This property determines the style of the headers in Details view. It has no meaning when the View property is set to anything else, because only the Details view has columns. The possible settings of the HeaderStyle property are shown in Table

Table: Settings of the HeaderStyle Property of VB.NET ListView Control

Setting	Description
Clickable	Visible column header that responds to clicking
Nonclickable (Default)	Visible column header that does not respond to clicking
None	No visible column header

AllowColumnReorder - This property is a True/False value that determines whether the user can reorder the columns at runtime, and it's meaningful only in Details view. If this property is set to True, the user can move a column to a new location by dragging its header with the mouse and dropping it in the place of another column.

Activation - This property, which specifies how items are activated with the mouse, can have one of the values shown in Table

Table: Settings of the Activation Property of VB.NET ListView Control

Setting	Description
OneClick	Items are activated with a single click. When the cursor is over an item, it changes shape, and the color of the item's text changes.
Standard (Default)	Items are activated with a double-click. No change in the selected item's text color takes place.
TwoClick	Items are activated with a double-click, and their text changes color as well.

- **FullRowSelect** - This property is a True/False value, indicating whether the user can select an entire row or just the item's text, and it's meaningful only in Details view. When this property is False, only the first item in the selected row is highlighted.
- **GridLines** - Another True/False property. If True, grid lines between items and subitems are drawn. This property is meaningful only in Details view.

- **Group** - The items of the ListView control can be grouped into categories. To use this feature, you must first define the groups by using the control's Group property, which is a collection of strings. You can add as many members to this collection as you want.
- **LabelEdit** - The LabelEdit property lets you specify whether the user will be allowed to edit the text of the items. The default value of this property is False. Notice that the LabelEdit property applies to the item's Text property only; you can't edit the subitems (unfortunately, you can't use the ListView control as an editable grid).
- **MultiSelect** - A True/False value, indicating whether the user can select multiple items from the control. To select multiple items, click them with the mouse while holding down the Shift or Ctrl key. If the control's ShowCheckboxes property is set to True, users can select multiple items by marking the check box in front of the corresponding item(s).
- **Scrollable** - A True/False value that determines whether the scroll bars are visible. Even if the scroll bars are invisible, users can still bring any item into view. All they have to do is select an item and then press the arrow keys as many times as needed to scroll the desired item into view.
- **Sorting** - This property determines how the items will be sorted, and its setting can be None, Ascending, or Descending. To sort the items of the control, call the Sort method, which sorts the items according to their caption. It's also possible to sort the items according to any of their subitems, as explained in the section "Sorting the ListView Control" later in this chapter.

The Columns Collection of ListView Control in VB.NET 2008

To display items in Details view, you must first set up the appropriate columns. The first column corresponds to the item's caption, and the following columns correspond to its subitems. If you don't set up at least one column, no items will be displayed in Details view. Conversely, the Columns collection is meaningful only when the ListView control is used in Details view.
The items of the Columns collection are of the ColumnHeader type. The simplest way to set up the appropriate columns is to do so at design time by using a visual tool. Locate and select the Columns property in the Properties window, and click the ellipsis button next to the property.

The ColumnHeader Collection Editor dialog box will appear, as shown in Figure 4.29, in which you can add and edit the appropriate columns.

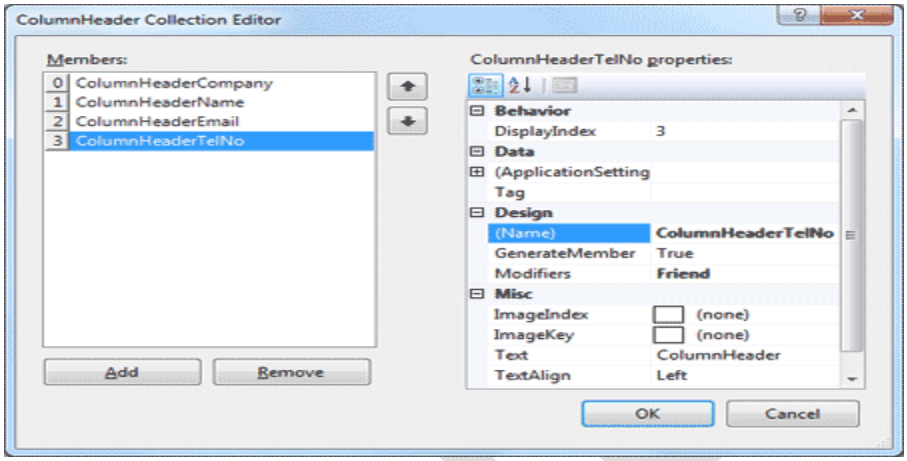


Figure - ListView Control's Column Header Collection Editor Dialog Box

Adding columns to a ListView control and setting their properties through the dialog box shown in Figure is quite simple. Don't forget to size the columns according to the data you anticipate storing in them and to set their headers.

It is also possible to manipulate the Columns collection fromwithin your code as follows. Create a ColumnHeader object for each column in your code, set its properties, and then add it to the control's Columns collection:

```
Dim ListViewCol As New ColumnHeader
ListViewCol.Text = "New Column"
ListViewCol.TextAlign = HorizontalAlignment.Center
ListViewCol.Width = 125
ListView1.Columns.Add(ListViewCol)
```

Adding and Removing Columns at Runtime

To add a new column to the control, use the Add method of the Columns collection. The syntax of the Add method is as follows:

```
ListView1.Columns.Add(header, width, textAlign)
```

The header argument is the column's header (the string that appears on top of the items). The width argument is the column's width in pixels, and the last argument determines how the text will be aligned. The textAlign argument can be Center, Left, or Right.

The Add method returns a ColumnHeader object, which you can use later in your code to manipulate the corresponding column. The ColumnHeader object exposes a Name property, which can't be set with the Add method:

```
Header1 = TreeView1.Add("Column 1", 60, ColAlignment.Left)
Header1.Name = "Column1"
```

After the execution of these statements, the first column can be accessed not only by index, but also by name.

To remove a column, call the Remove method:

```
ListView1.Columns(3).Remove
```

The indices of the following columns are automatically decreased by one. The Clear method removes all columns from the Columns collection. Like all collections, the Columns collection exposes the Count property, which returns the number of columns in the control.

The Items and SubItems collection

As with the TreeView control, the ListView control can be populated either at design time or at runtime. To add items at design time, click the ellipsis button next to the ListItems property in the Properties window. When the ListViewItem Collection Editor dialog box pops up, you can enter the items, including their subitems, as shown in Figure

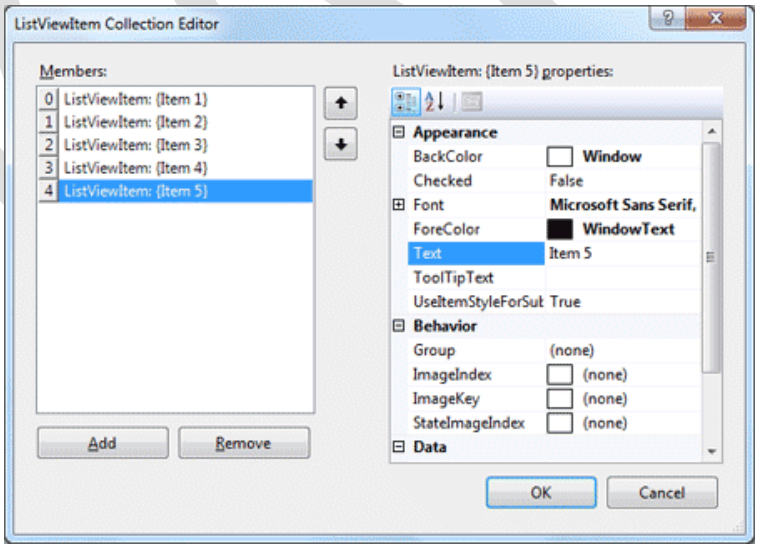


Figure - ListViewItem Collection Editor Dialog Box

Unlike the TreeView control, the ListView control allows you to specify a different appearance for each item and each subitem. To set the appearance of the items, use the Font, BackColor, and ForeColor properties of the ListViewItem object.

These members are as follows:

BackColor/ForeColor properties - These properties set or return the background/foreground colors of the current item or subitem.

Checked property - This property controls the status of an item. If it's True, the item has been selected. You can also select an item from within your code by setting its Checked property to True. The check boxes in front of each item won't be visible unless you set the control's ShowCheckBoxes property to True.

Font property - This property sets the font of the current item. Subitems can be displayed in a different font if you specify one by using the Font property of the corresponding subitem (see the section titled ‘‘The SubItems Collection," later in this chapter). By default, subitems inherit the style of the basic item. To use a different style for the subitems, set the item's UseItemStyleForSubItems property to False.

Text property - This property indicates the caption of the current item or subitem.

SubItems collection - This property holds the subitems of a ListViewItem. To retrieve a specific subitem, use a statement such as the following:

```
sitem = ListView1.Items(idx1).SubItems(idx2)
```

where **idx1** is the index of the item, and **idx2** is the index of the desired subitem.*

To add a new subitem to the SubItems collection, use the Add method, passing the text of the subitem as an argument:

```
LItem.SubItems.Add("subitem's caption")
```

The argument of the Add method can also be a ListViewItem object. Create a ListViewItem, populate it, and then add it to the Items collection as shown here:

```
Dim LI As New ListViewItem
LI.Text = "A New Item"
Li.SubItems.Add("Its first subitem")
Li.SubItems.Add("Its second subitem")
```

```
' statements to add more subitems
ListView1.Items.Add(LI)
LItem.SubItems.Insert(idx, subitem)
```

POSSIBLE QUESTIONS

PART A (1 Mark)

(Online Examinations)

PART B (6 Marks)

1. Discuss in detail about TrackBar, ListBox and TextBox with example
2. Explain about Rich TextBox controls.
3. Describe in detail about ListBox, CheckedListBox and ComboBox Controls.
4. Explain about ListView controls
5. Elucidate about Color, Font, Print Dialog Box controls in VB.NET.
6. Explain SrollBar and CheckedListBox controls with example.
7. Give Explanation about TreeView controls with neat diagram.
8. Elaborate RichTextBox Control properties, methods with an example.
9. Elucidate in detail about ListView Controls
10. Differentiate between list box and combo box.

PART A (1 Mark) – Unit 1II

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
1	Which is the base class for all built-in controls?	User Control	Custom Control	Control	ActiveX Control	Control
2	Which class is used to run the EXE application file in VB.NET	Process	Application	Exe	Execute	Process
3	Which controls do not have events?	TextBox	Label	ToolTip	ImageList	ImageList
4	What is the property used to enlarge the image in picture box?	Size	SizeMode	Mode	Stretch	SizeMode
5	What is the default event for Picture Box?	Click	Disposed	Layout	Resize	Click

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
6	A ------- is a component used to accept input from the user or display the information on the form	text	container	control	counter	control
7	The textbox can accept a maximum of ------ characters	1024.00	2048.00	156.00	1028.00	2048.00
8	The -------- property allows you to display multiple lines of text in a textbox control	Text	Multiline	PasswordChar	Autosize	Multiline
9	The -------- property allows automatic resizing of the label control according to the length of its caption	Text	Multiline	PasswordChar	Autosize	Autosize

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
10	The ---------- is used to display the text as a link	label	textbox	linklabel	listview	linklabel
11	The ---------- property is used to get or set the color used to display the active link	LinkColor	DisabledLinkColor	ActiveLinlColor	LinkVisited	ActiveLinlColor
12	The ---------- property is used to get or set a value indicating whether a link should be displayed as though it was visited	LinkColor	DisabledLinkColor	LinkVisited	ActiveLinlColor	LinkVisited
13	The ------- property is used to get or set the mode behavior of the listbox control	Sorted	SelectionMode	SelectedIndex	SelectedItem	SelectionMode

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
14	The ------- property is used to set or retrieve the currently selected item in the combobox control	Sorted	SelectionMode	SelectedIndex	SelectedItem	SelectedItem
15	The --------- control is used to set Yes/No options	CheckBox	RadioButton	GroupBox	Button	CheckBox
16	The --------- control is used to group related controls together	RadioButton	StatusBar	GroupBox	CheckBox	GroupBox
17	The -------- property is used to specify whether or not the statusbar should display panels	Text	Checked	SelectedIndex	ShowPanels	ShowPanels
18	The -------- property is used to specify the location of a control in terms of X and Y coordinates	Name	Visible	Location	Enabled	Location

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
19	In VB.Net the -------- class is the base class for displaying common dialog boxes.	Inherits	String	CommonDialog	MyBase	CommonDialog
20	The classes that are inherited from the CommonDialog class are categorized as --------	4.00	6.00	5.00	3.00	5.00
21	Button class is based on ------- class	Stirng	TextBoxBase	ButtonBase	Windows	ButtonBase
22	The ------ property doesn't allow the user to enter	Enabled	Multiline	ReadOnly	TextAllign	ReadOnly
23	The default event of the CheckBox is ----------	Click	CheckedChange	Changed	DoubleClick	CheckedChange

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
24	RadioButton control is based on the ------- - class	Stirng	TextBoxBase	ButtonBase	Windows	ButtonBase
25	The ListBox control is based on the ------- - class	Stirng	TextBoxBase	ButtonBase	ListControl	ListControl
26	To display the list as multiple columns in list box --------- property is used	SelectionMode	SelectedIndex	SelectedItem	MultiColumn	MultiColumn
27	The default event of ListBox is the ---------	Click	CheckedChange	DoubleClick	SelectedIndexChanged	SelectedIndexChanged
28	The ---------- property in the Appearance section of the properties window	TextAllign	ReadOnly	Enabled	DropDownStyle	DropDownStyle

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
29	The ---------- Gets/Sets whether the tree node is checked	ReadOnly	Checked	IsEditing	IsSelected	Checked
30	The --------- Gets the collection of nodes in the current node	Checked	IsEditing	IsSelected	Nodes	Nodes
31	Default event of the Tree View control is the ----------	Click	Selected	AfterSelect	Load	AfterSelect
32	---------- is a combination of a ListBox and a CheckBox	DropDownBox	CheckedListBox	LinkBox	TreeView	CheckedListBox
33	-------- cannot display captions where as GroupBoxes can	Panels	PictureBox	Splitter	ToolTip	Panels

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
34	To assign ToolTip's with controls ------- is used	SetTip	SetToolTip	GetTip	SetTool	SetToolTip
35	Notable property in ErrorProvider is ----------	AutoScroll	SetTip	Allign	BlinkRate	BlinkRate
36	The default event of the MenuItem is ----------	CheckedChange	AfterSelect	Click	Active	Click
37	The ------- property is used to display a menu item as a radio button	RadioCheck	Checked	Shortcut	DefaultItem	RadioCheck
38	The menus that appear on the menu bar are created using the -------- object	MenuItem	MainMenu	Context	MenuDesigner	MainMenu

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
39	The --------- property allows to set the initial directory which should open while using the OpenFileDialog.	InitialDirectory	FilterIndex	RestoreIndex	ShowHelp	InitialDirectory
40	The ------ property checks whether the specified path exists before returning from the dialog.	InitialDirectory	CheckPathExists	RestoreIndex	FilterIndex	CheckPathExists
41	WindowState property is ---------- -- by default	Normal	Maximized	Minimized	Flat	Normal
42	This property is used to change/display the titile of the form	Name	Text	Title	Form	Text

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
43	The default BackColor of the Form is the system color named	gray	white	pale	control	control
44	Toolbar items are part of ---------- collection	items	Buttons	properties	Opions	Buttons
45	The control with the tab index ------- first gets focus when the form is shown	0.00	1.00	Maximum value	Minimum value	0.00
46	What is the return type of InputBox() function	Integer	Object	String	Double	String
47	In Message Box which is the required parameter, that must be supplied a value?	prompt	button	title	name	prompt

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
48	If the number of items exceed the value that can be displayed, _____ bars will automatically appear on the control	Icon	option button	command button	scroll bars	scroll bars
49	The _____ method is used to add items to a list at run time.	item	index	remove item	add item	add item
50	The _____ property sets the index number of the currently selected item	index number	list index	list count	add item	list index
51	The sorted property is set to ____ to enable a list to appear in alphanumeric order	0.00	1.00	TRUE	FALSE	TRUE

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
52	_____ box saves the space on a form	list box	tool box	combo box	TreeView	combo box
53	_____ used in groups to display multiple choices from which the user can select one or more.	option button	check box	combo box	label box	check box
54	In _____ control the user can set the control's value by sliding the indicator or by clicking on either side of the indicator.	Scroll Bar	Track Bar	status bar	image bar	Track Bar
55	The _____ method is used to remove an item from the list	item	index	remove item	add item	remove item

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
56	When a common dialog control is added, a new icon appears in the ____ of the form	component tray	component list	status bar	component bar	component tray
57	The _____ property is used to wrap the text in Textbox control when text reaches the right edge	Wrap Word	Word Wrap	Accept returns	Accept Tab	Word Wrap
58	Each item in a Tree View is called _____	branch	subtree	leaf	node	node
59	In _____ Dialog control the user review and select a font and then set its size and style	Color Dialog	Font Dialog	Open Dialog	Format Dialog	Font Dialog

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
60	Which property is used to specify the type(s) of files displayed in the dialog box	DereferenceLinks	Filter	File Name	Object Property	Filter

UNIT – IV

SYLLABUS

Handling Strings, characters and Dates: Handling Strings and Characters – Handling Dates. Working with Folders and Files: Accessing Folders and Files – Accessing Files. Drawing and Painting with Visual Basic: Displaying Images – Drawing with GDI – Co-ordinate Transformation – Bitmaps.

Handling Strings, Characters and Dates

The .NET Framework provides two basic classes for manipulating text: the String and String-Builder classes.

The distinction between the two classes is that the String class is better suited for static strings, whereas the StringBuilder class is better suited for dynamic strings. Use the String class for strings that don't change frequently in the course of an application, and use the StringBuilder class for strings that grow and shrink dynamically. The two classes expose similar methods, but the String class's methods return new strings; if you need to manipulate large strings extensively, using the String class might fill the memory quite quickly.

Handling String and Characters

The Char Class

The Char data type stores characters as individual, double-byte (16-bit), Unicode values; and it exposes methods for classifying the character stored in a Char variable. You can use methods such as IsDigit and IsPunctuation on a Char variable to determine its type, and other similar methods that can simplify your string validation code.

To use a character variable in your application, you must declare it with a statement such as the following one:

```
Dim ch As Char
ch = Convert.ToChar("A")
```

Properties

The Char class provides two trivial properties: MaxValue and MinValue. They return the largest and smallest character values you can represent with the Char data type.

Methods

The Char data type exposes several useful methods for handling characters. All the methods described here have the same syntax: They accept either a single argument, which is the character they act upon, or a string and the index of a character in the string on which they act.

GetNumericValue

This method returns a positive numeric value if called with an argument that is a digit, and the value −1 otherwise. If you call the GetNumericValue with the argument 5, it will return the numeric value 5. If you call it with the symbol @, it will return the value −1.

GetUnicodeCategory

This method returns a numeric value that is a member of the UnicodeCategory enumeration and identifies the Unicode group to which the character belongs. The Unicode groups characters into categories such as math symbols, currency symbols, and quotation marks. Look up the UnicodeCategory enumeration in the documentation for more information.

IsLetter, IsDigit, IsLetterOrDigit

These methods return a True/False value indicating whether their argument, which is a character, is a letter, decimal digit, or letter/digit, respectively. You can write an event handler by using the IsDigit method to accept numeric keystrokes and to reject letters and punctuation symbols.

IsLower, IsUpper

These methods return a True/False value indicating whether the specified character is lowercase or uppercase, respectively.

IsNumber

This method returns a True/False value indicating whether the specified character is a number. The IsNumber method takes into consideration hexadecimal digits (the characters 0123456789-ABCDEF) in the same way as the IsDigit method does for decimal numbers.

IsPunctuation, IsSymbol, IsControl

These methods return a True/False value indicating whether the specified character is a punctuation mark, symbol, or control character, respectively. The Backspace and Esc keys, for example, are ISO (International Organization for Standardization) control characters.

IsSeparator

This method returns a True/False value indicating whether the character is categorized as a separator (space, new-line character, and so on).

IsWhiteSpace

This method returns a True/False value indicating whether the specified character is white space. Any sequence of spaces, tabs, line feeds, and form feeds is considered white space. Use this method along with the IsPunctuation method to remove all characters in a string that are not words.

ToLower, ToUpper

These methods convert their argument to a lowercase or uppercase character, respectively, and return it as another character.

ToString

This method converts a character to a string. It returns a single-character string, which you can use with other string-manipulation methods or functions.

The String Class

The String class implements the String data type, which is one of the richest data types in terms of the members it exposes. We have used strings extensively in earlier chapters, but this is a formal discussion of the String data type and all of the functionality it exposes.

To create a new instance of the String class, you simply declare a variable of the String type. You can also initialize it by assigning to the corresponding variable a text value:

```
Dim title As String = "Visual Basic 2008 Tutorial"
```

The Replace method, like all other methods of the String class, doesn't operate directly on the string to which it's applied. Instead, it creates a new string and returns it as a new string. You can also use Visual Basic's string-manipulation functions to work with strings. For example, you can replace the string VB with Visual Basic by using the following statement:

```
newTitle = Replace(title, "VB", "Visual Basic")
```

Like the methods of the String class, the string-manipulation functions don't act on the original string; they return a new string.

Properties

The String class exposes only two properties, the Length and Chars properties, which return a string's length and its characters, respectively. Both properties are read-only.

Length

The Length property returns the number of characters in the string and is read-only. To find out the number of characters in a string variable, use the following statement:

```
chars = myString.Length
```

Chars

The Chars property is an array of characters that holds all the characters in the string.

Methods

All the functionality of the String class is available through methods, which are described next. They are all shared methods: They act on a string and return a new string with the modified value.

Compare

This method compares two strings and returns a negative value if the first string is less than the second, a positive value if the second string is less than the first, and zero if the two strings are equal. Of course, the simplest method of comparing two strings is to use the comparison operators, as shown here:

```
If name1 < name 2 Then
' name1 is alphabetically smaller than name 2
Else If name 1 > name 2 Then
' name2 is alphabetically smaller than name 1
Else
' name1 is the same as name2
End If
```

CompareOrdinal

The CompareOrdinal method compares two strings similar to the Compare method, but it doesn't take into consideration the current locale. This method returns zero if the two strings are the same, and a positive or negative value if they're different. These values, however, are not 1 and −1; they represent the numeric difference between the Unicode values of the first two characters that are different in the two strings.

Concat

This method concatenates two or more strings (places them one after the other) and forms a new string. The simpler form of the Concat method has the following syntax and it is equivalent to the & operator:

```
newString = String.Concat(string1, string2)
```

This statement is equivalent to the following:

newString = string1 & string2

Copy

The Copy method copies the value of one string variable to another. Notice that the value to be copied must be passed to the method as an argument. The Copy method doesn't apply to the current instance of the String class. Most programmers will use the assignment operator and will never bother with the Copy method.

EndsWith, StartsWith

These two methods return True if their argument ends or starts with a user-supplied substring. The syntax of these methods is as follows:

found = str.EndsWith(string)

found = str.StartsWith(string)

These two methods are equivalent to the Left() and Right() functions, which extract a given number of characters from the left or right end of the string, respectively.

IndexOf, LastIndexOf

These two methods locate a substring in a larger string. The IndexOf method starts searching from the beginning of the string, and the LastIndexOf method starts searching from the end of the string. Both methods return an integer, which is the order of the substring's first character in the larger string (the order of the first character is zero).

