Overview of Visual Basic for Applications (Supplement – especially if you didn't buy book)

I. Variables, Constants, and Data Types

In Visual Basic for Applications, as in all high-level programming languages, you use variables and constants to store values. Variables can contain data represented by any supported data type.

VBA Data Types

The following table lists the fundamental data types that VBA supports.

Data type	Description	Range
Byte	1-byte binary data	0 to 255
Integer	2-byte integer	– 32,768 to 32,767
Long	4-byte integer	– 2,147,483,648 to 2,147,483,647
Single	4-byte floating-point number	(negative values). 1.401298E – 45 to 3.402823E38 (positive values)
Double	8-byte floating-point number	– 1.79769313486231E308 to– 4.94065645841247E – 324 (negative values). 4.94065645841247E – 324 to 1.79769313486231E308 (positive values)
Currency	8-byte number with a fixed decimal point	– 922,337,203,685,477.5808 to 922,337,203,685,477.5807
String	String of characters	Zero to approximately two billion characters
Variant	Date/time, floating-point number, integer, string, or object. 16 bytes, plus 1 byte for each character if the value is a string value.	Date values: January 1, 100 to December 31, 9999. Numeric values: same range as Double. String values: same range as String. Can also contain Error or Null values.
Boolean	2 bytes	True or False
Date	8-byte date/time value	January 1, 100 to December 31, 9999
Object	4 bytes	Any object reference

Declaring a Constant, Variable, or Array

You declare a constant for use in place of a literal value by using the Const statement. You can specify private or public scope, specify a data type, and assign a value to the constant, as shown in the following declarations.

```
Const MyVar = 459
Public Const MyString = "HELP"
Private Const MyInt As Integer = 5
Const MyStr = "Hello", MyDouble As Double = 3.4567
```

If you don't specify scope, the constant has private scope by default. If you don't explicitly specify a data type when you declare a constant, Visual Basic gives the constant the data type that best matches the expression assigned to the constant. For more information, see "Const Statement," "Public Statement," "Private Statement," and "As" in Help.

You declare a variable by using the Dim, Private, Public, or Static keyword. Use the As keyword to explicitly specify a data type for the variable, as shown in the following declarations.

```
Private I
Dim Amt
Static YourName As String
Public BillsPaid As Currency
Private YourName As String, BillsPaid As Currency
Private Test, Amount, J As Integer
```

If you don't declare a variable as static, when a procedure that contains it ends, the variable's value isn't preserved and the memory that the variable used is reclaimed. If you don't explicitly declare a data type, VBA gives the variable the Variant data type by default.

Note: Not all variables in the same declaration statement have the same specified type. For example, the variables Test and Amount in the last line in the preceding example are of the Variant data type.

The steps you take to declare an array are very similar to the steps you take to declare a variable. You use the Private, Public, Dim, and Static keywords to declare the array, you use integer values to specify the upper and lower bounds for each dimension, and you use the As keyword to specify the data type for the array elements. You must explicitly declare an array before you can use it; you cannot implicitly declare an array.

When you declare an array, you specify the upper and lower bounds for each dimension within the parentheses following the array name. If you specify only one value for a dimension, VBA interprets the value as the upper bound and supplies a default lower bound. The default lower bound is 0 (zero) unless you set it to 1 by using the Option Base statement. The following declarations declare one-dimensional arrays containing 15 and 21 elements, respectively.

```
Dim counters(14) As Integer
Dim sums(20) As Double
```

You can also specify the lower bound of a dimension explicitly. To do this, separate the lower and upper bounds with the To keyword, as in the following declarations.

```
Dim counters(1 To 15) As Integer
Dim sums(100 To 120) As String
```

In the preceding declarations, the index numbers of counters range from 1 to 15, and the index numbers of sums range from 100 to 120.

Tip: You can use the LBound and UBound functions to determine the existing lower and upper bounds of an array.

