Introduction

Objectives

- To learn how to declare list and table arrays and access their elements.
- To learn how to use the ListBox control object for displaying arrays.

Data Structures

Up to now, the kinds of data you have been working with can be described for the most part as "single valued." Numerical values define unique real or integer numbers, and Boolean values are unique values as well. The exception is the String data type, although until now this Study Guide has treated it as if it were a unique value too, since it is not possible to change any of the characters in a string constant, just as it is not possible to change any of the digits in a numeric constant.

However the string data type is really what is referred to as a "structured" data type. The reason is that it is really not a single entity but rather a list of individual values each corresponding to one character in the string. When a programmer wishes to represent a list or a table of values as a single object, he or she can represent them as a structure called an "array."

The term "list" will be used in this study guide to refer to a data structure often called a "1-dimensional array," because it conveys the idea of a sequence of single items arranged in some order. For example, in every grocery list there is a 1st item, a 2nd item, and a so on. In Visual Basic, the items are single valued.

"Table" is the word that will be used in place of the term "2-dimensional array" because "table" more intuitively conveys the notion of single items arranged in rows and columns.
The implementation of arrays such as list and tables in Visual Basic and other programming languages imposes two important restrictions:

1. The data type of any item in a list or table can be Boolean, Integer, Double, or String, but all items in the same list or table must have the same data type.
2. The maximum size of the list or table must be known in advance.

**List Arrays**

To define a list array, we need to indicate how many values will be contained in the array and what the data type will be for each of those values. All values must be of the same data type. This information is captured in an "array declaration statement":

```vbnet
Dim array-name(upper bound) As data type
```

This statement is nearly the same as the simple declaration statement that was used to declare the data types of single-valued variables. In fact, the only difference is the provision of a number in parentheses, called the "upper bound" on the array, and it is often synonymous with the "number of values in the list" that the programmer wishes to represent. However, it is actually possible to store one more value in the list than suggested by the number because of the way in which the components of the list array are retrieved and stored.

Suppose you need to create a list array of 10 integer values. Then you can declare such an array as follows:

```vbnet
Dim numList(10) as Integer
```

To access any of the integers in this list array, you simply name the array and indicate, in parentheses, which of the 10 values you want. For example, numList(5) refers to the integer at position 5 of the list array, while numList(10) refers to the integer in the last position (position 10) of the list array. Suppose you want to add up the values in the odd positions of the list array. One way to do so is with the assignment statement:

```vbnet
sum = numList(1) + numList(3) + numList(5) + numList(7) + numList(9)
```

Since you are really repeating a pattern, namely, `numList(i)` for various values of `i`, this is an example of iteration, and so a better solution is to use a loop:

```vbnet
For i As Integer = 1 to 9 Step 2
    sum = sum + numList(i)
Next i
```

In the example above either solution could be used, but if we wanted to sum all the elements in the list, a much longer assignment statement would be needed, while the For-Next block remains essentially the same:
For i As Integer = 1 To 10 Step 1
    sum = sum + numList(i)
Next i

This simple example also hints at the fact that arrays often involve iteration, and so programs that use arrays are likely to use Do-While or For-Next blocks as well.

The number enclosed in parentheses that specifies a position in the list array is called a subscript because of the similarity of its role to subscripts in mathematical notation. It is also referred to as an index. So when you see the notation numList(4), for example, you should read this as numList sub 4, subscript 4 of numList, index 4 of numList, or position 4 of numList.

However it is misleading, and not strictly correct, to refer to "numList(4)" as the fourth (4th) value in numList. It is actually the fifth (5th) value! In Visual Basic every list array includes a position 0, and so numList(0) is a valid reference, even if you choose not to use it. Thus, position 4 of numList is really the fifth value of the list, preceded by numList(0), numList(1), numList(2), and numList(3).

A common question that often arises is whether you should begin storing numbers in an array beginning at position 0 or position 1? The answer is that you may do either: that is, you can choose to use position 0 or not, and usually the decision will depend on what is most convenient for you to do as a programmer.

