

UNIT-I

SYLLABUS

Introduction: 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.Working with forms: The appearance of Forms-Loading and Showing Forms-Designing Menus.

Introduction to .NET

- .NET technology was introduced by Microsoft.
- It is a platform neutral framework.
- It is a layer between the operating system and the programming language. It supports many programming languages.
- .NET provides a common set of class libraries, which can be accessed from any .NET based programming language.

.NET supports the following languages:

- C#
- VB.NET
- C++
- J#

.NET Framework:

The components that make up the .NET platform are collectively called the .NET Framework. It is designed for cross-language compatibility. Cross language compatibility means, an application written in Visual Basic .NET may reference a DLL file written in C#.

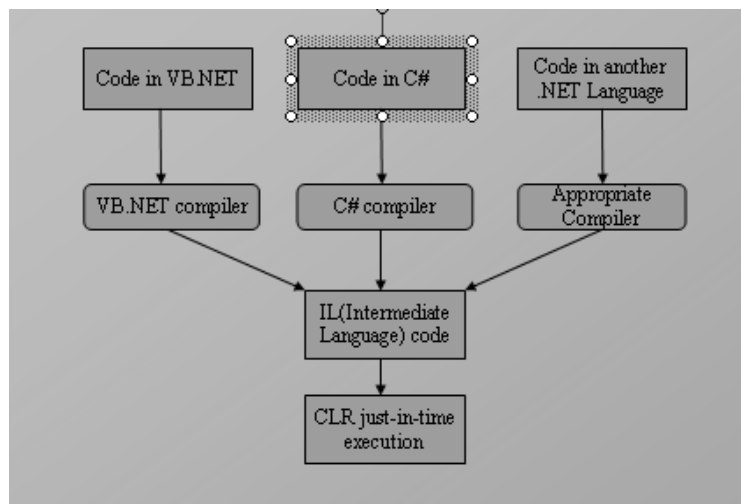


The .NET Framework consists of two main components:

- ▶ Common Language Runtime (CLR)
- ▶ Class Libraries

Common Language Runtime (CLR)

1. The CLR is described as the "execution engine" of .NET. It provides the environment within which the programs run.
2. CLR that manages the execution of programs and provides core services, such as code compilation, memory allocation, thread management, and garbage collection.
3. When the .NET program is compiled, the output of the compiler is not an executable file but a file that contains a special type of code called the [Microsoft](#) Intermediate Language (MSIL), which is a low-level set of instructions understood by the common language run time.
4. The JIT compiler converts MSIL into native code(executable code).



Class Libraries

- Class library is designed to integrate with the common language runtime.
- The class library consists of lots of prewritten code that all the applications created in VB .NET and Visual Studio .NET will use. The code for all the elements like forms, controls and the rest in VB .NET applications actually comes from the class library.

Features of .NET Framework

1. Consistent Programming Model
2. Direct Support for Security
3. Simplified Development Efforts
4. Easy Application Deployment and Maintenance

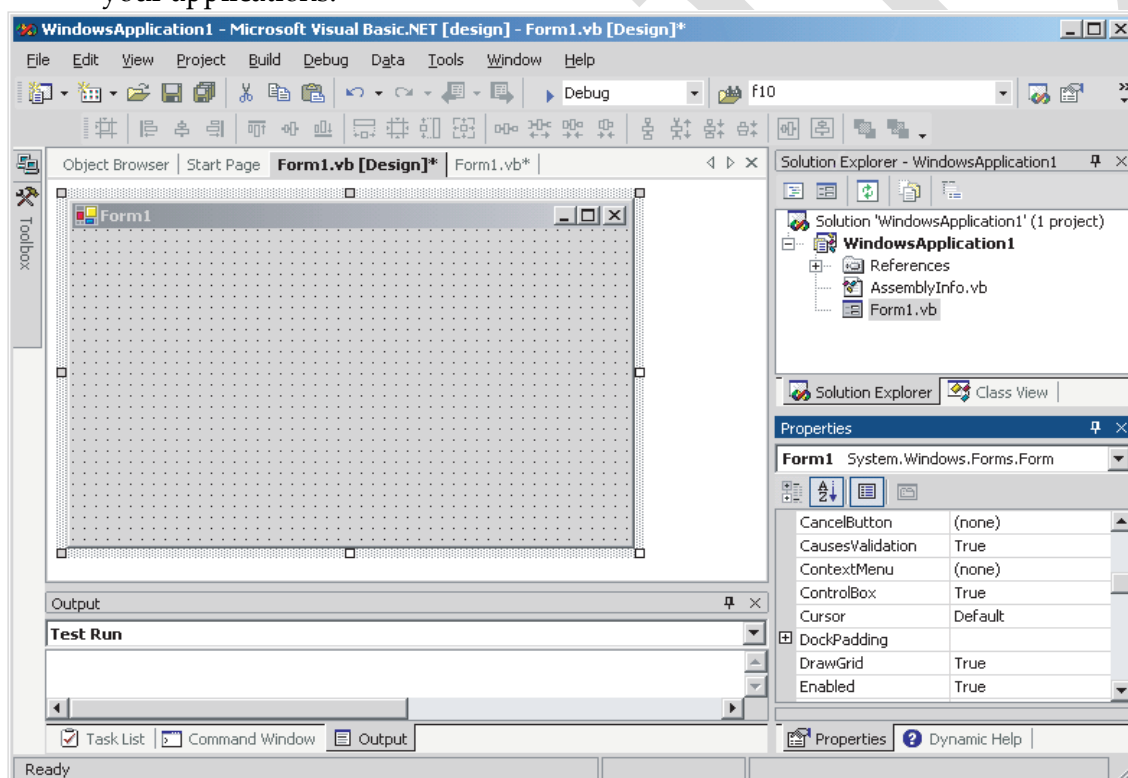
Getting Started with VB.NET

- Visual Studio .NET is an integrated environment for building, testing, and debugging a variety of Applications: Windows applications, Web applications, classes and custom controls, console applications.

- It provides numerous tools for automating the development process, visual tools to perform many common design and programming tasks.

The Integrated Development Environment

- To simplify the process of application development, Visual Studio .NET provides an environment that's common to all languages, which is known as *integrated development environment (IDE)*.
- The purpose of the IDE is to enable the developer to do as much as possible with visual tools, before writing code. The IDE provides tools for designing, executing, and debugging your applications.



1. The IDE Menu

The IDE main menu provides the following commands, which lead to submenus.

- **File Menu**

The File menu contains commands for opening and saving projects, or project items.

Edit Menu

The Edit menu contains the usual editing commands. It also contains two more special commands.

- I) The Advanced command
- II) The IntelliSense command.

Advanced Submenu

Advanced submenu is visible only when the code editor opens.

View White Space Space characters 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.

IntelliSense submenu:

IntelliSense is a feature of the editor that displays as much information as possible, whenever possible

The IntelliSense submenu includes the following options.

- i) **List Members** When this option is on, the editor lists all the members (properties, methods, events, and argument list) in a drop-down list. This list will appear when you enter the name of an object or control followed by a period.

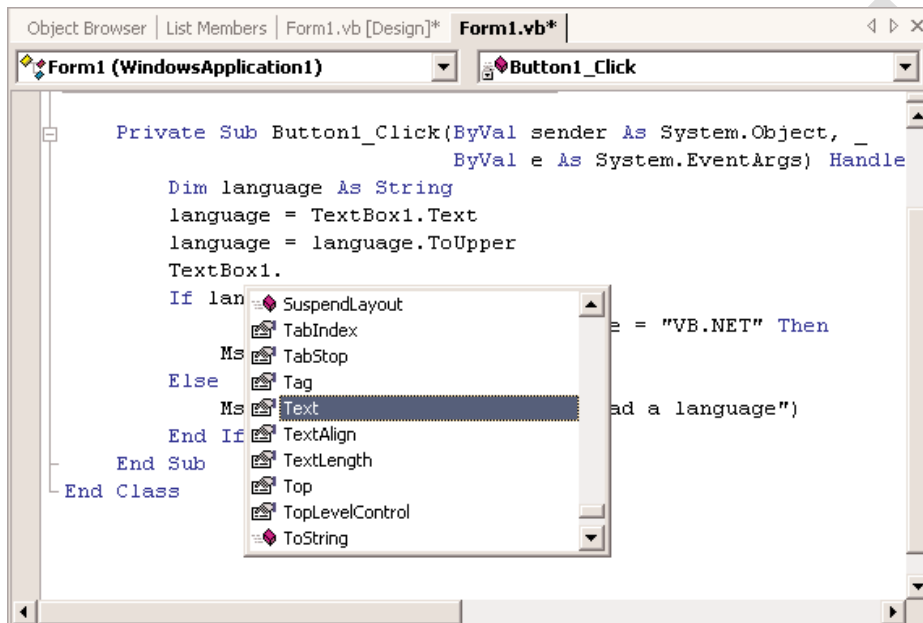
Example:

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"



ii) **Parameter Info** While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow tooltip.

iii) **Quick Info** This is another IntelliSense feature that displays information about commands and functions.

iv) **Complete Word:** The Complete Word feature enables you to complete the current word by pressing Ctrl+spacebar.