You can declare arrays of up to 60 dimensions. The following declaration creates an array with three dimensions, whose sizes are 4, 10, and 15. The total number of elements is the product of these three dimensions, or 600.

```
Dim multiD(4, 1 To 10, 1 To 15)
```

Tip: When you start adding dimensions to an array, the total amount of storage needed by the array increases dramatically, so use multidimensional arrays with care. Be especially careful with Variant arrays, because they're larger than arrays of other data types.

You declare a dynamic array just as you would declare a fixed-size array, but without specifying dimension sizes within the parentheses following the array name, as in the following declaration.

```
Dim dynArray() As Integer
```

Somewhere in a procedure, allocate the actual number of elements with a ReDim statement, as in the following example.

```
ReDim DynArray(X + 1)
```

Use the Preserve keyword to change the size of an array without losing the data in it. You can enlarge an array by one element without losing the values of the existing elements, as in the following example.

```
ReDim Preserve myArray(UBound(myArray) + 1)
```

II. Sub Procedures vs. Function Procedures

With Visual Basic, you can create two types of procedures: Sub procedures and Function procedures.

A Sub procedure is a unit of code enclosed between the Sub and End Sub statements that performs a task but doesn't return a value. The following example is a Sub procedure.

```
 Sub DisplayWelcome()
     MsgBox "Welcome"
 End Sub
```

A Function procedure is a unit of code enclosed between the Function and End Function statements. Like a Sub procedure, a Function procedure performs a specific task. Unlike a Sub procedure, however, a Function procedure also returns a value. The following example is a Function procedure.

```
Function AddThree(OriginalValue As Long)
     AddThree = OriginalValue + 3
End Function
```

Public Procedures vs. Private Procedures

You can call a public procedure, declared with the Public keyword, from any procedure in any module in your application. You can call a private procedure, declared with the Private keyword, only from other procedures in the same module. Both Sub procedures and Function procedures can be either public or private. The following are examples of private procedures.

```
 Private Sub Test1()
     MsgBox "This is the Test1 procedure running"
 End Sub

 Private Function AddThree(OriginalValue As Long)
     AddThree = OriginalValue + 3
 End Function
```

The following are examples of public procedures.

```
Public Sub Test1()
     MsgBox "This is the Test1 procedure running"
End Sub

Public Function AddThree(OriginalValue As Long)
     AddThree = OriginalValue + 3
End Function
```

If you don't use either the Public or Private keyword to declare a procedure, the procedure will be public by default. Therefore, the following are also examples of public procedures.

```
Sub Test1()
     MsgBox "This is the Test1 procedure running"
End Sub

Function AddThree(OriginalValue As Long)
    AddThree = OriginalValue + 3
End Function
```

Although it's not necessary to use the Public keyword when creating a public procedure, including it in procedure declarations makes it easier to see at a glance which procedures are public and which are private. For more information, see "Public" or "Private" in Help.

Using the Value Returned from a Function

For a function to return a value, it must include a function assignment statement that assigns a value to the name of the function. In the following example, the value assigned to ConeSurface will be the value returned by the function.

```
Function ConeSurface(radius, height)
    Const Pi = 3.14159
    coneBase = Pi * radius ^ 2
    coneCirc = 2 * Pi * radius
    coneSide = Sqr(radius ^ 2 + height ^ 2) * coneCirc / 2
    ConeSurface = coneBase + coneSide
End Function
```

The information that must be supplied to a Sub procedure or Function procedure for it to perform its task (radius and height in the preceding example) is passed in the form of arguments. For more information about arguments, see "Passing Arguments to a Procedure" later in this chapter.