**Implicit Arrays**

Instead of specifying explicitly the size of an array in its declaration statement, it is possible to define the size of the array implicitly by specifying the values that are to be stored in the array initially.

The general form of an implicit array declaration is:

```vbnet
Dim arrayname( ) As data type = {list of initial values}
```

For example, suppose that you wish to create an array called "assignments" representing a list of the marks obtained on a set of assignments. The initial values are "0" for all the assignments. This can be accomplished with the following declaration statement:

```vbnet
Dim assignment As Integer = {0, 0, 0, 0, 0}
```

This declaration is called an "implicit array declaration" and creates an array of five locations, with the integer value "0" assigned to each location.
Of course, this could also be done with an explicit array declaration statement:

```vbnet
Dim assignment(4) As Integer
```

Then the array would be initialized with five assignment statements:

```vbnet
assignment(0) = 0
assignment(1) = 0
assignment(2) = 0
assignment(3) = 0
assignment(4) = 0
```

These assignment statements will not be needed if the list is defined implicitly as shown. However, if at some point during execution the list needs to be reinitialized, then only assignment statements should be used. This is because the initialization of arrays using the implicit declaration statement occurs only once during each call to the subprogram where it is declared. So, if the array needs to be reinitialized while the subprogram in which it is declared is still active, then an implicit array declaration cannot be used to reinitialize it. Otherwise, the implicit array declaration is more compact.

Finally, you cannot declare an array explicitly and also initialize it using the syntax of the implicit array declaration. That is, the following is invalid:

```vbnet
Dim assignment(4) As Integer = {0, 0, 0, 0, 0}
```

### The ListBox Control

Until now the control object for output used in this Study Guide has been the `TextBox` control. While you can use `TextBoxes` to output the values of an array, there is another control object that is often more convenient for this purpose, the ListBox. If you have been examining the examples in the textbook, you will have observed that the author uses the ListBox for output rather than the `TextBox`. Either control object is acceptable for the examples presented so far in this Study Guide. However, the ListBox will prove more useful for displaying list arrays. Furthermore, the ListBox control object possesses several methods that allow the programmer to manage the list being displayed.

ListBoxes do not use the Text property to insert values. Instead, they have a special property called "Items." By assigning values to the Items property of a ListBox, the values can be displayed on the Form as a list. However, because the value of the Items is an array of elements that are objects, you use methods defined specifically for that object. Recall that methods are a type of subprogram associated specifically with objects for which they are defined. You should review Unit 5 if you are unsure of the concept.
Because the Items property of a ListBox is an array, you can retrieve any element using a subscript. So to retrieve the third element that is currently displayed in a ListBox called `ListBox1` you use the expression:

```csharp
ListBox1.Items(2)
```

Remember that the first element in any array has subscript 0.

Item objects, the elements of a ListBox, have a number of interesting methods that make the ListBox a useful and versatile way of displaying output, particularly lists of values. Some of these methods are:

- **Add** (<value>) adds the `<value>` to an existing Items array. In particular, the ListBox will display the value on a new line. Where the value is inserted in the existing list, however, depends upon another property of the ListBox object, called the "Sorted" property. This property can be assigned the Boolean value True or False. If the value of the Sorted property of the ListBox is False, then the value is placed at the end of the existing list. However, if the value is True, then the value is inserted so that the list remains in alphabetical order.

- **Remove** (<value>) removes the first occurrence of an element in the Items list that matches the `<value>` supplied as an argument.

- **RemoveAt** (<value>) removes the element from the Items list that is at the position given by the argument `<value>`.

- **Clear()** removes all items from the ListBox.

Items objects also possess the **Count** property that specifies the number of items currently in the Items list. ListBoxes themselves have additional properties that are useful in managing the information displayed:

- The **SelectedIndex** property of a ListBox displays the position of the element that is currently highlighted in the display of the list on the Form. If no element is highlighted, the value of SelectedIndex is -1. Also, setting the SelectedIndex property to -1 will "unhighlight" an element of a ListBox.

- The **Text** property of a ListBox is the value of the currently highlighted element.