View Menu

This menu contains commands to display any toolbar or window of the IDE.

Project Menu

This menu contains commands for adding items to the current project.

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 to access data for the project.

Format Menu

The Format menu 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.

Window Menu

It contains command to open and hide windows.

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

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

The Windows Forms tab contains the icons of the controls which can be placed on a Windows form.

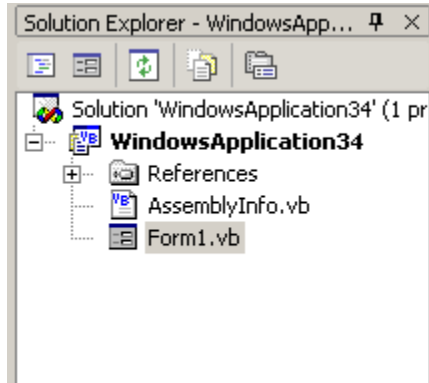
The Web Forms and HTML tabs contain the icons of the controls you can place on a Web form.

The Data tab contains the icons which are used to build data-driven applications.

The XML Schema tab contains the tools to work with schema XML files.

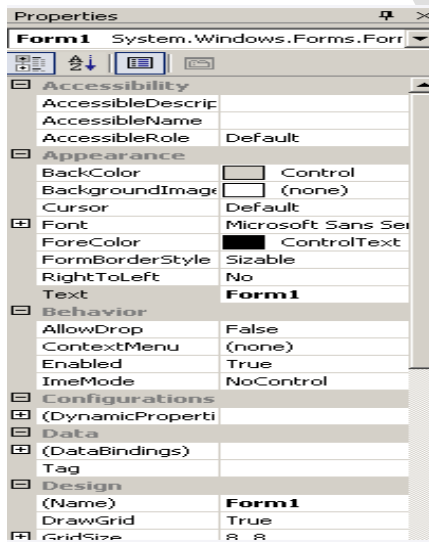
The Solution Explorer

This window contains a list of the items in the current solution. A solution may contain multiple projects, and each project may contain multiple items. The Solution Explorer displays a hierarchical list of all the components, organized by project.



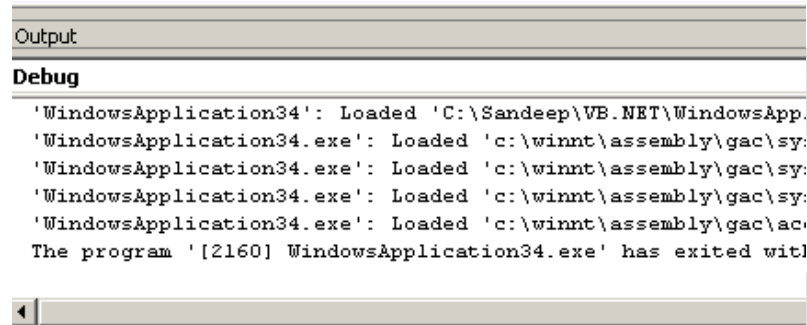
5. Properties Window

The properties window allows 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.



The Output Window

The Output window is where the compiler send its output. Every time you start an application, a series of messages is displayed on the Output window. If the Output window is not visible, select the View ➤ Other Windows ➤ Output command from the menu.



The Command Window

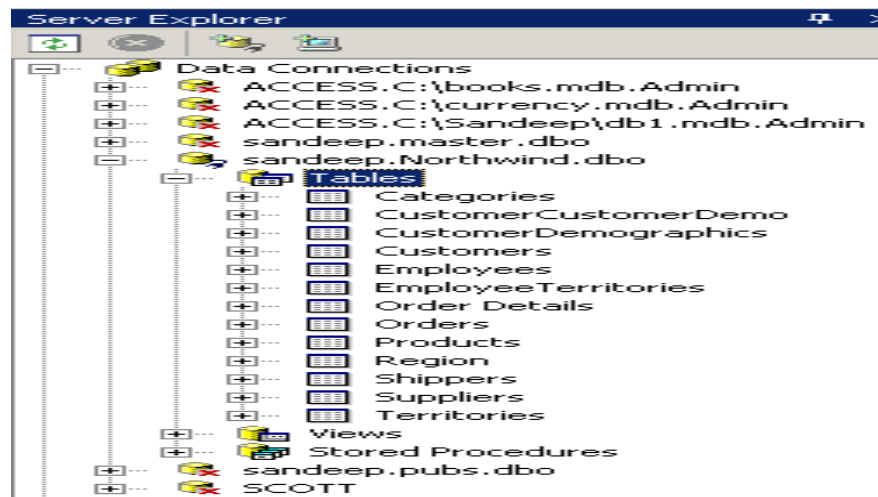
The execution of a program can be interrupted by inserting a breakpoint. When the breakpoint is reached, the program's execution is suspended and a statement in the Command window can be executed.

The Task List Window

This window is usually populated by the compiler with error messages, if the code can't be successfully compiled. You can double-click an error message in this window, and the IDE will take you to the line with the statement in error.

Server Explorer Window

The Server Explorer window is helped to work with databases in an easy graphical environment. For example, if we drag and drop a database table onto a form, VB .NET automatically creates connection and command objects that are needed to access that table. The image below displays Server Explorer window.



Environment Options

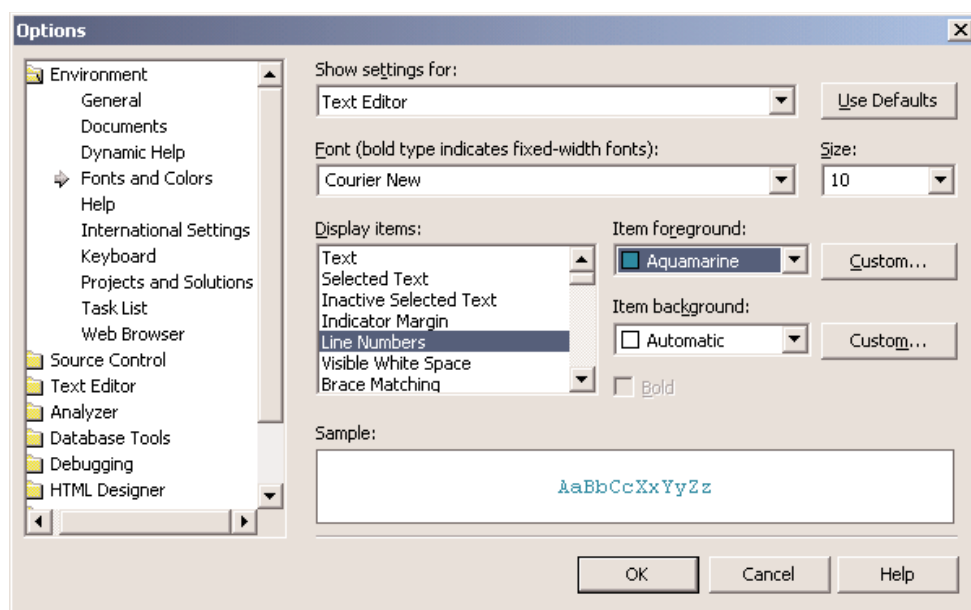
The visual studio IDE is highly customizable, Environment options deals with the customizable options exists in the visual studio IDE environment.

Font and Color Environment

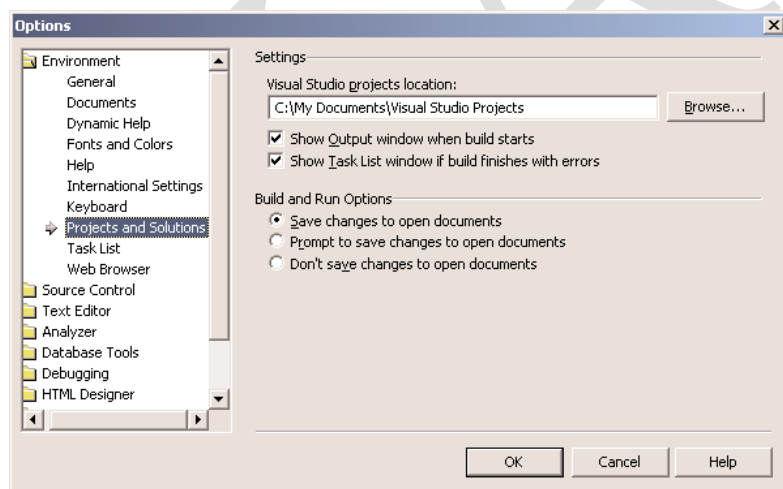
Open the Tools menu and select Options (the last item in the menu). The Options dialog box will appear where you can set all the options regarding the environment.

