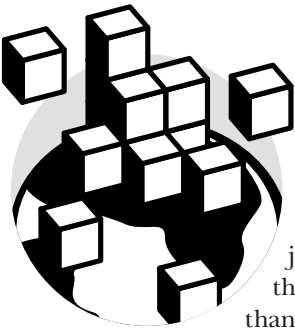


7

A VISUAL BASIC .NET CRASH COURSE



Visual Basic (VB), for all intensive purposes, has arrived, and it's just as powerful and flexible as any other .NET language, although this may well be due more to the strength of the .NET Framework than to Visual Basic as a programming language.

All versions of VB prior to Visual Basic .NET (let's refer to these versions as "classic" VB) were criticized as being non-object oriented programming languages not worthy of enterprise level or mission critical applications. However, while earlier versions lacked true full inheritance, they have been widely used to deliver mission critical applications successfully.

The fact is, a well-written Visual C++ application will nearly always out perform a Visual Basic application. But Visual Basic, while lacking in some flexibility and power, is easier to implement than Visual C++. Furthermore, because VB lacks flexibility and power, developers are less likely to create multithread problems or memory leaks at the cost of performance and stability.

This chapter provides an overview of many new Visual Basic .NET features and concepts. (For a detailed language reference, see the MSDN library online or *The Book of VB .NET, .NET Insight for VB Developers*.)

What's New in Visual Basic .NET?

The number of language enhancements in Visual Basic .NET nearly justifies the creation of a new language. Let's look at a few of the more visible changes.

Option Explicit

Forcing the explicit declaration of all variables reduces potential bugs. Classic VB required an `Option Explicit` statement in the declaration section of code if we wanted the Visual Basic compiler to enforce variable declaration which held the potential for problems because variables were often misspelled. When using `Option Explicit`, an entirely new variable is created if not already declared. Unlike classic VB, Visual Basic .NET implements `Option Explicit` by default, preventing the accidental creation of new variables and protecting the use of declared ones.

Option Strict

`Option Strict` is similar to `Option Explicit` in that it tells the compiler to require a variable declaration and requires all data conversions to be explicit. In classic Visual Basic, implicit conversions are not possible when `Option Strict` is on. (This setting is off by default.)

Option Compare

`Option Compare`, as you might guess, determines how strings will be evaluated. The two possible parameter values are `binary` and `text`. `Binary` compares the literal binary values of the two values being compared. A `binary` compare would mean the upper- and lower-case values cannot be equal, in effect enforcing case-sensitive compares. `Text` allows the evaluation of two variables to be case-insensitive.

Your application requirements will determine which `Option Compare` option you will use.

Option Base

`Option Base` is a retired option of classic Visual Basic that allowed developers to determine whether or not arrays will be 0 or 1 based. Visual Basic .NET no longer recognizes this option and sets all arrays to base 0.

Variables

Variables in .NET come in two flavors: value types and reference types. All primitive data types with the exception of the string data type are value type variables. All classes including the string data type are reference types.

The most significant difference between the types is in how they are stored in memory. Value types are stored in a stack (which requires a smaller memory footprint), while reference types are stored in a heap.

Boxing

Boxing occurs when a value type is converted to a reference type and recreated on the heap. Boxing should be used sparingly as the ability to move values from the stack to the heap is performance intensive.

The most common occurrence of boxing is when a value type variable is passed to a procedure that accepts the System.Object data type. System.Object is the equivalent of the classic Visual Basic variant data type.

ReDim

The ReDim statement, available in classic Visual Basic, is still available in Visual Basic .NET. Classic Visual Basic not only allowed developers to redimension an array, but also initialize the array. Visual Basic .NET allows the use of ReDim to redimension an array but not to initialize an array.

StringBuilder

The StringBuilder class is an impressive class optimizing string manipulation. You'll better understand its advantages once you understand how string manipulation has historically worked.

Classic Visual Basic hid the actual implementation code supporting functions available in the Visual Basic library, and string manipulation was no exception.

One common string function is the concatenation of two strings. Unfortunately, Visual Basic doesn't simply add the two strings together; instead, the windows system determines the space required for the new string, allocates memory, and places the new concatenated value into the newly allocated memory.

The StringBuilder class is implemented as an array of characters. This allows it to implement methods to manipulate what appears to be a string without the overhead incurred by an actual string. The Insert method of the StringBuilder class is used to add to the character array in a way that is much more efficient than classic string manipulation, increasing performance of many common programming scenarios. (You'll find the StringBuilder class in the System.Text namespace.)

Using the StringBuilder

This example will show you how to use the `StringBuilder` class and will compare its performance against the performance of classic Visual Basic string concatenation. To begin, follow these steps:

1. Create a windows project and build a window that looks the same as Figure 7-1, using the parameters in Table 7-1.

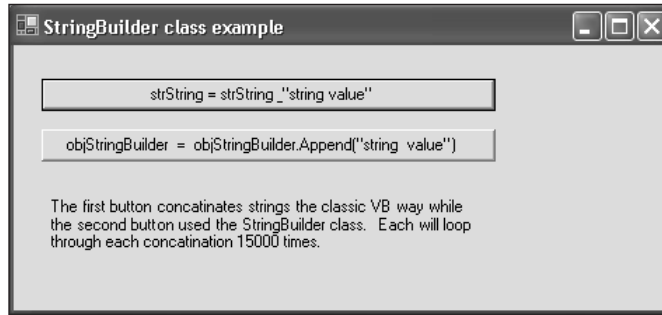


Figure 7-1: Using the `StringBuilder` class.

Table 7-1: Parameters for the `StringBuilder` Class

Control	Property	Value
Button	Name	btnString
	Text	strString = strString & "string value"
Button	Name	BtnStringBuilder
	Text	objStringBuilder = objStringBuilder.Append("string value")
Label	Name	lblStringDisplay
Label	Name	lblStringBuildingDisplay
Label	Text	The first button concatenates strings the classic VB way while the second button used the <code>StringBuilder</code> class. Each will loop through each concatenation 15000 times.

2. Add the following code segment to the click event of the btnString button.

```
.....
Dim dateStart As Date
Dim strString As String
Dim i As Integer

dateStart = DateAndTime.TimeOfDay

For i = 1 To 15000
    strString = strString & "string value "
Next i
lblStringDisplay.Text = DateAndTime.DateDiff(DateInterval.Second, _
dateStart, DateAndTime.TimeOfDay) & " Seconds"
.....
```

3. Add the following code segment to the click event of the btnStringBuilder button.

```
.....
Dim dateStart As Date
Dim objString As New System.Text.StringBuilder()
Dim i As Integer

dateStart = DateAndTime.TimeOfDay

For i = 1 To 15000
    objString = objString.Append("string value ")
Next i
lblStringBuilderDisplay.Text = DateAndTime.DateDiff( _
DateInterval.Second, dateStart, _
DateAndTime.TimeOfDay) & " Seconds"
.....
```

4. Now run the example and press each button. You will see a significant difference between the performances of the two methods of string concatenation.

NOTE

Previously, strings were built by simply adding one onto the end of another. This only seems to be what is happening. What is actually occurring is something different. When adding one string to another, you begin with the original string in memory, then a new string is allocated in memory for the string being added. Next, a new string representing the new concatenated string is created and the new string placed into it, and finally, the original string and the added string are de-allocated, leaving only the newly concatenated string in memory.