When the Function procedure returns a value, this value can then become part of a larger expression. For example, the following statement in another procedure incorporates the return value of the ConeSurface and ScoopSurface functions in its calculations.

```
totalSurface = ConeSurface(3, 11) + 2 * ScoopSurface(3)
```

Some Notes On The Range Object & Cells Collection

One of the most fundamental yet difficult concepts to master in VBA programming in Excel is the Range object. A Range object is simply a set of one or more cells on a worksheet. However, there are a variety of ways to access Range objects, and the best technique depends a great deal on the circumstances and what you are trying to do. To complicate matters, Excel interprets references to Range objects differently depending on where your code is located. For instance, consider the following statement:

```
Range(“A3”).Value = 7
```

This statement is ‘unqualified’ in the sense that we have not specified what worksheet contains the cell we are referring to (A3). If the above line appears in the code module for a worksheet (say, Sheet1) then it would change the value of cell A3 on Sheet1 to 7 – even if another sheet besides Sheet1 is selected or active. On the other hand, if the above line appeared in a ‘generic’ code module (that is, a module not associated with a particular worksheet) then it would change the value of cell A3 on the active sheet to 7.

The Cells collection can also be used to access and manipulate cells (or ranges of cells) on a worksheet. To use the Cells collection of a worksheet object we must specify the row number and column ***number*** (not letter) of a cell. For instance,

```
Cells(3,1).Value = 7
```

changes the value of the cell in row 3 and column 1 (aka cell “A3”) to 7. It is often more convenient to use the Cells collection when writing code that loops across columns of a worksheet (or range) as it is easier to loop through a series of numbers than a series of letters. Here again, the interpretation of ‘unqualified’ Cells statements is dependent on where the code is located (i.e., a ‘generic’ code module vs. a worksheet code module).

With the Range object and Cells collection it is also possible to refer to several cells (or a range of cells) in a single statement. For example, suppose a worksheet named Sample contains data in the range A5:D9. We might assign a name to this range as follows:

```
Worksheets(“Sample”).Range(“A5:D9”).Name = “Data”
```

or, equivalently,

```
With Worksheets(“Sample”)
    .Range(.Cells(5,1),.Cells(9,4)).Name = “Data”
End With
```

Alternatively, we might place the Rand() function in each cell in this range as follows:

```
Worksheets(“Sample”).Range(“A5:D9”).Formula = “=Rand()”
```

or, equivalently,

```
With Worksheets(“Sample”)
    .Range(.Cells(5,1),.Cells(9,4)). Formula = “=Rand()”
End With
```

In the above examples, we used the Cells collection (of the “Sample” worksheet object) within a Range object to specify the upper lefthand cell and lower righthand cell of the area of interest (separated by a comma).

Like a worksheet object, a Range object also consists of a collection of cells. As a result, there is also a Cells collection associated with any Range object. For instance, the statement

```
Worksheets(“Sample”).Range(“A5:D9”).Cells(1,1).Value = 7
```

changes the value in cell A5 to 7. Why? Because cell A5 is in row 1 and column 1 of the range “A5:D9”. Similarly,

```
Worksheets(“Sample”).Range(“C5:F9”).Cells(5,3).Value = 11
```

changes the value in cell E9 to 11. Why? Because cell E9 is in row 5 and column 3 of the range “C5:F9”. Thus, when using the Cells collection it is important to keep in mind whether you are referring to the collection of cells in a worksheet object or a range object.

