This chapter goes beyond the fundamentals of Visual Basic programming and introduces a variety of features that make it easier for you to create powerful, flexible applications.

For example, you can load multiple projects into a single session of the programming environment, work with Windows registry settings, or selectively compile certain parts of your program.

Beyond the fundamentals of writing code, Visual Basic provides a variety of language elements that enhance your code. The last three topics in this chapter discuss three of these language elements: user-defined types, enumerated constants, and collections.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

Contents

- Working with Resource Files
- Working with Templates
- Creating Your Own Data Types
- Using Enumerations to Work with Sets of Constants
- Using Collections as an Alternative to Arrays

Working with Resource Files

A resource file allows you to collect all of the version-specific text and bitmaps for an application in one place. This can include constant declarations, icons, screen text, and other material that may change between localized versions or between revisions or specific configurations.

Adding Resources to a Project

You create a resource file using a text editor and resource compiler, such as those provided with Microsoft Visual C++. The compiled resource file will have a .res file name extension.

The actual file consists of a series of individual strings, bitmaps, or other items, each of which has a unique identifier. The identifier is either a Long or a String, depending on the type of data represented by the resource. Strings, for example, have a Long identifier, while bitmaps have a Long or String identifier. To retrieve resources in your code, learn the identifier for each resource. The function parameters referring to the resources can use the Variant data type.

For More Information For more information on resource files, see "Using Resource Files for Localization" in "International Issues."

3

Note Windows resource files are specific to 16-bit or 32-bit applications. Visual Basic will generate an error message if you try to add a 16-bit resource file to a project.

4

To add the resource file to your project, from the Project menu, choose the Add File command, just as you would when adding any other file to the project. A single project can have only one resource file; if you add a second file with a .res extension, an error occurs.

Using Resources in Code

Visual Basic provides three functions for retrieving data from the resource file for use in code.

Function	Description
LoadResString	Returns a text string.
LoadResPicture	Returns a Picture object, such as a bitmap, icon, or cursor.
LoadResData	Returns a Byte array. This is used for .wav files, for example.

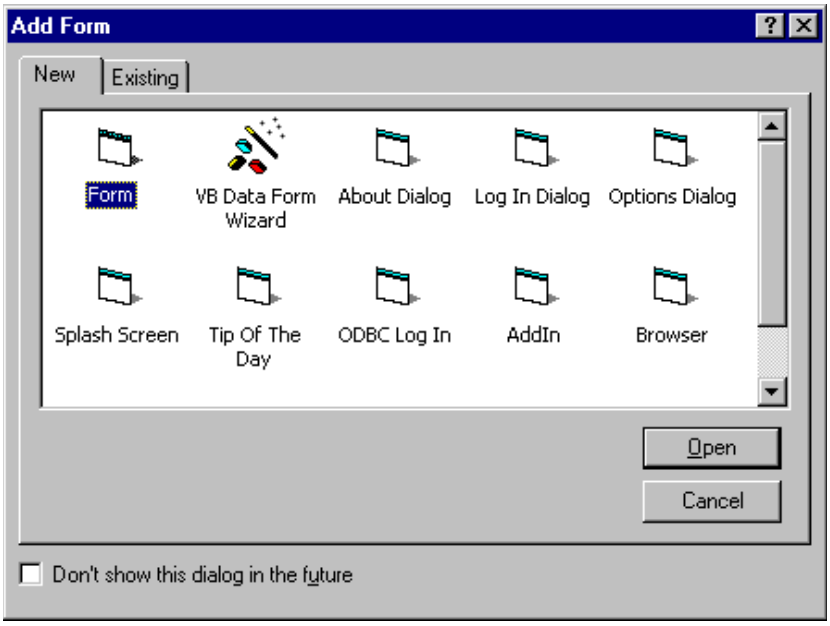
5

Working with Templates

Visual Basic provides a variety of templates for creating common application components. Rather than creating all the pieces of your application from scratch, you can customize an existing template. You can also reuse custom components in multiple applications by creating your own templates.

You can open an existing template by selecting its icon in the Add Object dialog box when you create a new form, module, control, property page, or document. For example, Visual Basic provides built-in form templates for creating an About dialog box, Options dialog box, or splash screen.