To locate a string within a larger one, use the following forms of the IndexOf method:

pos = str.IndexOf(searchString)

pos = str.IndexOf(SearchString, startIndex)

pos = str.IndexOf(SearchString, startIndex, endIndex)

The startIndex and the endIndex arguments delimit the section of the string where the search will take place, and pos is an integer variable.

The last three overloaded forms of the IndexOf method search for an array of characters in the string:

str.IndexOf(Char())

str.IndexOf(Char(), startIndex)

str.IndexOf(Char(), startIndex, endIndex)

IndexOfAny

This is an interesting method that accepts as an argument an array of arguments and returns the first occurrence of any of the array's characters in the string. The syntax of the IndexOfAny method is

Dim pos As Integer = str.IndexOfAny(chars)

where chars is an array of characters.

This method attempts to locate the first instance of any member of the chars array in the string. If the character is found, its index is returned. If not, the process is repeated with the second character, and so on until an instance is found or the array has been exhausted.

Insert

The Insert method inserts one or more characters at a specified location in a string and returns the new string. The syntax of the Insert method is as follows:

newString = str.Insert(startIndex, subString)

startIndex is the position in the str variable, where the string specified by the second argument will be inserted.

Join

This method joins two or more strings and returns a single string with a separator between the original strings. Its syntax is the following, where separator is the string that will be used as the separator, and strings is an array with the strings to be joined:

newString = String.Join(separator, strings)

Split

Just as you can join strings, you can split a long string into smaller ones by using the Split method, whose syntax is the following, where delimiters is an array of characters and str is the string to be split:

strings() = String.Split(delimiters, str)

The string is split into sections that are separated by any one of the delimiters specified with the first argument. These strings are returned as an array of strings.

Splitting Strings with Multiple Separators

The delimiters array allows you to specify multiple delimiters, which makes it a great tool for isolating words in a text. You can specify all the characters that separate words in text (spaces, tabs, periods, exclamation marks, and so on) as delimiters and pass them along with the text to be parsed to the Split method.

The statements in Listing isolate the parts of a path, which are delimited by a backslash character.

Listing: Extracting a Path's Components

```
Dim path As String = "c:\My Documents\Business\Expenses"
Dim delimiters() As Char = {"\"c}
Dim parts() As String
parts = path.Split(delimiters)
Dim iPart As IEnumerator
iPart = parts.GetEnumerator
While iPart.MoveNext
Debug.WriteLine(iPart.Current.tostring)
End While
```

Remove

The Remove method removes a given number of characters from a string, starting at a specific location, and returns the result as a new string. Its syntax is the following, where startIndex is the index of the first character to be removed in the str string variable and count is the number of characters to be removed:

```
newSrting = str.Remove(startIndex, count)
```

Replace

This method replaces all instances of a specified character (or substring) in a string with a new one. It creates a new instance of the string, replaces the characters as specified by its arguments, and returns this string. The syntax of this method is

```
newString = str.Replace(oldChar, newChar)
```

where oldChar is the character in the str variable to be replaced, and newChar is the character to replace the occurrences of oldChar.

You can change the last statement to replace tabs with a specific number of spaces — usually three, four, or five spaces.

```
Dim txt, newTxt As String
Dim vbTab As String = vbCrLf
txt = "some text with two tabs"
newTxt = txt.Replace(vbTab, " ")
```

PadLeft, PadRight

These two methods align the string left or right in a specified field and return a fixed-length string with spaces to the right (for right-padded strings) or to the left (for left-padded strings). After the execution of these statements

```
Dim LPString, RPString As String
RPString = "[" & "Learning VB".PadRight(20) & "]"
LPString = "[" & "Learning VB".PadLeft(20) & "]"
```

the values of the LPString and RPString variables are as follows:

[Mastering VB]

[Mastering VB]

There are eight spaces to the left of the left-padded string and eight spaces to the right of the right-padded string.

The StringBulider Class

The StringBuilder class stores dynamic strings and exposes methods to manipulate them much faster than the String class. As you will see, the StringBuilder class is extremely fast, but it uses considerably more memory than the string it holds. To use the StringBuilder class in an application, you must import the System.Text namespace (unless you want to fully qualify each instance of the StringBuilder class in your code). Assuming that you have imported the System.Text class in your code module, you can create a new instance of the class via the following statement:

```
Dim txt As New StringBuilder
```

To create a new instance of the StringBuilder class, you can call its constructor without any arguments, or pass the initial string as an argument:

```
Dim txt As New StringBuilder("some string")
```

Properties

You have already seen the two basic properties of the StringBuilder class: the Capacity and MaxCapacity properties. In addition, the StringBuilder class provides the Length and Chars properties, which are the same as the corresponding properties of the String class. The Length property returns the number of characters in the current instance of the StringBuilder class, and the Chars property is an array of characters. Unlike the Chars property of the String class, this one is read/write.

Methods

Many of the methods of the StringBuilder class are equivalent to the methods of the String class, but they act directly on the string to which they're applied, and they don't return a new string.

Append

The Append method appends a base type to the current instance of the StringBuilder class, and its syntax is the following, where the value argument can be a single character, a string, a date, or any numeric value:

SB.Append(value)

When you append numeric values to a StringBuilder, they're converted to strings; the value appended is the string returned by the type's ToString method. You can also append an object to the StringBuilder — the actual string that will be appended is the value of the object's ToString property.

AppendFormat

The AppendFormat method is similar to the Append method. Before appending the string, however, AppendFormat formats it. The string to be appended contains format specifications and the appropriate values. The syntax of the AppendFormat method is as follows:

SB.AppendFormat(string, values)

The first argument is a string with embedded format specifications, and values is an array with values (objects, in general

Insert

This method inserts a string into the current instance of the StringBuilder class, and its syntax is as follows:

SB.Insert(index, value)

The index argument is the location where the new string will be inserted in the current instance of the StringBuilder, and value is the string to be inserted.

Remove

This method removes a number of characters from the current StringBuilder, starting at a specified location; its syntax is the following, where startIndex is the position of the first character to be removed from the string, and count is the number of characters to be removed:

SB.Remove(startIndex, count)

Replace

This method replaces all instances of a string in the current StringBuilder object with another string. The syntax of the Replace method is the following, where the two arguments can be either strings or characters:

SB.Replace(oldValue, newValue)

Unlike the String class, the replacement takes place in the current instance of the StringBuilder class and the method doesn't return another string.

ToString

Use this method to convert the StringBuilder instance to a string and assign it to a String variable. The ToString method returns the string represented by the StringBuilder variable to which it's applied.

Handling Dates

The Date Time Class

The DateTime class is used for storing date and time values, and it's one of the Framework's base data types. Date and time values are stored internally as Double numbers. The integer part of the value corresponds to the date, and the fractional part corresponds to the time. To convert a DateTime variable to a Double value, use the method ToOADateTime, which returns a value that is an OLE (Object Linking and Embedding) Automation-compatible date. The value 0 corresponds to midnight of December 30, 1899.

To initialize a DateTime variable, supply a date value enclosed in a pair of pound symbols. If the value contains time information, separate it from the date part by using a space:

```
Dim date1 As Date = #4/15/2007#
Dim date2 As Date = #4/15/2007 2:01:59#
```

Properties

The DateTime class exposes the following properties, which are straightforward.

Date, TimeOfDay

The Date property returns the date from a date/time value and sets the time to midnight. The TimeOfDay property returns the time part of the date. The following statements

```
Dim date1 As DateTime
date1 = Now()
Debug.WriteLine(date1)
Debug.WriteLine(date1.Date)
Debug.WriteLine(date1.TimeOfDay)
```

will print something like the following values in the Output window:

```
8/5/2007 9:41:55 AM
8/5/2007 12:00:00 AM
09:41:55.5296000
```

DayOfWeek, DayOfYear

Hour, Minute, Second, Millisecond

These properties return the corresponding time part of the date value passed as an argument. If the current time is 9:47:24 p.m., the three properties of the DateTime class will return the integer values 9, 47, and 24 when applied to the current date and time:

```
Debug.WriteLine("The current time is " & Date.Now.ToString)
Debug.WriteLine("The hour is " & Date.Now.Hour)
Debug.WriteLine("The minute is " & Date.Now.Minute)
Debug.WriteLine("The second is " & Date.Now.Second)
```

Day, Month, Year

These three properties return the day of the month, the month, and the year of a DateTime value, respectively. The Day and Month properties are numeric values, but you can convert them to the appropriate string (the name of the day or month) with the WeekDayName() and MonthName() functions.

Ticks

This property returns the number of ticks from a date/time value. Each tick is 100 nanoseconds (or 0.0001 milliseconds). To convert ticks to milliseconds, multiply them by 10,000 (or use the TimeSpan object's TicksPerMillisecond property.

Methods

The DateTime class exposes several methods for manipulating dates. The most practical methods add and subtract time intervals to and from an instance of the DateTime class.

Compare

Compare is a shared method that compares two date/time values and returns an integer value indicating the relative order of the two values. The syntax of the Compare method is the following, where date1 and date2 are the two values to be compared:

order = System.DateTime.Compare(date1, date2)

DaysInMonth

This shared method returns the number of days in a specific month. Because February contains a variable number of days depending on the year, the DaysInMonth method accepts as arguments both the month and the year:

monDays = DateTime.DaysInMonth(year, month)

FromOADate

This shared method creates a date/time value from an OLE Automation-compatible date.

newDate = DateTime.FromOADate(dtvalue)

The argument dtvalue must be a Double value in the range from −657,434 (first day of year 100) to 2,958,465 (last day of year 9999).

IsLeapYear

This shared method returns a True/False value that indicates whether the specified year is a leap year:

Dim leapYear As Boolean = DateTime.IsLeapYear(year)

Add

This method adds a TimeSpan object to the current instance of the DateTime class.

```
Dim TS As New TimeSpan()
Dim thisMoment As Date = Now()
TS = New TimeSpan(3, 6, 2, 50)
Debug.WriteLine(thisMoment)
Debug.WriteLine(thisMoment.Add(TS))
```

The values printed in the Output window when I tested this code segment were as follows:

```
9/1/2007 10:10:49 AM
9/4/2007 4:13:39 PM
```

Subtract

This method is the counterpart of the Add method; it subtracts a TimeSpan object from the current instance of the DateTime class and returns another Date value.

Adding Intervals to Dates

Various methods add specific intervals to a date/time value. Each method accepts the number of intervals to add (days, hours, milliseconds, and so on) to the current instance of the DateTime class. These methods are the following: AddYears, AddMonths, AddDays, AddHours, AddMinutes, AddSeconds, AddMilliseconds, and AddTicks.

To add 3 years and 12 hours to the current date, use the following statements:

```
Dim aDate As Date
aDate = Now()
aDate = aDate.AddYears(3)
aDate = aDate.AddHours(12)
```

If the argument is a negative value, the corresponding intervals are subtracted from the current instance of the class.

ToString

This method converts a date/time value to a string, using a specific format. The DateTime class recognizes numerous format patterns, which are listed in the following two tables. Table lists the standard format patterns, and Table lists the characters that can format individual parts of the date/time value. You can combine the custom format characters to format dates and times in any way you wish.

The syntax of the ToString method is the following, where formatSpec is a format specification:

```
aDate.ToString(formatSpec)
```

The D named date format, for example, formats a date value as a long date; the following statement will return the highlighted string shown below the statement:

```
Debug.Writeline(#9/17/2010#.ToString("D"))
Friday, September 17, 2010
```

Table - lists the named formats for the standard date and time patterns. The format characters are case-sensitive — for example, g and G represent slightly different patterns.

Named Format	Output	Format Name
d	MM/dd/yyyy	ShortDatePattern
D	dddd, MMMM dd, yyyy	LongDatePattern
F	dddd, MMMM dd, yyyy HH:mm:ss.mmm	FullDateTimePattern (long date and long time)
f	dddd, MMMM dd, yyyy HH:mm.ss	FullDateTimePattern (long date and short time)
g	MM/dd/yyyy HH:mm	general (short date and short time)
G	MM/dd/yyyy HH:mm:ss	General (short date and long time)
M	m MMMM dd	MonthDayPattern (month and day)
r, R	ddd, dd MMM yyyy HH:mm:ss GMT	RFC1123Pattern

Table: Date Format Specifier

Format Character	Description
d	The date of the month
dd	The day of the month with a leading zero for single-digit days
ddd	The abbreviated name of the day of the week (a member of the AbbreviatedDayNames enumeration)
dddd	The full name of the day of the week (a member of the DayNamesFormat enumeration)
M	The number of the month
MM	The number of the month with a leading zero for single-digit months
MMM	The abbreviated name of the month (a member of the AbbreviatedMonthNames enumeration)
MMMM	The full name of the month

The following examples format the current date by using all the format patterns listed in Table. An example of the output produced by each statement is shown under each statement, indented and highlighted.

```
Debug.WriteLine(now().ToString("d"))
6/1/2008
Debug.WriteLine(now().ToString("D"))
Sunday, June 01, 2008
Debug.WriteLine(now().ToString("f"))
Sunday, June 01, 2008 10:29 AM
Debug.WriteLine(now().ToString("F"))
Sunday, June 01, 2008 10:29:35 AM
```

```
Debug.WriteLine(now().ToString("g"))
6/1/2008 10:29 AM
Debug.WriteLine(now().ToString("G"))
6/1/2008 10:29:35 AM
```

To display the full month name and the day in the month, for instance, use the following statement:

```
Debug.WriteLine(now().ToString("MMMM d")).
```

Date Conversion Methods

The DateTime class supports methods for converting a date/time value to many of the other base types, which are presented here briefly.

ToFileTime, FromFileTime

The ToFileTime method converts the value of the current Date instance to the format of the local system file time. There's also an equivalent FromFileTime method, which converts a file time value to a Date value.

ToLongDateString, ToShortDateString

These two methods convert the date part of the current DateTime instance to a string with the long (or short) date format. The following statement will return a value like the one highlighted, which is the long date format:

```
Debug.WriteLine(Now().ToLongDateString)
Tuesday, July 15, 2008
```

ToLongTimeString, ToShortTimeString

These two methods convert the time part of the current instance of the Date class to a string with the long (or short) time format. The following statement will return a value like the one highlighted:

```
Debug.WriteLine(Now().ToLongTimeString)
6:40:53 PM
```

ToOADate

This method converts the DateTime instance into an OLE Automation-compatible date (a long value).

ToUniversalTime, ToLocalTime

ToUniversalTime converts the current instance of the DateTime class into universal coordinated time (UCT). The method ToLocalTime converts a UCT time value to local time.

Dates as Numeric Values

The Date type encapsulates complicated operations, and it's worth taking a look at the inner workings of the classes that handle dates and times. Let's declare two variables to experiment a little with dates: a Date variable, which is initialized to the current date, and a Double variable.

```
Dim Date1 As Date = Now()
Dim dbl As Double
```

Insert a couple of statements to convert the date to a Double value and print it:

```
dbl = Date1.ToOADate
Debug.WriteLine(dbl)
```

The TimeSpan Class

The last class discussed in this chapter is the TimeSpan class, which represents a time interval and can be expressed in many different units — from ticks and milliseconds to days. The TimeSpan is usually the difference between two date/time values, but you can also create a TimeSpan for a specific interval and use it in your calculations.

To use the TimeSpan variable in your code, just declare it with a statement such as the following:

```
Dim TS As New TimeSpan
```

You can initialize an instance of the TimeSpan object by creating two date/time values and getting their difference, as in the following statements:

```
Dim TS As New TimeSpan
Dim date1 As Date = #4/11/1985#
Dim date2 As Date = Now()
TS = date2.Subtract(date1)
Debug.WriteLine(TS)
```

Depending on the day on which you execute these statements, they will print something like the following in the Output window:

```
8086.15:37:01.6336000
```

Properties

The TimeSpan type exposes the properties described in the following sections. Most of these properties are shared.

Field Properties

TimeSpan exposes the simple properties shown in Table 13.3, which are known as fields and are all shared.

Table: The Fields of the TimeSpan Object

Property	Returns
Empty	An Empty TimeSpan object
MaxValue	The largest interval you can represent with a TimeSpan object
MinValue	The smallest interval you can represent with a TimeSpan object

TicksPerDay	The number of ticks in a day
TicksPerHour	The number of ticks in an hour
TicksPerMillisecond	The number of ticks in a millisecond
TicksPerMinute	The number of ticks in one minute
TicksPerSecond	The number of ticks in one second
Zero	A TimeSpan object of zero duration

Interval Properties

In addition to the fields, the TimeSpan class exposes two more groups of properties that return the various intervals in a TimeSpan value (shown in Tables 13.4 and 13.5). The members of the first group of properties return the number of specific intervals (days, hours, and so on) in a TimeSpan value. The second group of properties returns the entire TimeSpan's duration in one of the intervals recognized by the TimeSpan method.

Table 9.4: The Intervals of a TimeSpan Value

Property	Returns
Days	The number of whole days in the current TimeSpan.
Hours	The number of whole hours in the current TimeSpan.
Milliseconds	The number of whole milliseconds in the current TimeSpan. The largest value of this property is 999.
Minutes	The number of whole minutes in the current TimeSpan. The largest value of this property is 59.
Seconds	The number of whole seconds in the current TimeSpan. The largest value of this property is 59.
Ticks	The number of whole ticks in the current TimeSpan.

Table : The Total Intervals of a TimeSpan Value

Property	Returns
TotalDays	The number of days in the current TimeSpan
TotalHours	The number of hours in the current TimeSpan
TotalMilliseconds	The number of whole milliseconds in the current TimeSpan
TotalMinutes	The number of whole minutes in the current TimeSpan

Duration

This property returns the duration of the current instance of the TimeSpan class. The duration is expressed as the number of days followed by the number of hours, minutes, seconds, and milliseconds. The following statements create a TimeSpan object of a few seconds (or minutes, if you don't mind waiting) and print its duration in the Output window.

```
Dim T1, T2 As DateTime
T1 = Now
MsgBox("Click OK to continue")
T2 = Now
Dim TS As TimeSpan
TS = T2.Subtract(T1)
Debug.WriteLine("Total duration = " & TS.Duration.ToString)
Debug.WriteLine("Minutes = " & TS.Minutes.ToString)
Debug.WriteLine("Seconds = " & TS.Seconds.ToString)
Debug.WriteLine("Ticks = " & TS.Ticks.ToString)
Debug.WriteLine("Milliseconds = " & TS.TotalMilliseconds.ToString)
Debug.WriteLine("Total seconds = " & TS.TotalSeconds.ToString)
```

If you place these statements in a button's Click event handler and execute them, you'll see a series of values like the following in the Immediate window:

Total duration = 00:01:34.2154752

Minutes = 1

Seconds = 34

Ticks = 942154752

Milliseconds = 94215,4752

Total seconds = 94,2154752

Methods

There are various methods for creating and manipulating instances of the TimeSpan class, and they're described in the following sections.

Interval Methods

The methods in Table 13.6 create a new TimeSpan object of a specific duration. The TimeSpan's duration is specified as a number of intervals, accurate to the nearest millisecond. All methods accept a single argument, which is a Double value that represents the number of the corresponding intervals (days, hours, and so on).

Parse(string)

This method creates a new TimeSpan object from a string with the TimeSpan format (days;followed by a period; followed by the hours, minutes, and seconds separated by colons). The following statements create a new TimeSpan variable with a duration of 3 days, 12 hours, 20 minutes, 30 seconds, and 500 milliseconds:

```
Dim SP As New TimeSpan()
SP = TimeSpan.Parse("3.12:20:30.500")
Debug.WriteLine(SP)
```

3.12:20:30.5000000

Accessing Files and Folders

The Directory Class

The System.IO.Directory class exposes all the members you need to manipulate folders. Because the Directory class belongs to the System.IO namespace, you must import the IO namespace into any project that might require the Directory object's members with the following statement:

Imports System.IO

Methods

The Directory object exposesmethods for accessing folders and their contents, which are described in the following sections.

- CreateDirectory
- Delete
- Exists
- Move
- GetCurrentDirectory, SetCurrentDirectory
- GetDirectoryRoot
- GetDirectories
- GetFiles
- GetFileSystemEntries
- GetCreationTime, SetCreationTime
- GetLastAccessTime, SetLastAccessTime
- GetLastWriteTime, SetLastWriteTime
- GetLogicalDrives
- GetParent

CreateDirectory

This method creates a new folder, whose path is passed to the method as a string argument:

Directory.CreateDirectory(path)

The CreateDirectory method returns a DirectoryInfo object, which contains information about the newly created folder. The DirectoryInfo object is discussed later in this chapter, along with the FileInfo object. Notice that the CreateDirectory method can create multiple nested folders in a single call. The following statement will create the folder folder1 (if it doesn't exist), folder2 (if it doesn't exist) under folder1, and finally folder3 under folder2 in the C: drive:

```
Directory.CreateDirectory("C:\folder1\folder2\folder3")
```

Delete

This method deletes a folder and all the files in it. If the folder contains subfolders, the Delete method will optionally remove the entire directory tree under the node you're removing. The simplest form of the Delete method accepts as an argument the path of the folder to be deleted:

```
Directory.Delete(path)
```

To delete a folder recursively (that is, also delete any subfolders under it), use the following form of the Delete method, which accepts a second argument:

```
Directory.Delete(path, recursive)
```

Exists

This method accepts a path as an argument and returns a True/False value indicating whether the specified folder exists:

```
Directory.Exists(path)
```

Move

This method moves an entire folder to another location in the file system; its syntax is the following, where source is the name of the folder to be moved and destination is the name of the destination folder:

Directory.Move(source, destination)

GetCurrentDirectory, SetCurrentDirectory

Use these methods to retrieve and set the path of the current directory. By default, the GetCurrentDirectory method returns the folder in which the application is running. SetCurrentDirectory accepts a string argument, which is a path, and sets the current directory to the specified path. You can change the current folder by specifying an absolute or a relative path, such as the following:

Directory.SetCurrentDirectory("..\Resources")

GetDirectoryRoot

This method returns the root part of the path passed as argument, and its syntax is the following:

root = Directory.GetDirectoryRoot(path)

GetDirectories

This method retrieves all the subfolders of a specific folder and returns their names as an array of strings:

Dim Dirs() As String

Dirs = Directory.GetDirectories(path)

GetFiles

This method returns the names of the files in the specified folder as an array of strings. The syntax of the GetFiles method is the following, where path is the path of the folder whose files you want to retrieve and files is an array of strings that's filled with the names of the files:

Dim files() As String = Directory.GetFiles(path)

GetCreationTime, SetCreationTime

These methods read or set the date that a specific folder was created. The GetCreationTime method accepts a path as an argument and returns a Date value:

Dim CreatedOn As Date

CreatedOn = Directory.GetCreationTime(path)

SetCreationTime accepts a path and a date value as arguments and sets the specified folder's creation time to the value specified by the second argument:

Directory.SetCreationTime(path, datetime)

GetLastAccessTime, SetLastAccessTime

These two methods are equivalent to the GetCreationTime and SetCreationTime methods, except they return and set the most recent date and time that the file was accessed.

GetLastWriteTime, SetLastWriteTime

These two methods are equivalent to the GetCreationTime and SetCreationTime methods, but they return and set the most recent date and time the file was written to.

GetLogicalDrives

This method returns an array of strings, which are the names of the logical drives on the computer. The statements in Listing 11.5 print the names of all logical drives.

Listing: Retrieving the Names of All Drives on the Computer

```
Dim drives() As String
drives = Directory.GetLogicalDrives
Dim drive As String
For Each drive In drives
Debug.WriteLine(drive)
Next
```

When executed, these statements will produce a list such as the following:

```
C:\
D:\
E:\
F:\
```

Notice that the GetLogicalDrives method doesn't return any floppy drives, unless there's a disk inserted into the drive.