Range objects have a number of other properties and methods that can be quite useful. The following examples illustrate a number of these.

```
Set MyRange = Worksheets(1).Range(“C10:F25”)

‘ Returns the number of columns in MyRange (or 4)
c = MyRange.Columns.Count

‘ Returns number of rows in MyRange (or 16)
r=MyRange.Rows.Count

‘ Returns physical number of first column in MyRange (or 3 - for column C)
fc=MyRange.Columns(1).Column

‘ Returns physical number of last column in MyRange (or 6 - for column F)
lc=MyRange.Columns(MyRange.Columns.Count).Column

‘ Returns physical number of first row in MyRange (or 10)
fr=MyRange.Rows(1).Row

‘ Returns physical number of last row in MyRange (or 25)
lr=MyRange.Rows(MyRange.Rows.Count).Row

‘ Computes sum of all items in MyRange
s=WorksheetFunction.Sum(MyRange)

‘Computes standard deviation of all items in MyRange
sd=WorksheetFunction.StDev(MyRange)

‘Computes sum of 3rd column in MyRange
sc3= WorksheetFunction.Sum(MyRange.Columns(3))

‘Computes sum of 5th row in MyRange
sr5= WorksheetFunction.Sum(MyRange.Rows(5))

‘ Enters a formula to sum 4th column in MyRange at bottom of  the column
MyRange.Parent.Cells(lr + 1, fc + 3).Formula = "=Sum(r[-" & r & "]c:r[-1]c)"

or

MyRange.Cells(1, 1).Offset(r, 3).Formula = "=Sum(r[-" & r & "]c:r[-1]c)"

or

MyRange.Cells(r + 1, 4).Formula = "=Sum(r[-" & r & "]c:r[-1]c)"
```

```
‘ Set an object pointing to the region (block) of cells containing  D15
Set y = Workbooks(1).Worksheets(1).Cells(15,4).CurrentRegion

‘ Returns address of the cell in the last row and last column of MyRange (or $F$25)
‘ Note that this does not change the selected cell
addr = MyRange.End(xlToRight).End(xlDown).Address

‘ Selects the cell in the last row and column of MyRange
‘ Assuming worksheet containing MyRange is active or selected first
MyRange.End(xlToRight).End(xlDown).Select

' The End method accepts the arguments xlToRight, xlToLeft, xlUp and xlDown
```

Excel objects and some VBA code

```
Sub CheckOutApplication()
    ThisCaption = Application.Caption
    MsgBox ThisCaption
    Application.Caption = "This is Excel?"
    Application.Caption = Empty     'sets it to the default value
    MsgBox "The current path is " & Application.Path
    Application.WindowState = xlMaximized
    Application.WindowState = xlMinimized
    Application.WindowState = xlNormal
End Sub

Sub CheckOutWorkbooks()
    MsgBox Application.Workbooks(1).Name
    If Not (ActiveWorkbook.Saved) Then
      ActiveWorkbook.Save
    End If
    MsgBox "The path to this file is " & _
      Application.Workbooks("Some Easy VBA.xls").Path
    MsgBox "This Workbook is saved: " & Application.Workbooks(1).Saved _
      & " " & ActiveWorkbook.Saved
    ActiveWorkbook.Save
    'Application.Workbooks(1).SaveAs FileName:="Still Easy.xls"
    Application.Workbooks("Some Easy VBA.xls").Close
End Sub

Sub CheckOutWorksheets()
    Application.ActiveWorkbook.Worksheets(2).Select
    Application.ActiveWorkbook.Worksheets(2).Protect password:="wilck", contents:=True
    Application.ActiveWorkbook.Worksheets(2).Unprotect
    Application.ActiveWorkbook.Worksheets(2).Name = "Wilck"
    ActiveSheet.Name = "No, really..."
    Application.ActiveWorkbook.Worksheets(3).Name = "Sheet99"
    Application.ActiveWorkbook.Worksheets("Sheet99").Name = "Bed Sheet"
    ActiveSheet.Visible = False
    ActiveSheet.Visible = True
End Sub

Sub CheckOutRange()
    Application.ActiveWorkbook.Worksheets(1).Range("A1").Value = 1
    Application.ActiveWorkbook.Worksheets(1).Range("A2").Value = 2
    Application.ActiveWorkbook.Worksheets(1).Range("A3").Value = 3
    Application.ActiveWorkbook.Worksheets(1).Range("A4").Value = 4
    Application.ActiveWorkbook.Worksheets(1).Range("B1").Formula = "=AVERAGE(A1:A4)"
    Application.ActiveWorkbook.Worksheets(1).Range("B2").Formula = "=STDEV(A1:A4)"
    With Application.ActiveWorkbook.Worksheets(1).Range("B2")
      .Font.Bold = True
      .ColumnWidth = 36
      .HorizontalAlignment = xlCenter
      .Interior.Pattern = xlGray50
    End With
End Sub
```

THE WORLD AS OBJECTS

I. Some Basic Ideas

It is sometimes helpful to view the world as a collection of objects. Each object may have several properties and also may be able to do certain things.