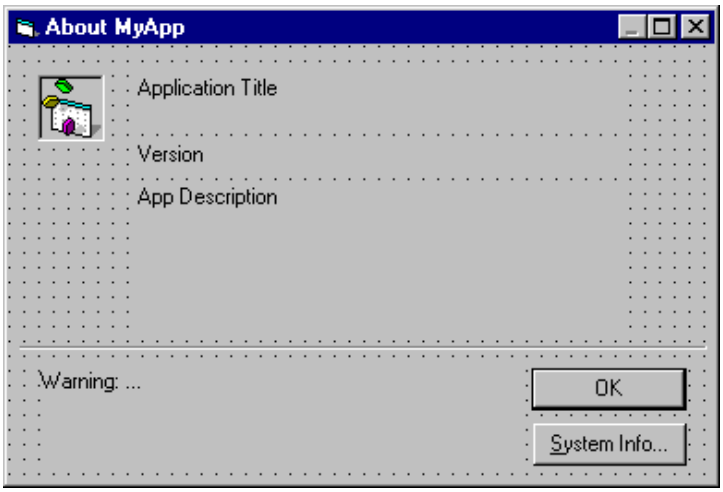
Figure 8.3 The Add Form dialog box



1

When you open a template, Visual Basic displays the object with placeholders that you can customize. For example, to create an About dialog box, open the About Dialog template and replace the Application Title, Version, and App Description placeholders with information specific to your application.

Figure 8.4 The About Dialog form template

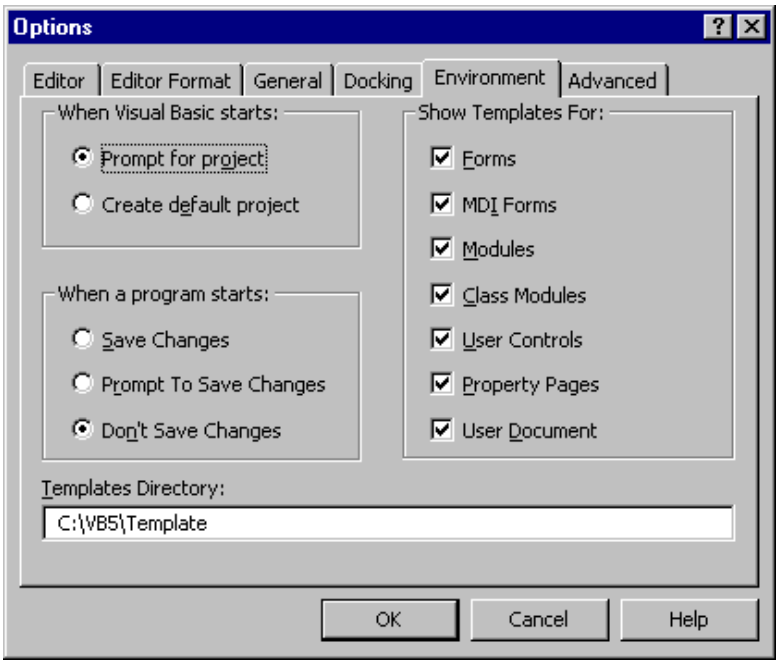


2

To create your own template, save the object that you want to use as a template, then copy it to the appropriate subdirectory of the Visual Basic Template directory. For example, to create a custom MyForm form template, save a form named MyForm, then copy the MyForm.frm file to the \VB\Template\Forms directory. When you select the Add Form command from the Project menu, Visual Basic displays the MyForm template in the Add Form dialog box, as shown in Figure 8.3.

You can disable display of templates in the Add object dialog box by selecting the Options command on the Tools menu and clearing the Show Templates options on the Environment tab of the Options dialog box. For example, to disable the display of form templates, clear the Forms option in the dialog box.

Figure 8.5 The Environment tab of the Options dialog box



3

Creating Your Own Data Types

You can combine variables of several different types to create user-defined types (known as *structs* in the C programming language). User-defined types are useful when you want to create a single variable that records several related pieces of information.

You create a user-defined type with the Type statement, which must be placed in the Declarations section of a module. User-defined types can be declared as Private or Public with the appropriate keyword. For example:

—4

```
Private Type MyDataType
    1– or –
Public Type MyDataType
```

6

For example, you could create a user-defined type that records information about a computer system:

```
' Declarations (of a standard module).
Private Type SystemInfo
    CPU As Variant
    Memory As Long
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

7

Declaring Variables of a User-Defined Type

You can declare local, private module-level, or public module-level variables of the same user-defined type:

```
Dim MySystem As SystemInfo, YourSystem As SystemInfo
```

8

The following table illustrates where, and with what scope, you can declare user-defined types and their variables.

Procedure/Module	You can *create* a user-defined type as...	*Variables* of a user-defined type can be declared...
Procedures	Not applicable	Local only
Standard modules	Private or public	Private or public
Form modules	Private only	Private only
Class modules	Private only	Private only

9

Assigning and Retrieving Values

Assigning and retrieving values from the elements of this variable is similar to setting and getting properties:

```
MySystem.CPU = "486"
If MySystem.PurchaseDate > #1/1/92# Then
```

10

You can also assign one variable to another if they are both of the same user-defined type. This assigns all the elements of one variable to the same elements in the other variable.