**Table Arrays**

A list is only one way of arranging a set of items. Tables are also structures where there is a set of items organized in a series of rows and columns. The modelling of data organized in this way can be achieved by using what are called "2-dimensional arrays" or "tables."
To declare a list array it is necessary to specify the number of items in the list. With a
table array, it is necessary to specify the number of rows and the number of columns:

```vbnet
Dim table-name(number of rows, number of columns) As data type
```

As with lists, all members of the table must be of the same data type, and this data type
is specified as part of the declaration statement.

To specify a particular value in a table array, you must provide both the row position and
the column position. Furthermore, just as there is a position 0 for each list array, so too
there is a row 0 and a column 0 in each table array. If you like to number your rows and
columns from 1 to whatever the last row or column number is, then you won't use row or
column 0. Again the decision about what is better depends on the application.

There are many examples where table arrays provide a convenient representation. One
obvious example, from linear algebra, is the representation of a matrix. The entries in
the array will have a numeric data type, such as Integer or Double. Another example is
the representation of the board in a game such as tic-tac-toe, checkers, or chess. In this
case, the entries will be non-numeric since they will represent the playing pieces on the
board.

For example, in the game of tic-tac-toe it is necessary to track the status of each of nine
positions arranged in a 3 X 3 table. The status of each position within the table can be
"X", "O", or "blank." To check if there is a win for "X", it is necessary to see if any row,
column, or diagonal of the table contains all "X"s. The following program segment
illustrates how this can be achieved:

```vbnet
Dim board(3, 3) As String
Dim winForX As Boolean

winForX = False
' Check the rows:
For r As Integer = 1 To 3
    winForX = (board(r, 1) = "X" And board(r, 2) = "X" And board(r, 3) = "X") Or winForX
Next r
' Check the columns:
For c As Integer = 1 To 3
    winForX = (board(1, c) = "X" And board(2, c) = "X" And board(3, c) = "X") Or winForX
Next c
' Check the diagonals:
```
After executing this program segment, the truth value of "winForX" determines if "X" has won the game. Notice that this algorithm checks all eight possibilities regardless of whether a win is found among the first few tests. A more efficient implementation is provided in the full implementation of a tic-tac-toe game provided shortly in Program Implementation section of this unit.

Control Arrays

Not only can numbers and strings be stored in lists or tables, but any type of object can be stored as an element in an array. In particular, since TextBoxes, Buttons, and Labels are objects, lists or tables of these objects can be defined. Since such objects were previously defined to be control objects, the corresponding lists or tables are called "control arrays."

EXAMPLE: The following Form shows the user interface of a tic-tac-toe game as it would appear during execution (Note that a win has been detected for "X"):

Each button has three possible labels: "X", "O", or "blank." Initially all buttons are labelled with a "blank". Players alternate turns by "clicking" one of the buttons that are labelled with a "blank." When a "blank" button is selected, it is re-labelled with the symbol originally chosen by the player whose turn it is to play. Buttons with the label of either player's symbol cannot be clicked.

To implement the game board, nine button control objects are created and placed on a form. At the time each button is selected, a new button object is created and named Button1, Button2, through Button9. Since the label on the button is defined by the value of the "text" property of the button, different labels can be assigned during execution by changing the Text property of the appropriate button. Since the size and font used are also properties, they can also be assigned during execution.
To facilitate the management of button labels, a control array is used. Each button is assigned as the value of one location of an array "btn." Initially all the labels should be "blank." The following program segment achieves this:

' Create a Control Array of Buttons:
Dim btn(9) As Button 'The data type of each element is "Button"

' Define the font of each button label to be 26 point, Sans Serif:
Dim btnFont As New Font("Microsoft Sans Serif", 26, FontStyle.Regular)

' Initialize the button array, each element a different Button:
btn(1) = Button1
btn(2) = Button2
btn(3) = Button3
btn(4) = Button4
btn(5) = Button5
btn(6) = Button6
btn(7) = Button7
btn(8) = Button8
btn(9) = Button9