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



Font and Color Window



Project and Solution Window

Variables:

- Variables are used to store data.
- Each variable has a name and a value.
- Variables are declared with the Dim keyword. Dim stands for Dimension.

Declaring Variables

- A variable should be declared before it is used in program.
- It should be declared with the datatype. It specifies the type of data it can hold.

Declare integer variables:

Dim width As Integer

Multiple variables declaration of the same type can be done in a single line

Dim width, depth, height As Integer

The following statement will create **two integer and two Double variables**:

Dim width, height As Integer, area, volume As Double

Variable-Naming Conventions

- Start with a letter.
- The only special character that can appear in the variable's name is the underscore character.
- Limits of the characters are 255.
- Must be unique within the scope.
- Variable names in VB.NET are case-insensitive.

Variable initialization

The variables can be initialized in the same line that declares them.

Here declares an integer variable and initializes in to 50:

Dim instance as Integer = 50

Declare and initialize multiple variables, of the same or different type, on the same line:

Dim quantity As Integer = 1, discount As Single = 0.25

Types of Variables

Visual Basic recognizes the following five categories of variables:

- _ Numeric
- _ String
- _ Boolean
- _ Date
- _ Object

Numeric Datatype:

All programming languages provide a variety of numeric data types, including the following:

- _ Integers
- _ Decimals
- _ Single, or floating-point numbers with limited precision
- _ Double, or floating-point numbers with extreme precision

Visual Basic Numeric Data Types

Data Type	Memory Representation	Stores
Short (Int16)	2 bytes	Integer values in the range -32,768 to 32,767.
Integer (Int32)	4 bytes	Integer values in the range -2,147,483,648 to 2,147,483,647.
Long (Int64)	8 bytes	Very large integer values
Single	4 bytes	Single-precision floating-point numbers. It can represent negative numbers in the range $-3.402823E38$ to $-1.401298E-45$ and positive numbers in the range $1.401298E-45$ to $3.402823E38$.
Double	8 bytes	Double-precision floating-point numbers. It can represent negative numbers in the range $-1.79769313486232E308$ to $-4.94065645841247E-324$ and positive numbers in the range $4.94065645841247E-324$ to $1.79769313486232E308$.
Decimal	16 bytes	Integer and floating-point numbers scaled by a factor in the range from 0 to 28

Example;

Dim a As Single, b As Double

a = 1 / 3

Console.WriteLine(a)

Console.WriteLine(a)

Output:

.3333333

0.3333333333333333

Not a Number (NaN)

The value NaN indicates that the result of an operation can't be defined: it's not a regular number, not zero, and not Infinity.

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)

The result of the above statements will be a NaN.

The Decimal Data Type

Decimal, are stored internally as integers in 16 bytes.

```
Dim a As Decimal = 328.558
```

```
Dim b As Decimal = 12.4051
```

```
Dim c As Decimal
```

```
c = a * b
```

```
Console.WriteLine(c)
```

The result of the above statements will be truncated to 3 decimal digits: 4,075.795.

The Byte Data Type

The Byte type holds an integer in the range 0 to 255. 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:

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

Boolean Variables

The Boolean data type stores True/False values.

Boolean variables are declared as:

```
Dim failure As Boolean
```

and they are initialized to False.

String Variables

The String data type stores only text. It is declared with the String type:

```
Dim someText As String
```

Example:

```
Dim aString As String
```

```
aString = "Now is the time for all good men to come to the aid of their country"
```

Character Variables

Character variables store a single Unicode character in two bytes. Characters are unsigned short integers (UInt16).

CChar() function --> to convert integers to characters

CInt() function --> to convert characters to their equivalent integer values.

To declare a character variable, use the Char keyword:

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

```
Dim char1 As Char = "a", char2 As Char = "ABC"
```

```
Console.WriteLine(char1)
```

```
Console.WriteLine(char2)
```

Char data type exposes interesting methods like IsLetter, IsDigit, IsPunctuation, and so on.

Processing Individual Characters

```
Dim password As String, ch As Char
```

```
Dim i As Integer
```

```
Dim valid As Boolean = False
```

While Not valid

password = InputBox("Please enter your password")

For i = 0 To password.Length - 1

ch = password.Chars(i)

If Not System.Char.IsLetterOrDigit(ch) Then

valid = True

Exit For

End If

Next

If valid Then

MsgBox("Your new password will be activated immediately!")

Else

MsgBox("Your password must contain at least one special symbol!")

End If

End While

Date Variables

A variable declared as Date can store both date and time values with a statement like the following:

Dim expiration As Date

The following are all valid assignments:

expiration = #01/01/2004#

expiration = #8/27/2001 6:29:11 PM#

expiration = "July 2, 2002"

expiration = Now()

Data Type Identifiers

They are special symbols, which can be appended to the variable name to denote the variable's type.

To create a string variable, you can use the statement:

Dim myText\$

Data Type Definition Characters

Symbol	Data Type	Example
\$	String	<i>A\$, messageText\$</i>
%	Integer	<i>counter%, var%</i>
&	Long	<i>population&, colorValue&</i>
!	Single	<i>distance!</i>
#	Double	<i>ExactDistance#</i>
@	Decimal	<i>Balance@</i>

Object Variables

- *Variants*—variables without a fixed data type
- Variants were the most flexible data type because they could accommodate all other types.

Object Variables

Variants—variables without a fixed data type

Variants are the opposite of strictly typed variables: they can store all types of values, from a single character to an object.

Variants were the most flexible data type 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

Converting Variable Types

CBool ----- Boolean

CByte ----- Byte

CChar----- Unicode character

CDate-----Date

CDbl ----- Double

CDec -----Decimal

CInt ----- Integer (4-byte integer, Int32)

CLng -----Long (8-byte integer, Int64)

CObj ----- Object

CShort -----Short (2-byte integer, Int16)

CSng -----Single

CStr ----- String

User-Defined Data Types

A structure for storing multiple values (of the same or different type) is called a *record*. For example, each check in a checkbook-balancing application is stored in a separate record.

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

To define a record in VB.NET, use the Structure statement, which has the following syntax:

Structure structureName

Dim variable1 As varType

Dim variable2 As varType

...

Dim variablen As varType

End Structure

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

The declaration for the record structure

Structure CheckRecord

Dim CheckNumber As Integer

Dim CheckDate As Date

Dim CheckAmount As Single

Dim CheckPaidTo As String

End Structure

A variable's 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*.

A variable is said to have *procedure-level* scope. It's visible within the procedure and invisible outside the procedure.

Variables declared outside any procedure in a module are visible from within all procedures in the same module, but they're invisible outside the module.

The Lifetime of a Variable

- The lifetime of a variable is the period for which they retain their value.
- Variables declared as Public exist for the lifetime of the application.
- Local variables declared within procedures with the Dim or Private statement.

Constants

Variables don't change value during the execution of a program.

Constants don't change value

This is a safety feature. Once 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 don't change value

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.

Example:

```
Public Const pi As Double = 3.14159265358979
```

```
Public Const pi2 As Double = 2 * pi
```

Arrays

Arrays can hold sets of data of the same type. An array has a name, as does a variable, and the values stored in it can be accessed by an index.

Declaring Arrays

Arrays must be declared with the Dim (or Public, or Private) statement followed by the name of the array and the index of the last element in the array in parentheses.

Example,

Dim Salaries(15) As Integer

Array Limits

The first element of an array has index 0. The number that appears in parentheses in the Dim statement is one less 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 function UBound(), which accepts as argument the array's name.

For the array

Dim myArray(19) As Integer

its upper bound is 19, and its capacity is 20 elements.