```
YourSystem = MySystem
```

11

User-Defined Types that Contain Arrays

A user-defined type can contain an ordinary (fixed-size) array. For example:

```
Type SystemInfo
    CPU As Variant
```

```
    Memory As Long
    DiskDrives(25) As String' Fixed-size array.
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

12

It can also contain a dynamic array.

```
Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives() As String          ' Dynamic array.
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

13

You can access the values in an array within a user-defined type in the same way that you access the property of an object.

```
Dim MySystem As SystemInfo
ReDim MySystem.DiskDrives(3)
MySystem.DiskDrives(0) = "1.44 MB"
```

14

You can also declare an array of user-defined types:

```
Dim AllSystems(100) As SystemInfo
```

15

Follow the same rules to access the components of this data structure.

```
AllSystems(5).CPU = "386SX"
AllSystems(X).DiskDrives(2) = "100M SCSI"
```

16

Passing User-Defined Types to Proecedures

You can pass procedure arguments using a user-defined type.

```
Sub FillSystem (SomeSystem As SystemInfo)
    SomeSystem.CPU = lstCPU.Text
    SomeSystem.Memory = txtMemory.Text
    SomeSystem.Cost = txtCost.Text
    SomeSystem.PurchaseDate = Now
End Sub
```

17

Note If you want to pass a user-defined type in a form or class module, the procedure must be private.

18

You can return user-defined types from functions, and you can pass a user-defined type variable to a procedure as one of the arguments. User-defined types are always passed by reference, so the procedure can modify the argument and return it to the calling procedure, as illustrated in the previous example.

For More Information To read more about passing by reference, see "Passing Arguments to Procedures" in "Programming Fundamentals."

19

User-Defined Types that Contain Objects

User-defined types can also contain objects.

```
Private Type AccountPack
    frmInput as Form
    dbPayRollAccount as Database
End Type
```

20

Tip Because the Variant data type can store many different types of data, a Variant array can be used in many situations where you might expect to use a user-defined type. A Variant array is actually more flexible than a user-defined type, because you can change the type of data you store in each element at any time, and you can make the array dynamic so that you can change its size as necessary. However, a Variant array always uses more memory than an equivalent user-defined type.

21

Nesting Data Structures

Nesting data structures can get as complex as you like. In fact, user-defined types can contain other user-defined types, as shown in the following example. To make your code more readable and easier to debug, try to keep all the code that defines user-defined data types in one module.

```
Type DriveInfo
    Type As String
    Size As Long
End Type

Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives(26) As DriveInfo
    Cost As Currency
    PurchaseDate As Variant
End Type

Dim AllSystems(100) As SystemInfo
AllSystems(1).DiskDrives(0).Type = "Floppy"
```

22

Using Enumerations to Work with Sets of Constants

Enumerations provide a convenient way to work with sets of related constants and to associate constant values with names. For example, you can declare an enumeration for a set of integer constants associated with the days of the week, then use the names of the days in code rather than their integer values.

You create an enumeration by declaring an enumeration type with the Enum statement in the Declarations section of a standard module or a public class module. Enumeration types can be declared as Private or Public with the appropriate keyword. For example:

```
Private Enum MyEnum
```

2– or –

```
Public Enum MyEnum
```

23

By default, the first constant in an enumeration is initialized to the value 0, and subsequent constants are initialized to the value of one more that the previous constant. For example the following enumeration, Days, contains a constant named Sunday with the value 0, a constant named Monday with the value 1, a constant named Tuesday with the value of 2, and so on.

```
Public Enum Days
    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
End Enum
```

24

Tip Visual Basic provides a built-in enumeration, vbDayOfWeek, containing constants for the days of the week. To view the enumeration's predefined constants, type vbDayOfWeek in the code window, followed by a period. Visual Basic automatically displays a list of the enumeration's constants.

25

You can explicitly assign values to constants in an enumeration by using an assignment statement. You can assign any long integer value, including negative numbers. For example you may want constants with values less than 0 to represent error conditions.

In the following enumeration, the constant Invalid is explicitly assigned the value –1, and the constant Sunday is assigned the value 0. Because it is the first constant in the enumeration, Saturday is also initialized to the value 0. Monday's value is 1 (one more than the value of Sunday), Tuesday's value is 2, and so on.