As you might imagine, this is a very inefficient process for simply adding two string values together. The StringBuilder class is a collection of characters. The StringBuilder character collection can allow values to be added and removed without the need to re-allocate and de-allocate memory blocks. As you will see, the performance difference is significant.

Structures

Classic Visual Basic allowed developers to create their own data types called User Defined Data Types or UDTs, which were implemented using the `Type` keyword. Visual Basic .NET has retired the `Type` keyword and replaced it with the keyword `Structure`, like so:

```
.....
Public Structure Person
    Dim strFirstName as String
    Dim strLastName as String
End Struct
.....
```

Variable Scope

All variables have a predefined scope that is assigned during initialization. Listed below are a few of the most common scope declarations and their definitions.

- *Private scope*: Defines a variables scope as restricted to the current method. A variable defined as having *private scope* is referred to as a member variable and is commonly prefixed with an “m”.
- *Public scope*: Allows the parent class, or calling class, access to the data held by a public variable or method.
- *Friend scope*: Similar to public scope as far as all code within a project is concerned. The difference between the *public scope* and *friend scope* is that variables or methods that are defined with the *friend scope* cannot be accessed by a parent class outside of the project.
- *Protected scope*: A new scope declaration that allows access to classes that inherit from the variables class.

Regions

The `#Region` directive allows you to organize your code into collapsible blocks which help to make the code window easier to work with by displaying only those functions you are working with. Each region can be defined with a name helping each region to be more easily identifiable, as shown here

```
.....
#Region "MyRegion"
    'some code
#End Region
.....
```

When you are done writing “some code,” you can collapse the region and begin working on the next segment of code.

Windows Forms

Visual Basic .NET implements Windows Forms as classes that inherit windows functionality from the Form class found in the System.Windows.Forms namespace. Developing Win32 applications in Visual Basic .NET is still very similar to classic Visual Basic windows development in that windows controls can be dragged and dropped onto the form designer. The difference is that none of the implementation code is hidden.

For example, here's the implementation code for the Windows Form discussed in the previous example of the StringBuilder class. While this type of code must be implemented in classic Visual Basic forms, it is hidden. As you can see, the code is no longer hidden; however, I would strongly recommend leaving this code alone unless you really know what you are doing and have a specific need to fill. Take a look at the code below and notice that the entire form is actually a class that inherits the System.Windows.Forms.form class. As mentioned earlier in this book, everything in .NET is a class. There are no exceptions.

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

    #Region " Windows Form Designer generated code "

        Public Sub New()
            MyBase.New()

            'This call is required by the Windows Form Designer.
            InitializeComponent()

            'Add any initialization after the InitializeComponent() call

        End Sub

        'Form overrides dispose to clean up the component list.
        Protected Overrides Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
                If Not (components Is Nothing) Then
                    components.Dispose()
                End If
            End If
            MyBase.Dispose(disposing)
        End Sub

        'Required by the Windows Form Designer
        Private components As System.ComponentModel.IContainer
```

(continued on next page)

```
'NOTE: The following procedure is required by the Windows Form Designer
```

```
End Sub
```

```
#End Region
```

```
End Class
```

The implementation code for all the controls on the form were stored in the “Windows Form Designer generated code” region. (This information has been removed so you won’t be distracted from the Windows Form’s own implementation.)

Project Structure

While the Visual Basic compiler used a file’s extension to determine what type of project file it was, Visual Basic .NET implements all code through classes. All Visual Studio .NET needs to know is the language the file is written in.

Project groups in Visual Studio have proved to be a powerful tool for managing, building, and debugging multiple project solutions. Visual Studio .NET replaces the Group Project with a Project Solution, which is one or more projects and the supporting files. Because solutions actually contain projects and items, they are often referred to as Solution Containers. And, because projects also contain files, it should come as no surprise that projects are referred to as Project Containers.

Solutions and project files each have their own extensions so that Visual Studio .NET knows what kind of container they are:

- **.sln:** The file extension of a solution file which maintains all solution specific information.
- **.suo:** The file extension of all Solution User Options files which maintains all of the user’s preference information for the solution.
- **.vbproj:** The file extension of all Visual Basic project files.
- **.vb** and **.cs:** The file extensions of Visual Basic .NET and C# files, respectively. This is a significant improvement from previous versions of Visual Studio when forms, classes, and other components were given component specific file which offered no clue as to the language used to build the file. Project items built using a specific language will always have that language’s file extension, thus allowing Visual Studio .NET to know which compiler it must use. (For additional information on file extensions of project items, refer to the MSDN article entitled, “File Types and File Extensions in Visual Basic and Visual C#”.)

ErrorProvider

One of the more interesting Windows and Web Form improvements is the ability to alert the user of exceptions without interrupting them until they press a button that performs validation, providing better overall user experience. The `ErrorProvider` component is a non-visual component that allows you to perform data validation on form controls. If a data violation occurs, you can set a message to be displayed as a tool tip near the offending control.

Ideally, you should implement data validation, a type of business rule enforcement, at the lowest common layer. The most ideal place to do so is at the database level because this is the only application layer that cannot be bypassed. Furthermore, rules enforced here are not duplicated as they would be if you implemented business rules in the presentation layer. For example, if a name can be equal to or less than 20 characters and the rule is implemented in the presentation layer, then every form that supports the use of the name must implement the same rule. If the rule is changed, it must be changed in every form that uses the name. This is both sloppy and error prone.

Current technology does not lend itself to this level of data validation very easily; however, over time Microsoft will devise better validation schemes and developers will build custom solutions. The challenge is to provide solid data validation without compromising user experience. To implement data validation in the presentation layer means that we are duplicating business rule enforcement because the same rules are surely implemented in the database as well. Of course the risk is that when the database schema changes, we may miss making the same changes in the presentation layer.

If all business rules are implemented in the Component layer, it is possible to bypass the rules by ignoring the Component layer or building another one that does not implement the rule. In such a case, you have the potential of corrupting data that will be more expensive to repair than it would have cost to devise a sound business rules enforcement schema.

NOTE

The XML Schema is excellent place to begin looking for sound business rule implementation. By pulling the database schema from the database and persisting it in memory, you can leverage XML Schema to enforce data types and constraints. Also, when building your web page dynamically you can enforce these data specific rules through the `ErrorProvider`. In this case, you are implementing rules at the business level layer, which were defined in the database at design time; the violation can be made known through the `ErrorProvider` at the presentation layer. This is the ideal way to enforce data specific business rules; all other programmatic business rules should be implemented in the business level layer. Never enforce business rules in the presentation layer.

Implementing Namespaces

Namespaces make it easy to organize classes, functions, data types, and structures into a hierarchy. Namespaces allow you to quickly access classes and methods buried in the .NET Framework Class Library or any other application that provides a namespace. The .NET Framework Class Library provides hundreds of classes and thousands of functions as well as data types and structures.

Use the Imports statement to import a namespace for easy access to its classes and methods. Once imported it is no longer necessary to use a fully qualified path to the desired class or method. For example:

```
Imports system.text ' Give access to the StringBuilder class
```

Table 7-2 lists the namespaces that are used most commonly and their general functionality.

Table 7-2: Commonly Used Namespaces

Namespace	Functionality
Microsoft.VisualBasic	Contains the Visual Basic .NET runtime, classes, and methods used for compiling Visual Basic code.
Microsoft.VSA	Provides a host of scripting functionality, allowing you to give users the ability to customize of your application.
Microsoft.Win32	Provides access to the Windows Registry and the ability to handle events raised by the operating system.
System	Provides basic classes and methods commonly used by all .NET languages. Includes data types, arrays, exception classes, Math functions, Garbage collector, conversion classes, console access, and the Object class along with many more commonly used components.
System.Collection	Provides interfaces and classes used for creating a variety of collection types. Collection types are <ul style="list-style-type: none"> • ArrayList: Basic collection object. • BitArray: A collection of bits. (0 and 1 values) • DictionaryBase: Implements base class for strongly typed collections supporting key (name) value pairs. • Hashtable: High performing collection of key (name) value pairs. • Queue: FIFO (First-in First-out) collection. • SortedList: A collection of key (name) value pairs sorted by the key (name). • Stack: LIFO (Last-in First-out) collection. • Specialized.ListDictionary: A faster collection than a Hashtable for 10 items or less. • Specialized.HybridDictionary: Acts as a ListDictionary until the collection gets larger where it converts to a Hashtable. • Specialized.StringCollection: A collection of strings.

Table 7-2: Commonly Used Namespaces (continued)

Namespace	Functionality
System.ComponentModel	<p>Provides classes and interfaces for runtime and design time behavior including the ability to contain or to be contained.</p> <p>The Container interface allows the Visual Studio .NET development environment to provide a graphical interface when developing a component.</p>
System.Data	Provides classes and interfaces to support data access including ADO.NET, XML, OLEDB, and SQL Server access.
System.Diagnostics	<p>Provides classes, allowing access to</p> <ul style="list-style-type: none"> • Event Log: A mechanism employed by the operating systems providing a common area to record application and system events. Events can include errors, warnings, and informational data. • Processes: Provides the ability start and stop processes as well as monitor processes on remote machines. • Performance Counters: Provides the ability to monitor performance of local and remote machines including the ability to define and create custom performance counters.
System.DirectoryServices	<p>Provides the ability to search and interact with Active Directory services providers:</p> <ul style="list-style-type: none"> • Active Directory is a hierarchical mean for logically organizing network and system resources. • Active Directory providers include IIS, LDAP (Lightweight Directory Access Protocol), NDS (Novel NetWare Directory Service), and WinNT directory services.
System.EnterpriseService	Provides the ability to employ COM+ functionality for building enterprise applications.
System.Globalization	Provides classes supporting multiple languages and cultures including date formats, calendars, and currencies.
System.IO	<p>Provides read and write access to data streams and files. Additional access is provided to related tasks including:</p> <ul style="list-style-type: none"> • Manipulation of the creation, modification, and deletion of file directories. • Manipulation of files through creation, modification, deletion, coping, and moving. • Provides information about files and directories, such as the existence of a file or directory, the extension of files, and the full path of files. • Allows access to system events including file system changes.

(continued on next page)

Table 7-2: Commonly Used Namespaces (continued)

Namespace	Functionality
System.Management	Provides access to information provided by WMI (Windows Management Instrumentation) including information about a systems drive space and CPU Utilization. Please refer to WMI for additional information concerning data that can be derived from the System.Management namespace.
System.Messaging	Provides classes for managing Message Queues.
System.Net	Provides classes for commonly used network protocols.
System.Reflection	Provides classes for access to component metadata stored in an assemblies manifest.
System.Runtime.Remoting	Provides classes designed for building distributed applications similar to classic DCOM (Distributed Component Object Model).
System.Runtime.Serialization	Provides classes designed to serialize objects into a sequence of bits for storage or transfer to another system.
System.Security	Implements the CLR's (Common Language Runtime) security components.
System.ServiceProcess	Provides classes required for building Windows Services.
System.Text	Provides classes for manipulating string data. Most notable is the StringBuilder class defining a modern approach to string manipulation.
System.Threading	Provides classes for building multi-threaded applications.
System.Timer	Provides classes for implementing non-visual timed events allowing actions to be taken on a given interval.
System.Web	Provides classes and additional subordinate namespaces encompassing all aspects of web development including ASP .NET, Web Services, and web controls and more.
System.Windows.Forms	Provides classes required for developing windows form applications.
System.XML	Provides classes for manipulating and using XML.

Structured Exception Handling

One of the more complex and important aspects of application development is error handling. Errors that are not handled can have devastating effects on the success of any application because, in most cases, the application will not be able to recover or shut down gracefully. Among the challenges is the lack of comprehensive error handling across languages and platforms.

To complicate matters further, handling errors is only the tip of the proverbial iceberg. As important as error handling is, an application must be

able to handle unacceptable application level events or exceptions that are not necessarily system generated error. Now, the term “error handling” is no longer sufficient.

Exception handling deals with any system or application generated error, what we now refer to as an exception. This ability to handle all exceptions, both system and application generated, goes a long way toward giving the user a stable application and a better overall user experience.

The CLR is a language independent means for exception handling that places all raised or thrown errors into an exception object that can be manipulated like any other object. In addition, the exception object can be created and used, programmatically, anywhere within an application.

The CLR implements exception handling with the Try/Catch/Throw model. This model of exception handling, while foreign to Visual Basic programmers, is well known by C++ programmers. This is a structured exception handling model that is time tested and proven as a solid means for handling or dealing with thrown exceptions.

Throwing an exception is similar to raising an error in Visual Basic. The simplest exception structure is as follows:

```
.....
Public Sub MySub()
    Try
        'Some code
    Catch
        'Deals with any exception that may occur.
    End Try
End Sub
.....
```

In its most basic form, all application and exception code goes into what is called a Try block. In this example, the Try block is the area between the Try and the End Try statements.

The Try block can be divided into three clauses. The area between the Try and Catch statements is where you place your applications code. The area between the Catch and, in this case, the End Try statements are where your exception handling code resides. Another section, defined as the finally clause, is available in the last clause where the Try blocks code executes. Our simple exception handling example does not use the finally clause as it is not required when a Catch statement is available.

Exception Handling Rules

A set of rules governs how the Try block can be used. But before we have a look at these rules, let’s take a quick look at some of the classes that support exception handling.

Table 7-3 lists several common exception classes you can look forward to using. Exception classes are thrown when a related exception is thrown; when looking for exceptions look for the following:

Table 7-3: Exception Classes

Exception Class	Reason Exception Class Is Thrown
System.AppDomainUnloadedException	Thrown when attempting to use an unloaded application domain.
System.ApplicationException	Thrown when a non-fatal application error has occurred.
System.ArgumentException	Thrown when an argument passed is not valid.
System.ArgumentNullException	Thrown when a null is passed as a method parameter that does not accept null values.
System.ArgumentOutOfRangeException	Thrown when a passed value is outside the range of a methods parameter.
System.ArithmeticException	Thrown when an error occurs while performing arithmetic and conversion operations.
System.ArrayTypeMismatchException	Thrown when adding a value of the incorrect data type to an array.
System.DivideByZeroException	Thrown whenever a value is divided by zero.
System.DllNotFoundException	Thrown when a DLL referenced as imported is not available.
System.IndexOutOfRangeException	Thrown when trying to access an invalid index in an array.
System.InvalidCastException	Thrown when an invalid conversion attempt is made.
System.NullReferenceException	Thrown when attempting to dereference a null object reference.
System.OutOfMemoryException	Thrown when memory is not available to perform the specified task.
System.OverflowException	Thrown when an operation overflow occurs.
System.RankException	Thrown when an array with the wrong number of dimensions is passed to a methods parameter.
System.SystemException	Is the base class for all exception classes in the System namespace.
System.Data.ConstraintException	Thrown when a constraint is violated.
System.Data.DataException	Thrown when an ADO.NET component generates an error.
System.Data.DBConcurrencyException	Thrown when the number of rows affected in an update procedure is zero.
System.Data.DeletedRowInaccessibleException	Thrown when attempting to perform data manipulation operations on a data row that has been deleted.
System.Data.InvalidConstraintException	Thrown when a data relationship is violated.

Table 7-3: Exception Classes (continued)

Exception Class	Reason Exception Class Is Thrown
System.Data. NotNullAllowedException	Thrown when inserting a null value where one is not accepted.
System.IO. DirectoryNotFoundException	Thrown when a specified directory cannot be found.
System.IO.FileLoadException	Thrown when a file cannot be loaded.
System.IO.IOException	Thrown when an I/O error occurs.
System.IO.PathTooLongException	Thrown when a path or file name are too long.
System.Runtime.Remoting. RemotingException	Thrown when an error occurs during a remote operation.
System.Runtime.Remoting. RemotingTimeoutException	Thrown when the response of a server or client exceed a predefined interval.
System.Runtime.Remoting. ServerException	Thrown when an error occurs while working with a remote component that is an unmanaged application incapable of throwing an exception.
System.Runtime.Serialization. SerializationException	Thrown when an error occurs during the serialization or deserialization of a component.
System.Web.HttpException	Allow an http exception to be thrown.
System.XML.XmlException	Provides exception information about the last XML exception.

Basic Rules

Basic rules govern the use of a Try block. (They will quickly become obvious after using Try blocks a few times.)

- All Try blocks must employ at least one catch or finally clause.
- A Catch clause with no other parameters will catch all unhandled exceptions.
- The finally clause always executes when available except when an Exit Try occurs. As such, the finally clause is a good place to perform component cleanup. If an Exit Try statement is used anywhere in the Try block then it is better to perform cleanup after the End Try block statement.
- Developers familiar with the On Error statement may still perform error handling as they did in Visual Basic only when a Try block does not exist in the procedure.

Exception Handling Examples

For the following examples we will create a single Windows Form and add a button for each example. To begin, create a new project and name it “Exception Handler”.

Try . . . Catch

One popular and easy way to understand an example of error handling has always been the divide by zero error, or in .NET terms “exception.” You will use the divide by zero exception wherever possible so as to not distract you from what the chapter is trying to convey. (You will examine a few other specific exception conditions later in this chapter.)

This example demonstrates the simplest of all exception structures:

1. Drag a button onto your Windows Form and label it “Try... Catch”.
2. Change the buttons name to “btnTryCatch”.
3. Apply the following code in the click event of the button.

```
.....  
Dim intResult as Integer"  
Dim int1 as Integer = 5  
Dim int2 as Integer = 0  
Try  
    intResult = int1 / int2  
Catch objException as System.OverflowException  
    MessageBox.Show("Divide by zero has occurred")  
End Try  
.....
```

The preceding example evaluates the exception object by using the catch clause. The catch clause checks to see if the exception object contains an “OverflowException” exception. If so, and in this case it will, the code in the catch clause executes.

The Finally Clause

This example employs the same code as the one previously, except it demonstrates that the Finally clause always executes:

1. Drag another button onto your Windows Form and label it “Finally”.
2. Change the buttons name to “btnFinally”.
3. Apply the following code in the click event of the button.

```
.....  
Dim intResult as Integer  
Dim int1 as Integer = 5  
Dim int2 as Integer = 0  
Try  
    intResult = int1 / int2  
Catch objException as System.OverflowException
```

```

        MessageBox.Show("Divide by zero exception has occurred")
    Finally
        Dim obj As New System.Text.StringBuilder()
        obj = obj.Append("Regardless of whether or not an ")
        obj = obj.Append("exception occurs, the Finally clause ")
        obj = obj.Append("will execute.")
        MessageBox.Show(obj.ToString)
        obj = Nothing
    End Try

```

Feel free to remove the errant code with “intResult = int1 / 1” and observe that the finally clause still executes.

The Exit Try Statement

This example demonstrates the Exit Try statement. You could simply add the Exit Try statement to a previous example: however, go ahead and create a new button to keep each example separate for future reference.

1. Drag another button onto your Windows Form and label it “Exit Try”.
2. Change the button name to “btnExitTry”.
3. Cut and paste code from the Finally button to the Exit Try button.
4. In the Catch clause, place the “Exit Try” statement after the messagebox statement:

```

Dim intResult as Integer
Dim int1 as Integer = 5
Dim int2 as Integer = 0
Try
    intResult = int1 / int2
Catch objException as System.OverflowException
    MessageBox.Show("Divide by zero exception has occurred")
    Exit Try
Finally
    Dim obj As New System.Text.StringBuilder()
    obj = obj.Append("Regardless of whether or not an ")
    obj = obj.Append("exception occurs, the Finally clause ")
    obj = obj.Append("will execute.")
    MessageBox.Show(obj.ToString)
    obj = Nothing
End Try

```

You will notice when the exception occurs, the exceptions message is displayed and the Try block is exited. In this case, the Finally block does not execute and is not a good place to clean up objects in memory.

Multiple Catch Statements

It is often preferable to use Multiple Catch statements in a single Try block, although the placement of Catch statements can impact performance.

Once an exception is caught, the processing of the remaining Catch statements is aborted. Subsequently, once the Catch clause completes processing, the Finally clause is processed if available. To increase the performance of your Try blocks, place the most likely one to error or more common exceptions in the first Catch blocks while placing the least likely errors toward the end.

Here's how to use Multiple Catch statements:

1. Drag another button onto your Windows Form and label it "Multiple Catch".
2. Change the button name to "btnMultipleCatch".
3. Cut and paste code from the Finally button to the Multiple Catch button.
4. Make the following changes as highlighted in the code below:

```

.....
Dim intResult As Integer
Dim int1 As Integer = 5
Dim int2 As Integer = 0
Dim str1 As String