Initializing Arrays

Initialization and declaration of variables can be done by the following constructor:

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) = "Joe Doe"
```

```
names(1) = "Peter Smack"
```

Multidimensional Arrays

A two-dimensional array has two indices. The first identifies the row, and the second identifies the column. To access the name and temperature of the third city in the two-dimensional array, use the following indices:

```
Temperatures(2, 0) ' the third city's name
```

```
Temperatures(2, 1) ' the third city's average temperature
```

The array could be declared as follows:

```
Dim Board(9, 9) As Integer
```

Dynamic Arrays

The size of a dynamic array can vary during the course of the program. To create a dynamic array, declare it as usual with the Dim statement but don't specify its dimensions:

```
Dim DynArray() As Integer
```

ReDim is executable—it forces the application to carry out an action at runtime. Dim statements aren't executable, and they can appear outside procedures.

A dynamic array also can be redimensioned to multiple dimensions. Declare it with the Dim statement outside any procedure as follows:

```
Dim Matrix() As Double
```

and then use the ReDim statement in a procedure to declare a three-dimensional array:

ReDim Matrix(9, 9, 9)

The Preserve Keyword

Preserve keyword, which forces it to resize the array without discarding the existing data.

ReDim Preserve DynamicArray(UBound(DynArray) + 1)

Example:

Public Class Form1

Private Sub Button1_Click(ByVal sender As System.Object, _

ByVal e As System.EventArgs) Handles Button1.Click

Dim i As Integer

Dim scores() As Integer

ReDim scores(1)

scores(0) = 100

scores(1) = 200

For i = 0 To scores.Length - 1

MsgBox(scores(i))

Next

ReDim Preserve scores(2)

scores(2) = 300

For i = 0 To scores.Length - 1

MsgBox(scores(i))

Next

End Sub

End Class

Arrays of Arrays

If an array is declared as Object, you can assign other types to its elements, including arrays. Suppose you have declared and populated two arrays, one with integers and another with strings.

You can then declare an Object array with two elements and populate it with the two arrays

```
Dim IntArray(9) As Integer
```

```
Dim StrArray(99) As String
```

```
Dim BigArray(1) As Object
```

```
Dim i As Integer
```

```
' populate array IntArray
```

```
For i = 0 To 9
```

```
IntArray(i) = i
```

```
Next
```

```
' populate array StrArray
```

```
For i = 0 To 99
```

```
StrArray(i) = "ITEM " & i.ToString("0000")
```

```
Next
```

```
BigArray(0) = IntArray
```

```
BigArray(1) = StrArray
```

```
Console.WriteLine(BigArray(0)(7))
```

```
Console.WriteLine(BigArray(1)(16))
```

The last two statements will print the following values on the Output window:

7

ITEM 0016

Variables as Objects

An *object* is a collection of data and code. An integer variable, *intVar*, is an object because it has a value and some properties and methods.

Properties and methods are implemented as functions. The method *intVar.ToString* for instance, returns the numeric value held in the variable as a string, so that you can use it in string operations. In other words, an Integer variable is an object that knows about itself.

It knows that it holds a whole number; it knows how to convert itself to a string; it knows the minimum and maximum values it can store (properties *MinValue* and *MaxValue*); and so on.

Formatting Numbers

The *ToString* method, exposed by all data types except the *String* data type, converts a value to the equivalent string and formats it at the same time

```
ToString(formatString)
```

The *formatString* argument is a format specifier (a string that specifies the exact format to be applied to the variable) this argument can be a specific character that corresponds to a predetermined format (*standard numeric format string*, as it's called) or a string of characters that

have special meaning in formatting numeric values (a *picture numeric format string*). Use standard format strings for the most common operations and picture strings to specify unusual formatting requirements. To format the value 9959.95 as a dollar amount, you can use the following standard currency format string:

```
Dim int As Single = 9959.95
```

```
Dim strInt As String
```

```
strInt = int.ToString("C")
```

or the following picture numeric format string:

```
strInt = int.ToString("$###,###.00")
```

Both statements will format the value as "\$9,959.95"

Formatting Dates

If the variable *birthDate* contains the value #1/1/2000#, the following expressions return the values shown below them, in bold:

```
Console.WriteLine(birthDate.ToString("d"))
```

1/1/2000

```
Console.WriteLine(birthDate.ToString("D"))
```

Saturday, January 01, 2000

```
Console.WriteLine(birthDate.ToString("f"))
```

Saturday, January 01, 2000 12:00 AM

```
Console.WriteLine(birthDate.ToString("s"))
```

2000-01-01T00:00:00

```
Console.WriteLine(birthDate.ToString("U"))
```

Saturday, January 01, 2000 12:00:00 AM

Flow Control Statements

The one that differentiates the computers from calculators is, as a programmer one can control the flow of the program according to the external programs

Test Structures

An application needs a built-in capability to test conditions and take a different course of action depending on the outcome of the test. Visual Basic provides three such decision structures:

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

If...Then

The If...Then statement tests the condition specified; if it's True, the program executes the statement(s) that follow. The If structure can have a single-line or a multiple-line syntax. To execute one statement conditionally, use the single-line syntax as follows:

If condition Then statement

Visual Basic evaluates the *condition*, and if it's true, executes the statement that follows. If the condition is False, the application continues with the statement following the If statement.

You can also execute multiple statements by separating them with colons:

If condition Then statement: statement: statement

Here's an example of a single-line If statement:

If Month(expDate) > 12 Then expYear = expYear + 1: expMonth = 1

You can break this statement into multiple lines by using End If, as shown here:

If expDate.Month > 12 Then

expYear = expYear + 1

expMonth = 1

End If

If condition Then

statementblock1

Else

statementblock2

End If

If...Then...Else

A variation of the If...Then statement is the If...Then...Else statement, which executes one block of statements if the condition is True and another block of statements if the condition is False.

The syntax of the If...Then...Else statement is as follows:

If condition1 Then

statementblock1

ElseIf condition2 Then

statementblock2

ElseIf condition3 Then

statementblock3

Else

statementblock4

End If

For example

score = InputBox("Enter score")

If score < 50 Then

Result = "Failed"

ElseIf score < 75 Then

Result = "Pass"

ElseIf score < 90 Then

Result = "Very Good"

Else

Result = "Excellent"

End If

MsgBox Result

Select Case

An alternative to the efficient, but difficult-to-read, code of the multiple-ElseIf structure is the select Case structure, which compares one expression to different values. The advantage of the Select Case statement over multiple If...Then...Else/ElseIf statements is that it makes the

code easier to read and maintain.

The Select Case structure tests a single expression, which is evaluated once at the top of the structure. The result of the test is then compared with several values, and if it matches one of them, the corresponding block of statements is executed.

Here's the syntax of the Select Case

statement:

Select Case expression

Case value1

statementblock1

Case value2

statementblock2

.

.

.

Case Else

statementblockN

End Select

For Example

Dim message As String

Select Case Now.DayOfWeek

Case DayOfWeek.Monday

message = "Have a nice week"

Case DayOfWeek.Friday

message = "Have a nice weekend"

Case Else

message = "Welcome back!"

End Select

MsgBox(message)

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively. Many tasks consist of trivial operations that must be repeated over and over again, and looping structures are an important part of any programming language.

Visual Basic supports the following loop structures:

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

For...Next

The For...Next loop is one of the oldest loop structures in programming languages. Unlike the other two loops, the For...Next loop requires that you know how many times the statements in

the loop will be executed. The For...Next loop uses a variable (it's called the loop's *counter*) that increases or decreases in value during each repetition of the loop.

The For...Next loop has the following syntax:

For counter = start To end [Step increment]

statements

Next [counter]

The keywords in the square brackets are optional. The arguments *counter*, *start*, *end*, and *increment* are all numeric. The loop is executed as many times as required for the *counter* to reach (or exceed) the *end* value.

In executing a For...Next loop, Visual Basic completes the following steps:

1. Sets *counter* equal to *start*
2. Tests to see if *counter* is greater than *end*. If so, it exits the loop. If *increment* is negative, Visual Basic tests to see if *counter* is less than *end*. If it is, it exits the loop.
3. Executes the statements in the block
4. Increments *counter* by the amount specified with the *increment* argument. If the *increment* argument isn't specified, *counter* is incremented by 1
5. Repeat the Statements

For Example

```
Dim i As Integer, total As Double
```

```
For i = 0 To data.GetUpperBound(0)
```

```
total = total + data(i)
```

```
Next i
```

```
Console.WriteLine (total / data.Length)
```

The single most important thing to keep in mind when working with For...Next loops is that the loop's *counter* is set at the beginning of the loop. Changing the value of the *end* variable in the loop's body won't have any effect.

For example, the following loop will be executed 10 times, not 100 times:

```
endValue = 10
```

```
For i = 0 To endValue
```

```
endValue = 100
```

```
{ more statements }
```

```
Next i
```

You can, however, adjust the value of the *counter* from within the loop. The following is an example of an endless (or infinite) loop:

```
For i = 0 To 10
```

```
Console.WriteLine(i)
```

```
i = i - 1
```

```
Next i
```

This loop never ends because the loop's *counter*, in effect, is never increased.

Do...Loop

The Do...Loop executes a block of statements for as long as a condition is True. Visual Basic evaluates an expression, and if it's True, the statements are executed. When the end of block is reached, the expression is evaluated again and, if it's True, the statements are repeated. If the expression is False, the program continues and the statement following the loop is executed.

There are two variations of the Do...Loop statement; both use the same basic model.

A loop can be executed either while the condition is True or until the condition becomes True. These two variations use the keywords While and Until to specify how long the statements are executed.

To execute a block of statements while a condition is True, use the following syntax:

Do While condition

statement-block

Loop

Here's a typical example of using a Do...Loop. Suppose the string *MyText* holds a piece of text, and you want to count the words in the text.

To locate an instance of a character in a string, use the InStr() function, which accepts three arguments:

- The starting location of the search
- The text to be searched
- The character being searched

The following loop repeats for as long as there are spaces in the text. Each time the InStr() function finds another space in the text, it returns the location (a positive number) of the space. When there are no more spaces in the text, the InStr() function returns zero, which signals the end of the loop, as shown:

```
Dim MyText As String = "The quick brown fox jumped over the lazy dog"
```

```
Dim position, words As Integer
```

```
position = 1
```

```
Do While position > 0
```

```
position = InStr(position + 1, MyText, " ")
```

```
words = words + 1
```

```
Loop
```

```
Console.WriteLine "There are " & words & " words in the text"
```

The Do...Loop is executed while the InStr() function returns a positive number, which happens for as long as there are more words in the text. The variable *position* holds the location of each successive space character in the text.

The search for the next space starts at the location of the current space plus 1 (so that the program won't keep finding the same space). For each space found the program increments the value of the *words* variable, which holds the total number of words when the loop ends.

Nested Control Structures

Control structures in Visual Basic can be nested in as many levels as you want, It's common practice to indent the bodies of nested decision and loop structures to make the program easier to read.

The following pseudocode demonstrates how to nest several flow-control statements:

```
For a = 1 To 100
```

```
{ statements }
```

```
If a = 99 Then
```

```
{ statements }
```

End If

While b < a

{ statements }

If total <= 0 Then

{ statements }

End If

End While

For c = 1 to a

{ statements }

Next

Next

For Example

Dim Array2D(6, 4) As Integer

Dim iRow, iCol As Integer

For iRow = 0 To Array2D.GetUpperBound(0)

For iCol = 0 To Array2D.GetUpperBound(1)

Array2D(iRow, iCol) = iRow * 100 + iCol

Console.Write(iRow & ", " & iCol & " = " & Array2D(iRow, iCol) & " ")

Next iCol

Console.WriteLine()

Next iRow

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 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 Sqrt() function will generate a runtime error), you might want to halt the operation if the array contains an invalid value.

To exit the loop prematurely, use the Exit For statement as follows:

```
For i = 0 To UBound(nArray)  
If nArray(i) < 0 Then Exit For  
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) and the While loop (Exit While), as well as 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 the function itself:

```
For i = 0 To nArray.GetUpperBound()  
If nArray(i) < 0 Then
```

MsgBox "Negative value found, terminating calculations"

Exit Function

End If

nArray(i) = Sqr(nArray(i))

Next

If this code is part of a subroutine procedure, you use the Exit Sub statement. The Exit statements for loops are Exit For, Exit While, and Exit Do.

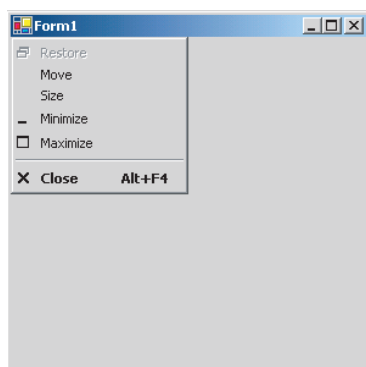
Working with Forms

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.

The terms form and window describe the same entity. A window is what the user sees on the desktop when the application is running. A form is the same entity at design time.

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



Elements of the Form

Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands. 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.

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 mouse
Size	Lets the user resize the form with the mouse
Minimize	Minimizes the form
Maximize	Maximizes the form
Close	Closes the current form

Properties of the Form Control

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 Cancel Button 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. You can also set them to the name of the corresponding button from within your code.

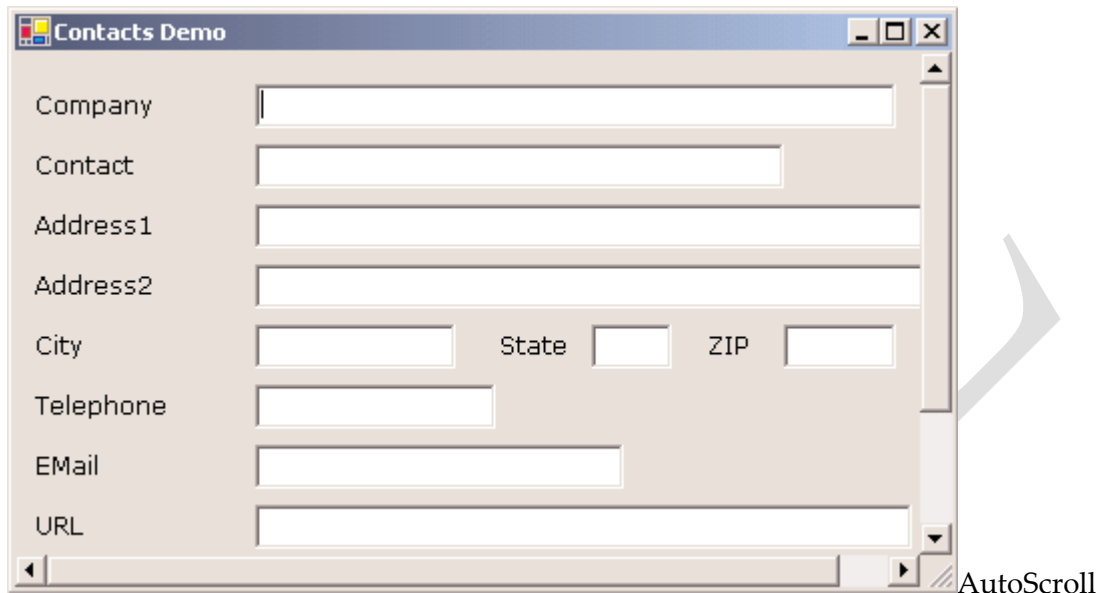
AutoScale

This property is a True/False value that determines whether the controls you place on the form are automatically scaled to the height of the current font. When you place a TextBox control on the form, for example, and the AutoScale property is True, the control will be tall enough to display a single line of text in the current font. The default value is True, which is why you can't make the controls smaller than a given size.

This is a property of the form, but it affects the controls on the form. If you change the Font property of the form after you have placed a few controls on it, the existing controls won't be affected. The controls are adjusted to the current font of the form the moment they're placed on it.

AutoScroll

This is one of the most needed of the Form object's new properties. The AutoScroll property is a True/False value that indicates whether scroll bars will be automatically attached to the form if it's resized to a point that not all its controls are visible.



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.

If you expand the AutoScrollMargin property in the Properties window, you will see that it's an object (a Size object, to be specific). It exposes two members, the Width and Height properties, and you must set both values. The default value is (0,0).

To set this property from within your code, use statements like these:

Me.AutoScrollMargin.Width = 40

Me.AutoScrollMargin.Height = 40

ControlBox

This property is also True by default. Set it to False to hide the 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.

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 other keystroke-related events), and you can handle the keystroke from within the control's appropriate handler.

In most cases, we let the control handle the keystroke and don't write any form code for that. If you want to use "universal" keystrokes in your application, you must set the KeyPreview property to True.

Doing so enables the form to intercept all keystrokes, so that you can process them from within the form's keystroke events. The same keystrokes are then passed to the control with the focus, unless you "kill" the keystroke by setting its Handled property to True when you process it on the form's level. For more information on processing keystrokes at the Form level and using special keystrokes throughout your application

MinimizeBox, MaximizeBox

These two properties are True by default. Set them to False to hide the corresponding buttons on the 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 with the MinimumSize property and no larger than the dimensions specified by MaximumSize.

The `MinimumSize` property is a `Size` object, and you can set it with a statement like the following:

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

Top, Left

These two properties set or return the coordinates of the form's top-left corner in pixels. These properties are rarely used in the code, since the location of the window on the desktop is determined by the user 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 in rare occasions.

Some dialog boxes, like the Find and Replace dialog box of any text processing application, are always visible, even when they don't have the focus. To make a form remain visible while it's open, set its `TopMost` property to `True`.

Width, Height

These two properties set or return the form's width and height in pixels. They are usually set from within the form's `Resize` event handler, to keep the size of the form at a minimum size. The forms width and height are usually controlled by the user at runtime.

Placing Controls on Forms

Designing a form means placing Windows controls on it, setting their properties, and then writing code to handle the events of interest. Visual Studio.NET is a rapid application development (RAD) environment.

It has come to mean that you can rapidly prototype an application and show something to the customer. And this is made possible through the visual tools that come with VS.NET, especially the new Form Designer.

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 with the mouse.

Next, you set the control's properties in the Properties window. Each control's dimensions can also be set in the Properties window, through the Width and Height properties.