GetParent

This method returns a DirectoryInfo object that represents the properties of a folder's parent folder. The syntax of the GetParent method is as follows:

```
Dim parent As DirectoryInfo = Directory.GetParent(path)
```

The name of the parent folder, for example, is parent.Name, and its full name is parent.FullName.

The File Class

The System.IO.File class exposes methods for manipulating files (copying them, moving them around, opening them, and closing them), similar to the methods of the Directory class. The names of the methods are self-descriptive, and most of them accept as an argument the path of the file on which they act. Use these methods to implement the common operations that users normally perform through the Windows interface, from within your application.

Methods

Many of the following methods allow you to open existing or create new files. We'll use some of these methods later in the chapter to write data to, and read from, text and binary files.

AppendText

This method appends some text to a file, whose path is passed to the method as an argument, along with the text to be written:

File.AppendText(path, text)

Copy

This method copies an existing file to a new location; its syntax is the following, where source is the path of the file to be copied and destination is the path where the file will be copied to:

File.Copy(source, destination)

The destination file exists, the Copy method will fail. An exception will be thrown also if either the source or the destination folder does not exist.

Create

This method creates a new file and returns a FileStream object, which you can use to write to or read from the file. (The FileStream object is discussed in detail later in this chapter, along with the methods for writing to or reading from the file.) The simplest form of the Create method accepts a single argument, which is the path of the file you want to create:

Dim FStream As FileStream = File.Create(path)

CreateText

This method is similar to the Create method, but it creates a text file and returns a StreamWriter object for writing to the file. The StreamWriter object is similar to the FileStream object but is used for text files only, whereas the FileStream object can be used with both text and binary files.

Dim SW As StreamWriter = File.CreateText(path)

Delete

This method removes the specified file from the file system. The syntax of the Delete method is the following, where path is the path of the file you want to delete:

File.Delete(path)

Exists

This method accepts as an argument the path of a file and returns a True/False value that indicates whether a file exists. The following statements delete a file, after making sure that the file exists:

```
If File.Exists(path) Then
File.Delete(path)
Else
MsgBox("The file " & path & " doesn't exist")
End If
```

GetAttributes

The GetAttributes method accepts a file path as an argument and returns the attributes of the specified file as a FileAttributes object. A file can have more than a single attribute (for instance, it can be hidden and compressed).

GetCreationTime, SetCreationTime

The GetCreationTime method returns a date value, which is the date and time the file was created. This value is set by the operating system, but you can change it with the SetCreationTime method. SetCreationTime accepts as an argument the file's path and the new creation time:

File.SetCreationTime(path, datetime)

GetLastAccessTime, SetLastAccessTime

The GetLastAccessTime method returns a date value, which is the date and time the specified file was accessed for the last time. Use the SetLastAccessTime method to set this value.

GetLastWriteTime, SetLastWriteTime

The GetLastWriteTime method returns a date value, which is the date and time that the specified file was written to for the last time. To change this attribute, use the SetLastWriteTime method.

Move

This method moves the specified file to a new location. You can also use the Move method to rename a file by simply moving it to another name in the same folder. Moving a file is equivalent to copying it to another location and then deleting the original file. The Move method works across volumes:

File.Move(sourceFileName, destFileName)

Open

This method opens an existing file for read-write operations. The simplest form of the method is the following, which opens the file specified by the path argument and returns a FileStream object to this file:

FStream = File.Open(path)

You can use the FStream object's methods to write to or read from the file. The following form of the method allows you to specify the mode in which you want to open the file, where the fileMode argument can have one of the values shown in Table 11.3.

FStream = File.Open(path, fileMode)

Table: FileMode Enumeration

Value	Effect
Append	Opens the file in write mode, and all the data you write to the file are appended to its existing contents.
Create	Requests the creation of a new file. If a file by the same name exists, this will be overwritten.
CreateNew	Requests the creation of a new file. If a file by the same name exists, an exception will be thrown. This mode will create and open a file only if it doesn't already exist and it's the safest mode.
Open	Requests that an existing file be opened.
OpenOrCreate	Opens the file in read-write mode if the file exists, or creates a new file and opens it in read-write mode if the file doesn't exist.
Truncate	Opens an existing file and resets its size to zero bytes. As you can guess, this file must be opened in write mode.

OpenRead

This method opens an existing file in read mode and returns a FileStream object associated with this file. You can use this stream to read from the file. The syntax of the OpenRead method is the following:

Dim FStream As FileStream = File.OpenRead(path)

The OpenRead method is equivalent to opening an existing file with read-only access via the Open method.

OpenText

This method opens an existing text file for reading and returns a StreamReader object associated with this file. Its syntax is the following:

Dim SR As StreamReader = File.OpenText(path)

OpenWrite

This method opens an existing file in write mode and returns a FileStrem object associated with this file. The syntax of the OpenRead method is as follows, where path is the path of the file:

Dim FStream As FileStream = File.OpenWrite(path)

The DirectoryInfo Class

To create a new instance of the DirectoryInfo class that references a specific folder, supply the folder's path in the class's constructor:

Dim DI As New DirectoryInfo(path)

CreateSubdirectory

This method creates a subfolder under the folder specified by the current instance of the class, and its syntax is as follows:

DI.CreateSubdirectory(path)

GetFileSystemInfos

This method returns an array of FileSystemInfo objects, one for each item in the folder referenced by the current instance of the class. The items can be either folders or files. To retrieve information about all the entries in a folder, create an instance of the DirectoryInfo class and then call its GetFileSystemInfos method:

```
Dim DI As New DirectoryInfo(path)
Dim itemsInfo() As FileSystemInfo
itemsInfo = DI.GetFileSystemInfos()
```

The FileInfo Class

The FileInfo class exposes many properties and methods, which are equivalent to the members of the File class, so I'm not going to repeat all of them here. The Copy/Delete/Move methods allow you to manipulate the file represented by the current instance of the FileInfo class, similar to the methods by the same name of the File class.

Length Property

This property returns the size of the file represented by the FileInfo object in bytes. The File class doesn't provide an equivalent property or method.

CreationTime, LastAccessTime, LastWriteTime Properties

These properties return a date value, which is the date the file was created, accessed for the last time, or written to for the last time, respectively. They are equivalent to the methods of the File object by the same name and the Get prefix.

Name, FullName, Extension Properties

These properties return the filename, full path, and extension, respectively, of the file represented by the current instance of the FileInfo class. They have no equivalents in the File class because the File class's methods require that you specify the path of the file, so its path and extension are known.

CopyTo, MoveTo Methods

These two methods copy or move, respectively, the file represented by the current instance of the FileInfo class. Both methods accept a single argument, which is the destination of the operation (the path to which the file will be copied or moved). If the destination file

exists already, you can overwrite it by specifying a second optional argument, which has a True/False value:

FileInfo.CopyTo(path, force)

Directory Method

This method returns a DirectoryInfo value that contains information about the file's parent directory.

DirectoryName Method

This method returns a string with the name of the file's parent directory. The following statements return the two (identical) strings shown highlighted in this code segment:

Dim FI As FileInfo

FI = New FileInfo("c:\folder1\folder2\folder3\test.txt")

Debug.WriteLine(FI.Directory().FullName)

c:\folder1\folder2\folder3

Debug.WriteLine(FI.DirectoryName()) c:\folder1\folder2\folder3

The Path Class

The Path class contains an interesting collection of methods, which you can think of as utilities. The Path class's methods perform simple tasks such as retrieving a file's name and extension, returning the full path description of a relative path, and so on. The Path class's members are shared, and you must specify the path on which they will act as an argument.

Properties

The Path class exposes the following properties. Notice that none of these properties applies to a specific path; they're general properties that return settings of the operating system. The FileSystem component doesn't provide equivalent properties to the ones discussed in this section.

DirectorySeparatorChar

This property returns the directory separator character, which is the backslash character (\).

InvalidPathChars

This property returns the list of invalid characters in a path as an array of the following characters:

/ \ " < > —

You can use these characters to validate user input or pathnames read from a file. If you have a choice, let the user select the files through the Open dialog box, so that their pathnames will always be valid.

PathSeparator, VolumeSeparatorChar

These properties return the separator characters that appear between multiple paths (:) and volumes (;), respectively.

Methods

The most useful methods exposed by the Path class are utilities for manipulating filenames and pathnames, described in the following sections. Notice that the methods of the Path class are shared: You must specify the path on which they will act as an argument.

ChangeExtension

This method changes the extension of a file. Its syntax is as follows:

newExtension = Path.ChangeExtension(path, extension)

Combine

This method combines two path specifications into one. Its syntax is as follows:

newPath = Path.Combine(path1, path2)

Use this method to combine a folder path with a file path. The following expression will return the highlighted string:

Path.Combine("c:\textFiles", "test.txt")

c:\textFiles\test.txt

GetDirectoryName

This method returns the directory name of a path. The following statement:

Path.GetDirectoryName("C:\folder1\folder2\folder3\Test.txt")

will return this string:

C:\folder1\folder2\folder3

GetFileName, GetFileNameWithoutExtension

These two methods return the filename in a path, with and without its extension, respectively.

GetFullPath

This method returns the full path of the specified path; you can use it to convert relative pathnames to fully qualified pathnames. The following statement returned the highlighted string on my computer (it will be quite different on your computer, depending on the current directory):

Console.WriteLine(Path.GetFullPath("..\..\Test.txt"))

C:\WorkFiles\Learn VB\Chapters\Chapter 11\Projects\Test.txt

GetTempFile, GetTempPath

The GetTempFile method returns a unique filename, which you can use as a temporary storage area from within your application. The name of the temporary file can be anything, because no user will ever access it. In addition, the GetTempFile method creates a zero-length file on the disk, which you can open with the Open method. A typical temporary filename is the following:

C:\DOCUME~1\TOOLKI~1\LOCALS~1\Temp\tmp105.tmp

It was returned by the following statement on my system:

Debug.WriteLine(Path.GetTempFile)

HasExtension

This method returns a True/False value, indicating whether a path includes a file extension.

Accessing Files

There are two types of files: text files and binary files. To access a file, you must first set up a Stream object. Stream objects are created by the various methods that open or create files, as you have seen in the previous sections, and they return information about the file they're connected to.

Using Streams

Another benefit of using streams is that you can combine them. The typical example is that of encrypting and decrypting data. Data is encrypted through a special type of Stream, the CryptoStream.

The FileStream Class

The Stream class is an abstract one, and you can't use it directly in your code. To prepare your application to write to a file, you must set up a FileStream object, which is the channel between your application and the file. The methods for writing and reading data are provided by the StreamReader/StreamWriter or BinaryReader/BinaryWriter classes, which are created on top of the FileStream object.

Properties

You can use the following properties of the FileStream object to retrieve information about the underlying file.

Length

This read-only property returns the length of the file associated with the FileStream current object in bytes.

Position

This property gets or sets the current position within the stream. You can compare the Position property to the Length property to find out whether you have reached the end of an existing file. When these two properties are equal, there are no more data to read.

Methods

The FileStream object exposes a fewmethods, which are discussed here. Themethods for accessing a file's contents are discussed in the following section.

Lock

This method allows you to lock the file you're accessing, or part of it. The syntax of the Lock method is the following, where position is the starting position and length is the length of the range to be locked:

Lock(position, length)

To lock the entire file, use this statement:

FileStream.Lock(1, FileStream.Length)

Seek

This method sets the current position in the file represented by the FileStream object:

FileStream.Seek(offset, origin)

The new position is offset bytes from the origin. In place of the origin argument, use one of he SeekOrigin enumeration members, listed in Table.

Table : SeekOrigin Enumeration

Value	Effect
Begin	The offset is relative to the beginning of the file.
Current	The offset is relative to the current position in the file.
End	The offset is relative to the end of the file.

SetLength

This method sets the length of the file represented by the FileStream object. Use this method after you have written to an existing file to truncate its length. The syntax of the SetLength method is this:

FileStream.SetLength(newLength)

The StreamWriter Class

The StreamWriter class is the channel through which you send data to a text file. To create a new StreamWriter object, declare a variable of the StreamWriter type. The first overloaded form of the constructor accepts a file's path as an argument and creates a new StreamWriter object for the file:

Dim SW As New StreamWriter(path)

NewLine Property

The StreamWriter object provides a handy property, the NewLine property, which allows you to change the string used to terminate each line in the file. This terminator is written to the text file by the WriteLine method, following the text. The default line-terminator string is a carriage return followed by a line feed (\r\n). The StreamReader object doesn't provide a similar property. It reads lines terminated by the carriage return (\r), line feed (\n), or carriage return/line feed (\r\n) characters only.

Methods

To send information to the underlying file, use the following methods of the StreamWriter object.

AutoFlush

This property is a True/False value that determines whether the methods that write to the file (the Write and WriteString methods) will also flush their buffer. If you set this property to False, the buffer will be flushed when the operating system gets a chance, when the Flush method is called, or when you close the FileStream object. When AutoFlush is True, the buffer is flushed with every write operation.

Close

This method closes the StreamWriter object and releases the resources associated with it to the system. Always call the Close method after you finish using the StreamWriter object. If you have created the StreamWriter object on top of a FileStream object, you must also close the underlying stream too.

Flush

This method writes any data in the buffer to the underlying file.

WriteLine(data)

This method is identical to the Write method, but it appends a line break after saving the data to the file. You will find examples on using the StreamWriter class after we discuss the methods of the StreamReader class.

The StreamReader Class

The StreamReader class provides the necessary methods for reading from a text file and exposes methods that match those of the StreamWriter class (the Write and WriteLine methods). The StreamReader class's constructor is overloaded. You can specify the FileStream

object it will use to read data from the file, the encoding scheme, and the buffer size. The simplest form of the constructor is the following:

Dim SR As New StreamReader(FS)

Methods

The StreamReader class provides the following methods for writing data to the underlying file.

Close

The Close method closes the current instance of the StreamReader class and releases any system resources associated with this object.

Peek

The Peek method returns the next character as an integer value, without actually removing it from the input stream. The Peek method doesn't change the current position in the stream. If there are no more characters left in the stream, the value −1 is returned. The Peek method will also return −1 if the current stream doesn't allow peeking.

Read

This method reads a number of characters from the StreamReader class to which it's applied and returns the number of characters read. The syntax of the Read method is as follows, where count is the number of characters to be read, starting at the startIndex location in the file:

charsRead = SR.Read(chars, startIndex, count)

ReadBlock

This method reads a number of characters from a text file and stores them in an array of characters. It accepts the same arguments as the Read method and returns the number of characters read.

```
Dim chars(count - 1) As Char
charsRead = SR.Read(chars, startIndex, count)
```

ReadLine

This method reads the next line from the text file associated with the StreamReader class and returns a string. If you're at the end of the file, the method returns the Null value. The syntax of the ReadLine method is the following:

```
Dim txtLine As String
txtLine = SR.ReadLine()
```

ReadToEnd

The last method for reading characters from a text file reads all the characters from the current position to the end of the file. We usually call this method once to read the entire file with a single statement and store its contents to a string variable. The syntax of the ReadToEnd method is as follows:

```
allText = SR.ReadToEnd()
```

The BinaryWriter Class

To prepare your application to write to a binary file, you must set up a BinaryWriter object, with the statement shown here, where FS is a properly initialized FileStream object:

```
Dim BW As New BinaryWriter(FS)
```

To specify the encoding of the text in the binary file, use the following form of the method:

```
Dim BW As New BinaryWriter(FS, encoding)
Dim BW As New BinaryWriter(path, encoding)
```

Methods

The BinaryWriter class exposes the following methods for manipulating binary files.

Close

This method flushes and closes the current BinaryWriter and releases any system resources associated with it.

Flush

This method clears all buffers for the current writer and writes all buffered data to the underlying file.

Seek

This method sets the position within the current stream. Its syntax is the following, where origin is a member of the SeekOrigin enumeration and offset is the distance from the origin:

Seek(offset, origin)

Write

The Write method writes a value to the current stream. This method is heavily overloaded, but it accepts a single argument, which is the value to be written to the file. The data type of its argument determines how it will be written. The Write method can save all the base types to the file in their native format, unlike the Write method of the TextWriter class, which stores them as strings.

WriteString

Whereas all other data types can be written to a binary file with the Write method, strings must be written with the WriteString method. This method writes a length-prefixed

string to the file and advances the current position by the appropriate number of bytes. The string is encoded by the current encoding scheme, and the default value is UTF8Encoding.

The BinaryReader Class

The BinaryReader class provides the methods you need to read data from a binary file. As you have seen, binary files might also hold text, and the BinaryReader class provides the ReadString method to read strings written to the file by the WriteString method.

To use the methods of the BinaryReader class in your code, you must first create an instance of the class. The BinaryReader object must be associated with a FileStream object, and the simplest form of its constructor is the following, where streamObj is the FileStream object:

```
Dim BR As New BinaryReader(streamObj)
```

.

Methods

The BinaryReader class exposes the following methods for accessing the contents of a binary file.

Close

This method is the same as the Close method of the StreamReader class. It closes the current reader and releases the underlying stream.

PeekChar

This method returns the next available character from the streamwithout repositioning the current pointer. The character read is returned as an integer, or −1 if there are no more characters to be read from the stream.

Drawing and Painting with Visual Basic

In general, graphics fall into two major categories: vector and bitmap. Vector graphics are images generated by graphics methods such as DrawLine and DrawEllipse. The drawing you create is based on mathematical descriptions of the various shapes. Bitmap graphics are images made up of pixels arranged in rows and columns. Each pixel is represented by a Long numeric value, which is the pixel's color.

Display and size images. - The most appropriate control for displaying images is the PictureBox control. You can assign an image to the control through its Image property, either at design time or at runtime. To display a user-supplied image at runtime, call the DrawImage method of the control's Graphics object.

Generate graphics by using the drawing methods. - Every object you draw on, such as forms and PictureBox controls, exposes the CreateGraphics method, which returns a Graphics object. The Paint event's e argument also exposes the Graphics object of the control or form. To draw something on a control, retrieve its Graphics object and then call the Graphics object's drawing methods.

Display text in various ways, including gradient fills. - The Graphics object provides the DrawString method, which prints a user-supplied string on a control. You can also specify the coordinates of the string's upper-left corner and its font. To position the string, you need to know its dimensions..

Drawing with GDI+

The most recent version on GDI is called GDI+.One of the basic characteristics of GDI is that it's stateless. This means that each graphics operation is totally independent of the previous one and can't affect the following one. To draw a line, you must specify a Pen object and the two endpoints of the line.

The GDI+ classes reside in the following namespaces, and you must import one or more of them in your projects: System.Drawing, System.Drawing2D,

System.Drawing.Imaging, and System.Drawing.Text. This chapter explores all three aspects of GDI+ — namely vector drawing, imaging, and typography.

Here are the statements to draw a line on the form:

```
Dim redPen As Pen = New Pen(Color.Red, 2)
Dim point1 As Point = New Point(10,10)
Dim point2 As Point = New Point(120,180)
Me.CreateGraphics.DrawLine(redPen, point1, point2)
```

The Basic Drawing Objects

This is a good point to introduce some of the objects we'll be using all the time when drawing. No matter what you draw or which drawing instrument you use, one or more of the objects discussed in this section will be required.

The Graphics Object

The Graphics object is the drawing surface — your canvas. All the controls you can draw on expose a Graphics property, which is an object, and you can retrieve it with the CreateGraphics method. Start by declaring a variable of the Graphics type and initialize it to the Graphics object returned by the control's CreateGraphics method:

```
Dim G As Graphics
G = PictureBox1.CreateGraphics
```

DpiX, DpiY - These two properties return the horizontal and vertical resolutions of the drawing surface, respectively. Resolution is expressed in pixels per inch (or dots per inch, if the drawing surface is your printer). On an average monitor, these two properties return a resolution of 96 dots per inch (dpi).

PageUnit - This property determines the units in which you want to express the coordinates on the Graphics object; its value can be a member of the GraphicsUnit enumeration

TextRenderingHint - This property specifies how the Graphics object will render text; its value is one of the members of the TextRenderingHint enumeration: AntiAlias, AntiAliasGrid-Fit, ClearTypeGridFit, SingleBitPerPixel, SingleBitPerPixelGridFit, and SystemDefault.

SmoothingMode - This property is similar to the TextRenderingHint, but it applies to shapes drawn with the Graphics object's drawing methods. Its value is one of the members of the SmoothingMode enumeration: AntiAlias, Default, HighQuality, HighSpeed, Invalid, and None.

The Point Class

The Point class represents a point on the drawing surface and is expressed as a pair of (x, y) coordinates. The x-coordinate is its horizontal distance from the origin, and the y-coordinate is its vertical distance from the origin. The origin is the point with coordinates (0, 0), and this is the top-left corner of the drawing surface.

The Rectangle Class

Another class that is often used in drawing is the Rectangle class. The Rectangle object is used to specify areas on the drawing surface. Its constructor accepts as arguments the coordinates of the rectangle's top-left corner and its dimensions:

```
Dim box As Rectangle
box = New Rectangle(X, Y, width, height)
```

The following statement creates a rectangle whose top-left corner is 1 pixel to the right and 1 pixel down from the origin, and its dimensions are 100 by 20 pixels:

```
box = New Rectangle(1, 1, 100, 20)
```

The Size Class

The Size class represents the dimensions of a rectangle; it's similar to a Rectangle object, but it doesn't have an origin, just dimensions. To create a new Size object, use the following constructor:

```
Dim S1 As New Size(100, 400)
```

The Color Class

The Color class represents colors, and there are many ways to specify a color. We'll discuss the Color class in more detail in Chapter 19, "Manipulating Images and Bitmaps." In the meantime, you can specify colors by name. Declare a variable of the Color type and initialize it to one of the named colors exposed as properties of the Color class:

```
Dim myColor As Color
myColor = Color.Azure
```

The Font Class

The Font class represents fonts, which are used when rendering strings via the DrawString method. To specify a font, you must create a new Font object; set its family name, size, and style; and then pass it as argument to the DrawString method. To create a new Font object, use a statement like the following:

```
Dim drawFont As New Font(”Verdana”, 12, FontStyle.Bold)
```

The Pen object exposes these properties:

Alignment - Determines the alignment of the Pen, and its value is one of the members of the PenAlignment enumeration: Center or Inset. When set to Center, the width of the pen is centered on the outline (half the width is inside the shape, and half is outside). When set to Inset, the entire width of the pen is inside the shape. The default value of this property isPenAlignment.Center.

LineJoin - Determines how two consecutive line segments will be joined. Its value is one of the members of the LineJoin enumeration: Bevel, Miter, MiterClipped, and Round. StartCap, EndCap Determines the caps at the two ends of a line segment, respectively. Their value is one of the members of the LineCap enumeration: Round, Square, Flat, Diamond, and so on.

DashCap - Determines the caps to be used at the beginning and end of a dashed line. Its value is one of the members of the DashCap enumeration: Flat, Round, and Triangle.

DashStyle - Determines the style of the dashed lines drawn with the specific Pen. Its value is one of the members of the DashStyle enumeration (Solid, Dash, DashDot, DashDotDot, Dot, and Custom).

PenType - Determines the style of the Pen; its value is one of the members of the PenType enumeration: HatchFilled, LinearGradient, PathGradient, SolidColor, and TextureFill.

The Brush Class

The Brush class represents the instrument for filling shapes; you can create brushes that fill with a solid color, a pattern, or a bitmap. In reality, there's no Brush object. The Brush class is actually an abstract class that is inherited by all the classes that implement a brush, but you can't declare a variable of the Brush type in your code. The brush objects are shown in Table.