' Set the Text, Enabled, and Font properties of each Button:
For i As Integer = 1 To 9
    btn(i).Text = "" ' Set the Text property of button i to blank
    btn(i).Enabled = True 'Ensure the click method of Button i is called
    btn(i).Font = btnFont ' define the font for the label on the button
Next i

An Example Using Arrays

The best way to understand how arrays can used is to observe their application in programs. The textbook provides a number of such examples. The following extended example incorporates many of the features of Visual Basic that have been introduced in this Study Guide. You should spend some time examining the source code to be sure you understand each line’s purpose. In particular, note the use of arrays to model the application. The analysis following the example should give you further insights.
**Problem Description**

Create a tic-tac-toe board in an interactive window that permits two players to play "X's" and "O's." The application represents a board using 9 Buttons arranged in a 3 x 3 table. The application provides each player with the opportunity to define a symbol to represent their pieces on the board and keeps track of whose turn is next. When that person clicks on a vacant square, the appropriate piece is displayed on the board. When one player has three pieces in a row or column or on the diagonal, the window displays a message indicating who the winner is.

**Requirements Analysis**

**INPUT REQUIREMENTS**

<table>
<thead>
<tr>
<th>INPUT</th>
<th>TYPE</th>
<th>PURPOSE</th>
<th>REPRESENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player 1 symbol</td>
<td>String</td>
<td>symbol used by p1</td>
<td>TextBox (1 char)</td>
</tr>
<tr>
<td>Player 2 symbol</td>
<td>String</td>
<td>symbol used by p2</td>
<td>TextBox (1 char)</td>
</tr>
</tbody>
</table>

**OUTPUT REQUIREMENTS**

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>TYPE</th>
<th>PURPOSE</th>
<th>REPRESENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message console</td>
<td>String</td>
<td>display messages</td>
<td>TextBox (multi-line)</td>
</tr>
</tbody>
</table>

**EVENT REQUIREMENTS**

<table>
<thead>
<tr>
<th>EVENT</th>
<th>EFFECT</th>
<th>REPRESENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Game</td>
<td>Initialize board</td>
<td>Button</td>
</tr>
<tr>
<td>Quit</td>
<td>Stop the program</td>
<td>Button</td>
</tr>
<tr>
<td>Button1, Button2, ..., Button9</td>
<td>Board positions</td>
<td>Buttons (nine)</td>
</tr>
</tbody>
</table>
From these specifications the following user interface is defined:

![User Interface Diagram]

**Program Implementation**

The source code for the program consists of three major sections:

1. Class variables to be shared among the procedure and methods that are required for the implementation.
2. User defined procedures and functions that perform the tasks of managing the state of the game board.
3. Methods for the events associated with pushing each of the buttons defined on the Form.

*Class Variables:*

```vbnet
Public Class Form1

'CLASS VARIABLE DEFINITIONS
Dim player As Integer
'player = 0 (for Player 1) or 1 (for Player 2)
Dim played(3, 3) As Boolean
'played(r,c) = True if used, False if not yet played
Dim symbol(2) As String
'symbol(0) is player 1’s symbol, symbol(1), player 2’s symbol
Dim btn(9) As Button
Dim board(3, 3) As String
'The nine board “buttons”
'Keeps track of which pieces are on the board.
```
User-defined Procedures:

' USER DEFINED PROCEDURES AND FUNCTIONS:

'PLAY: Procedure to process the selection of a board position by either
'player, to update the board information, and to test for a win
'by either player, or a draw:
Sub Play(ByVal button As Button, _
    ByVal row As Integer, _
    ByVal col As Integer)
    If Not played(row, col) Then
        TextBox3.Text = ""
        button.Text = symbol(player)
        board(row, col) = symbol(player)
        played(row, col) = True
        If WinFor(symbol(player)) Then
            TextBox3.Text = symbol(player) & " wins!!!"
            DisableButtons()
        ElseIf DrawGame() Then
            TextBox3.Text = "Game is a draw"
            DisableButtons()
        End If
        player = (player + 1) Mod 2
    Else
        TextBox3.Text = "Position already occupied."
    End If
End Sub