Try
    str1 = int1 / int2
    Throw (New Exception("A different exception"))
Catch objException As System.OverflowException
    MessageBox.Show("Divide by zero exception has occurred")
Catch
    MessageBox.Show("Some other exception has occurred.")
Finally
    Dim obj As New System.Text.StringBuilder()
    obj = obj.Append("Regardless of whether or not an ")
    obj = obj.Append("exception occurs, the Finally clause ")
    obj = obj.Append("will execute.")
    MessageBox.Show(obj.ToString)
obj = Nothing
End Try
.....

```

As you step through the procedure you will notice that you no longer receive an overflow exception. You will also notice that the string value receiving the results of the calculation has the value "infinity" when dividing a number by zero. The overflow exception does not occur, however, when we threw an exception to the calling method. The first Catch clause is completely ignored, but the second Catch clause is not looking for any specific exception; as a result, the second Catch clause catches all exceptions not already caught.

NOTE

Historically, when you were building COM components, the best practice was to ensure that all methods and components handled their own exceptions. The best practice with .NET components is to pass the exception to the client and allow the client to determine the next course of action. This is due in part because all languages now understand how to deal with each other's languages exceptions, therefore, it is no longer critical for the language catching the exception to also deal with the exception. Also, exceptions don't always correlate to an error that occurred. Often an exception simply indicates an application state that is not what the method requires. This could be as simple as the database is not available. In this case, no error has occurred in the code; however, because the database is unavailable, the method cannot complete its assigned task.

Getting Exception Information

The exception class has several properties and methods. You'll learn about a few of the more notable ones here and then examine an example of each:

- **Source property:** The Source property of the exception class is intended to hold the application or object name generating the exception. It can also be programmatically set, but if it is not set, the property returns the assembly name where the exception occurred.
- **Message property:** The Message property is a string containing a description of the current exception.
- **TargetSite property:** The TargetSite property is a string containing the name of the procedure where the exception occurred.
- **GetType method:** The GetType method is inherited from the System.Object class and returns the type of exception that has occurred.
- **ToString method:** The ToString method returns a string describing the current exception including information provided by several other exception class properties and methods.

Exception Class Properties and Methods Example

This example demonstrates the use of some exception class properties and methods:

1. Drag another button onto your Windows Form and label it "Properties/Methods."
2. Change the button name to "btnPropMeth."

3. Cut and paste code from the Finally button to the Properties/Methods button. 4. Add the following code to the newly pasted code:

```

.....
Dim intResult as Integer
Dim int1 as Integer = 5
Dim int2 as Integer = 0
Try
    intResult = int1 / int2
Catch objException as System.OverflowException
    MessageBox.Show("Divide by zero exception has occurred")

    'Source, Message, TargetSite, GetType, ToString
    MessageBox.Show("Source: " & objException.Source())
    MessageBox.Show("Message: " & objException.Message())
    MessageBox.Show("TargetSite: " & objException.TargetSite.Name)
    MessageBox.Show("GetType.Name: " & objException.GetType.Name)
    MessageBox.Show("ToString: " & objException.ToString())
Finally
    Dim obj As New System.Text.StringBuilder()
    obj = obj.Append("Regardless of whether or not an ")
    obj = obj.Append("exception occurs, the Finally clause ")
    obj = obj.Append("will execute.")
    MessageBox.Show(obj.ToString)
    obj = Nothing
End Try
.....

```

Object-Oriented Programming (OOP)

As mentioned earlier, Visual Basic did not meet the test as a true object-oriented language that implements true object-oriented programming as defined by abstraction, encapsulation, polymorphism, and inheritance. Visual Basic .NET not only supports inheritance, but also supports a variety of inheritance implementations including interface, forms, and implementation or polymorphism inheritance.

Before you continue any further, let's briefly discuss the four main concepts of object orientation and the implementation code for each. Each brief discussion is followed with an example that demonstrates the discussed OOP concept.

Abstraction

Abstraction is the easiest of the OOP concepts to understand and is often something we implement naturally without realizing it. In short, abstraction is the implementation of code to mimic the actions and characteristics of any real-world entity.

The most commonly-used example for describing abstraction is the abstraction of a person. Imagine that we want to create an object from a class that represents a person. A person class will need to describe its characteristics through the implementation of properties. Actions of the person class are performed by methods.

Encapsulation

Encapsulation is the programmatic equivalent of a black box. An actual black box may have a switch and dials. Inside the box would be the mechanisms to perform the actions provided by the black box.

We expose properties and methods through abstraction, but we implement the actual workings of our component through encapsulation. A few encapsulated actions might include data access, data validation, calculations, adding data to an array or collection, or calling other methods or other components.

Exposing our components interface while hiding the component's implementation code effectively separates interface implementation from our black box implementation. This separation helps to modularize components to perform a more specific task while requiring minimal knowledge of how the black box actually works.

One of the more useful applications of encapsulation is in making a complex component. For example, your program may require interaction with a third party system, but interaction with this system can only be achieved through a complex API. Rather than requiring all developers on a project to spend valuable time figuring out how to correctly use the third party API or even find ways to misuse it, one developer could study the API then encapsulate it in a component that exposes a less complex interface. This is a common practice that saves time and reduces potential bugs.

Like abstraction, encapsulation isn't as much a technology as it is a method of code implementation. In the case of encapsulation, our method of implementation separates the exposed interface from the actual implementation code.

Polymorphism

Abstraction is an interface implemented to represent a real-world object; encapsulation is the implementation of a black box through interface and implementation separation; and polymorphism is the ability to implement the interface of another class into multiple classes or to implement multiple interfaces on a single class. This method of implementation is referred to as *interface-based programming*.

A vehicle is a good example of polymorphism. A vehicle interface would only have those properties and methods that all vehicles have, a few of which might include paint color, number of doors, accelerator, and ignition. These properties and methods would apply to all types of vehicles including cars, trucks, and semi-trucks.

Polymorphism will not implement code behind the vehicle's properties and methods. (That's the job of inheritance covered in the next section.) Instead, polymorphism is the implementation of an interface. If the car, truck, and semi-truck all implement the same vehicle interface, then the client code for all three classes can be exactly the same.

Implementing the vehicle interface only requires the declaration of properties and methods. To create a new interface, use the `Interface` keyword in place

of the Class keyword. The client implementing the new interface can do so by using the Implements keyword as shown in the example:

```
.....
Implements IVehicle
.....
```

After using the Implements keyword, you will notice that Intellisense displays the properties and methods of the IVehicle interface. Using the Implements keyword will only give access to the properties and methods of the IVehicle interface; however, you must provide your own code behind the methods and property declarations to match the interface.

Inheritance

Inheritance is the ability to apply another class's interface and code to your own class. Remember, with polymorphism, you got the interface; however, you must apply your own code. The power of inheritance is the ability to inherit code, saving developers time. This type of inheritance is called *implementation inheritance*. To inherit another class, use the Inherits keyword.

Visual inheritance is the ability to inherit another form's look and feel onto another. Remember, everything in .NET is a class, including forms. If you create a project that exists in the MyApp namespace, create a form name MyBaseForm. The following code will inherit the MyBaseForm within our new form:

```
.....
Public Class MyNewForm
    Inherits MyApp.MyBaseForm
End Class
.....
```

Properties

Properties are part of a program's interface and describe the characteristics of a class. These properties hold information about a class or, when loaded into memory, an object. Properties, as they exist in classes, are often referred to as "data." When a reference is made to a class's data, you will know that the reference is actually directed toward a class's property.

To create a property, use the Property keyword and then define the type of property you are implementing. Properties can be read-only, write-only, or read and writable. To define the characteristics of properties, use the keywords ReadOnly, WriteOnly, or supply no definition at all to implement both read and write ability.

Visual Studio .NET makes properties easier to implement by adding the basic shell of property code based on the property's scope definition. Unlike Visual Basic, Visual Basic .NET automatically supplies code for both read and write functionality: "Get" for read ability method and "Set" for write access to a property.

Create a new class, type the following code, and press ENTER:

```
.....
Public Property FName() As String
.....
```

Visual Studio .NET will automatically fill in the rest of the code that is required by the FName property:

```
.....
Get
Return m_FName
End Get
    Set(ByVal Value As String)
        m_FName = Value
    End Set
End Property
.....
```

Methods

Methods are the actions exposed by a class in the form of either functions or sub-procedures. Sub-procedures and functions both execute code on behalf of the calling application, but sub-procedures simply execute code while functions execute code, then return a value.

The .NET Framework provides at least two new changes to how you can use procedures. In Visual Basic, you could call a procedure without the use of parameters, including procedures that required no parameters at all. The .NET Framework requires parenthesis to follow all methods even when parameters are not required. For example:

- Visual Basic 6 method call:

```
.....
intResult = DoSomething
.....
```

- Visual Basic .NET method call:

```
.....
intResult = DoSomething()
.....
```

Another change is the addition of the Return keyword. When returning a value for a function in Visual Basic, you set the function's name equal to the value being returned. With Visual Basic .NET, you can set the keyword Return equal to a value and the value will be returned with the function. This is very useful when making code more generic. For instance, you can easily cut and past a method's code without regard to another method's function name because the keyword Return is used for setting the method equal to a return value. Examples of the old versus new method for returning values of a function are:

- Visual Basic 6 method call:

```
.....
Public Function DoSomething() as Int32
    DoSomething = 10
End Function
.....
```

- Visual Basic .NET method call:

```
.....
Public Function DoSomething() as Int32
    Return = 10
End Function
.....
```

If you look closely at a function's supporting properties you will find that the `Return` keyword is used by default. You can set the function name equal to a given value.

The third significant change is in how parameters are passed. Visual Basic passed a parameter value `ByRef` by default. The preferred method for passing parameter values is to explicitly define whether a value is passed by `ByRef` or `ByVal`. Finally, when using the `Option` keyword, you must define a default value similar to how C has worked for many years now.

Declaration Options

We have covered a few of the most common declaration methods including those that describe the scope of a property or method. Several description options will extend or restrict scope.

Here is a list of the most commonly-used declaration options with brief descriptions of each:

- **Private:** The `Private` keyword defines a variable or method as accessible only by code within the context of where the declaration occurred; outside code is not permitted access. Variables and methods defined as private are often referred to as member variables or methods, and commonly prefixed with an "m".
- **Public:** The `Public` keyword declares a property or method as accessible by anyone within the calling application or within the class itself.
- **Friend:** The `Friend` keyword defines a property or method as accessible by members within the class it is declared in.
- **Protected:** The `Protected` keyword defines a property or method as accessible only by members of its class or by members of an inheriting class.
- **Default:** A `Default` property is a single property of a class that can be set as the default. This allows developers that use your class to work more easily with your default property because they do not need to make a direct reference to the property. Default properties cannot be initialized as `Shared` or `Private` and all must be accepted at least on argument or parameter. Default properties do not promote good code readability, so use this option sparingly.
- **Overloads:** The `Overloads` property allows a function to be described using deferent combinations of parameters. Each combination is considered a signature, thereby uniquely defining an instance of the method being defined. You can define a function with multiple signatures without using the keyword `Overloads`, but if you use the `Overloads` keyword in one, you must use it in all of the function's Overloaded signatures.

- **Shared:** The Shared keyword is used in an inherited or base class to define a property or method as being shared among all instances of a given class. If multiple instances of a class with shared properties or methods are loaded, the shared properties or methods will provide the same data across each instance of the class. When one class alters the value for a shared property, all instances of that class will reflect the change. Shared properties of all instances of the class point to the same memory location.
- **Overridable:** The Overridable keyword is used when defining a property or method of an inherited class, as overridable by the inheriting class.
- **Overrides:** The Overrides keyword allows the inheriting class to disregard the property or method of the inherited class and implements its own code.
- **NotOverridable:** The NotOverridable keyword explicitly declares a property or method as not overridable by an inheriting class, and all properties are “not overridable” by default. The only real advantage to using this keyword is to make your code more readable.
- **MustOverride:** The MustOverride keyword forces the inheriting class to implement its own code for the property or method.
- **Shadows:** The Shadows keyword works like the Overloads keyword except that with shadows we do not have to follow rules such as implementing the same signature. The Shadows keyword does not require the consent (override ability) of the inherited class to replace the property or method’s implementation code. A method does not have to be defined as overridable for the Shadows keyword to work.

Object Instantiation

When you drag and drop controls onto a Windows Form, you are using objects. When you observe your code, you are looking at a class; when that code is loaded into memory, at runtime, it is considered an object. The importance of the distinction is simply to describe that a class is a template, while an object is an instance of that template in memory. Also, many copies of the template can exist in memory at the same time as objects.

Fortunately, we do not have to depend on the component designer to work with classes; we can build our own classes and components. This is nothing new for a moderately experienced developer; what is new is how Visual Basic .NET permits us to instantiate classes.

Classic COM relied on the Windows Registry to store its exposed properties, methods, events, and enumerations; a client application could only access these exposed interfaces through the Registry. As a result, the way you instantiate classes when using classic COM components in COM+ is very important. Visual Basic .NET accepts a number of instantiation methods without performance impacts, although all variables must first be diminished and then instantiated before they can be used.

The two methods for instantiating classic COM are the `CreateObject` and `New` keywords. `CreateObject` uses the Windows Registry to obtain the interface of the class being instantiated. Because `CreateObject` depends on windows for access to the register, COM+ can apply a context for use by the COM+ services.

The `New` keyword in classic COM also depends on Windows for access to the Windows Registry. The catch is that it doesn't always have to access the Windows Registry to discover a class's interface if the class resides inside the same component as the calling class. Because the `New` keyword has no problem accessing a class's interface within the same component, a class can be instantiated by passing COM+ services that would normally add a context or other component service. While this will not prevent you from loading a class into COM+, to take full advantage of COM+ services you should use the `CreateObject` keyword.

Having said all that, the `CreateObject` keyword cannot be used to instantiate .NET classes, although it can be used to instantiate classes that exist within classic COM components. Because .NET components don't rely on the Windows Registry, the `New` keyword is used when loading all .NET components.

Here are several examples of how you might define and load classes into memory. First, the variable is diminished as `MyClass`:

```
.....  
Dim obj As MyClass  
.....
```

Second, you load the class "MyClass" into memory. An instance of a class loaded into memory is referred to as an object. Notice there is no "Set" keyword used.