Ex: We may define a generic Airplane as an object with four properties and two "actions", as follows:

Property1: *FleetNumber*
Property2: *AirlineName*
Property3: *FlightNumber*
Property4: *TypeJet*
Action1: *TransportPassengers*
Action2: *TransportCargo*

We can then talk about a particular airplane as an instance of our airplane object. The object (above) is the "template" or "mold" by which we will make all instances of the airplane.

Ex: An instance of an Airplane

Property1 = "J1723"
Property2 = "USAir"
Property3 = 327
Property4 = "Boeing 747"
Action1 = TransportPassengers
Action2 = TransportCargo

Sometimes the things (“actions”) that objects can do are further quantified, e.g.,

TransportPassengers FromCity:="Blacksburg", ToCity:="Beijing"

These “quantifyings” are called arguments.

Sometimes/often objects can be arranged in hierarchies:

Ex: A University

President
- Dean
- Department Head
- Professors
- Students
- Dirt
- Graduate Students (*Just kidding*)
- Post-Docs

Finally, objects sometimes exist individually, and sometimes they exist as collections. For example, there is only one President at W&M, but there are many Deans. The Deans are members of a collection. To refer to a particular Dean in this collection, we provide the collection name with an appropriate specific qualifier in parentheses.

Ex: Dean("Business") or Dean(3) or Dean("Pulley")

II. Dot notation and syntax

As discussed above, objects have both properties and "actions" (called methods in VB terminology) associated with them. Properties describe the objects (adjectives), and methods describe what the objects can do (verbs).

Recall that sometimes objects exist in hierarchies, and that their instances will either exist in collections or individually.

Properties

We represent properties using "dot notation" as follows:

```
ObjectName.PropertyName
```

Ex:
```
WristWatch.Type
Airplane.FlightNumber
Ball.Color
```

1. We can **set the values** of object properties with the following five components:

Component	Example
ObjectName	**Ball**
Dot Operator	**.**
PropertyName	**Color**
Equals Operator	=
PropertyValue	**"red"**

Another example is

```
WristWatch.Type = "Rolex"
```
or
```
Car.Price = 55000
```

2. For objects in a collection, we must specify which item we are talking about in order to prevent confusion; e.g.,

```
Dean("Business").Name = "Sorensen"
```

Here the information in parentheses indicates which Dean we are talking about.

3. We also can **retrieve the values** of object properties as follows:

Component	Example
Variable	**ThisColor**
Equals Operator	=
ObjectName	**Ball**
Dot Operator	**.**
PropertyName	**Color**

Another example is:

```
MyFlight = Airplane.FlightNumber
```
or
```
grade = Student(SSN).FinalAverage
```

N.B.: Student is a collection. SSN is a variable that will hold the text specifying to which student in the collection we are referring.

4. When using objects in a hierarchy we will specify all the objects in that hierarchy that are necessary to prevent confusion, by using dot notation.

Ex:
```
MyCar.Tire("Front Left").Size("75R13")
House.Room("Living").Window("Back").Lock = "unlatched"
```

Methods

We represent object methods using “dot notation” in much the same way as above:

```
ObjectName.Method
```

Ex:
```
WristWatch.Wind
Airplane.TransportPassengers
Ball.Bounce
```

Just as an action does not have a “value”, methods do not have values that can be set or retrieved. Instead, methods have optional arguments, like the arguments to a function, which give additional information about the action being taken.

```
ObjectName.Method  argument(s)
```

These arguments can be explicitly specified by their name(s):

Ex:
```
Professor.Teach  course:=”MSDS460”, days:=”TTh”, time:=”14T”
```

Or, as long as they are in the correct order, they can simply be specified by their value:

Ex:
```
Professor.Teach “MSDS460”, “TTh”, “14T”
```

*** Most methods have default settings for their arguments, and if no values are explicitly specified, these default settings will be assumed.