'WINFOR: Check 8 possible ways to win:
Function WinFor(ByVal player As String) As Boolean
    'Check for row win
    For i As Integer = 1 To 3
        If board(i, 1) = player And board(i, 2) = player And board(i, 3) = player Then
            Return True
        End If
    Next
    'Check for column win
    For c As Integer = 1 To 3
        If board(1, c) = player And board(2, c) = player And board(3, c) = player Then
            Return True
        End If
    Next
    'Check for diagonal win
    If board(1, 1) = player And board(2, 2) = player And board(3, 3) = player Then
        Return True
    Else
        Return (board(3, 3) = player And board(2, 2) = player)
    End If
End Function
'DRAWGAME: Check Board for free spaces (none means a draw):
Function DrawGame() As Boolean
    For row As Integer = 1 To 3
        For col As Integer = 1 To 3
            If Not played(row, col) Then
                Return False
            End If
        Next
    Next
Return True
End Function

Sub DisableButtons()
    For i As Integer = 1 To 9
        btn(i).Enabled = False
    Next
    ' Enable players to change their symbols for another game.
    TextBox1.ReadOnly = False
    TextBox2.ReadOnly = False
End Sub

' INITIALIZEGAME: Procedure to initialize the board buttons and the arrays
' "played" and "board":
Sub InitializeGame()
    Dim btnFont As New Font("Microsoft Sans Serif", 26, FontStyle.Regular)
    For i As Integer = 1 To 9
        btn(i).Text = ""
        btn(i).Enabled = True
        btn(i).Font = btnFont
    Next i
    For row As Integer = 1 To 3
        For col As Integer = 1 To 3
            played(row, col) = False
            board(row, col) = ""
        Next col
    Next row
End Sub

' GETPLAYERSYMBOLS:
' Procedure to obtain the playing symbols to be used by each player:
Sub GetPlayersSymbols()
    TextBox3.Text = ""
    If TextBox1.Text = "" Then
        TextBox3.Text = "Enter Player 1 symbol (one character)."
    ElseIf TextBox2.Text = "" Then
        TextBox3.Text = "Enter Player 2 symbol (one character)."
    Else
        symbol(0) = TextBox1.Text
        TextBox1.ReadOnly = True
        symbol(1) = TextBox2.Text
        TextBox2.ReadOnly = True
    End If
End Sub
Methods to Manage Button Events:

' METHODS TO HANDLE ALL EVENTS:
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Play(Button1, 1, 1)
End Sub

Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    Play(Button2, 1, 2)
End Sub

Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    Play(Button3, 1, 3)
End Sub

Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
    Play(Button4, 2, 1)
End Sub

Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
    Play(Button5, 2, 2)
End Sub

Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
    Play(Button6, 2, 3)
End Sub

Private Sub Button7_Click(sender As Object, e As EventArgs) Handles Button7.Click
    Play(Button7, 3, 1)
End Sub

Private Sub Button8_Click(sender As Object, e As EventArgs) Handles Button8.Click
    Play(Button8, 3, 2)
End Sub

' Button to terminate execution of the application:
Private Sub Button9_Click(sender As Object, e As EventArgs) Handles Button9.Click
    Me.Close()
End Sub

' Button to start a new game:
Private Sub Button10_Click(sender As Object, e As EventArgs) Handles Button10.Click
    player = 0
    InitializeGame()
    GetPlayerSymbols()
End Sub

' ON START UP: Button1 to Button9 control objects are stored in the list "btn"
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    btn(1) = Button1
    btn(2) = Button2
    btn(3) = Button3
    btn(4) = Button4
    btn(5) = Button5
    btn(6) = Button6
    btn(7) = Button7
    btn(8) = Button8
    btn(9) = Button9
    TextBox1.ReadOnly = False
    TextBox1.MaxLength = 1
    TextBox2.ReadOnly = False
    TextBox2.MaxLength = 1
    TextBox3.Text = "Enter Player symbols"
    InitializeGame()
End Sub