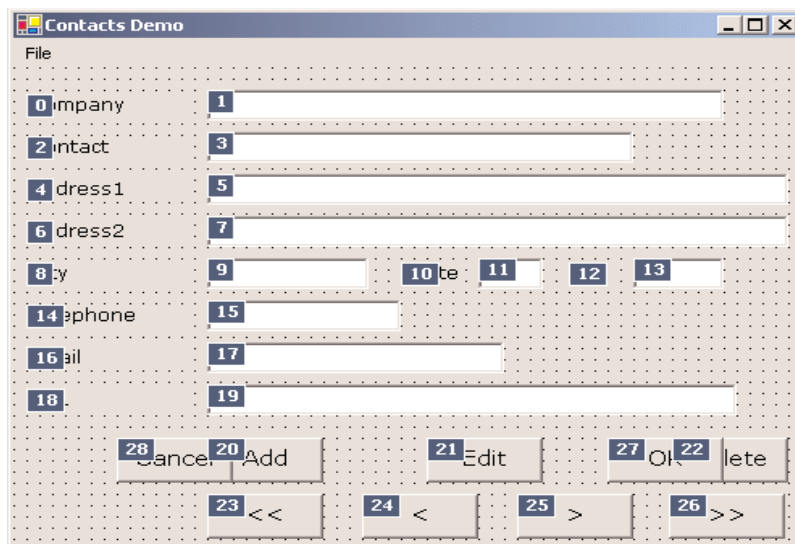
These two properties are expressed in pixels. You can also call the Width and Height properties from within your code to read the dimensions of a control. Likewise, the Top and Left properties return (or set) the coordinates of the top-left corner of the control.

Setting the TabOrder

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 isn't determined by the form; you specify the order when you design the application, with the help of the TabOrder property.

Each control has its own TabOrder 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 TabOrder of the various controls on the form need not be consecutive.

To specify the tab order of the various controls, you can set their TabOrder property in the properties window, or you can select the Tab Order command from the View menu. The tab order of each control will be displayed on the corresponding control.



Setting the TabOrder of the controls

To set the tab order of the controls, click each control in the order in which you want them to receive the focus. Notice that you can't change the tab order of a few controls only. You must click all of them in the desired order, starting with the first control in the tab order.

The tab order need not be the same as the physical order of the controls on the form, but controls that are next to each other in the tab order should be placed next to each other on the form as well.

The Form's Events

The Form object triggers several events, the most important of them being Activate, Deactivate, Closing, Resize, and Paint.

The Activate and Deactivate Events

When more than one form is displayed, the user can switch from one to the other with the mouse or by pressing Alt+Tab. Each time a form is activated, the Activate event takes place. Likewise, when a form is activated, the previously active form receives the Deactivate event.

The Closing Event

This 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 as well. For example, you may display a warning if the user has unsaved data, or you may have to update a database, and so on.

Place the code that performs these tasks in a subroutine and call it from within your menu's Exit command, as well as from within the Closing event's handler. You can cancel the closing of a form by setting the Cancel property to True.

For Example

```
Public Sub Form1_Closing(ByVal sender As Object, _
```

```
ByVal e As System.ComponentModel.CancelEventArgs) _
```

```
Handles Form1.Closing
```

```
Dim reply As MsgBoxResult
```

```
reply = MsgBox("Current document has been edited. Click OK to terminate " & _
```

```
"the application, or Cancel to return to your document.", _
```

```
MsgBoxStyle.OKCancel)
```

```
If reply = MsgBoxResult.Cancel Then
```

e.Cancel = True

End If

End Sub

The Resize Event

The Resize event is fired every time the user resizes the form with 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.

Many VB applications used the Resize event to impose a minimum size for the form. To make sure that the user can't make the form smaller than, say 300 by 200 pixels, you would insert these lines into the Resize event's handler:

Private Form1_Resize(ByVal sender As Object, ByVal e As System.EventArgs) _

Handles Form1.Resize

If Me.Width < minWidth Then Me.Width = minWidth

If Me.Height < minHeight Then Me.Height = minHeight

End Sub

The Paint Event

This event takes place every time the form must be refreshed. When you switch to another form that partially or totally overlaps the current form and then switch back to the first form, the Paint event will be fired to notify your application that it must redraw the form.

In this event's handler, we insert the code that draws on the form. The form will refresh its controls automatically, but any custom drawing on the form won't be refreshed automatically.

For Example



Loading and Showing Forms

To access a form from within another form, you must first create a variable that references the second form. Let's say your application has two forms, named Form1 and Form2, and that Form1 is the project's startup form.

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. All procedures in Form1 must see the same instance of the Form2, so that no matter how many procedures show

Form2, or how many times they do it, they'll always bring up the same single instance of Form2.) Then, to invoke Form2 from within Form1, execute the following statement:

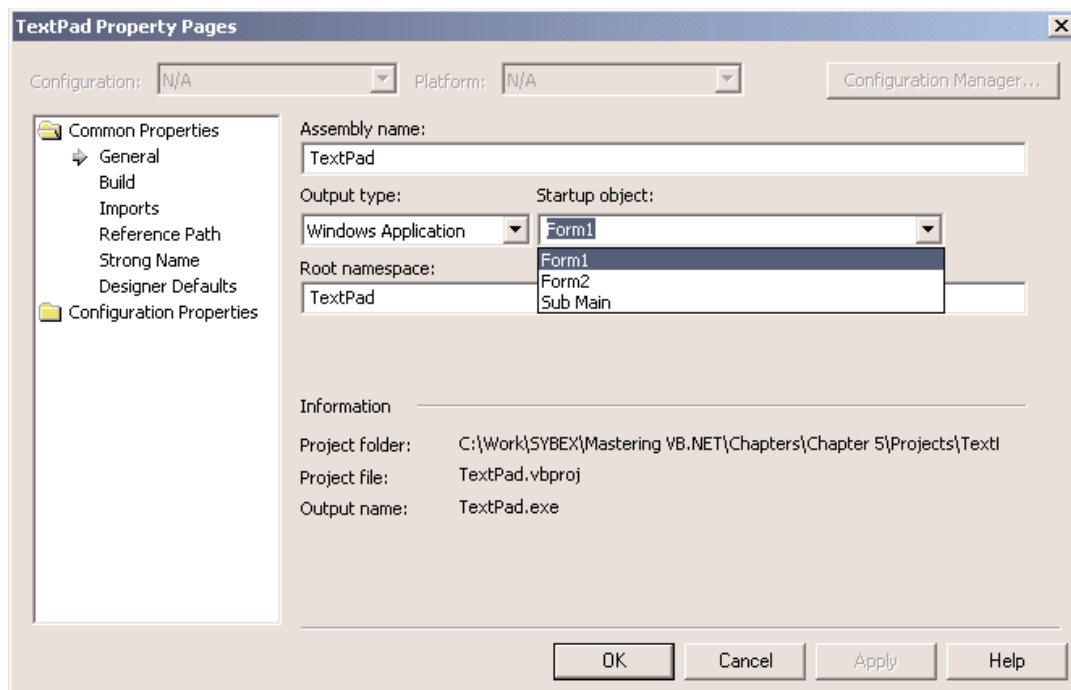
frm.Show

Startup Forms

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 see other parameters in the same window, which are discussed elsewhere in this book.

By default, Visual Basic suggests the name of the first form it created, which is Form1. If you change the name of the form, Visual Basic will continue using the same form as startup form, with its new name.



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 (I'm showing the entire module's code):

Module StartUpModule

Sub Main()

System.Windows.Forms.Application.Run(New AuxiliaryForm())