Table - Brush Styles

Brush	Fill Effect
SolidBrush	Fills shapes with a solid color
HatchBrush	Fills shapes with a hatched pattern
LinearGradientBrush	Fills shapes with a linear gradient
PathGradientBrush	Fills shapes with a gradient that has one starting color and many ending colors

Solid Brushes

To fill a shape with a solid color, you must create a SolidBrush object with the following constructor, where brushColor is a color value, specified with the help of the Color object: Dim sBrush As SolidBrush

sBrush = New SolidBrush(brushColor)

Every filled object you draw with the sBrush object will be filled with the color of the brush.

Hatched Brushes

To fill a shape with a hatch pattern, you must create a HatchBrush object with the following constructor:

Dim hBrush As HatchBrush

HBrush = New HatchBrush(hatchStyle, hatchColor, backColor)

The HatchStyle enumeration has 54 members, so Table 14.3 shows only a few common patterns.

Table - The HatchStyle Enumeration

Value	Effect
BackwardDiagonal	Diagonal lines from top-right to bottom-left
Cross	Vertical and horizontal crossing lines
DiagonalCross	Diagonally crossing lines
ForwardDiagonal	Diagonal lines from top-left to bottom-right
Horizontal	Horizontal lines
Vertical	Vertical lines

Gradient Brushes

A gradient brush fills a shape with a specified gradient. The LinearGradientBrush fills a shape with a linear gradient, and the PathGradientBrush fills a shape with a gradient that has

one starting color and one or more ending colors. Gradient brushes are discussed in detail in the section titled "Gradients," later in this chapter.

Textured Brushes

In addition to solid and hatched shapes, you can fill a shape with a texture by using a TextureBrush object. The texture is a bitmap that is tiled as needed to fill the shape. Textured brushes are used to create rather fancy graphics, and we won't explore them in this tutorial.

The Path Class

The Path class represents shapes made up of various drawing entities, such as lines, rectangles, and curves. You can combine as many of these drawing entities as you'd like and build a new entity, which is called a path. Paths are usually closed and filled with a color, a gradient, or a bitmap. You can create a path in several ways. The simplest method is to create a new Path object and then use one of the following methods to append the appropriate shape to the path:

- AddArc
- AddEllipse
- AddPolygon
- AddBezier
- AddLine
- AddRectangle
- AddCurve
- AddPie
- AddString

The following method draws an ellipse:

Me.CreateGraphics.DrawEllipse(mypen, 10, 30, 40, 50)

To add the same ellipse to a Path object, use the following statement:

```
Dim myPath As New Path
myPath.AddEllipse(10, 30, 40, 50)
```

To display the path, call the DrawPath method, passing a Pen and Path object as arguments:

```
Me.CreateGraphics.DrawPath(myPen, myPath)
```

Drawing Shapes

Before getting into the details of the drawing methods, however, let's write a simple application that draws a couple of simple shapes on a form. First, we must create a Graphics object with the following statements:

```
Dim G As Graphics
G = Me.CreateGraphics
```

Everything you'll draw on the surface represented by the G object will appear on the form. Then, we must create a Pen object to draw with. The following statement creates a Pen object that's 1 pixel wide and draws in blue:

```
Dim P As New Pen(Color.Blue)
```

Persistent Drawing

If you switch to the Visual Studio IDE or any other window, and then return to the form of the SimpleShapes application, you'll see that the drawing has disappeared! The same will happen if you minimize the window and then restore it to its normal size. Everything you draw on the Graphics object is temporary. It doesn't become part of the Graphics object and is visible only while the control, or the form, need not be redrawn. As soon as the form is redrawn, the shapes disappear.

Drawing Methods

The Framework provides several drawing methods, one for each basic shape. All drawing methods have a few things in common. The first argument is always a Pen object, which will be used to render the shape on the Graphics object.

Table shows the names of the drawing methods. The first column contains the methods for drawing stroked shapes, and the second column contains the corresponding methods for drawing filled shapes (if there's a matching method).

Table - The Drawing Methods

Drawing Method	Description
DrawArc	Draws an arc
DrawBezier	Draws very smooth curves with fixed endpoints, whose exact shape is determined by two control points
DrawBeziers	Draws multiple Bezier curves in a single call
DrawClosedCurve	Draws a closed curve
DrawCurve	Draws curves that pass through certain points

DrawLine

The DrawLine method draws a straight-line segment between two points with a pen supplied as an argument. The simplest forms of the DrawLine method are the following, where point1 and point2 are either Point or PointF objects, depending on the coordinate system in use:

Graphics.DrawLine(pen, X1, Y1, X2, Y2)

Graphics.DrawLine(pen, point1, point2)

DrawRectangle

The DrawRectangle method draws a stroked rectangle and has two forms:

Graphics.DrawRectangle(pen, rectangle)

Graphics.DrawRectangle(pen, X1, Y1, width, height)

DrawEllipse

An ellipse is an oval or circular shape, determined by the rectangle that encloses it. To draw an ellipse, call the DrawEllipse method, which has two basic forms:

Graphics.DrawEllipse(pen, rectangle)

Graphics.DrawEllipse(pen, X1, Y1, width, height)

The arguments are the same as with the DrawRectangle method because an ellipse is basically a circle deformed to fit in a rectangle. The two ellipses and their enclosing rectangles shown in Figure 14.7 were generated with the statements of Listing 14.5.

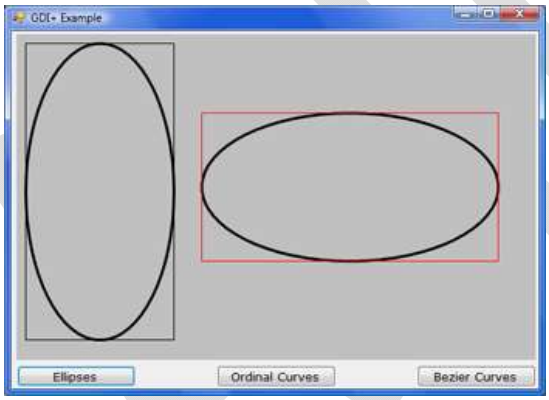


Figure - Two ellipses with their enclosing rectangles

Listing: Drawing Ellipses and Their Enclosing Rectangles

```
Private Sub bttnEllipses_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles bttnEllipses.Click
Dim G As Graphics
G = PictureBox1.CreateGraphics
G.Clear(PictureBox1.BackColor)
G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
G.FillRectangle(Brushes.Silver, ClientRectangle)
```

```
Dim R1, R2 As Rectangle
R1 = New Rectangle(10, 10, 160, 320)
R2 = New Rectangle(200, 85, 320, 160)
G.DrawEllipse(New Pen(Color.Black, 3), R1)
G.DrawRectangle(Pens.Black, R1)
G.DrawEllipse(New Pen(Color.Black, 3), R2)
G.DrawRectangle(Pens.Red, R2)
End Sub
```

DrawPie

A pie is a shape similar to a slice of pie (an arc along with the two line segments that connect its endpoints to the center of the circle or the ellipse, to which the arc belongs). The DrawPie method has two forms:

```
Graphics.DrawPie(pen, rectangle, start, sweep)
Graphics.DrawPie(pen, X, Y, width, height, start, sweep)
```

The statements of Listing create a pie chart by drawing individual pie slices. Each pie starts where the previous one ends, and the sweeping angles of all pies add up to 360 degrees, which corresponds to a full rotation (a full circle). Unlike the other samples of this section, I've used the FillPie method, because we hardly ever draw the outlines of the pies; we fill each one with a different color instead. Figure shows the output produced by Listing 14.6.

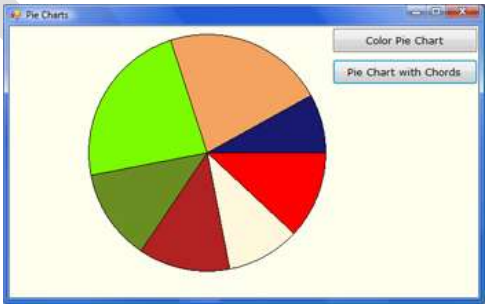


Figure - A simple pie chart generated with the FillPie method

Listing: Drawing a Simple Pie Chart with the FillPie Methods

```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click
Dim g As System.Drawing.Graphics
g = Me.CreateGraphics
Dim brush As System.Drawing.SolidBrush
Dim rect As Rectangle
brush = New System.Drawing.SolidBrush(Color.Green)
Dim Angles() As Single = {0, 43, 79, 124, 169, 252, 331, 360}
Dim colors() As Color = {Color.Red, Color.Cornsilk, _
Color.Firebrick, Color.OliveDrab, _
Color.LawnGreen, Color.SandyBrown, Color.MidnightBlue}
g.Clear(Color.Ivory)
rect = New Rectangle(100, 10, 300, 300)
Dim angle As Integer
For angle = 1 To Angles.GetUpperBound(0)
brush.Color = colors(angle - 1)
g.FillPie(brush, rect, Angles(angle - 1), Angles(angle) - Angles(angle - 1))
Next
g.DrawEllipse(Pens.Black, rect)
End Sub
```

DrawPolygon

The DrawPolygon method draws an arbitrary polygon. It accepts two arguments: the Pen that it will use to render the polygon and an array of points that define the polygon. The syntax of the DrawPolygon method is the following:

```
Graphics.DrawPolygon(pen, points())
```

where points is an array of points, which can be declared with a statement like the following:

Dim points() As Point = {New Point(x1, y1), New Point(x2, y2), ...}

DrawCurve

Curves are smooth lines drawn as cardinal splines. The simplest form of the DrawCurve method has the following syntax, where points is an array of points:

Graphics.DrawCurve(pen, points, tension)

DrawBezier

The DrawBezier method draws Bezier curves, which are smoother than cardinal splines. A Bezier curve is defined by two endpoints and two control points. The DrawBezier method accepts a pen and four points as arguments:

Graphics.DrawBexier(pen, X1, Y1, X2, Y2, X3, Y3, X4, Y4)

Graphics.DrawBezier(pen, point1, point2, point3, point4)

DrawPath

This method accepts a Pen object and a Path object as arguments and renders the specified path on the screen:

Graphics.DrawPath(pen, path)

DrawString, MeasureString

The DrawString method renders a string in a single line or multiple lines. As a reminder, the TextRenderingHint property of the Graphics object allows you to specify the quality of the rendered text. The simplest f orm of the DrawString method is the following:

```
Graphics.DrawString(string, font, brush, X, Y)
```

The simplest form of the MeasureString method is the following, where string is the string to be rendered and font is the font in which the string will be rendered:

```
Dim textSize As SizeF
textSize = Me.Graphics.MeasureString(string, font)
```

The StringFormat Object

Some of the overloaded forms of the DrawString method accept an argument of the StringFormat type. This argument determines characteristics of the text and exposes a few properties of its own, which include the following:

- **Alignment** - Determines the alignment of the text; its value is a member of the StringAlignment enumeration: Center (text is aligned in the center of the layout rectangle), Far (text is aligned far from the origin of the layout rectangle), and Near (text is aligned near the origin of the layout rectangle).
- **Trimming** - Determines how text will be trimmed if it doesn't fit in the layout rectangle. Its value is one of the members of the StringTrimming enumeration: Character (text is trimmed to the nearest character), EllipsisCharacter (text is trimmed to the nearest character and an ellipsis is inserted at the end to indicate that some of the text is missing), EllipsisPath (text at the middle of the string is removed and replaced by an ellipsis), EllipsisWord (text is trimmed to the nearest word and an ellipsis is inserted at the end), None (no trimming), and Word (text is trimmed to the nearest word).
- **FormatFlags** - Specifies layout information for the string. Its value can be one of the members of the StringFormatFlags enumeration. The two members of this enumeration that you might need often are DirectionRightToLeft (prints to the left of the specified point) and DirectionVertical.

DrawImage

The DrawImage method, which renders an image on the Graphics object, is a heavily overloaded and quite flexiblemethod. The following form of themethod draws the image at the specified location. Both the image and the location of its top-left corner are passed to the method as arguments (as Image and Point arguments, respectively):

```
Graphics.DrawImage(img, point)
```

Gradients

In this section, you'll look at the tools for creating gradients. The techniques for gradients can get quite complicated, but I will limit the discussion to the types of gradients you'll need for business or simple graphics applications.

Linear Gradients

To draw a linear gradient, you must create an instance of the LinearGradientBrush class with a statement like the following:

```
Dim lgBrush As LinearGradientBrush
lgBrush = New LinearGradientBrush(rect, startColor, endColor, gradientMode)
```

Path Gradients

This is the ultimate gradient tool. Using a PathGradientBrush, you can create a gradient that starts at a single point and fades into multiple different colors in different directions. You can fill a rectangle starting from a point in the interior of the rectangle, which is colored, say, black.

Each corner of the rectangle might have a different ending color. The PathGradientBrush will change color in the interior of the shape and will generate a gradient that's smooth in all directions. Figure shows a rectangle filled with a path gradient, although the gray shades on the printed page won't show the full impact of the gradient. Open the Gradients project you downloaded earlier to see the same figure in color (use the Path Gradient button).

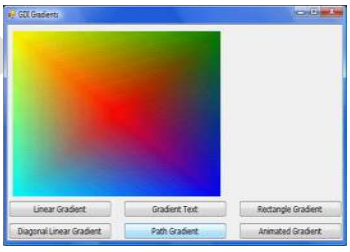


Figure - A path gradient starting at the middle of the rectangle

Clipping

The SetClip method has the following forms:

```
Graphics.SetClip(Graphics)
Graphics.SetClip(Rectangle)
Graphics.SetClip(GraphicsPath)
Graphics.SetClip(Region)
```

Applying Transformations

In computer graphics, there are three types of transformations: scaling, translation, and rotation:

The scaling transformation changes the dimensions of a shape but not its basic form. If you scale an ellipse by 0.5, you'll get another ellipse that's half as wide and half as tall as the original one. The translation transformation moves a shape by a specified distance. If you translate a rectangle by 30 pixels along the x-axis and 90 pixels along the y-axis, the new origin will be 30 pixels to the right and 90 pixels down from the original rectangle's top-left corner.

The rotation transformation rotates a shape by a specified angle, expressed in degrees; 360 degrees correspond to a full rotation, and the shape appears the same. A rotation by 180 degrees is equivalent to flipping the shape vertically and horizontally.

Transformations are stored in a 5 × 5 matrix, but you need not set it up yourself. The Graphics object provides the ScaleTransform, TranslateTransform, and RotateTransform methods, and you can specify the transformation to be applied to the shape by calling one or more of these methods and passing the appropriate argument(s).

The ScaleTransform method accepts as arguments scaling factors for the horizontal and vertical directions:

```
Graphics.ScaleTransformation(Sx, Sy)
```

The TranslateTransform method accepts two arguments, which are the displacements along the horizontal and vertical directions:

```
Graphics.TranslateTransform(Tx, Ty)
```

The Tx and Ty arguments are expressed in the coordinates of the current coordinate system. The shape is moved to the right by Tx units and down by Ty units. If one of the arguments is negative, the shape is moved in the opposite direction (to the left or up).

The RotateTransform method accepts a single argument, which is the angle of rotation expressed in degrees:

```
Graphics.RotateTransform(rotation)
```

The rotation takes place about the origin. As you will see, the final position and orientation of a shape is different if two identical rotation and translation transformations are applied in a different order.

Every time you call one of these methods, the elements of the transformation matrix are set accordingly. All transformations are stored in this matrix, and they have a cumulative effect. If you specify two translation transformations, for example, the shape will be translated by the sum of the corresponding arguments in either direction. These two transformations:

```
Graphics.TranslateTransform(10, 40)
Graphics.TranslateTransform(20, 20)
```

are equivalent to the following one:

```
Graphics.TranslateTransform(30, 60)
```

To start a new transformation after drawing some shapes on the Graphics object, call the Reset-Transform method, which clears the transformation matrix.

The effect of multiple transformations might be cumulative, but the order in which transformations are performed makes a big difference.

Bitmaps

Specifying Colors

The model of designing colors based on the intensities of their RGB components is called the RGB model, and it's a fundamental concept in computer graphics. If you aren't familiar with this model, this section is well worth reading. Nearly every color you can imagine can be constructed by mixing the appropriate percentages of the three basic colors.

Defining Colors

To manipulate colors, use the Color class of the Framework. This is a shared class, and you need not create new Color objects; just call the appropriate property or method of the Color class. The Color class exposes 128 predefined colors as properties, which you can access by name, and additional members for specifying custom colors. For example, you can define colors by using the FromARGB method of the Color class. This method accepts three arguments, which are the components of the primary colors in the desired color:

```
Color.FromARGB(Red, Green, Blue)
```

The method returns a Color value, which you can assign to a variable of the same type, or use it directly as the value of a Color property. To change the form's background color to yellow, you can assign the value returned by the FromARGB method to the BackColor property of a form or control:

```
Form1.BackColor = FromARGB(255, 128, 128)
```

Alpha Blending

Besides the red, green, and blue components, a Color value might also contain a transparency component. This value determines whether the color is opaque (255) or transparent (0). In the case of transparent colors, you can specify the degree of transparency.

This component is the alpha component. The following statement creates a new color value, which is yellow and 25 percent transparent:

```
Dim trYellow As Color
trYellow = Color.FromARGB(192, Color.Yellow)
```

The preceding statements print the logo at two locations on the image of the PictureBox1 control with different colors, as shown in Figure

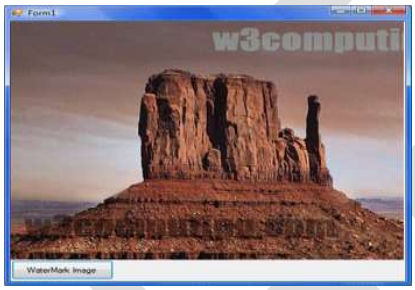


Figure - Watermarking an image with a semitransparent string

Figure - Creating a 3D effect by superimposing transparency on an opaque and a semitransparent string

The code behind the Draw Semi-Transparent Text button is quite simple, really. First it draws the string with the solid blue brush:

```
brush = New SolidBrush(Color.FromARGB(255, 0, 0, 255))
```

Processing Bitmaps

A bitmap is a two-dimensional array of color values. These values are stored in disk files, and when an image is displayed on a PictureBox or Form control, each of its color values

is mapped to a pixel on the PictureBox or form. This is true when the image isn't resized, of course.

Refreshing the Image

When you draw on a bitmap, which is associated with the Image property of a PictureBox control, the image on the control isn't refreshed every time the bitmap is modified. Instead, the image is modified when the Paint event has a chance to be serviced. The processing is implemented with two nested loops that iterate through the bitmap's rows and columns, as in the following code:

```
For pxlCol As Integer = 0 To PictureBox1.Image.Height - 1
For pxlRow As Integer = 0 To PictureBox1.Image.Width - 1
' statements to process current pixel:
' (pxlRow, pxlCol)
Next
Next
```

The image on the control won't be refreshed until the outer loop has finished. As a result, users can't see the progress of the operation; they will see the new image after all its pixels have been processed.

To force the PictureBox control to refresh its image, you must call the Refresh method.

POSSIBLE QUESTIONS

PART A (1 Mark)

(Online Examinations)

PART B (6 Marks)

1. Describe The Date Class with example.
2. Describe Co-ordinate transformations.
3. Write a program to animate a picture using animation control.
4. Explain in detail about handling strings in VB.NET with examples.
5. Discuss about displaying images in Vb.NET
6. Explain in detail about The Char Class in VB.NET
7. Explain about drawing with GDI+.
8. Describe The Date Class with example.
9. Elaborate Directory Class and File Class with example.
10. Explain in detail about Bitmaps.

PART A (1 Mark) – Unit 1V

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
1	The ____ data type stores characters as individual	char	character	both a & b	Object Property	char
2	The _____ method to accept numeric keystrokes and to reject letters and punctuation symbols.	IsDigit	IsLetter	IsLetter/IsDigit	Object Property	IsDigit
3	The ____method takes into consideration hexadecimal digits	IsLetter	IsLetter/IsNumber	IsNumber	Object Property	IsNumber
4	_____methods convert their argument to the lowercase character	ToUpper	ToLower	IsUpper	IsLower	ToLower
5	_____methods convert their argument to the uppercase character	ToUpper	ToLower	IsUpper	IsLower	ToUpper

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
6	The String class implements the _____ data type	char	String	charstring	property	String
7	The____method concatenates the two or more strings	strcat	cat	concat	property	concat
8	The ____method copies the value of one string variable to another	string	Copy	strcat	property	Copy
9	The_____method inserts one or more characters at a specified location in a string	add	Insert	both a&b	delete	Insert
10	The ____method joins two or more strings	concat	merge	join	both a&b	join

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
11	split a long string into smaller ones by using the____ method	strsplit()	Split	strspliting	property	Split
12	The ____ method removes a given number of characters from a string	Remove	delete	both a &b	add item	Remove
13	The____Method replaces all instances of a specified character	remove	Replace	ReplaceAll	resize	Replace
14	There are eight spaces to the left of the _____string	left	right-padded	left-padded	resize	left-padded
15	The tick proerty in DateTime Class, Each tick represents _____ nanoseconds	10.00	100.00	1000.00	property	100.00

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
16	The _____ method is used to find the given character is lower case	. Islower()	IslowerCase()	Tolower()	Isletter()	Tolower()
17	. _____ class is used to store the string and also to manipulate the string	The string class	the char class	stringbuilder class	both a&b	the string class
18	The _____ method appends a base type to the current instance of the StringBuilder class,	Append Format	Append	both a&b	functions	Append
19	_____method returns the number of days in a specific month	DaysInMonth	month	daymonth	functions	DaysInMonth
20	The day of the month with a leading zero for single-digit days_____	d	dd	ddd	dddd	dd

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
21	_____The full name of the month	mm	mmm	mmmm	m	mmmm
22	_____time converts the current instance of the DateTime class into universal coordinated time (UCT).	ToUniversalTime	ToLocalTime	UniversalTime	LocalTime	ToUniversalTime
23	_____ method converts a UCT time value to local time.	ToUniversalTime	LocalTime	ToLocalTime	UniversalTime	ToLocalTime
24	_____ method creates a new folder	CreateDirectory	Directory.CreateDirectory(path)	directory(path)	createpath	CreateDirectory
25	_____ method deletes a folder and all the files in it.	delete	remove	both a&b	update	delete

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
26	The____ method accepts a path as an argument and returns a True/False value indicating whether the specified folder exists	exit	exists	exitsub	entry	exists
27	The_____ method moves an entire folder to another location in the file system	move	moveall	both a&b	tranfer	move
28	_____accepts a string argument, which is a path, and sets the current directory to the specified path.	GetCurrentDirectory	SetCurrentDirectory	GetDirectories	current Director	SetCurrentDirectory
29	_____ method accepts a path as an argument and returns a Date value	GetCreationTime	SetCreationTime	creationTime	time	GetCreationTime
30	____ time accepts a path and a date value as arguments and sets the specified folder's	GetCreationTime	SetCreationTime	creationTime	time	SetCreationTime

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
31	The _____File class exposes methods for manipulating files	System.IO.	import system.io.	imports system.io.	imports	System.IO.
32	The___ method creates a new file and returns a FileStream object	create	add	insert	new	create
33	The___ character returns the directory separator character	/	\	"	<>	\
34	The___ method changes the extension of a file	ChangeExtension	ChangingExtension	Extension	change	ChangeExtension
35	The___method sets the current position in the file represented by the FileStream object	seekorigin	seek	seekoffset	Object Property	seek

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
36	The___ property gets or sets the current position within the stream.	Position	PositionOn	OnPosition	name property	Position
37	The ____ class is the channel through which you send data to a text file.	StreamWriter	FileStreamWriter	FileWriter	file	StreamWriter
38	The___ method writes any data in the buffer to the underlying file.	AutoFlush	Flush	Auto	Flush Auto	Flush
39	The____ method doesn't change the current position in the stream.	seek	SeekOrigin	Peek	PeekOrigin	Peek
40	The ____ class provides the methods you need to read data from a binary file.	BinaryReader	BinaryWriter	StreamReader	StreamWriter	BinaryReader

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
41	The_____ method returns the next available character from the stream without repositioning the current pointer	Peek	PeekChar	Seek	SeekChar	PeekChar
42	_____ is represented by a Long numeric value, which is the pixel's color.	pixelcolor	pixel	color	megapixel	pixel
43	The most recent version on GDI is called____	GDI+.	GDI	GDI-	GDI*	GDI+.
44	The _____ properties return the horizontal and vertical resolutions of the drawing surface	DpiX, DpiY	x,y	xDipY	DpiXy	DpiX, DpiY
45	The____ coordinate is its horizontal distance from the origin	y	x	(0,0)	(x.y)	x

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
46	The____ coordinate is its vertical distance from the origin	y	x	(0,0)	(x.y)	y
47	the ____ coordinate to top-left corner of the drawing surface.	y	x	(0,0)	(x.y)	(0,0)
48	The _____class represents the dimensions of a rectangle	NewSize	Size	Sizeobject	sizeproperty	Size
49	____Determines how two consecutive line segments will be joined	merge	join	LineJoin	mergejoin	LineJoin
50	_____Determines the caps to be used at the beginning and end of a dashed line	Dashcap	startcap	endcap	Linecap	DashCap

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
51	_____Determines the style of the dashed lines drawn with the specific Pen	DashCap	DashStyle	DashDot	DashDotDot	DashStyle
52	The ____ class represents the instrument for filling shapes	Brush	SolidBrush	HatchBrush	PathGradientBrush	Brush
53	_____ brush Fills shapes with a gradient that has one starting color and many ending colors	Brush	PathGradientBrush	HatchBrush	SolidBrush	PathGradientBrush
54	____text is trimmed to the nearest word and an ellipsis is inserted at the end	EllipsisPath	EllipsisWord	EllipsisCharacter	Ellipsis	EllipsisPath
55	____text is aligned far from the origin of the layout rectangle	Center	Far	Near	close	Far

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
56	_____ control executes timer events at specifies intervals of time	time	watch	timer	seconds	timer
57	What increments of time is applied interval property of the timer control____	Seconds	Nanoseconds	milliseconds	minutes	
58	The _____ transformation changes the dimensions of a shape but not its basic form	Rotation	Translation	scaling	Scaling - Rotaion	scaling
59	RGB components is called _____	ARGB	RGB	custom colors	GB	RGB
60	A rotation by ____ degrees is equivalent to flipping the shape vertically and horizontally.	360.00	90.00	180.00	0.00	180.00

UNIT-V

SYLLABUS

Databases: Architecture and Basic Concepts: What is database? - Server Explorer – Structured Query Language – The Query Builder – Building database Application with ADO.Net: The Architecture of ADO.Net-Creating the dataset – Data Binding – Programming the Data Adapter Objects – The Command and Data Reader Object. Programming the ADO.Net objects: The Structure of the dataset – The DataForm Wizard – Transactions – Performing Update Operations.