End Class
Analysis of the Program

Although this is a much larger example than you have been asked to review previously in this Study Guide, all of the Visual Basic syntax should be familiar to you. However, it is important that you take note of the following major ideas that have been incorporated into this implementation:

*Objects can be assigned as "values."*

In this example, Button objects are used as values in the following ways:

1. The data type of the array called `btn` is declared to be `Button`. `Button1` through `Button9` are the names of nine instances of the Button. Thus "btn(i)" represents Button i, for i having the values from 1 to 9. By storing the Button objects in an array, a For-Next loop can be used whenever it is necessary to access all the Button objects that make up the playing area. Such a need occurs in two cases:

   To initialize the game board in the subprogram `InitGame`, it is necessary to reset three of the properties of each of the nine buttons. Without a loop, doing so would require 27 assignment statements.

   The nine buttons that define the game board are disabled when a win occurs so that no further changes can be made. This is implemented in a loop that occurs in the subprogram `DisableButtons`.

2. Buttons are passed as an argument value in a subprogram call to `Play`. This subprogram is the primary procedure for processing the event of pushing a board Button. No matter which button is pushed, the same tasks need to be performed. Rather than writing nine Sub procedures, one for each of the nine game buttons, a single procedure, `Play`, was written. When a button is pushed, the corresponding Button object is passed to the procedure `Play`. Notice how each of the `Button_Click` event procedures (all nine of them) calls the Sub procedure `Play`, conveying to that procedure through the argument list which Button object was clicked and where that Button is on the board by specifying in which row and which column it is located.

*Control Objects can be Enabled or Disabled.*

In this example, extensive use is made of properties that control whether the user can initiate an event on some of the control objects on the Form:

1. `TextBox` objects can be restricted from data entry by setting the property "ReadOnly" to True. This setting is used in the subprogram `GetPlayerSymbols` to prevent the users from changing their playing symbols once they are entered. At the end of the game, this property is set to False so that the symbols can be changed at the start of a new game.
2. TextBox objects can have the length of the string that can be entered, restricted to a fixed length by setting the property MaxLength. In the Form1_Load method, the length of symbol that players can choose to represent their playing symbols is restricted to 1 character by setting the MaxLength property of the corresponding TextBoxes to 1. Note that each player's symbol is stored in a list array called "symbol."

3. Button objects can be disabled from calling their corresponding Button_Click methods by setting the "Enabled" property of each Button to False, as is done in the Sub procedure DisableButtons. When a new game is initiated and the Button objects need to be enabled, this property must be set to True for each button, which is handled using the For-Next loop in the sub procedure InitializeGame.

**Fonts are objects too.**

If you have successfully entered and run this program you will note that the users' playing symbols are displayed on the buttons when they are clicked. Furthermore, they are displayed in a much larger font size. This effect is achieved in the following way:

1. By changing the Text property of the Button object clicked so that it displays the appropriate player's symbol. By knowing which player clicked the button (given by the value of "player"), the corresponding player's symbol is retrieved from the list "symbol" and assigned to the Text property of the appropriate Button object by the assignment statement button.Text = symbol(player) in the procedure "Play." Note that the variable button is assigned the value of Button1, Button2, ..., or Button9 by argument provided in the call to Play.

2. The 26 pt, sans-serif font used to display the players' characters on the buttons is achieved by redefining the font to be used for displaying the Text property of the Button on the Form. To redefine the property, a new Font object is created whose font type, size, and style, the components of a "font object," have the required values. This is provided in the procedure "InitializeGame" by the declaration:

   ```vbnet
   Dim btnFont As New Font("Microsoft Sans Serif", 26, Fontstyle.Regular)
   ```

   If you haven't yet done so, take the time to enter and run this program and observe the various changes that take place as a result of different events. Then try to determine by inspecting the source code, how each effect is achieved. When you feel you understand how the program behaves the way it does and you feel confident you can duplicate these behaviours in programs you write, you will have made significant progress in the development of your skills as a Visual Basic programmer.