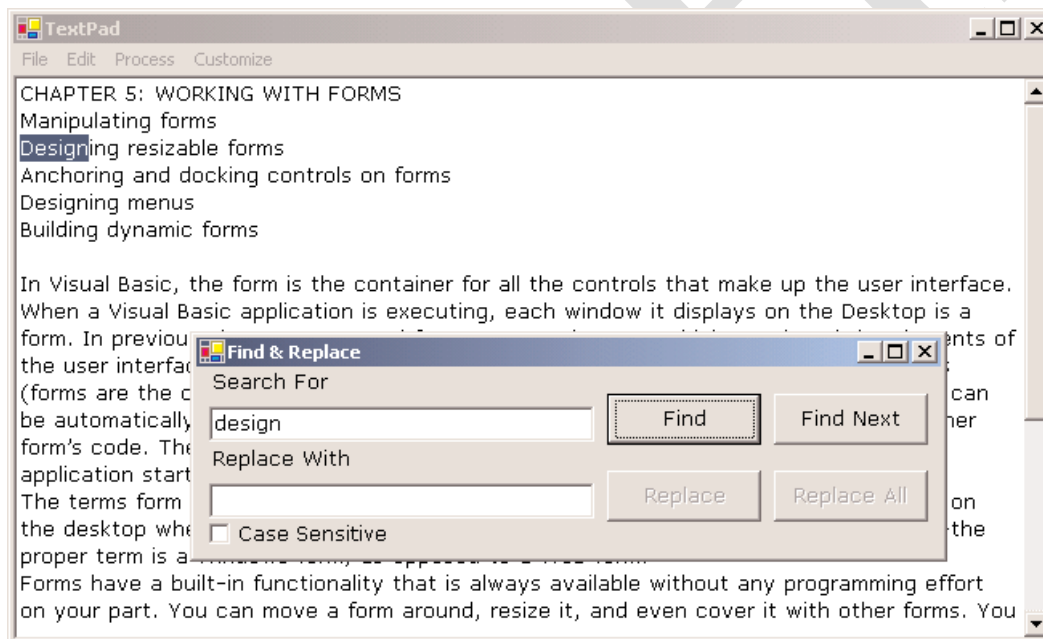
End Sub

End Module

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

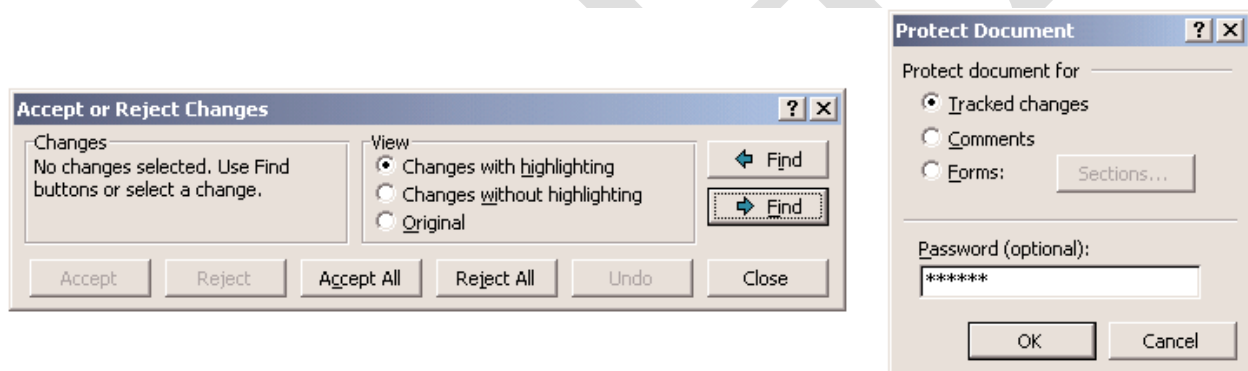
When the user clicks one of the Find & Replace form's buttons, the corresponding code must access the text on the main form of the application and search for a word or replace a string with another. The Find & Replace dialog box not only interacts with the TextBox control on the main form, it also remains visible at all times while it's open, even if it doesn't have the focus, because its TopMost property was set to True



Forms vs. Dialog Boxes

Dialog boxes are special types of forms with rather limited functionality, which we use to prompt the user for data. The Open and Save dialog boxes are two of the most familiar dialog boxes in Windows.

They're so common, they're actually known as *common dialog boxes*. Technically, a dialog box is a good old Form with its BorderStyle property set to FixedDialog. Like forms, dialog boxes may contain a few simple controls, such as Labels, TextBoxes, and Buttons. You can't overload a dialog box with controls and functionality, because you'll end up with a regular form.



Typical dialog boxes of Word

Another difference between forms and dialog boxes is that forms usually interact with each other. If you need to keep two windows open and allow the user to switch from one to the other, you need to implement them as regular forms. If one of them is modal, then you should implement it as a dialog box. A characteristic of dialog boxes is that they provide an OK and a Cancel button. The OK button tells the application that you're done using the dialog box and the application can process the information on it.

The Cancel button tells the application that it should ignore the information on the dialog box and cancel the current operation. As you will see, dialog boxes allow you to quickly find out which button was clicked to close them, so that your application can take a different action in each case.

In short, the difference between forms and dialog boxes is artificial. If it were really important to distinguish between the two, they'd be implemented as two different objects—but they're the same object.

To create a dialog box, start with a Windows Form, set its `BorderStyle` property to `FixedDialog` and set the `ControlBox`, `MinimizeBox`, and `MaximizeBox` properties to `False`. Then add the necessary controls on the form and code the appropriate events, as you would do with a regular Windows form.

Designing Menus

Menus are one of the most common and characteristic elements of the Windows user interface. Even in the old days of character-based displays, menus were used to display methodically organized choices and guide the user through an application. Despite the visually rich interfaces of Windows applications and the many alternatives, menus are still the most popular means of organizing a large number of options.

Many applications duplicate some or all of their menus in the form of toolbar icons, but the menu is a standard fixture of a form. You can turn the toolbars on and off, but not the menus.

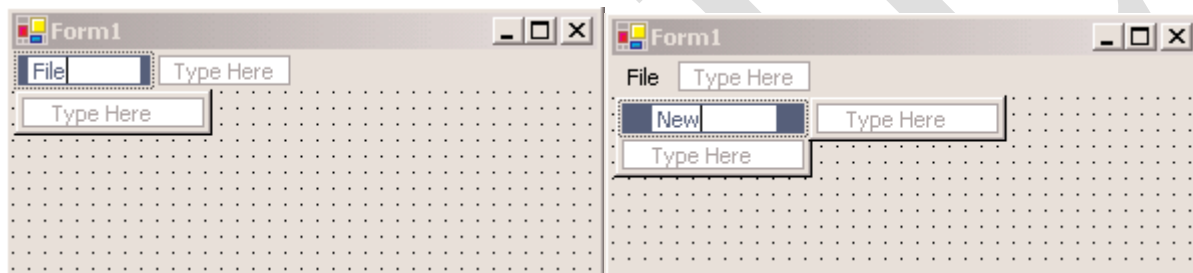
The Menu Editor

Menus can be attached only to forms, and they're implemented through the `MainMenu` control. The items that make up the menu are `MenuItem` objects. The `MainMenu` control and `MenuItem` objects give you absolute control over the structure and appearance of the menus of your application.

The IDE provides a visual tool for designing menus, and then you can program their Click event handlers. In principle, that's all there is to a menu: you design it, then you program each command's actions.

Depending on the needs of your application, you may wish to enable and disable certain commands, add context menus to some of the controls on your form, and so on. Because each item (command) in a menu is represented by a MenuItem object, you can control the application's menus

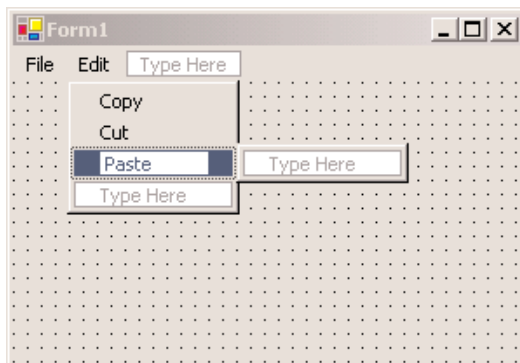
Double-click the MainMenu icon on the Toolbox. The MainMenu control will be added to the form, and a single menu command will appear on your form. Its caption will be Type Here. If you don't see the first menu item on the Form right away, select the MainMenu control in the Components tray below the form. Do as the caption says; click it and enter the first command's caption,



Enter the remaining items of the File menu—**Open**, **Save**, and **Exit**—and then click somewhere on the form. All the temporary items (the ones with the Type Here caption) will disappear, and the menu will be finalized on the form.

At any point, you can add more items by right-clicking one of the existing menu items and selecting Insert New. To add the Edit menu, select the MainMenu icon to activate the visual menu editor and then click the File item.

In the new item that appears next to that, enter the string **Edit**. Press Enter and you'll switch to the first item of the Edit menu.



Each menu item has a name, which allows you to access the properties of the menu item from within your code. The same name is also used in naming the Click event handler of the item. The default names of the menu items you add visually to the application's menu are MenuItem1, MenuItem2, and so on.

To change the default names to something more meaningful, you can change the Name property in the Properties window. To view the properties of a menu item, select it with the left mouse button, then rightclick it and select Properties from the context menu.

The MenuItem Object's Properties

The MenuItem object represents a menu command, at any level. If a command leads to a submenu, it's still represented by a MenuItem object, which has its own collection of MenuItem objects. Each individual command is represented by a MenuItem object. The MenuItem object provides the following properties, which you can set in the Properties window at design time or manipulate from within your code:

Checked

Some menu commands act as toggles, and they are usually checked to indicate that they are on or unchecked to indicate that they are off. To initially display a check mark next to a menu command, right-click the menu item, select Properties, and check the Checked box in its Properties window.

You can also access this property from within your code to change the checked status of a menu command at runtime. For example, to toggle the status of a menu command called `FntBold`, use the statement:

`FntBold.Checked = Not FntBold.Checked`

DefaultItem

This property is a True/False value that indicates whether the `MenuItem` is the default item in a submenu. The default item is displayed in bold and is automatically activated when the user double-clicks a menu that contains it.

Enabled

Some menu commands aren't always available. The `Paste` command, for example, has no meaning if the Clipboard is empty (or if it contains data that can't be pasted in the current application).