DATABASES: ARCHITECTURE AND BASIC CONCEPTS

What Is a Database?

A database is an object for storing complex, structured information. The same is true for a file, or even for the file system on your hard disk. What makes a database unique is the fact that databases are designed to make data easily retrievable. The purpose of a database is not so much the storage of information as its quick retrieval. In other words, you must structure your database so that it can be queried quickly and efficiently.

Databases are maintained by special programs, such as Access and SQL Server. These programs are called database management systems (DBMS), and they're among the most complicated applications. A fundamental characteristic of a DBMS is that it isolates much of the complexity of the database from the developer. Regardless of how each DBMS stores data on disk, you see your data organized in tables with relationships between tables. To access the data stored in the database and to update the database, you use a special language, Structured Query Language (SQL).

The Visual Database Tools

To simplify the development of database applications, Visual Studio.NET comes with some visual tools, the most important of which are discussed in the following sections.

The Server Explorer This is the first and most prominent tool. The Server Explorer is the Toolbox for database applications, in the sense that it contains all the basic tools for connecting to databases and manipulating their objects.

The Query Builder This is a tool for creating SQL queries (statements that retrieve the data we want from a database, or update the data in the database)..

The Database Designer and Tables Designer These tools allow you to work with an entire database or its tables.

The Server Explorer

Your starting point for developing database applications with VB.NET is the Server Explorer. This toolbox is your gateway to the databases on your system or network, and you can use it to locate and retrieve the tables you're interested in. Place the pointer over the Server Explorer tab to expand the corresponding toolbox, which looks something like the one shown in Figure 20.5. The two main objects in the Server Explorer are Data Connections and the Servers object.

Right-click the Data Connections icon and, from the context menu, select the Add Connection command. You may also see one or more connections to your databases, if you have already created some. Every new connection you add remains under the Data Connections branch until you decide to remove it, and you can use it in any number of projects.

Database Diagrams This is where you can examine the various diagrams of the database. A database diagram is a visual representation of a set of related tables, with the relations between the tables. Relations are indicated with line segments between two related tables, and you can quickly learn a lot about the structure of a database by looking at a database diagram.

Tables This is where you can select a table and edit it, or add a new table to the database. Finally, you can view the table's rows and edit them, add new rows, or delete existing rows.

Views This is where you specify the various views you want to use in your applications.

Stored Procedures Stored procedures are (usually small) programs that are stored in the database and perform very specific, and often repeated, tasks. By coding many of the operations you want to perform against the database as stored procedures, you won't have to access the database directly.

Functions The functions of SQL Server are just like the VB functions. They perform specific tasks on the database (retrieve or update data) taking into consideration the arguments passed to the functions when they were called.

Structured Query Language

SQL (Structured Query Language) is a universal language for manipulating tables, and every database management system (DBMS) supports it, so you should invest the time and effort to learn it. You can generate SQL statements with point-and-click operations (the Query Builder is a visual tool for generating SQL statements), but this is no substitute for understanding SQL and writing your own statements. SQL is a *nonprocedural* language. This means that SQL doesn't provide traditional programming structures like IF statements or loops.

To retrieve all the company names from the Customers table of the Northwind database, you issue a statement like this one:

```
SELECT CompanyName
FROM Customers
```

To select customers from a specific country, you issue the following statement:

```
SELECT CompanyName
FROM Customers
WHERE Country = 'Germany'
```

SQL statements are categorized into two major categories, which are actually considered separate languages: the statements for manipulating the data, which form the

Data Manipulation Language (DML); and the statements for defining database objects, such as tables or their indexes, which form the Data Definition Language (DDL).

Executing SQL Statements

The Query Analyzer executes SQL statements you design. The Query Builder lets you build the statements with visual tools.

Using the Query Analyzer

One of the applications installed with SQL Server is the Query Analyzer. To start it, select Start ➢Programs ➢SQL Server ➢Query Analyzer. Initially, its window will be empty. First, select the desired database's name in the Database drop-down list and then enter the SQL statement you want to execute in the upper pane. The SQL statement will be executed against the selected database when you press Ctrl+E, or click the Run button (the button with the green arrow on the toolbar).

Alternatively, you can prefix the SQL statement with the USE statement, which specifies the database against which the statement will be executed. To retrieve all the Northwind customers located in Germany, enter this statement:

```
USE Northwind
SELECT CompanyName FROM Customers
WHERE Country = 'Germany'
```

Selection Queries

The simplest form of the SELECT statement is

```
SELECT fields
FROM tables
```

where *fields* and *tables* are comma-separated lists of the fields you want to retrieve from the database and the tables they belong to.

To select the contact information from all the companies in the Customers table, use this statement:

```
USE Northwind
SELECT CompanyName, ContactName, ContactTitle
FROM Customers
```

To retrieve all the fields, use the asterisk (*) or the ALL keyword. The statement

```
SELECT * FROM Customers
```

will select all the fields from the Customers table.

WHERE Clause

The most common form of the SELECT statement is the following:

```
SELECT fields
FROM tables
WHERE condition
```

The *condition* argument can be a relational expression, like the ones you use in VB. To select all the customers from Germany, use the following condition:

```
WHERE Country = 'Germany'
```

To select customers from multiple countries, use the OR operator to combine multiple conditions:

```
WHERE Country = 'Germany' OR
Country = 'Austria'
```

You can also combine multiple conditions with the AND operator.

The statement

```
USE Northwind
SELECT ProductName, CategoryName
FROM Products, Categories
WHERE Products.CategoryID = Categories.CategoryID
```

will retrieve the names of all products, along with their category names.

AS Keyword

By default, each column of a query is labeled after the actual field name in the output. If a table contains two fields named CustLName and CustFName, you can display them with different labels using the AS keyword. The SELECT statement

```
SELECT CustLName, CustFName
```

will produce two columns labeled CustLName and CustFName. The query's output will look much better if you change the labels of these two columns with a statement like the following one:

```
SELECT CustLName AS [Last Name],
CustFName AS [First Name]
```

TOP Keyword

The TOP keyword is used only when the rows are ordered according to some meaningful criteria. Limiting a query's output to the alphabetically top *N* rows isn't very

practical. When the rowsare sorted according to items sold, revenue generated, and so on, it makes sense to limit the query's output to *N* rows.

DISTINCT Keyword

The DISTINCT keyword eliminates any duplicates from the cursor retrieved by the SELECT statement. To eliminate them, use the DISTINCT keyword, as shown in the following statement:

```
USE NORTHWIND
SELECT DISTINCT Country
FROM Customers
```

LIKE Operator

The LIKE operator uses pattern-matching characters, like the ones you use to select multiple files in DOS. The LIKE operator recognizes several pattern-matching characters (or *wildcard* characters) to match one or more characters, numeric digits, ranges of letters, and so on; these are listed in

Null Values

A very common operation in manipulating and maintaining databases is to locate Null values in fields. The expressions IS NULL and IS NOT NULL find field values that are (or are not) Null. A zero-length string is not the same as a Null field. To locate the rows which have a Null value in their CompanyName column, use the following WHERE clause:

```
WHERE CompanyName IS NULL
```

ORDER Keyword

The rows of a query are not in any particular order. To request that the rows be returned in a specific order, use the ORDER BY clause, whose syntax is

ORDER BY col1, col2, . . .

You can specify any number of columns in the ORDER list.

The statement

```
USE NORTHWIND
SELECT CompanyName, ContactName
FROM Customers
ORDER BY Country, City
```

will display the customers ordered by country and by city within each country.

SQL Joins

Joins specify how you connect multiple tables in a query, and there are four types of joins:

_ Left outer, or left join

_ Right outer, or right join

_ Full outer, or full join

_ Inner join

Left Joins

This join displays all the records in the left table and only those records of the table on the right that match certain user-supplied criteria.

This join has the following syntax:

FROM (primary table) LEFT JOIN (secondary table) ON (primary table).(field) (comparison) (secondary table).(field)

Right Joins

This join is similar to the left outer join, except that all rows in the table on the right are displayed and only the matching rows from the left table are displayed. This join has the following syntax:

FROM (secondary table) RIGHT JOIN (primary table) ON (secondary table).(field) (comparison) (primary table).(field)

Full Joins

The full join returns all the rows of the two tables, regardless of whether there are matching rows or not. In effect, it's a combination of left and right joins. To retrieve all the titles and all publishers, and match publishers to their titles, use the following join:

```
USE PUBS
SELECT title, pub_name
FROM titles FULL JOIN publishers
ON titles.pub_id = publishers.pub_id
```

Inner Joins

This join returns the matching rows of both tables, similar to the WHERE clause, and has the following syntax:

FROM (primary table) INNER JOIN (secondary table) ON (primary table).(field) (comparison) (secondary table).(field)

Limiting Groups with HAVING

The HAVING clause limits the groups that will appear in the cursor. In a way, it is similar to the WHERE clause, but the HAVING clause allows you to use aggregate functions. The following statement will return the IDs of the products whose sales exceed 1,000 units:

```
USE NORTHWIND
SELECT ProductID, SUM(Quantity)
FROM [Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 1000
```

IN and NOT IN Keywords

The IN and NOT IN keywords are used in a WHERE clause to specify a list of values that a column must match (or not match). They are more of a shorthand notation for multiple OR operators. The following is statement that retrieves the names of the customers in all German-speaking countries:

```
USE NORTHWIND
SELECT CompanyName
FROM Customers
WHERE Country IN ('Germany', 'Austria', 'Switzerland')
```

The BETWEEN Keyword

The BETWEEN keyword lets you specify a range of values and limit the selection to the rows that have a specific column in this range. The BETWEEN keyword is a shorthand notation for an expression like

column >= minValue AND column <= maxValue

Editing Existing Rows

The UPDATE statement edits a row's fields, and its syntax is

```
UPDATE table_name SET field1 = value1, field2 = value2, …
WHERE criteria
```

The *criteria* expression is no different than the criteria you specify in the WHERE clause of selection query. To change the country from "UK" to "United Kingdom" in the Customers table, use the following statement:

```
UPDATE Customers SET Country='United Kingdom'
WHERE Country = 'UK'
```

The Query Builder

The Query Builder is a visual tool for building SQL statements. It's a highly useful tool that generates SQL statements for you—you just specify the data you want to retrieve with point-and-click operations, instead of typing complicated expressions.

The Query Builder is a visual tool for building SQL statements. It's a highly useful tool that generates SQL statements for you—you just specify the data you want to retrieve with point-and-clickoperations, instead of typing complicated expressions. A basic understanding of SQL is obviously required, and this is why I've described the basic keywords of SQL in the last section, but it is possible to build SQL queries with the Query Builder without knowing anything about SQL. It's a great tool for beginners, but you can't get far by ignoring SQL.

The Query Builder is also a great tool for learning SQL, as you specify the query with point-and-click operations but the Query Builder builds the appropriate SQL statements. You can also edit the SQL statement manually and execute it.

The Query Builder Interface

The Query Builder contains four panes: Diagram, Grid, SQL, and Results.

Diagram Pane

This is where you select the tables you want to use in your queries—the tables in which the required data reside. To select a table, right-click anywhere on the Diagram pane and you will see the Add Table dialog box. Select as many tables as you need and then close the Add Table dialog box.

Grid Pane

The Grid pane contains the selected fields. Some fields may not be part of the output—you may use them only for selection purposes—but their names will appear on this pane. To exclude them from the output, clear the box in the Output column. The Alias column contains a name for the field. By default, the column's name is the alias. This is the heading of each column in the output, and you can change the default name to any string that suits you.

SQL Pane

As you build the statement with point-and-click operations, the Query Builder generates the SQL statement that must be executed against the database to retrieve the specified data. The statement that retrieves product names along with their categories is shown next:

```
SELECT dbo.Products.ProductName, dbo.Categories.CategoryName
FROM dbo.Categories INNER JOIN dbo.Products
ON dbo.Categories.CategoryID = dbo.Products.CategoryID
```

Results Pane

To execute a query, right-click somewhere on the SQL pane and select Run from the context menu. The Query Builder will execute the statement it generated and will

display the results in the Results pane at the bottom of the window. The heading of each column is the column's name, unless you've specified an alias for the column

SQL at Work: Counting Rows

Let's say you want to find out the number of orders in which each product appears. Go back to the Server Explorer and open the previous view (or the Query Analyzer). Add the Orders table, which will be automatically related to the Order Details table with the OrderID field. Click the OrderID field in the Orders table. A new line will be added to the Grid pane, and its Group By column will be set automatically to Group By. Set it to Count Distinct and its alias to "# Of Orders." We're going to sum the orders in which each product appears. The Count Distinct aggregate function is similar to the Count function, but it will not include the same order twice (if the same product appears in two rows of the same order). Run the query. This time you'll get one line per product.

The Alice Mutton item has been ordered 37 times, and the total items sold are 978.
Alice Mutton 978 37
Aniseed Syrup 328 12
Boston Crab Meat 1103 41
Camembert Pierrot 1577 51

The SELECT statement generated by the SQL Builder is the following. Notice that the Orders table isn't involved in the query. All the information we need resides in the Order Details table.
The Products table is included so that we can display product names instead of product IDs.

```
SELECT TOP 100 PERCENT dbo.Products.ProductName,
SUM(dbo.[Order Details].Quantity) AS [Total Items],
SUM(dbo.[Order Details].OrderID) AS [# Of Orders]
```

```
FROM dbo.Products
INNER JOIN dbo.[Order Details] ON
dbo.Products.ProductID = dbo.[Order Details].ProductID
GROUP BY dbo.Products.ProductName
ORDER BY dbo.Products.ProductName
```

The phrase TOP 100 PERCENT tells SQL Server to return all qualifying rows and is optional. The Query Builder inserted it so that you can change the value and limit the number of selected rows. Change the default aliases of the two calculated columns and execute the query again by clicking the button with the exclamation mark.

Parameterized Queries

How about running the same query with different dates? Let's modify our query once again, and make the two dates parameters of the queries. Each time you'll be executing the new query, you'll be prompted to specify the starting and ending dates. Replace the two dates in the Criteria column of the Grid pane with a question mark. The revised expression should now read:

Between ? And ?

Calculated Columns

Let's add yet another step of complexity to our query. We'll modify our query so that it calculates the total revenues generated by each product. Move down in the Field column of the Grid pane, and in the first free cell, enter the following expression:

Quantity * UnitPrice * (1 – Discount)

The wizard will replace the field names with fully qualified names:

(dbo.[Order Details].Quantity * dbo.Products.UnitPrice) * (1 - dbo.[Order

Details].Discount)

This expression calculates the subtotal for each line in the Order Details table. We multiply the price with the quantity, taking into consideration the discount. Shortly, we're going to sum the subtotals for each product. Because this is a calculated column, its Alias becomes Expr1. Change this value to Revenue. In the Group By column, select Sum. Make sure the Output column is checked and then run the query. Same results as before, only this time with an extra column, which is the revenue generated by the corresponding product:

Alice Mutton 978 37 38142
Aniseed Syrup 328 12 3280
Boston Crab Meat 1103 41 20295.2
The SQL statement generated by the SQL Builder is:

```
SELECT dbo.Products.ProductName,
SUM(dbo.[Order Details].Quantity) AS [Total Items],
COUNT(DISTINCT dbo.Orders.OrderID) AS [Times Ordered],
SUM(dbo.[Order Details].Quantity * dbo.Products.UnitPrice) AS Revenue
FROM dbo.Products
INNER JOIN dbo.[Order Details] ON
dbo.Products.ProductID = dbo.[Order Details].ProductID
INNER JOIN dbo.Orders ON
dbo.[Order Details].OrderID = dbo.Orders.OrderID
WHERE (dbo.Orders.OrderDate > @FromDate) AND
(dbo.Orders.OrderDate < @ToDate)
GROUP BY dbo.Products.ProductName
ORDER BY dbo.Products.ProductName
```

BUILDING DATABASE APPLICATION WITH ADO.NET

The Architecture of ADO.Net - ADO .NET

Most applications need data access at one point of time making it a crucial component when working with applications. Data access is making the application interact with a database, where all the data is stored. Different applications have different requirements for database access. VB .NET uses ADO .NET (Active X Data Object) as it's data access and manipulation protocol which also enables us to work with data on the Internet. Let's take a look why ADO .NET came into picture replacing ADO.

Evolution of ADO.NET

The first data access model, DAO (data access model) was created for local databases with the built-in Jet engine which had performance and functionality issues. Next came RDO (Remote Data Object) and ADO (Active Data Object) which were designed for Client Server architectures but soon ADO took over RDO. ADO was a good architecture but as the language changes so is the technology. With ADO, all the data is contained in a recordset object which had problems when implemented on the network and penetrating firewalls.

ADO was a connected data access, which means that when a connection to the database is established the connection remains open until the application is closed. Leaving the connection open for the lifetime of the application raises concerns about database security and network traffic. Also, as databases are becoming increasingly important and as they are serving more people, a connected data access model makes us think about its productivity.

Example : an application with connected data access may do well when connected to two clients, the same may do poorly when connected to 10 and might be unusable

when connected to 100 or more. Also, open database connections use system resources to a maximum extent making the system performance less effective.

Why ADO.NET?

To cope up with some of the problems mentioned above, ADO .NET came into existence. ADO .NET addresses the above mentioned problems by maintaining a disconnected database access model which means, when an application interacts with the database, the connection is opened to serve the request of the application and is closed as soon as the request is completed.

Likewise, if a database is Updated, the connection is opened long enough to complete the Update operation and is closed. By keeping connections open for only a minimum period of time, ADO .NET conserves system resources and provides maximum security for databases and also has less impact on system performance. Also, ADO .NET when interacting with the database uses XML and converts all the data into XML format for database related operations making them more efficient.

The ADO.NET Data Architecture

DataSet

The dataset is a disconnected, in-memory representation of data. It can be considered as a local copy of the relevant portions of the database. The DataSet is persisted in memory and the data in it can be manipulated and updated independent of the database. When the use of this DataSet is finished, changes can be made back to the central database for updating. The data in DataSet can be loaded from any valid data source like Microsoft SQL server database, an Oracle database or from a Microsoft Access database.

Data Provider

The Data Provider is responsible for providing and maintaining the connection to the database. A DataProvider is a set of related components that work together to provide data in an efficient and performance driven manner. The .NET Framework currently comes with two DataProviders: the SQL Data Provider which is designed only to work with Microsoft's SQL Server 7.0 or later and the OleDb DataProvider which allows us to connect to other types of databases like Access and Oracle. Each DataProvider consists of the following component classes:

1. The Connection object which provides a connection to the database
2. The Command object which is used to execute a command
3. The DataReader object which provides a forward-only, read only, connected recordset
4. The DataAdapter object which populates a disconnected DataSet with data and performs update

Data access with ADO.NET can be summarized as follows:

1. A connection object establishes the connection for the application with the database.
2. The command object provides direct execution of the command to the database. If the command returns more than a single value, the command object returns a DataReader to provide the data. Alternatively, the DataAdapter can be used to fill the Dataset object. The database can be updated using the command object or the DataAdapter.

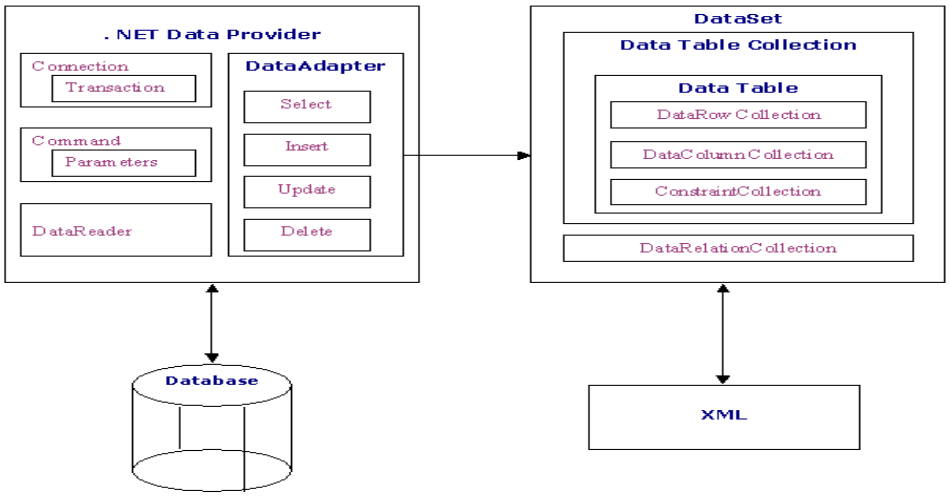
Component classes that make up the Data Providers

The Connection Object

The Connection object creates the connection to the database. Microsoft Visual Studio .NET provides two types of Connection classes: the SqlConnection object, which is designed specifically to connect to Microsoft SQL Server 7.0 or later, and the OleDbConnection object, which can provide connections to a wide range of database types like Microsoft Access and Oracle. The Connection object contains all of the information required to open a connection to the database.