Different Formats for Different Numeric Values (Format Function)

A user-defined format expression for numbers can have from one to four sections separated by semicolons. If the format argument contains one of the named numeric formats, only one section is allowed.

If you use	**The result is**
One section only	The format expression applies to all values.
Two sections	The first section applies to positive values and zeros, the second to negative values.
Three sections	The first section applies to positive values, the second to negative values, and the third to zeros.
Four sections	The first section applies to positive values, the second to negative values, the third to zeros, and the fourth to Null values.

The following example has two sections: the first defines the format for positive values and zeros; the second section defines the format for negative values.

"$#,##0;($#,##0)"

If you include semicolons with nothing between them, the missing section is printed using the format of the positive value. For example, the following format displays positive and negative values using the format in the first section and displays "Zero" if the value is zero.

"$#,##0;;\Z\e\r\o"

It is for this reason that we can use the format ";;;" to hide all values in a given cell or range of cells.

If you use a number sign (#) in your formatting, a character will only be printed if it exists. If you use a zero (0), however, it acts as a firm placeholder, and a character will always be printed to the screen. To the left of the decimal point, if there are more characters than placeholders, all characters will be printed – to the right of the decimal point, if there are more characters than placeholders then the value will be rounded off to display only the number of placeholders.

To illustrate this, the following formats applied to the number 35.247 give the following results:

Format	**Value displayed**
"$#,##0.##"	$35.25
"$#0,000.#"	$0,035.2
"#.00000"	35.24700
"0000.0000"	0035.2470
"0.0"	35.2

These number formats can be applied in several different ways, including the following:

1) Select a range of cells and choose "Cells" under the "Format" menu in Excel, specifying the Custom category under Number formatting.

2) Use the "NumberFormat" property of the Range object:
   Ex: ActiveSheet.Range("A1").NumberFormat = "#,##0.##;-#,##0.##"

3) Use the Format function***:
   Ex: ActiveSheet.Range("A1").Value = Format(5459.4378, "##,##0.000")

*** The Format function changes the actual format of the specified number, but it does not change the number format which is displayed in the Range in which that number is placed. Therefore, although the statement above changes the number 5459.4378 into the number 5459.438, what will be displayed in that cell because of the default cell formatting settings in Excel (unless you actually change the NumberFormat for that cell) will be 5459.44. Any calculations performed on the value in that cell, however, will use its actual value, 5459.438.

VBA Control Structures

Using control structures, you can control the flow of your program's execution. If left unchecked by control-flow statements, a program's logic will flow through statements from left to right, and from top to bottom. Although you can write very simple programs with only this unidirectional flow, and although you can control a certain amount of flow by using operators to regulate precedence of operations, most of the power and utility of any programming language comes from its ability to change statement order with structures and loops.

Decision Structures

Visual Basic procedures can test conditions and then, depending on the results of that test, perform different operations. The Visual Basic decision structures are listed in the following table.

To test	Use
A single condition and run a single statement or a block of statements	If...Then
A single condition and choose between two statement blocks	If...Then...Else
More than one condition and run one of several statement blocks	If...Then...ElseIf
A single condition and run one of several statement blocks	Select Case

If...Then

Use the If...Then statement to run one or more statements when the specified condition is True. You can use either a single-line syntax or a multiple-line "block" syntax. The following pair of examples illustrate the two types of syntax.

```
If thisVal < 0 Then thisVal = 0

If thisVal > 5 Then
    thatVal = thisVal + 25
    thisVal = 0
End If
```

Notice that the single-line form of the If...Then statement doesn't use an End If statement. If you want to run more than one line of code when the condition is True, you must use the multiple-line If...Then...End If syntax.

Note When the condition you're evaluating contains two expressions joined by an Or operator — for example, If (thisVal > 5 Or thatVal < 9) — both expressions are tested, even if the first one is True. In rare circumstances, this behavior can affect the outcome of the statement; for example, it can cause a run-time error if a variable in the second expression contains an error value.