```
.....  
Obj = New MyClass  
.....
```

Another method is to declare and instantiate an object in a single line:

```
.....  
Dim obj2 As MyClass = New MyClass()  
.....
```

Finally, you can implicitly diminish a variable with a class you are attempting to load. This is the shortest method and is perfectly acceptable:

```
.....  
Dim obj3 As New MyClass  
.....
```

Early and Late Binding

Binding is something we do when diminishing a variable, though many developers may not realize the importance of how they bind a class.

Early binding, often referred to as strong typing, refers to explicitly declaring the class used to define a variable. Early binding has several benefits. For example, when programming, Visual Studio .NET can give access to the class's interface with Intellisense which greatly reduces potential for typos and promotes rapid development. Also, when early binding a class, the Visual Basic compiler can enforce the proper use of a class's interface by providing warnings and

refusing to complete the compile until the error is resolved. But performance gains are probably the most important reason to bind early: Early binding allows your program to access your class's interface directly, rather than through the Windows Registry or at runtime. If the compiler knows ahead of time which classes you will be using in your application, it can make the appropriate compilation optimizations.

Late binding can be useful when developing against non-existent components or ones that are being developed. Late binding allows you to continue compiling your code until the component is available; once the class is available, you can modify your code to early bind. You might also use late binding when you truly don't know the object type that will be passed to your function, in which case it is perfectly acceptable to accept any type of object.

Before late binding can occur, the Option Strict option must be set to off. Option strict is off by default:

```
.....
Option Strict Off
.....
```

To declare a variable as late bound, simply diminish the variable as type object:

```
.....
Option Strict Off
Dim obj As Object
'or
Dim obj As System.Object
.....
```

The System.Object class is the class from which all other classes are derived. While it has no specific characteristics that prevent it from acting as any other class, it is used for late binding.

Components

A class defines something that can exist in memory. It defines an object's interface including properties, methods, events, and enumerations as well as implementation code. An object is an instance of a class in memory; while a class may exist only once, multiple instances of that class may reside in memory as objects.

When adding items to a project, you can add a "class" or a "component class". In essence, these are the same thing with one exception: a component class implements the IComponent interface, enabling Visual Studio .NET to drag non-visual controls, such as the timer control, onto a component designer.

Visual Studio .NET provides a designer for building components, which allows you to drag visual controls onto your class or component, and begin coding. For example, if you want to program a delay into your class, you can use a component item with the designer to drag the timer control onto your component. The implementation of a graphical designer (IE Component Designer) and container are available to you when you selected "component class" as a

new item. Visual Studio creates your class by adding a line of code that inherits everything your class needs to be a component, as follows:

```
.....
Inherits System.ComponentModel.Component
.....
```

Then the designer creates a components object using the IContainer interface so that the designer can allow drag and drop capabilities:

```
.....
Private components As System.ComponentModel.IContainer
.....
```

Simple OOP Examples

Now you'll build an example application that employs abstraction, encapsulation, polymorphism, and inheritance:

1. Create a new class library project named "PersonProj".
2. Rename the Class1.vb file to "Person.vb".
3. Add a new Windows Application project to the solution named "TestClient".
4. Right-click on the TestClient project and select Set as StartUp Project.

Adding Abstraction

Now add abstraction:

1. Replace the default class in Person.vb with the following code:

```
.....
Public Class clsPerson
    Public FName As String
    Public LName As String
    Public FullName As String
    Public BirthDate As Date
    Public Age As Integer
    Public TotalHours As Integer

    Public Sub Work(ByVal intHours As Integer)
        TotalHours += intHours
    End Sub
End Class
.....
```

2. Create a form that looks like Figure 7-2, and name it "frmAbstraction.vb." Use the parameters in Table 7-4 to build the new Windows Form.

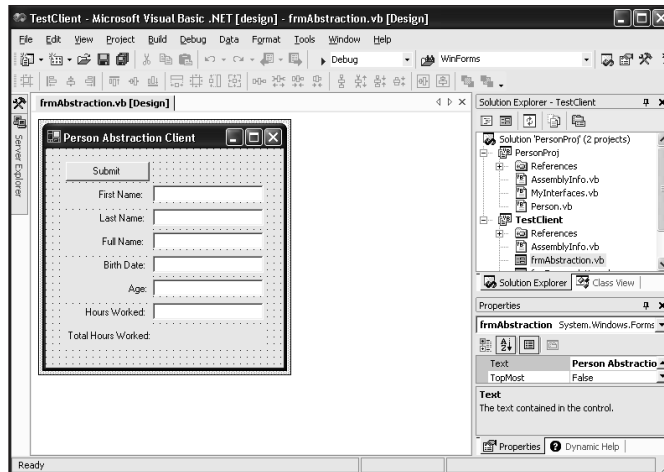


Figure 7-2: Creating a new Windows Form.

Table 7-4: Form Parameters

Control	Property	Value
Button	Name	btnSubmit
	Text	Submit
Textbox	Name	txtFName
	Name	txtLName
Textbox	Name	txtFullName
Textbox	Name	txtBirthDate
Textbox	Name	txtAge
Textbox	Name	txtHoursWorked
Label	Name	lblTotalHoursWorked

3. Right-click on the TestClient project and select Properties. Change the Startup object to “frmAbstraction.vb”.
4. Add a reference to the PersonProj. Right-click on the TestClient project and select Add Reference. Select the Projects tab and press the “Select” button then press OK.
5. Add the following object declaration to the initialization of frmAbstraction form:

```

.....
' Here we are initializing the Person class. Normally this would be done
' when the class was needed for data access but in this case we are using
' the Person class to maintain our data.
Dim objPerson As New PersonProj.clsPerson()
.....

```

6. Add the following code to the submit buttons click event:

```
.....
objPerson.FName = txtFName.Text
objPerson.LName = txtLName.Text
objPerson.FullName = txtFullName.Text

If IsDate(txtBirthDate.Text) Then
    objPerson.BirthDate = CDate(txtBirthDate.Text)
End If

If IsNumeric(txtAge.Text) Then
    objPerson.Age = CInt(txtAge.Text)
End If

If IsNumeric(txtHoursWorked.Text) Then
    objPerson.Work(CInt(txtHoursWorked.Text))
End If

lblTotalHoursWorked.Text = "Total Hours Worked: " _
& objPerson.TotalHours.ToString
.....
```

7. Now run the application.

You should be able to place any value you wish into the First and Last name fields, then completely contradict yourself when filling in your full name. The same should hold true for entering your birth date and age. This example abstracts a person but does not hide any implementation; each time you press the Submit button, the Total Hours Worked is summed and displayed.

Adding Encapsulation

The encapsulation example implements the `clsPerson` class and encapsulated code, hiding the implementation code for all the properties and methods.

In the abstraction example, the user has to enter both their birthday and age. As you encapsulate the implementation for the `Person` object, you will hide the implementation of their age. Age will be derived from the person's birth date, thus preventing a user from creating an invalid age and birth date values. Properties are also encapsulated, allowing the class to derive the full name from the first and last names that have been entered.

1. Add a new Windows Form item named "frmEncapsulation.vb".
2. Add the controls and parameters listed in Table 7-5 to the frmEncapsulation.vb form.

Table 7-5: Controls for the Form

Control	Property	Value
Button	Name	btnSubmit
	Text	Submit
Textbox	Name	txtFName
Textbox	Name	txtLName
Label	Name	lblFullNameDisplay
Textbox	Name	txtBirthDate
Label	Name	lblAgeDisplay
Textbox	Name	txtHoursWorked
Label	Name	lblTotalHoursWorked

3. Create a new class in the Person class using the following code:

```

.....
Public Class clsPerson2
    Private m_FName As String
    Private m_LName As String
    Private m_BirthDate As Date
    Private m_TotalHours As Integer

    Public Property FName() As String
        Get
            Return m_FName
        End Get
        Set(ByVal Value As String)
            m_FName = Value
        End Set
    End Property

    Public Property LName() As String
        Get
            Return m_LName
        End Get
        Set(ByVal Value As String)
            m_LName = Value
        End Set
    End Property

    Public ReadOnly Property FullName() As String
        Get
            Return m_FName & " " & m_LName
        End Get
    End Property

```

```

Public Property BirthDate() As Date
    Get
        Return m_BirthDate
    End Get
    Set(ByVal Value As Date)
        If IsDate(Value) Then
            m_BirthDate = Value
        End If
    End Set
End Property

Public ReadOnly Property Age() As Integer
    Get
        If DatePart(DateInterval.Year, m_BirthDate) = 1 Then
Exit Property
        End If
        Return DateDiff(DateInterval.Year, m_BirthDate, Now)
    End Get
End Property

'Method for adding hours to m_TotalHours worked.
Public Sub Work(ByVal intHours As Integer)
    m_TotalHours += intHours
End Sub

Public ReadOnly Property TotalHoursWorked() As Integer
    Get
        Return m_TotalHours
    End Get
End Property

End Class

```

-
4. Add the following object declaration to the initialization of frmAbstraction form:

```

' Here we are initializing the Person class. Normally this would be done
' when the class was needed for data access but in this case we are using
' the Person class to maintain our data.
Dim objPerson As New PersonProj.clsPerson2()

```

5. Right-click and select Properties then change the StartUp object to “frmEncapsulation”.
6. Add the following code to the Submit button:

```

.....
objPerson.FName = txtFName.Text
objPerson.LName = txtLName.Text

If IsDate(txtBirthDate.Text) Then
    objPerson.BirthDate = CDate(txtBirthDate.Text)
Else
    MsgBox("Please provide a valid Birth Date.")
End If

If IsNumeric(txtHoursWorked.Text) Then
    objPerson.Work(CInt(txtHoursWorked.Text))
    txtHoursWorked.Text = ""
End If

lblFullNameDisplay.Text = objPerson.FullName.ToString
lblAgeDisplay.Text = objPerson.Age.ToString

lblTotalHoursWorked.Text = "Total Hours Worked: " _
    & objPerson.TotalHoursWorked.ToString
.....

```

7. Now run the application and enter the information.

You’ll notice that your age is calculated for you so that it cannot contradict what the age should be based on the birth date, and the full name is derived from the first and last name.

Adding Polymorphism or Interface-based Inheritance

This example features an interface called `IPerson` and a class named `Employee` that uses the `IPerson` interface:

1. First create a new Windows Form with the same controls as used in the encapsulation example and name it “frmPolymorphism”.
2. Add a new Module to the `PersonProj` project and name it “MyInterfaces.vb”.

3. Apply the following code to the MyInterface.vb module. The code defines the interface.

```

.....
Public Interface IPerson
    Property FName() As String
    Property LName() As String
    ReadOnly Property FullName() As String
    Property BirthDate() As Date
    ReadOnly Property Age() As Integer

    'Method for adding hours to m_TotalHours worked.
    Sub Work(ByVal intHours As Integer)
        ReadOnly Property TotalHoursWorked() As Integer
    End Interface
.....

```

4. Right-click and select Properties, then change the StartUp object to “frmPolymorphism”.
5. Create a new class to the Person.vb module. You will use it to inherit the new interface:

```

.....
Public Class clsPerson3
    Implements IPerson
    Private m_FName As String
    Private m_LName As String
    Private m_BirthDate As Date
    Private m_TotalHours As Integer
    Private m_HrRate As Integer
    Private m_TotalPay As Double
    Public Property FName() As String _
Implements IPerson.FName
    Get
        Return m_FName
    End Get
    Set(ByVal Value As String)
        m_FName = Value
    End Set
End Property

    Public Property LName() As String _
Implements IPerson.LName
    Get
        Return m_LName
    End Get
    Set(ByVal Value As String)
        m_LName = Value
    End Set
.....

```

```
End Property

Public ReadOnly Property FullName() As String _
Implements IPerson.FullName
    Get
        Return m_FName & " " & m_LName
    End Get
End Property

Public Property BirthDate() As Date _
Implements IPerson.BirthDate
    Get
        Return m_BirthDate
    End Get
    Set(ByVal Value As Date)
        If IsDate(Value) Then
            m_BirthDate = Value
        End If
    End Set
End Property

Public ReadOnly Property Age() As Integer _
Implements IPerson.Age
    Get
        If DatePart(DateInterval.Year, _
m_BirthDate) = 1 Then Exit Property
        Return DateDiff(DateInterval.Year, _
m_BirthDate, Now)
    End Get
End Property

'Method for adding hours to m_TotalHours worked.
Public Sub Work(ByVal intHours As Integer) _
Implements IPerson.Work
    m_TotalHours += intHours
End Sub

Public ReadOnly Property TotalHoursWorked() As Integer _
Implements IPerson.TotalHoursWorked
    Get
        Return m_TotalHours
    End Get
End Property

'Additional Properties:
'HrRate
```

(continued on next page)

```

Public WriteOnly Property HrRate()
    Set(ByVal Value)
        m_HrRate = Value
    End Set
End Property

'TotalPay
Public ReadOnly Property TotalPay() As Double
    Get
        Return m_HrRate * m_TotalHours
    End Get
End Property

End Class

```

6. Now run this example just as you ran the previous ones.

You will not notice a difference in how the application runs, although the plumbing has changed quite a bit.

With this simple example, it is easy to question the usefulness of polymorphism. However, if you were to continue building an application that dealt with several aspects of a person, you might find polymorphism more helpful to use if you had to deal with Employees, Customers, Managers, and Contractors.

Working with Inheritance

This inheritance example inherits the interface and implementation code of the `clsPerson3` class:

1. Add a new Windows Form item named “frmInheritance.vb”. Use the same controls as we used in the encapsulation example.
2. Add the following code to the initialization section of the form:

```

' Here we are initializing the Person class. Normally this would be done
' when the class was needed for data access but in this case we are using
' the Person class to maintain our data.
Dim objPerson As New Person.clsEmployee()

```

3. Add the following class to the `Person.vb` module of the `PersonProj` project:

```

Public Class clsEmployee
    Inherits clsPerson3
End Class

```

4. Right-click and select Properties, then change the StartUp object to “frmInheritance”.
5. Now Run the inheritance example as you did the previous ones.

Notice that the `clsEmployee` class inherits the functionality of the `clsPerson3` class, which in turn implements the `IPerson` interface. This example demonstrates both polymorphism and inheritance that have been combined to form a single solution.

Summary

In this chapter, you learned that Visual Basic has come a long way from a reduced featured-set language that promoted RAPID application development to a full featured language. Now employing full inheritance, Visual Basic promises to aid in the delivery of enterprise level applications that may previously have been better delivered in another OOP language.