To indicate that a command can't be used at the time, you set its `Enabled` property to `False`. The command then appears grayed in the menu, and although it can be highlighted, it can't be activated.

The following statements enable and disable the `Undo` command depending on whether the `TextBox1` control can undo the most recent operation.

`If TextBox1.CanUndo Then`

`cmdUndo.Enabled = True`

`Else`

`cmdUndo.Enabled = False`

`End If`

cmdUndo is the name of the Undo command in the application's Edit menu. The *CanUndo* property of the TextBox control returns a True/False value indicating whether the last action can be undone or not.

IsParent

If the menu command, represented by a MenuItem object, leads to a submenu, then that MenuItem's IsParent property is True. Otherwise, it's False. The IsParent property is read-only.

Mnemonic

This read-only property returns the character that was assigned as an access key to the specific menu item. If no access key is associated with a MenuItem, the character 0 will be returned.

Visible

To remove a command temporarily from the menu, set the command's Visible property to False. The Visible property isn't used frequently in menu design.

MDIList This property is used with Multiple Document Interface (MDI) applications to maintain a list of all open windows.

Programming Menu Commands

Menu commands are similar to controls. They have certain properties that you can manipulate from within your code, and they trigger a Click event when they're clicked with the mouse or selected with the Enter key. If you double-click a menu command at design time, Visual Basic opens the code for the Click event in the code window.

The name of the event handler for the Click event is composed of the command's name followed by an underscore character and the event's name, as with all other controls.

To program a menu item, insert the appropriate code in the MenuItem's Click event handler.

A related event is the Select event, which is fired when the cursor is placed over a menu item, even if it's not clicked. The Exit command's code would be something like:

```
Sub menuExit(ByVal sender As Object, ByVal e As System.EventArgs) _
```

```
Handles menuExit.Click
```

```
End
```

```
End Sub
```

If you need to execute any clean-up code before the application ends, place it in the Cleanup() subroutine and call this subroutine from within the Exit item's Click event handler:

```
Sub menuExit(ByVal sender As Object, ByVal e As System.EventArgs) _
```

```
Handles menuExit.Click
```

```
Cleanup()
```

```
End
```

```
End Sub
```

The same subroutine must also be called from within the Closing event handler of the application's main form, as some users might terminate the application by clicking the form's Close button.

An application's Open menu command contains the code that prompts the user to select a file and then open it. All you really need to know is that each menu item is a MenuItem object, and it fires the Click event every time it's selected with the mouse or the keyboard. In most cases, you can treat the Click event handler of a MenuItem object just like the Click event handler of a Button.

You can also program multiple menu items with a single event handler. Let's say you have a Zoom menu that allows the user to select one of several zoom factors. Instead of inserting the same statements in each menu item's Click event handler, you can program all the items of the Zoom menu with a single event handler. Select all the items that share the same event handler (click them with the mouse while holding down the Shift button).

Then click the Event button on the Properties window and select the event that you want to be common for all selected items. The handler of the Click event of a menu item has the following declaration:

```
Private Sub Zoom200_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Zoom200.Click  
End Sub
```

This subroutine handles the menu item 200%, which magnifies an image by 200%. Let's say the same menu contains the options 100%, 75%, 50%, and 25%, and that the names of these commands are Zoom100, Zoom75, and so on. The common handler for their Click event will have the following declaration:

```
Private Sub Zoom200_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Zoom200.Click, _  
Zoom100.Click, Zoom75.Click, Zoom50.Click, Zoom25.Click  
End Sub
```

The common event handler wouldn't do you any good, unless you could figure out which item was clicked from within the handler's code. This information is in the event's *sender* argument. Convert this argument to the MenuItem type, then look up all the properties of the

MenuItem object that received the event. The following statement will print the name of the menu item that was clicked

Console.WriteLine(CType(sender, MenuItem).Text)

When you program multiple menu items with a single event handler, set up a Select Case statement based on the caption of the selected menu item, like the following:

Select Case sender.Text

Case "Zoom In"

{ statements to process Zoom In command }

Case "Zoom Out"

{ statements to process Zoom Out command }

Case "Fit"

{ statements to process Fit command }

End Select

It's also common to manipulate the MenuItem's properties from within its Click event handler. These properties are the same properties you set at design time, through the Menu Editor window. Menu commands don't have methods you can call. Most menu object properties are toggles. To change the Checked property of the FontBold command, for instance, use the following statement:

FontBold.Checked = Not FontBold.Checked

If the command is checked, the check mark will be removed. If the command is unchecked, the check mark will be inserted in front of its name. You can also change the command's caption at runtime, although this practice isn't common. The Text property is manipulated only when you create dynamic menus, as you will see in the section "Adding and Removing Commands at Runtime."

You can change the caption of simple commands such as Show Tools and Hide Tools. These two captions are mutually exclusive, and it makes sense to implement them with a single command. The code behind this MenuItem examines the caption of the command, performs the necessary operations, and then changes the caption to reflect the new state of the application:

If ShowMenu.Text = "Show Tools" Then

{ code to show the toolbar }

ShowMenu.Text = "Hide Tools"

Else

{ code to hide the toolbar }

ShowMenu.Text = "Show Tools"

End If

Manipulating Menus at Runtime

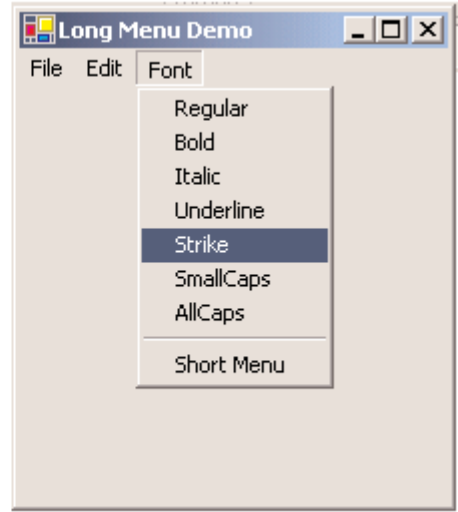
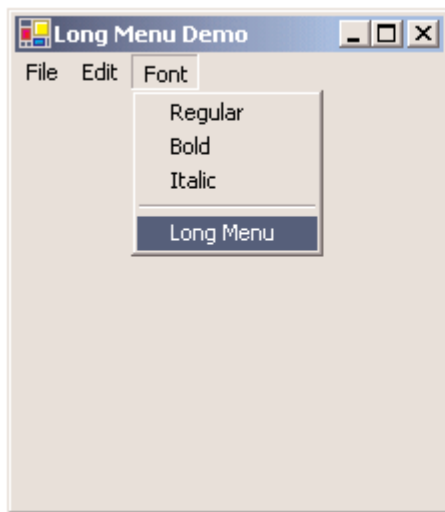
Dynamic menus change at runtime to display more or fewer commands, depending on the current status of the program. Generally there are two techniques to implement the menus as follows

- 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.



```
Protected Sub menuSize_Click(ByVal sender As Object, _  
ByVal e As System.EventArgs)  
If MenuSize.text = "Short Menu" Then  
MenuSize.text = "Long Menu"  
Else  
MenuSize.text = "Short Menu"  
End If  
  
mFontUnderline.Visible = Not mFontUnderline.Visible  
  
mFontStrike.Visible = Not mFontStrike.Visible  
  
mFontSmallCaps.Visible = Not mFontSmallCaps.Visible  
  
mFontAllCaps.Visible = Not mFontAllCaps.Visible  
  
End Sub
```

Adding and Removing commands at runtime

Many applications maintain a list of the most recently opened files in their File menu. When you first start the application, this list is empty, and as you open and close files, it starts to grow.

The RunTimeMenu project 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.

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).

```
Protected Sub btnRemoveOption_Click(ByVal sender As Object, _  
ByVal e As System.EventArgs)  
If RunTimeMenu.MenuItems.Count > 0 Then  
RunTimeMenu.MenuItems.Remove(RunTimeMenu.MenuItems.count - 1)  
End If  
End Sub  
  
Protected Sub btnAddOption_Click(ByVal sender As Object, _  
ByVal e As System.EventArgs)  
RunTimeMenu.MenuItems.Add("Run Time Option " & _  
RunTimeMenu.MenuItems.Count.toString, _  
New EventHandler(AddressOf Me.OptionClick))  
End Sub
```

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.Menu`.

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 `MenuItem` object. All the `MenuItems` under a menu command form a `MenuItems` collection, which you can scan and retrieve the individual commands.

The first command in a menu is accessed with the expression `Me.Menu.MenuItems(0)`; this is the File command in a typical application. The expression `Me.Menu.MenuItems(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 `MenuItems` collection of the top command. The first command in the File menu can be accessed by the expression

`Me.Menu.MenuItems(0).MenuItems(0)`