```
Public Enum WorkDays
    Saturday
    Sunday = 0
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Invalid = -1
End Enum
```

26

Note Visual Basic treats constant values in an enumeration as long integers. If you assign a floating-point value to a constant in an enumeration, Visual Basic rounds the value to the nearest long integer.

27

By organizing sets of related constants in enumerations, you can use the same constant names in different contexts. For example, you can use the same names for the weekday constants in the Days and WorkDays enumerations.

To avoid ambiguous references when you refer to an individual constant, qualify the constant name with its enumeration. The following code refers to the Saturday constants in the Days and WorkDays enumerations, displaying their different values in the Immediate window.

```
Debug.Print "Days.Saturday = " & Days.Saturday
Debug.Print "WorkDays.Saturday = " & WorkDays.Saturday
```

28

You can also use the value of a constant in one enumeration when you assign the value of a constant in a second enumeration. For example, the following declaration for the WorkDays enumeration is equivalent to the previous declaration.

```
Public Enum WorkDays
    Sunday = 0
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday = Days.Saturday - 6
    Invalid = -1
End Enum
```

29

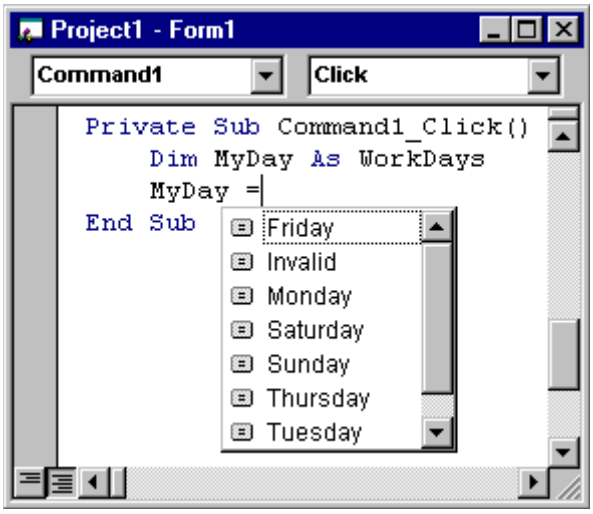
After you declare an enumeration type, you can declare a variable of that type, then use the variable to store the values of enumeration's constants. The following code uses a variable of the WorkDays type to store integer values associated with the constants in the WorkDays enumeration.

```
Dim MyDay As WorkDays
MyDay = Saturday            ' Saturday evaluates to 0.
If MyDay < Monday Then      ' Monday evaluates to 1,
                                ' so Visual Basic displays
                                ' a message box.
    MsgBox "It's the weekend. Invalid work day!"
End If
```

30

Note that when you type the second line of code in the example in the code window, Visual Basic automatically displays the WorkDays enumeration's constants in the Auto List Members list.

Figure 8.7 Visual Basic automatically displays an enumeration's constants



4

Because the constant Sunday also evaluates to 0, Visual Basic also displays the message box if you replace "Saturday" with "Sunday" in the second line of the example:

```
MyDay = Sunday              ' Sunday also evaluates to 0.
```

31

Note Although you normally assign only enumeration constant values to a variable declared as an enumeration type, you can assign any long integer value to the variable. Visual Basic will not generate an error if you assign a value to the variable that isn't associated with one of the enumeration's constants.

32

For More Information See "Providing Named Constants for Your Component" in "General Principals of Component Design."

33

Using Collections as an Alternative to Arrays

Although collections are most often used for working with objects, you can use a collection to work with any data type. In some circumstances, it may be more efficient to store items in a collection rather than an array.

You may want to use a collection if you're working with a small, dynamic set of items. The following code fragment shows how you might use a collection to save and display a list of URL addresses.

```
' Module-level collection.
```

```
Public colURLHistory As New Collection

' Code for adding a specified URL address
' to the collection.
Private Sub SaveURLHistory(URLAddress As String)
    colURLHistory.Add URLAddress
End Sub

' Code for displaying the list of URL addresses
' in the Immediate window.
Private Sub PrintURLHistory()
    Dim URLAddress As Variant
    For Each URLAddress in colURLHistory
        Debug.Print URLAddress
    Next URLAddress
End Sub
```

34

For More Information For more information on using collections, see "Programming With Your Own Objects" in "Programming with Objects." To learn more about using arrays, see "Arrays" in "Programming Fundamentals."

35