The Command Object

The Command object is represented by two corresponding classes: SqlCommand and OleDbCommand. Command objects are used to execute commands to a database across a data connection. The Command objects can be used to execute stored procedures on the database, SQL commands, or return complete tables directly. Command objects provide three methods that are used to execute commands on the database:



ADO .NET Data Architecture

1. ExecuteNonQuery: Executes commands that have no return values such as INSERT, UPDATE or DELETE
2. ExecuteScalar: Returns a single value from a database query
3. ExecuteReader: Returns a result set by way of a DataReader object

The DataReader Object

DataReader object provides a forward-only, read-only, connected stream recordset from a database. Unlike other components of the Data Provider, DataReader objects cannot be directly instantiated. Rather, the DataReader is returned as the result of the Command object's ExecuteReader method.

The SqlCommand.ExecuteReader method returns a SqlDataReader object, and the OleDbCommand.ExecuteReader method returns an OleDbDataReader object. The DataReader can provide rows of data directly to application logic when you do not need to keep the data cached in memory. Because only one row is in memory at a time, the DataReader provides the lowest overhead in terms of system performance but requires the exclusive use of an open Connection object for the lifetime of the DataReader.

The DataAdapter Object

The DataAdapter is the class at the core of ADO .NET's disconnected data access. It is essentially the middleman facilitating all communication between the database and a DataSet. The DataAdapter is used either to fill a DataTable or DataSet with data from the database with it's Fill method. After the memory-resident data has been manipulated, the DataAdapter can commit the changes to the database by calling the Update method. The DataAdapter provides four properties that represent database commands:

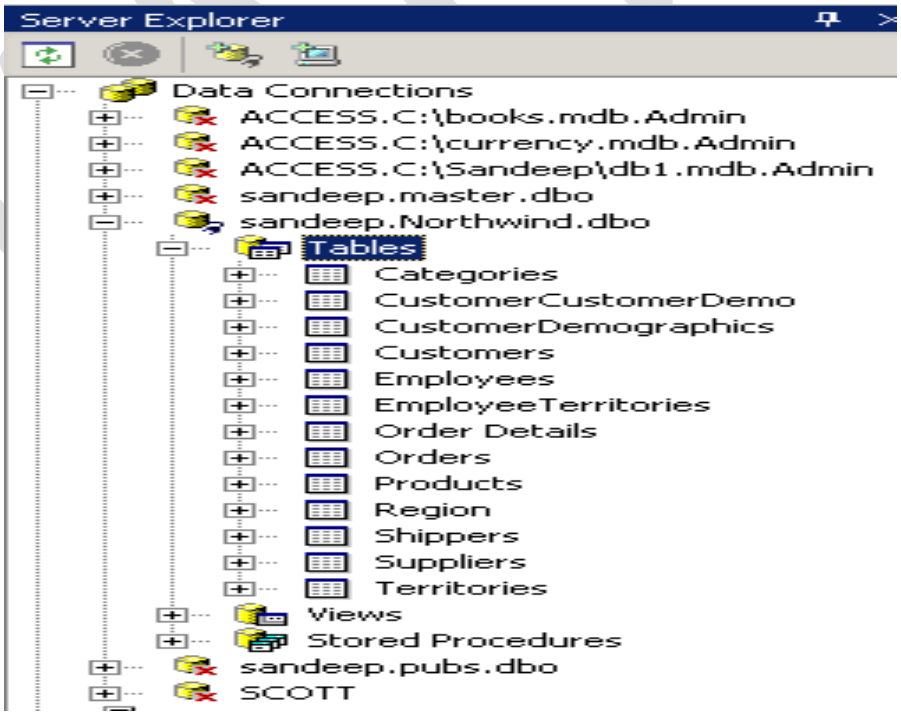
1. SelectCommand
2. InsertCommand

3. DeleteCommand
4. UpdateCommand

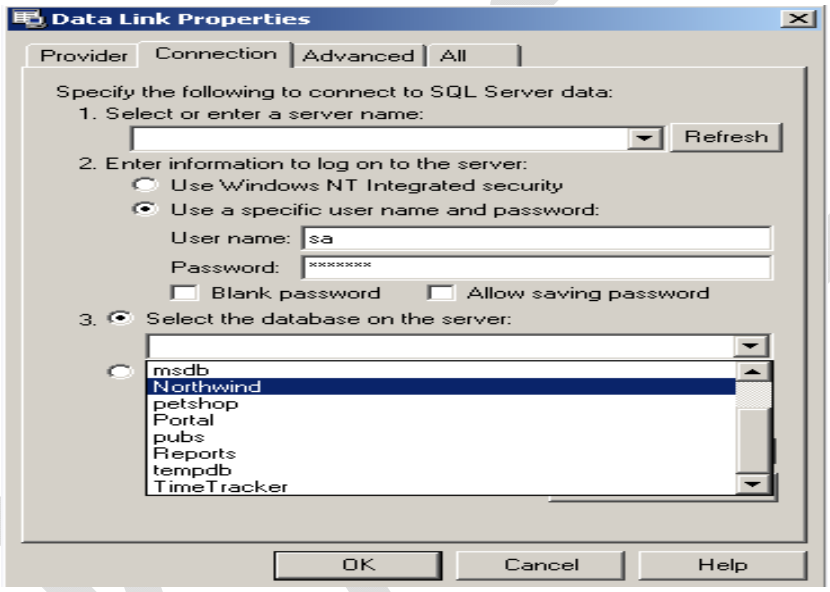
When the Update method is called, changes in the DataSet are copied back to the database and the appropriate InsertCommand, DeleteCommand, or UpdateCommand is executed.

Creating a DataSet

Visual Basic allows us to work with databases in two ways, visually and code. In Visual Basic, Server Explorer allows us to work with connections across different data sources visually. Lets see how we can do that with Server Explorer. Server Explorer can be viewed by selecting View->Server Explorer from the main menu or by pressing Ctrl+Alt+S on the keyboard. The window that is displayed is the Server Explorer which lets us create and examine data connections. The Image below displays the Server Explorer.



Let's start working with the Server Explorer. We will work with SQL Server, the default provider for .NET. We'll be displaying data from Customers table in sample North wind database in SQL Server. First, we need to establish a connection to this database. To do that, right-click on the Data Connections icon in Server Explorer and select Add Connection item. Doing that opens the Data Link Properties dialog which allows you to enter the name of the server you want to work along with login name and password. The Data Link properties window can be viewed in the Image below.

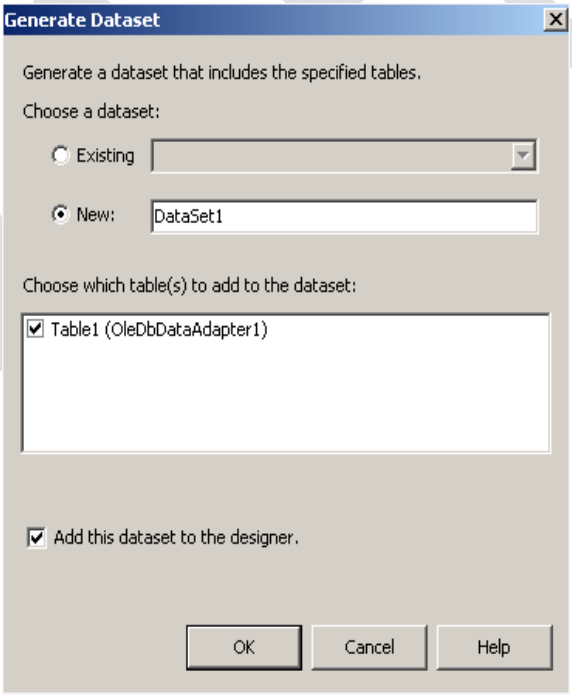


Since we are working with a database already on the server, select the option "select the database on the server". Selecting that lists the available databases on the server, select Northwind database from the list. Once you finish selecting the database, click on the Test Connection tab to test the connection. If the connection is successful, the message "Test Connection Succeeded" is displayed.

When connection to the database is set, click OK and close the Data Link Properties. Closing the data link properties adds a new Northwind database connection to

the Server Explorer and this connection which we created just now is part of the whole Visual Basic environment which can be accessed even when working with other applications. When you expand the connection node ("+" sign), it displays the Tables, Views and Stored Procedures in that Northwind sample database. Expanding the Tables node will display all the tables available in the database. In this example we will work with Customers table to display its data.

Now drag Customers table onto the form from the Server Explorer. Doing that creates SQLConnection1 and SQLDataAdapter1 objects which are the data connection and data adapter objects used to work with data. They are displayed on the component tray. Now we need to generate the dataset that holds data from the data adapter. To do that select Data->Generate DataSet from the main menu or right-click SQLDataAdapter1 object and select generate DataSet menu. Doing that displays the generate Dataset dialog box like the image below.



Once the dialogbox is displayed, select the radio button with New option to create a new dataset. Make sure Customers table is checked and click OK. Clicking OK adds a

dataset, DataSet11 to the component tray and that's the dataset with which we will work. Now, drag a DataGrid from toolbox. We will display Customers table in this data grid. Set the data grid's DataSource property to DataSet11 and it's DataMember property to Customers. Next, we need to fill the dataset with data from the data adapter. The following code does that:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs)_
Handles MyBase.Load
DataSet11.Clear()
SqlDataAdapter1.Fill(DataSet11)
'filling the dataset with the data adapter's fill method
End Sub
```

Once the application is executed, Customers table is displayed in the data grid. That's one of the simplest ways of displaying data using the Server Explorer window.

Data Binding

What you've done so far was to bind the DataGrid control to the rows of a DataSet. This process is called *data binding,* and it's not an exclusive feature of the DataGrid control. In fact, all controls can be bound to a DataSet and display a specific field of the current row from the DataSet. You can create a form with TextBox controls on it and bind each control's Text property to a different DataSet field.

As you move through the rows of the DataSet, the values on the controls will change to reflect the values of the corresponding fields in the current row. If you edit the TextBox controls, the new values will overwrite the ones in the DataSet. No changes, however, will be immediately sent to the data source, because the DataSet resides on the client computer and is disconnected from its source.

To update the underlying table(s), you must call the DataAdapter object's Update method. Figure 21.11 shows a simple interface built with data-bound TextBox controls. Each control is bound to a different field in the Customers table, and the control values change as you navigate through the rows of the table with the help of the buttons at the bottom of the form. Again, the Customers table resides in a DataSet object on the client. The form shown in the figure is the main form of the ViewEditCustomers project, which you will find on the CD.

Viewing and editing the Customers table through data-bound TextBox controls

The complex data-bound controls have a DataSource and a DataMember property. DataSource determines where the data will come from and is usually set to the name of DataSet object. If the DataSet contains multiple tables, then you must also specify which of the tables you want to display on the control. You do so by setting the control's DataMember property to the name of the appropriate table.

As you saw in the preceding section, the DataGrid control can display multiple related tables. If that's what you want, then don't set the DataMember property. A DataSet may contain (and usually does) multiple unrelated tables, in which case you must set the DataMember property to one of the tables in the DataSet.

Programming the DataAdapter Object

Each DataAdapter has four command objects, which provide the information needed to interact with the database: DeleteCommand, InsertCommand, UpdateCommand, and SelectCommand. If an application isn't going to alter the database, then you need only specify SelectCommand, which retrieves data with a SELECT statement. Each of these objects has a CommandText property, which is the name of the stored procedure or SQL statement that acts against the database; a Connection object, which determines the database the command object acts upon; and a collection of Parameter objects (the Parameters collection), which are the parameters expected by the SQL statement or stored procedure. These four Command objects are adequate to interact with the tables in the database.

The Command and DataReader Objects

As with the other major ADO.NET objects, there are two flavors of the DataReader object: the SqlDataReader and the OleDbDataReader objects. Use the SqlDataReader object for SQL Server databases and the OleDbDataReader for OLEDB-compliant databases. To create a DataReader object, you must execute a query against a database through a Command object. You've already set up Command objects, even though you didn't do so explicitly. This time we'll create a Command object and set its Connection and CommandText properties from within our code. Once the Command object has been set up, you can execute it by calling one of the following methods:

- **ExecuteReader** Executes the command and returns a DataReader object, which you can use to read the results, one row at a time.

- **ExecuteXMLReader** Executes the command and returns a XMLDataReader object, which you can use to read the results, one row at a time.

- **ExecuteScalar** Executes the command, returns the first column of the first row in the result, and ignores all other rows.
- **ExecuteNonQuery** Executes a SQL command against the database and returns the number of rows affected. Use this method to execute a command that updates the database.

The first two methods return a DataReader object, the ExecuteNonQuery method returns an integer (the number of rows affected), and the ExecuteScalar method returns an object (the first column of the first row in the result set). The DataReader is an abstract class and can't be used in an application. Instead, use the SqlDataReader or the OleDbDataReader object, depending on the
database you're connected to.

Programming the ADO.Net objects

The Structure of a DataSet

The main object of any ADO.NET application is the DataSet object, which is s a miniature database that lives on the client. The main purpose of the Connection and DataAdapter objects is to populate the DataSet object, as well as move information from the DataSet to the database and update the underlying table(s). The basic concept behind ADO.NET is to move the required data to the client, process them there, and then, optionally, update the database with the changes made by the client application to the local data. The data on the client is a copy of the data on the server the moment the DataSet was generated, and the DataSet is totally disconnected from the underlying
tables in the database.

The structure of the DataSet object is quite simple. VB6 programmers will have to get used to living with client-side data, but those of you new to VB.NET will find the DataSet a convenient method of working with subsets of tables. It's made up of tables,

which may or may not correspond to tables of the database. You can bring in an entire table, like the Categories table. Or you can select a few columns and/or a few rows from a table in the database and store them to a table in the Data- Set.

Finally, you can create a table by combining rows from multiple tables. For example, you can execute a query that retrieves all product names from the Products table along with the name of the category they belong to from the Categories tables and stores the returned rows to a new table in the DataSet. It is also possible to add and drop tables from a DataSet at any time during the course of the application.

Finally, you can create new tables from within your code. To do so, we create a DataTable object to represent the new table and then a series of DataRow objects to represent the table's rows. Each row must have a data type, an optional default value, a length, and so on. You can create the same table structures from within your code as you would do with the visual tools of Enterprise Manager. After specifying the structure of the tables, you can add relations between them. The DataSet's structure and its data are described with XML keywords.

Navigating the Tables of a DataSet

The DataSet object exposes members for accessing its contents. The tables in a DataSet are exposed through the Tables collection, which is made up of DataTable objects. If the tables are related, the relations are exposed by the Relations collection, which is made up of DataRelation objects. The following two loops print the names of the tables and relations in a DataSet:

```
Dim tbl As System.Data.DataTable
For Each tbl In AllOrders1.Tables
Console.WriteLine(tbl.TableName)
Next
```

```
Dim rel As System.Data.DataRelation
For Each rel In AllOrders1.Relations
Console.WriteLine(rel.RelationName)
Next
```

The DataTable object's most important property is the Rows property, which is a collection of DataRow objects. The DataRow object, in turn, exposes the Item property, which is the value of a specific column (field) in a row. If the *DSCustomers* DataSet contains the Customers table, the following statement returns the CustomerID field of the third row in the Customers table:

```
DSCustomers.Tables("Customers").Rows(2).Item("CustomerID")
```

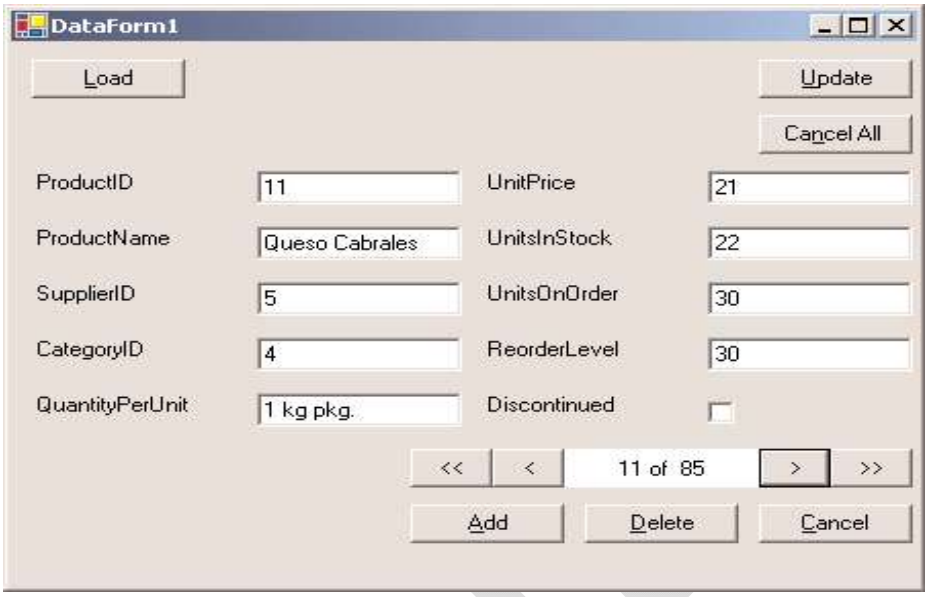
There's a simpler expression for retrieving the same value, which is the following:

```
DSCustomers.Customers(2).CustomerID
```

The DataForm Wizard

One of the tools that come with Visual Studio is the DataForm wizard, which creates data-entry forms for you. Let's look at this tool in action, then we'll discuss its limitations. You will also find interesting coding examples in the output generated by the wizard. The example of this section is the EditProducts project, whose main form is shown in Figure 22.2.

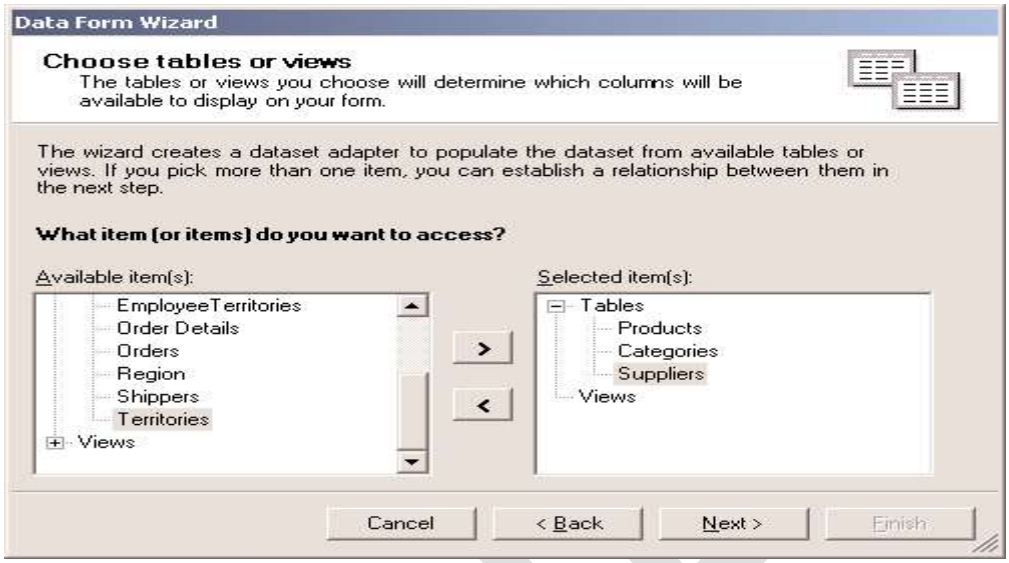
This form allows you to edit the rows of the Products table of the Northwind database, enter new rows, and delete existing rows. The interface and the code behind the controls were generated by the DataForm wizard. What you see in Figure 22.2 is the form of the EditProducts project as it was generated by the wizard. The main form of the EditProducts project on the CD is quite different, because we'll edit this form extensively in this section to make its interface more user-friendly.



Editing the Products table on a form generated by the DataForm wizard

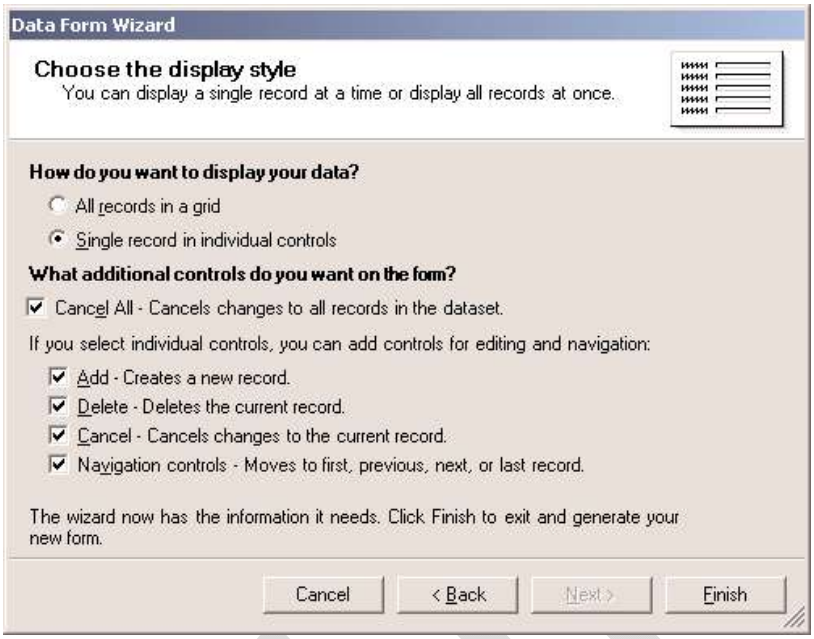
Start a new project, name it EditProducts, and delete the Form1 component. Then right-click the project's name and select Add ➢ Add New Item. In the dialog box that appears, select DataForm Wizard. A wizard starts that will take you through the steps of setting up a new DataForm.

The first screen displays a welcome message; click Next to skip it. On the next screen, you're prompted to specify the DataSet on which the DataForm will be based. Since the project doesn't contain a DataSet, specify the name of a new DataSet, which the wizard will create for you. Enter the name **DSProducts** and click Next to view the next screen of the wizard.



On the next screen of the wizard, you're prompted to choose the tables and columns that will be stored in the DataForm. the category and supplier of each product, rather than displaying all categories and all suppliers). By default, the wizard selects all the fields of the Products table. If you wanted to create a master/detail form, you'd have to specify the Detail table as well. For now click Next to see the next screen of the wizard.

On the last screen, you must select the display style. You can display all rows on a DataGrid control or create a new form with separate controls for each row. Check the radio button Single Record In Individual Controls, and the check boxes at the bottom of the window will be enabled. These check boxes allow you to specify whether the DataForm will contain an Add button (to add new rows to the Products table), a Delete button (to delete the current row), a Cancel button (to cancel any changes in the DataSet and reload all rows from the underlying table) and the Navigation controls (to move from row to row). Leave all the check boxes checked, as they are by default.



Transactions

A *transaction* is a series of actions that must either succeed, or fail, as a whole. Should one of the actions fail, then the entire transaction fails and all the changes made to the database so far must be undone ("rolled back" in proper database terminology). If all actions succeed, then they can be finalized ("committed" in proper database terminology) and become part of the database. A transaction takes place while you transfer money from one account to another. The two actions are the withdrawal of an amount from one account and the deposit of the same amount to another account.

The following pseudo-code is the skeleton of a transaction:

Begin Transaction
Try
{ statements to complete transaction }
Commit Transaction
Catch Exception
Rollback Transaction

End Try

Performing Update Operations

Updating the underlying tables is also straightforward, as long as all rows and all fields have been validated. As you have seen, there are two major approaches when working with ADO.NET: use the DataSet's update method, or use the Command object to execute SQL statements and stored procedures directly against the database. You can safely use DataSets to send data to other users. You can also safely receive DataSets from other users, probably from different databases.

The problem of two or more users attempting to update the same data is as old as computers (almost) and is known as *concurrency*. There are two ways to deal with concurrency, optimistic concurrency and pessimistic concurrency.

ADO.NET is based on optimistic concurrency. *Optimistic concurrency* means that no other users will attempt to access the same data while you're editing them. As you recall, the stored procedures generated by the DataAdapter wizard for updating the database won't update a row if even one of its fields has changed since we last read it.

A DataRow's Versions

To specify which version of a field's value you want to read, specify the second parameter of the DataRow.Item property. The following statement retrieves the Original value of the first column of the first row in the Products table of the *DSProducts1* **DataSet:**

DSProducts1.Products.Rows(0).Item("ProductName", DataRowVersion.Original)

A DataRow's States

In addition to versions, rows have states, too; a row can be in one of the following states:

- **Added** The row has been added to the DataTable, but it hasn't been accepted yet (rows are accepted after they're written to the database as well).

- **Deleted** The row has been deleted. However, it remains in the DataSet marked as Deleted, so that the Update method can delete the matching row of the underlying table.

- **Detached** The row has been created but it has not been added to a DataTable yet. A row is in this state while you set its fields and before you actually add it to a table.

- **Modified** The row has been modified, but it hasn't been accepted yet.

- **Unchanged** The row hasn't been changed yet.

Updating Tables Manually

To update the underlying table(s) from within a DataSet, you must call the DataAdapter object's Update method. The Update method sends all the changes to the database, and through the appropriate stored procedures, the tables are updated. The DataAdapter object exposes the Continue- UpdateOnError property, which determines how the DataAdapter will react when an error is encountered.

If the ContinueUpdateOnError property is False, which (mysteriously) is the default value, the DataAdapter will terminate the update process. If the first 10 edited rows in the DataSet contain no errors, they will be committed to the database. If the 11th row

contains an error, the DataAdapter won't even attempt to update the remaining rows. The ContinueUpdateOnError property should be set to True, so that the DataAdapter will update all the rows that don't contain errors.

POSSIBLE QUESTIONS

PART A (1 Mark)

(Online Examinations)

PART B (6 Marks)

1. Write about the Structure of a DataSet.
2. Elaborate in detail about DataReader and DataAdapter Object.
3. Write about the Data Binding
4. Write a VB.NET Program to maintain details of students. Use Crystal Report to generate report.
5. Explain in detail about SQL.
6. Discuss in detail about ADO.NET architecture.
7. How will you perform Update Operations in DataSets? Explain in detail with example.
8. Discuss about database in detail.
9. Elaborate Server Explorer.
10. Explain the Structure of a DataSet.

PART A (1 Mark) – Unit V

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
1	ADO Refers to _____	ActiveX Data object	Active Data Object	Application Development object	Associative Data Object	ActiveX Data object
2	Which of this is not a server component	Counter Component	Permission Checker Component	Distributed component	Content Linking	Distributed component
3	The Provider to access MS Access database is	OLEDB Data Provide	SQL Data Provider	ADO Data Provider	DOA Data Provider	OLEDB Data Provide
4	Which object is used to perform retrieve Operation	Connection Object	Command Object	Data object	Request Object	Command Object
5	_____ provides a language for describing Web Services	UDDI	WSDL	DDT	XML	WSDL

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
6	Drive,Folder,File Objects all belong to _____	TextStream Object	FileSystem Object	Dictionary Object	NetworkSystem Object	FileSystem Object
7	Which of these properties belong to TextStream Object	Drive	Line	Path	Size	Line
8	CTS Refers to _____	Common type system	Common type service	Central type system	Central type servive	Common type system
9	Whenever an application is created, a _____ is added.	Form	Class	Property	Object	form
10	SQL Data Provider is used For	MS Access	SQL Server	Oracle	MYBase	SQL Server

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
11	Which of the following operations can you NOT perform on an ADO.NET DataSet?	Development	A DataSet can be synchronised with a RecordSet	A DataSet can be converted to XML	You can infer the schema from a DataSet	A DataSet can be synchronised with a RecordSet
12	Which of this not a OLE DB Provider	ODBC drivers	DTP Packages OLAP services	Dataset	MSDataShape	DTP Packages OLAP services
13	_____ Object is specifically designed to run commands against a data store	Connection	Command	Dataset Object	DataReader Object	Command
14	_____ Object allows to connect to the data stores	Connection	Command	Dataset Object	DataReader Object	Connection
15	_____ ADO Object is to handle data not formatted in structured rows and columns	Connection	Command	Dataset Object	Record	Record

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
16	Which of these is a Access Data Type	Text	String	Char	Long	Text
17	RDO stands for _____	Remote data object	remote development object	Remote data oriented	Real Data Object	Remote data object
18	Data set is a _____ architecture	connected	disconnected	self constructed	locally connected	disconnected
19	_____is responsible for providing and maintaining connection to database	Data reader	Data adapter	data set	Data provider	Data provider
20	DAO stands for _____	Data access object	Data adapter object	Data available object	Data provider oriented	Data access object

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
21	Local copy of database is called as _____	Data base copy	Dataset	Data provider	Data tables	Dataset
22	ADO NET comes with _____ providers	2.00	3.00	4.00	6.00	2.00
23	OLEDB Data Provider is used For	MS Access	SQL Server	Oracle	SQL Server and Oracle	a
24	_____ was a connected data access	RDO	ADO.NET	ASP.NET	DAO	ADO
25	In ADO.NET, The data are converted in to _____ Format	XML	XAML	HTML	CSS	XML

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
26	_____is a disconnected, in-memory representation of data.	Dataset	Data Reader	Data Adapter	Data Provider	Dataset
27	The _____object which provides a forward-only, read only, connected recordset	Dataset	Data Reader	Data Adapter	Data Provider	Data Reader
28	The _____ object which populates a disconnected DataSet with data and performs update	Dataset	Data Reader	Data Adapter	Data Provider	Data Adapter
29	A _____object establishes the connection for the application with the database.	Connection	Command	Dataset Object	Record	Connection
30	_____is the Extension for Access Database	.MDB	.RTF	.XML	.GCC	.MDB

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
31	_____Returns a single value from a database query	ExecuteNonQuery	ExecuteScalar	ExecuteReader	Non ExecuteReader	ExecuteScalar
32	_____Returns a result set by way of a DataReader object	ExecuteNonQuery	ExecuteScalar	ExecuteReader	Non ExecuteReader	ExecuteReader
33	_____ is essentially the middleman facilitating all communication between the database and a DataSet.	Dataset	Data Reader	Data Adapter	Data Provider	Data Adapter
34	Which Command is used to insert the New Record to the Database Table	Insert	Add	Update	New	Insert
35	_____ is a collection of Record	Database	Table	File	Document	Table

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
36	Which of the Component is not a Component of ADO.NET	DataReader	DataProvider	DataAdapter	DataChanger	DataChanger
37	Which of the Following does not support Client Server Technology	DAO	ADO	RDO	ADO.NET	DAO
38	Which object is used to perform Update Operation	Connection Object	Command Object	Data Adapter	Request Object	Data Adapter
39	In Access, the Image data type are stored using _____ data type	Text	BLOB	Memo	Number	BLOB
40	In SQL, the Image data type are stored in _____ format	Text	Unicode	Binary	Special	Binary

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
41	Which of these is not a Access Data Type	Text	String	memo	date	String
42	Which of this is not a server component	Counter Component	Permission Checker Component	Distributed component	Content Linking	Distributed component
43	The Provider to access MS Access database is	OLEDB Data Provide	SQL Data Provider	ADO Data Provider	DOA Data Provider	OLEDB Data Provide
44	Which object is used to perform retrieve Operation	Connection Object	Command Object	Data object	Request Object	Command Object
45	SQL Data Provider is used For	MS Access	SQL Server	Oracle	database connectivity	SQL Server
46	The main object of any ADO.NET application is the	Connection Object	Command Object	Dataset Object	Request Object	Dataset Object

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
47	_____ ADO Object is to handle data not formatted in structured rows and colums	Connection	Command	Dataset Object	Record	Record
48	Drive,Folder,File Objects allbelong to _____	TextStream Object	FileSystem Object	Dictionary Object	NetworkSystem Object	FileSystem Object
49	Which of these properties belong to TextStream Object	Drive	Line	Path	Size	Line
50	Which of this not a OLE DB Provider	ODBC drivers	DTP Packages OLAP services		MSDataShape	DTP Packages OLAP services
51	GUIDs stands for	Globally Unqiue Identifiers	Global Unique Identifiers	Garphics User Identifiers	Garphics Unique Idenetifiers	Globally Unqiue Identifiers
52	To bind the data grid control to the rows of a Dataset is called as_____	Databinding	dataset object	Dictionary Object	Command Object	Databinding

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
53	SQL statements are categorized into _____ statements	two	three	four	one	two
54	The Query builder lets us to build the statements with _____	sql statements	object	commands	visual tools	visual tools
55	The simplest form of the SELECT Statement is	SELECT fields FROM tables	SELECT tables FROM Fields	SELECT tables	SELECT fields	SELECT fields FROM tables
56	The retrieve all the fields the symbol we use _____	*	$	#	@	*
57	The _____ keyword eliminates any duplicates from the cursor retrieved by the SELECT statement	INSERT	DISTINCT	UPDATE	SELECT	DISTINCT
58	_____ is the gateway to the databases on our system or network	Form Design	The code window	The Server Explorer	The toolbox	The Server Explorer

S.no.	Question	Choice 1	Choice - 2	Choice - 3	Choice - 4	Answer
59	The two main objects in the server explorer are _____	Data Connection, Servers object	Tools , Properties	database provider, dataset	Text stream object , File stream object	Data Connection, Servers object
60	A _____ is an object for storing complex, structured information	MDI	database	Dataset Object	The toolbox	database

Reg No…….

[17CCP204]

Karpagam Academy of Higher Education
(Deemed university Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021
(For the candidates admitted from 2017 onwards)
I M.Com (CA)
First Internal Test, January - 2018
Visual Basic.Net

Time: 2 hours Maximum: 50 marks
Date & sess: 31.1.2018 & FN

PART –A (20*1=20 Marks)
Multiple choice Questions

1 .VB.Net is a _____programming paradigm.

a) Procedural b) Structured c) Object Oriented d) Monolithic

2. Data members of a class are by default _____

a) public b) private c) static d)volatile

3. Member functions of a class are by default _____

a) public b) private c) static d) volatile

4 .IDE stands for _____

a) Internet Design Environment b) Integrated Development Environment

c) Internet Distributed Environment d) Interface Design Environment

5. The user action like key press, clicks, mouse movements are called _____

a) Handlers b) Triggers c) Events d) Methods

6. The _____ statement first executes the statement and then tests the condition after each execution

a) do….while b) while….do c) select….case d) while

7. _____ structure executes the statements until the condition is satisfied

a) do…loop b) do..loop until c) do while…loop d) do until

8. do…loop until is --------- loop

a) finite b) infinite c) long d) small

9. _____ function retrieves only date

a) for…next b) next…for c) exit for d) exit do

10. The ____ data type stores characters as individual

a) char b) character c) properties window d) Object Property

11. How Many Parent Form will be In MDI.

a) 2 b) 0 c) 1 d) Many

12. To attach the scroll bar automatically to the form, which property to set true.

a) Auto Scale b) Auto scroll c) Auto scroll bar d) Auto accept

13. Which of the following windows is useful for viewing the objects and codes?

a) Form window b) Menu Editor window

c) Form layout window d) Project explorer window

14. A sequence of variables by the same name can be referred using _____

a) arrays b) modules c) sub-routines d) functions

15 The _____ concept helps the user can supply arguments in any order

a) named arguments b)arguments value c)order arguments d)order value

16. Procedures that returns a value are called ----------

a) subroutines b) sub units c) parameters d) functions

17 To declare a variable, use the _____statement followed by the variable's name, the as keyword, and its type,

a) dim b) integer c) string d) dim as

18. The data type of the variable is defined by using the --------- clause

a) in b) where c) as d) is

19. Which of the given data types used to represent integer numbers

a) int b) character c) byte d) precision

20 Multiple implementations of the same function is called ____

a) poly overloading b)overloading function c)Override function d)Project

PART –B (3*2=6 Marks)
Answer All the Questions

21. Define Solution Explorer Window.

22. Define Overloading Functions with example.

23. Give Explanation: Text Box Control with its Properties.

PART –C (3*8=24 Marks)
Answer All the Questions

24. a. Discuss in detail about IDE components in VB.NET with neat sketch..

(Or)

b. Explain in detail about Flow-Control Statements with example.

25. a. Explain button control with example program

(Or)

b. Elucidate in detail about Argument Passing Mechanisms

26. a. Enumerate Appearance of forms with example form window.

(Or)

b. With a vb.net program to check whether given string palindrome or not.

Karpagam Academy of Higher Education
(Deemed university Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021
(For the candidates admitted from 2017 onwards)
I M.Com (CA)
First Internal Test, January - 2018
Visual Basic.Net – ANSWER KEY

Time: 2 hours **Maximum: 50 marks**
Date & sess: 31 .1.2018 & FN

PART –A (20*1=20 Marks)
Multiple choice Questions

1 .VB.Net is a _____programming paradigm.

a) Procedural b) Structured **c) Object Oriented** d) Monolithic

2. Data members of a class are by default _____

a) public **b) private** c) static d)volatile

3. Member functions of a class are by default _____

a) public b) private c) static d) volatile

4 .IDE stands for _____

a) Internet Design Environment **b) Integrated Development Environment**

c) Internet Distributed Environment d) Interface Design Environment

5. The user action like key press, clicks, mouse movements are called _____

a) Handlers b) Triggers c) **Events** d) Methods

6. The _____ statement first executes the statement and then tests the condition after each execution

a) **do**….**while** b) while….do c) select….case d) while

7. _____ structure executes the statements until the condition is satisfied

a) do…loop b) **do**..**loop until** c) do while…loop d) do until

8. do…loop until is --------- loop

a) finite b) **infinite** c) long d) small

9. _____ function retrieves only date

a) **for**…**next** b) next…for c) exit for d) exit do

10. The ____ data type stores characters as individual

a) **char** b) character c) properties window d) Object Property

11. How Many Parent Form will be In MDI.

a) 2 b) 0 **c) 1** d) Many

12. To attach the scroll bar automatically to the form, which property to set true.

a) Auto Scale **b) Auto scroll** c) Auto scroll bar d) Auto accept

13. Which of the following windows is useful for viewing the objects and codes?

a) Form window b) Menu Editor window

c) Form layout window d) **Project explorer window**

14. A sequence of variables by the same name can be referred using _____

a) **arrays** b) modules c) sub-routines d) functions

15 The _____ concept helps the user can supply arguments in any order

a) named arguments b)arguments value c)order arguments d)order value

16. Procedures that returns a value are called ----------

a) subroutines b) sub units c) parameters d) **functions**

17 To declare a variable, use the _____statement followed by the variable's name, the as keyword, and its type,

a) **dim** b) integer c) string d) dim as

18. The data type of the variable is defined by using the --------- clause

a) in b) where c) **as** d) is

19. Which of the given data types used to represent integer numbers

a) **int** b) character c) byte d) precision

20 Multiple implementations of the same function is called ____

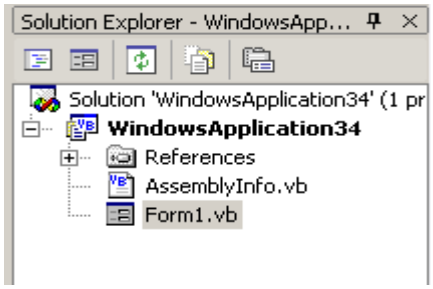
a) poly overloading b)**overloading function** c)Override function d)Project

PART –B (3*2=6 Marks)
Answer All the Questions

21. Define Solution Explorer Window.

Solution Explorer Window

The Solution Explorer window gives an overview of the solution which we are working with current window and lists all the files in the project. An image of the Solution Explorer window is shown on the right side of the IDE.



22. Define Overloading Functions with example.

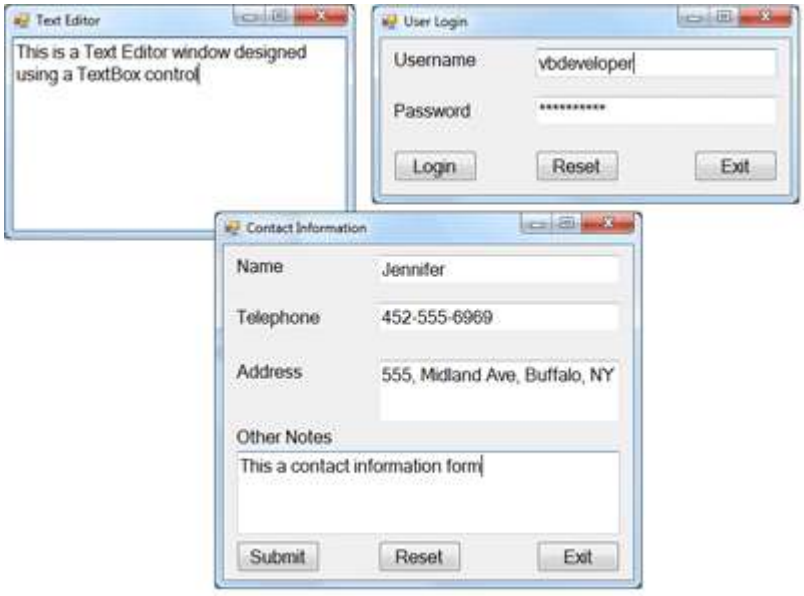
Function overloading, means that we can have multiple implementations of the same function, each with a different set of arguments and possibly a different return value. Yet all overloaded functions share the same name.

Eg:

```
Overloads Function Min(ByVal a As Double, ByVal b As Double) As Double
Min = Convert.ToDouble(If(a < b, a, b))
End Function
Overloads Function Min(ByVal a As String, ByVal b As String) As String
Min = Convert.ToString(If(a < b, a, b))
End Function
```

23. Give Explanation: Text Box Control with its Properties.

The TextBox control is the primary mechanism for displaying and entering text. It is a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.



TextAlign

This property sets (or returns) the alignment of the text on the control, and its value is a member of the Horizontal Alignment. Left, Right, or Center.

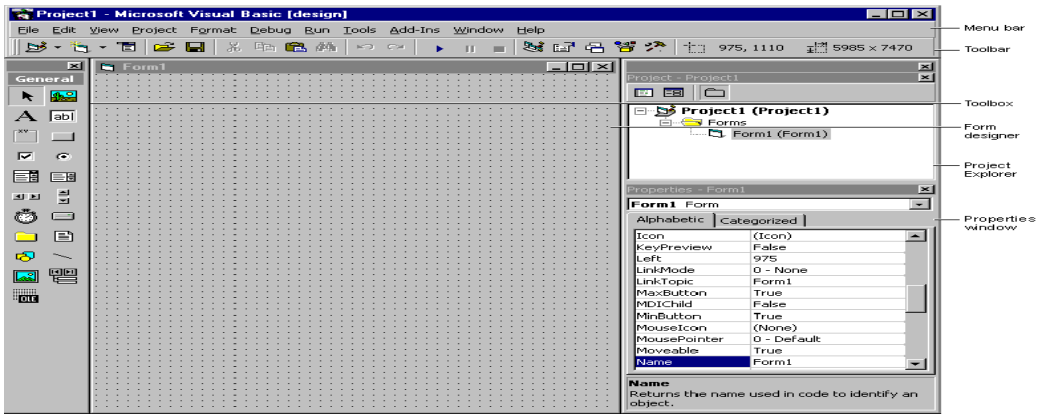
WordWrap

This property determines whether the text is wrapped automatically when it reaches the right edge of the control. The default value of this property is True. If the control has a horizontal scroll bar, however, you can enter very long lines of text.

PART –C (3*8=24 Marks)
Answer All the Questions

24. a. Discuss in detail about IDE components in VB.NET with neat sketch..

Figure: The Visual Basic Integrated Development Environment



The IDE Components

The IDE of Visual Studio.NET contains numerous components, and it will take you a while to explore them. It's practically impossible to explain what each tool, each window, and each menu does.

The IDE Menu - The IDE main menu provides the following commands, which lead to submenus. Notice that most menus can also be displayed as toolbars. Also, not all options are available at all times. The options that cannot possibly apply to the current state of the IDE are either invisible or disabled. The Edit menu is a typical example.

File Menu - The File menu contains commands for opening and saving projects, or project items, as well as the commands for adding new or existing items to the current project.

Edit Menu -The Edit menu contains the usual editing commands. Among the commands of the Edit menu are the advanced command and the IntelliSense command.

IntelliSense Submenu - The Edit -> IntelliSense menu item leads to a submenu with four options, which are described next. IntelliSense is a feature of the editor (and of other Microsoft applications) that displays as much information as possible, whenever possible.

Parameter Info - While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow tooltip

Quick Info - This is another IntelliSense feature that displays information about commands and functions. When you type the opening parenthesis following the name of a function, for example, the function's arguments will be displayed in a tooltip box (a yellow horizontal box).

View Menu - This menu contains commands to display any toolbar or window of the IDE. You have already seen the Toolbars menu (earlier, under "Starting a New Project"). The Other Windows command leads to submenu with the names of some standard windows, including the Output and Command windows.

The Output window is the console of the application. The compiler's messages, for example, are displayed in the Output window. The Command window allows you to enter and execute statements. When you debug an application, you can stop it and enter VB statements in the Command window.

Project Menu - This menu contains commands for adding items to the current project (an item can be a form, a file, a component, even another project). The last option in this menu is the Set As StartUp Project command, which lets you specify which of the projects in a multi project solution is the startup project (the one that will run when you press F5).

Build Menu - The Build menu contains commands for building (compiling) your project. The two basic commands in this menu are the Build and Rebuild All commands. The Build command compiles (builds the executable) of the entire solution, but it doesn't compile any components of the project that haven't changed since the last build. The Rebuild All command does the same, but it clears any existing files and builds the solution from scratch.

Debug Menu – This menu contains commands to start or end an application, as well as the basic debugging tools

Data Menu - This menu contains commands you will use with projects that access data.

Format Menu - The Format menu, which is visible only while you design a Windows or Web form, contains commands for aligning the controls on the form.

Tools Menu - This menu contains a list of tools, and most of them apply to C++. The Macros command of the Tools menu leads to a submenu with commands for creating macros. Just as you can create macros

in an Office application to simplify many tasks, you can create macros to automate many of the repetitive tasks you perform in the IDE..

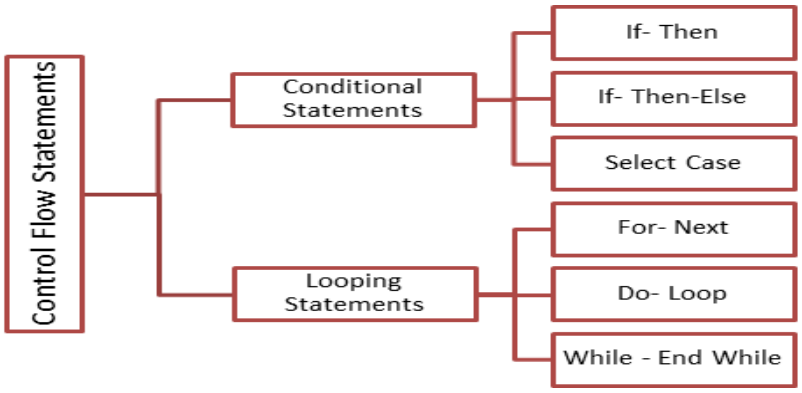
Window Menu -This is the typical Window menu of any Windows application. In addition to the list of open windows, it also contains the Hide command, which hides all Toolboxes and devotes the entire window of the IDE to the code editor or the Form Designer. The Toolboxes don't disappear completely. They're all retracted, and you can see their tabs on the left and right edges of the IDE window. To expand a Toolbox, just hover the mouse pointer over the corresponding tab.

Help Menu -This menu contains the various help options. The Dynamic Help command opens the Dynamic Help window, which is populated with topics that apply to the current operation. The Index command opens the Index window, where you can enter a topic and get help on the specific topic.

24. b. Explain in detail about Flow-Control Statements with example.

Control Flow statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false



Decision Statements

Applications need a mechanism to test conditions and take a different course of action depending on the outcome of the test. Visual Basic provides three such decision, or conditional, statements:

- If. . .Then
- If. . .Then. . .Else
- Select Case

Loop Statements

Loop statements allow you to execute one or more lines of code repetitively. Many tasks consist of operations that must be repeated over and over again, and loop statements are an important part of any programming language. Visual Basic supports the following loop statements:

- For. . .Next
- Do. . .Loop
- While. . .End While

1) If Then Statement

If Then statement is a control structure which executes a set of code only when the given condition is true.

Syntax:

```
If [Condition] Then
  [Statements]
```

In the above syntax when the **Condition** is true then the **Statements** after **Then** are executed.

Example:

```
Private Sub Button1_Click_1(ByVal sender As System.Object, ByVal e As system.EventArgs) Handles
Button1.Click
  If  Val(TextBox1.Text) > 25 Then
     TextBox2.Text = "Eligible"
  End If
```

Description:

In the above If Then example the button click event is used to check if the age got using **TextBox1** is greater than **25**, if true a message is displayed in **TextBox2**

2) If Then Else Statement

If Then Else statement is a control structure which executes different set of code statements when the given condition is true or false.

Syntax:

```
If [Condition] Then
  [Statements]
Else
  [Statements]
```

In the above syntax when the **Condition** is true, the **Statements** after **Then** are executed.If the condition is false then the statements after the **Else** part is executed.

Example:

```
Private Sub Button1_Click (ByVal sender As System.Object,  ByVal e As System.EventArgs) Handles
Button1.Click
  If  Val(TextBox1.Text) >= 40 Then
    MsgBox("GRADUATED")
      Else
    MsgBox("NOT GRADUATED")
  End If
End Sub
```

Description:

In the above If Then Else example the marks are entered in **TextBox1**.When a button is clicked a message **GRADUATED** is displayed if the condition (>40) is true and **NOT GRADUATED** if it is false.

3) Nested If Then Else Statement

Nested If..Then..Else statement is used to check multiple conditions using if then else statements nested inside one another.

Syntax:

```
If [Condition] Then
```

```
  If [Condition] Then
    [Statements]
  Else
    [Statements]
Else
  [Statements]
```

In the above syntax when the **Condition** of the first if then else is true, the second if then else is executed to check another two conditions. If false the statements under the Else part of the first statement is executed.

Example:

```
 Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Button1.Click
   If  Val(TextBox1.Text) >= 40 Then
      If  Val(TextBox1.Text) >= 60 Then
         MsgBox("You have FIRST Class")
      Else
         MsgBox("You have SECOND Class")
      End If
   Else
      MsgBox("Check your Average marks entered")
   End If
 End Sub
```

Description:

In the above nested if then else statement example first the average mark is checked if it is more than 40, if true the second if then else control is used check for first or second class. If the first condition is false the statements under the else part is executed.

4) Select Case Statement

Select case statement is used when the expected results for a condition can be known previously so that different set of operations can be done based on each condition.

Syntax:

```
Select Case Expression
  Case Expression1
    Statement1
  Case Expression2
    Statement2
  Case Expressionn
    Statementn
    ...
  Case Else
    Statement
End Select
```

In the above syntax, the value of the **Expression** is checked with **Expression1..n** to check if the condition is true. If none of the conditions are matched the statements under the **Case Else** is executed.

Description:

In the above example based on the color input in **TextBox1**, the color code for RGB colors are displayed, if the color is different then the statement under **Case Else** is executed. Thus we can easily execute the select case statement.

Loop Statements

1) Do While Loop Statement

Do While Loop Statement is used to execute a set of statements only if the condition is satisfied. But the loop gets executed once for a false condition once before exiting the loop. This is also known as **Entry Controlled loop**.

Syntax:

```
Do While [Condition]
  [Statements]
Loop
```

In the above syntax the **Statements** are executed till the **Condition** remains true.

Example:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
  Dim a As Integer
  a = 1
  Do While a < 100
    a = a * 2
    MsgBox("Product is::" & a)
  Loop
End Sub
```

Description:

In the above Do While Loop example the loop is continued after the value 64 to display 128 which is false according to the given condition and then the loop exits.

2) Do Loop While Statement

Do Loop While Statement executes a set of statements and checks the condition, this is repeated until the condition is true. .It is also known as an **Exit Control** loop

Syntax:

```
Do
  [Statements]
Loop While [Condition]
```

In the above syntax the **Statements** are executed first then the **Condition** is checked to find if it is true.

Example:

```
Private Sub Form1_Load(ByVal sender As System.Object,  ByVal e As System.EventArgs)
Handles MyBase.Load
```

```
    Dim cnt As Integer
      Do
     cnt = 10
       MsgBox("Value of cnt is::" & cnt)
    Loop While cnt <= 9
End Sub
```

Description:

In the above Do Loop While example, a message is displayed with a value 10 only after which the condition is checked, since it is not satisfied the loop exits.

3) For Next Loop Statement

For Next Loop Statement executes a set of statements repeatedly in a loop for the given initial, final value range with the specified step by step increment or decrement value.

Syntax:

```
For counter = start To end [Step]
  [Statement]
Next [counter]
```

In the above syntax the **Counter** is range of values specified using the **Start** ,**End** parameters. The **Step** specifies step increment or decrement value of the counter for which the statements are executed.

Example:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
    Dim i As Integer
    Dim j As Integer
    j = 0
    For i = 1 To 10 Step 1
     j = j + 1
     MsgBox("Value of j is::" & j)
```

```
    Next i
End Sub
```

Description:

In the above For Next Loop example the counter value of i is set to be in the range of 1 to 10 and is incremented by 1. The value of j is increased by 1 for 10 times as the loop is repeated.

Nested Control Structures

You can place, or nest, control structures inside other control structures (such as an If. . .Then block within a For. . .Next loop).

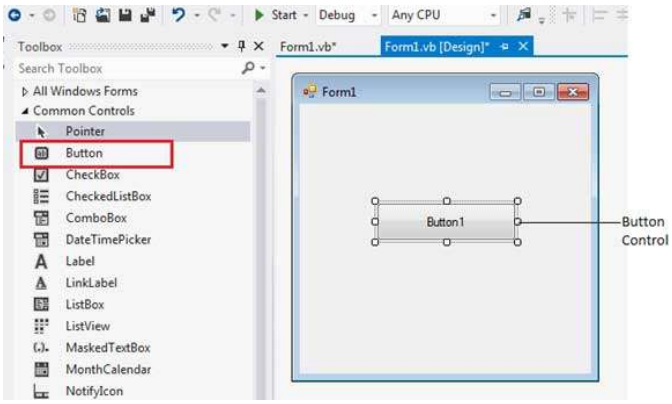
Example:

```
Dim Income As Decimal
Income=Convert.ToDecimal(InputBox("Enteryourincome"))
If Income >0 Then
If Income>12000 Then
MsgBox"You will pay taxes this year"
Else
MsgBox"You won't pay any taxes this year"
End If
Else
MsgBox"Bummer"
End If
```

25. a. Explain button control with example program

The Button control represents a standard Windows button. It is generally used to generate a Click event by providing a handler for the Click event.

Create a label by dragging a Button control from the Toolbox ad dropping it on the form.



Properties of the Button Control

The following are some of the commonly used properties of the Button control:

S.No	Property	Description
1	AutoSizeMode	Gets or sets the mode by which the Button automatically resizes itself.
2	BackColor	Gets or sets the background color of the control.
3	BackgroundImage	Gets or sets the background image displayed in the control.

Methods of the Button Control

The following are some of the commonly used methods of the Button control:

S.No	Method Name & Description
1	GetPreferredSize Retrieves the size of a rectangular area into which a control can be fitted.
2	NotifyDefault Notifies the Button whether it is the default button so that it can adjust its appearance accordingly.

Events of the Button Control

The following are some of the commonly used events of the Button control:

S.No	Event	Description
1	Click	Occurs when the control is clicked.
2	DoubleClick	Occurs when the user double-clicks the Button control.

Example :

```
Public Class Form1
 Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As  System.EventArgs)
Button1.Text = "Click Here"
End Sub
 Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
```

```
Button1.Click
MsgBox("http://vb.net-informations.com")
End Sub
End Class
```

25.b. Elucidate in detail about Argument Passing Mechanisms

Argument Passing Mechanisms

One of the most important topics in implementing your own procedures is the mechanism used to pass arguments. The examples so far have used the default mechanism: passing arguments by value. The other mechanism is passing them by reference. Although most programmers use the default mechanism, it's important to know the difference between the two mechanisms and when to use each.

- ✓ Passing arguments By Value
- ✓ Passing arguments by Reference
- ✓ Returning Multiple Values
- ✓ Passing Objects as Arguments

Passing arguments by value

This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter. The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument.

VB.Net, you declare the reference parameters using the **ByVal** keyword. The following example demonstrates the concept:

```
Module paramByval
  Sub swap(ByVal x As Integer, ByVal y As Integer)
    Dim temp As Integer
    temp = x ' save the value of x
    x = y    ' put y into x
```

```
    y = temp 'put temp into y
  End Sub
  Sub Main()
    ' local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    Console.WriteLine("Before swap, value of a : {0}", a)
    Console.WriteLine("Before swap, value of b : {0}", b)
    ' calling a function to swap the values '
    swap(a, b)
    Console.WriteLine("After swap, value of a : {0}", a)
    Console.WriteLine("After swap, value of b : {0}", b)
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

Passing Parameters by Reference

A reference parameter is a reference to a memory location of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these

parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

In VB.Net, you declare the reference parameters using the **ByRef** keyword. The following example demonstrates this:

```
Module paramByref
  Sub swap(ByRef x As Integer, ByRef y As Integer)
    Dim temp As Integer
    temp = x ' save the value of x
    x = y    ' put y into x
    y = temp 'put temp into y
  End Sub
  Sub Main()
    ' local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    Console.WriteLine("Before swap, value of a : {0}", a)
    Console.WriteLine("Before swap, value of b : {0}", b)
    ' calling a function to swap the values '
    swap(a, b)
    Console.WriteLine("After swap, value of a : {0}", a)
    Console.WriteLine("After swap, value of b : {0}", b)
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces following result:

```
Before swap, value of a : 100
Before swap, value of b : 200
```

```
After swap, value of a : 200
After swap, value of b : 100
```

Returning Multiple Values

If you want to write a function that returns more than a single result, you will most likely pass additional arguments by reference and set their values from within the function's code. The CalculateStatistics() function, calculates the basic statistics of a data set. The values of the data set are stored in an array, which is passed to the function by reference. The CalculateStatistics() function must return two values: the average and standard deviation of the data set. Here's the declaration of the CalculateStatistics() function:

```
Function CalculateStatistics(ByRef Data() As Double, ByRef Avg As Double, ByRef StDev As Double)
As Integer
```

The function returns an integer, which is the number of values in the data set. The two important values calculated by the function are returned in the Avg and StDev arguments:

```
Function CalculateStatistics(ByRef Data() As Double, ByRef Avg As Double, ByRef StDev As Double)
As Integer
Dim i As Integer, sum As Double, sumSqr As Double, points As Integer
points = Data.Length
For i = 0 To points - 1
sum = sum + Data(i)
sumSqr = sumSqr + Data(i) ^ 2
Next
Avg = sum / points
StDev = System.Math.Sqrt(sumSqr / points - Avg ^ 2)
Return(points)
End Function
```

To call the CalculateStatistics() function from within your code, set up an array of Doubles and declare two variables that will hold the average and standard deviation of the data set:

```
Dim Values(99) As Double
' Statements to populate the data set
Dim average, deviation As Double
Dim points As Integer
points = Stats(Values, average, deviation)
Debug.WriteLine points & " values processed."
Debug.WriteLine "The average is " & average & " and"
Debug.WriteLine "the standard deviation is " & deviation
```

Using ByRef arguments is the simplest method for a function to return multiple values.. Another problem with this technique is that it's not clear whether an argument must be set before calling the function. As you will see shortly, it is possible for a function to return an array or a custom structure with fields for any number of values.

Passing Objects as Arguments

When you pass objects as arguments, they're passed by reference, even if you have specified the ByVal keyword. The procedure can access and modify the members of the object passed as an argument, and the new value will be visible in the procedure that made the call.

The following code segment demonstrates this. The object is an ArrayList, which is an enhanced form of an array. Example: the Add method adds new items to the ArrayList, and you can access individual items with an index value, similar to an array's elements. In the Click event handler of a Button control, create a new instance of the ArrayList object and call the PopulateList() subroutine to populate the list. Even if the ArrayList object is passed to the subroutine by value, the subroutine has access to its items:

```
Private Sub Button1 Click(ByVal sender As System.Object, ByVal e As system.EventArgs) Handles
Button1.Click
Dim aList As New ArrayList()
PopulateList(aList)
Debug.WriteLine(aList(0).ToString)
```

```
Debug.WriteLine(aList(1).ToString)
Debug.WriteLine(aList(2).ToString)
End Sub

Sub PopulateList(ByVal list As ArrayList)
list.Add("1")
list.Add("2")
list.Add("3")
End Sub
```

The same is true for arrays and all other collections. Even if you specify the ByVal keyword, they're passed by reference.

26. a. Enumerate Appearance of forms with example form window.

APPEARANCE OF FORMS

Applications are made up of one or more forms (usually more than one), and the forms are what users see. You should craft your forms carefully, make them functional, and keep them simple and intuitive. You already know how to place controls on the form, but there's more to designing forms than populating them with controls. The main characteristic of a form is the title bar on which the form's caption is displayed.

Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands shown in Table . On the right end of the title bar are three buttons: Minimize, Maximize, and Close. Clicking these buttons performs the associated function. When a form is maximized, the Maximize button is replaced by the Restore button. When clicked, this button resets the form to the size and position before it was maximized. The Restore button is then replaced by the Maximize button

Commands of the Control Menu of the Form

Command	Effect
Restore	Restores a maximized form to the size it was before it was maximized; available only if the form has been maximized.
Move	Lets the user move the form around with the arrow keys.
Size	Lets the user resize the form with the arrow keys.
Minimize	Minimizes the form.
Maximize	Maximizes the form.
Close	Closes the current form

Properties of the Form Object

AcceptButton, CancelButton

These two properties let you specify the default Accept and Cancel buttons. The Accept button is the one that's automatically activated when you press Enter, no matter which control has the focus at the time, and is usually the button with the OK caption. Likewise, the Cancel button is the one that's automatically activated when you hit the Esc key and is usually the button with the Cancel caption. To specify the Accept and Cancel buttons on a form, locate the AcceptButton and CancelButton properties of the form and select the corresponding controls from a drop-down list, which contains the names of all the buttons on the form.

AutoScaleMode

This property determines how the control is scaled, and its value is a member of the AutoScale-Mode enumeration: None (automatic scaling is disabled), Font (the controls on the form are scaled

relative to the size of their font), Dpi, which stands for dots per inch (the controls on the form are scaled relative to the display resolution), and Inherit (the controls are scaled according to the AutoScaleMode property of their parent class). The default value is Font; if you change the form's font size, the controls on it are scaled to the new font size.

AutoScroll

The **AutoScroll property** is a True/False value that indicates whether scroll bars will be automatically attached to the form if the form is resized to a point that not all its controls are visible.

AutoScrollPosition

This property is available from within your code only (you can't set this property at design time), and it indicates the number of pixels that the form was scrolled up or down. Its initial value is zero, and it assumes a value when the user scrolls the form (provided that the form's AutoScroll property is True). Use this property to find out the visible controls from within your code, or scroll the form programmatically to bring a specific control into view.

AutoScrollMargin

This is a margin, expressed in pixels, that's added around all the controls on the form. If the form is smaller than the rectangle that encloses all the controls adjusted by the margin, the appropriate scroll bar(s) will be displayed automatically.

AutoScrollMinSize

This property lets you specify the minimum size of the form before the scroll bars are attached. If your form contains graphics that you want to be visible at all times, set the Width and Height members of the AutoScrollMinSize property to the dimensions of the graphics.

FormBorderStyle

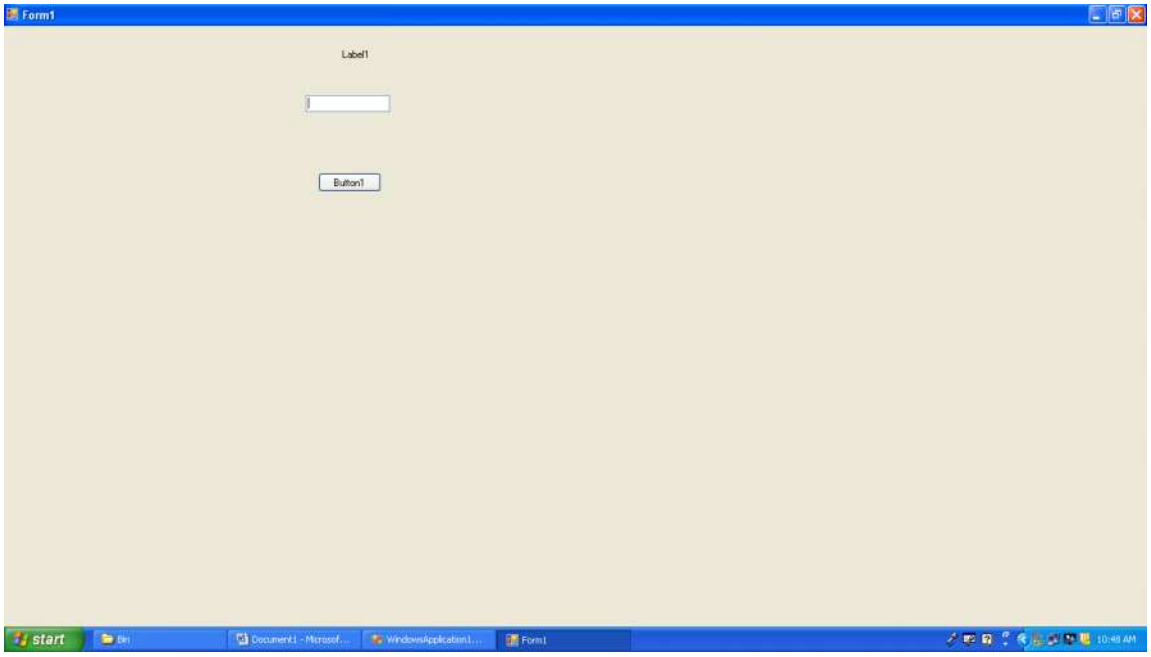
The FormBorderStyle property determines the style of the form's border; its value is one of the FormBorderStyle enumeration's members, which are shown in Table. You can make the form's title bar disappear altogether by setting the form's FormBorderStyle property to FixedToolWindow, the ControlBox property to False, and the Text property (the form's caption) to an empty string

Tabel - The FormBorderStyle Enumeration

Value	Effect
None	A borderless window that can't be resized. This setting is rarely used.
Sizable	(default) A resizable window that's used for displaying regular forms.
Fixed3D	A window with a fixed visible border, ‘‘raised" relative to the main area. Unlike the None setting, this setting allows users to minimize and close the window.
FixedDialog	A fixed window used to implement dialog boxes.
FixedSingle	A fixed window with a single-line border.
FixedToolWindow	A fixed window with a Close button only. It looks like a toolbar displayed by drawing and imaging applications.
SizableToolWindow	Same as the FixedToolWindow, but is resizable. In addition, its caption font is smaller than the usual.

26. b. With a vb.net program to check whether given string palindrome or not.

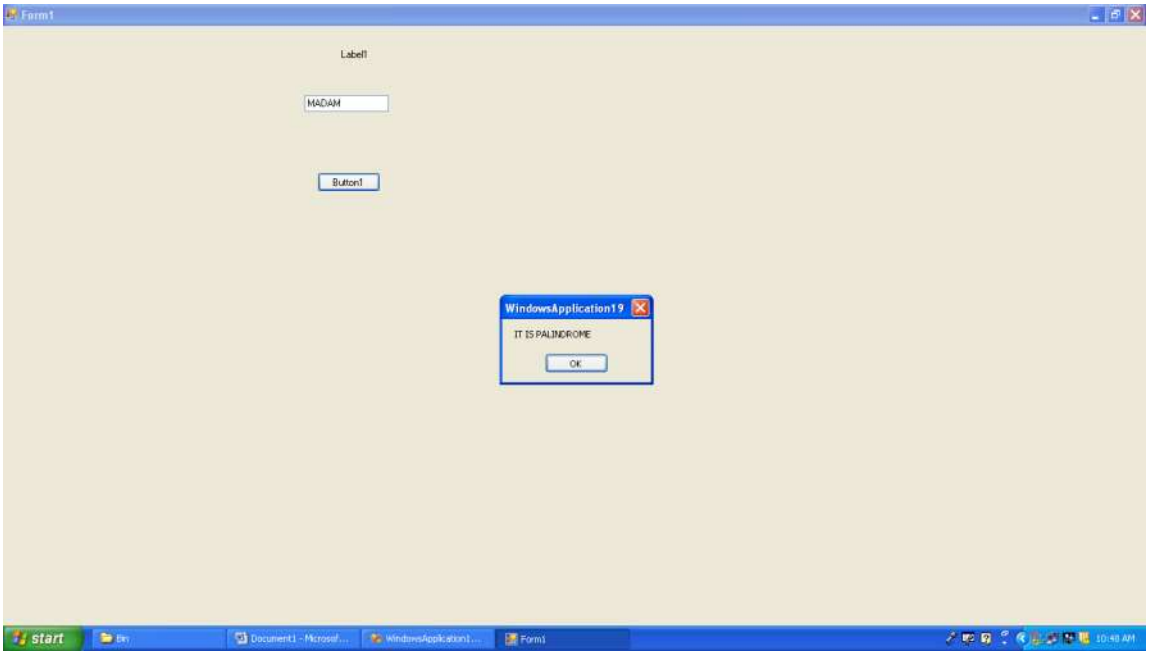
INPUT FORM



Coding

```
Public Class Form1
   Dim a As String
   Dim s As String
   Dim b As String
   Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click
      a = TextBox1.Text
      s = a.ToLower
      b = StrReverse(s)
      If s = b Then
         MsgBox("IT IS PALINDROME")
      Else
         MsgBox("IT IS NOT PALINDROME")
      End If
   End Sub
End Class
```

OUTPUT



Result: The above program is verified