If...Then...Else

Use the If...Then...Else statement to define two blocks of statements, as in the following example. One of the statements runs when the specified condition is True, and the other one runs when the condition is False.

```
If age < 16 Then
    MsgBox "You are not old enough for a license."
Else
    MsgBox "You can be tested for a license."
End If
```

If...Then...ElseIf

You can add ElseIf statements to test additional conditions without using nested If...Then statements, thus making your code shorter and easier to read. For example, suppose that you need to calculate employee bonuses using bonus rates that vary according to job classification. The Function procedure in the following example uses a series of ElseIf statements to test the job classification before calculating the bonus.

```
Function Bonus(jobClass, salary, rating)
        If jobClass = 1 Then
  Bonus = salary * 0.1 * rating / 10
        ElseIf jobClass = 2 Then
  Bonus = salary * 0.09 * rating / 10
        ElseIf jobClass = 3 Then
  Bonus = salary * 0.07 * rating / 10
        Else
  Bonus = 0
        End If
End Function
```

The If...Then...ElseIf statement block is very flexible. You can start with a simple If...Then statement and add Else and ElseIf clauses as necessary. However, this approach is unnecessarily tedious if each ElseIf statement compares the same expression with a different value. For this situation, you can use the Select Case statement.

Select Case

You can use the Select Case statement instead of multiple ElseIf statements in an If...Then...ElseIf structure when you want to compare the same expression with several different values. A Select Case statement provides a decision-making capability similar to the If...Then...ElseIf statement; however, Select Case makes the code more efficient and readable.

For instance, to add several more job classifications to the example in the preceding section, you can add more ElseIf statements, or you can write the function using a Select Case statement, as in the following example.

```
Function Bonus(jobClass, salary, rating)
        Select Case jobClass
  Case 1
          Bonus = salary * 0.1 * rating / 10
  Case 2
          Bonus = salary * 0.09 * rating / 10
  Case 3
          Bonus = salary * 0.07 * rating / 10
  Case 4, 5       'The expression list can contain several values...
          Bonus = salary * 0.05 * rating / 5
  Case 6 To 8     '...or be a range of values
          Bonus = 150
  Case Is > 8     '...or be compared to other values
          Bonus = 100
  Case Else
          Bonus = 0
        End Select
End Function
```

Notice that the Select Case structure evaluates a single expression at the top of the structure. In contrast, the If...Then...ElseIf structure can evaluate a different expression for each ElseIf statement. You can replace an If...Then...ElseIf structure with a Select Case structure only if each ElseIf statement evaluates the same expression.

Looping Structures

You can use loop structures to repeatedly run a section of your procedure. The Visual Basic loop structures are listed in the following table.

To	Use
Test a condition at the start of the loop, run the loop only if the condition is True, and continue until the condition becomes False	Do While...Loop
Test a condition at the start of the loop, run the loop only if the condition is False, and continue until the condition becomes True	Do Until...Loop
Always run the loop once, test a condition at the end of the loop, continue while the condition is True, and stop when the condition becomes False	Do...Loop While
Always run the loop once, test a condition at the end of the loop, continue while the condition is False, and stop when the condition becomes True	Do...Loop Until
Run a loop a set number of times, using a loop counter that starts and ends at specified values and that changes value by a specified amount each time through the loop	For...Next
Run a loop once for each object in a collection	For Each...Next

Note Visual Basic also includes the While…Wend statement, but it's a good idea to use the more flexible variations of the Do…Loop statement (such as Do While…Loop or Do…Loop While) instead.

Do...Loop

Use a Do...Loop statement to run a block of statements an indefinite number of times — that is, when you don't know how many times you need to run the statements in the loop. There are several variations of the Do...Loop statement, but each one evaluates a condition to determine whether or not to continue running. As with an If...Then statement, the condition must be a value or an expression that evaluates to either True or False. The different Do…Loop variations are described in this section. For more information about the Do...Loop statement, see "Do...Loop Statement" in Help.

Note If you want to run a block of statements a specific number of times, use a For…Next loop.

Do While...Loop

Use the Do While...Loop statement when you want to test a condition before you run the loop and then continue to run the loop while the condition is True.

Note: The statements in a Do While…Loop structure must eventually cause the condition to become False, or the loop will run forever (this is called an infinite loop). To stop an infinite loop, press CTRL+BREAK.

The Function procedure in the following example counts the occurrences of a target string within another string by looping as long as the target string is found. Because the test is at the beginning of the loop, the loop runs only if the string contains the target string.

```
Function CountStrings(longstring, target)
        position = 1
        Do While InStr(position, longstring, target) 'Returns True/False
           position = InStr(position, longstring, target) + 1
         Count = Count + 1
      Loop
      CountStrings = Count
 End Function
```

Do Until...Loop

Use the Do Until…Loop statement if you want to test the condition at the beginning of the loop and then run the loop until the test condition becomes True. If the condition is initially True, the statements inside the loop never run. With the test at the beginning of the loop in the following example, the loop won't run if Response is equal to vbNo.

```
Response = MsgBox("Do you want to process more data?", vbYesNo)
Do Until Response = vbNo
         ProcessUserData    'Call procedure to process data
         Response = MsgBox("Do you want to process more data?", vbYesNo)
Loop
```

Do...Loop While

When you want to make sure that the statements in a loop will run at least once, use Do…Loop While to put the test at the end of the loop. The statements will run as long as the condition is True. In the following Microsoft Excel example, the loop runs only if the Find method finds a cell that contains "test." If the text is found, the loop sets the color of the cell, and then searches for the next instance of "test." If no other instance is found, the loop ends.

```
 Sub MakeBlue()
         Set rSearch = Worksheets("sheet1").Range("a1:a10")
         Set c = rSearch.Find("test")
         If Not c Is Nothing Then
   first = c.Address
   Do
           c.Font.ColorIndex = 5
           Set c = rSearch.FindNext(c)
   Loop While (Not c Is Nothing) And (c.Address <> first)
         Else
   MsgBox "not found"
         End If
 End Sub
```

Do...Loop Until

With the Do…Loop Until statement, which puts the test at the end of the loop, the loop runs at least once and stops running when the condition becomes True, as shown in the following example.

```
 Do
        ProcessUserData    'Call procedure to process data
        response = MsgBox("Do you want to process more data?", vbYesNo)
 Loop Until response = vbNo
```

For...Next

When you know that you must run the statements a specific number of times, use a For...Next loop. Unlike the many variations of Do…Loop, a For...Next loop uses a counter variable that increases or decreases in value during each repetition of the loop. Whereas the variations of Do…Loop end when a test condition becomes True or False, a For...Next loop ends when the counter variable reaches a specified value.

The Sub procedure in the following example sounds a tone however many times you specify.

```
 Sub BeepSeveral()
        numBeeps = InputBox("How many beeps?")
        For counter = 1 To numBeeps
   Beep
        Next counter
 End Sub
```

Because you didn't specify otherwise, the counter variable in the preceding example increases by 1 each time the loop repeats. You can use the Step keyword to specify a different increment for the counter variable (if you specify a negative number, the counter variable decreases by the specified value each time through the loop). In the following Sub procedure, which replaces every other value in an array with 0 (zero), the counter variable increases by 2 each time the loop repeats.

```
Sub ClearArray(ByRef ArrayToClear())
        For i = LBound(ArrayToClear) To UBound(ArrayToClear) Step 2
   ArrayToClear(i) = 0
        Next i
End Sub
```

Note   The variable name after the Next statement is optional, but it can make your code easier to read, especially if you have several nested For loops.

For Each...Next

A For Each...Next loop is similar to a For...Next loop, except that it repeats a group of statements for each element in a collection of objects or in an array, instead of repeating the statements a specified number of times. This is especially useful if you don't know how many elements are in a collection, or if the contents of the collection might change as your procedure runs. The For Each…Next statement uses the following syntax.

```
For Each element In group
              statements
Next element
```

When Visual Basic runs a For Each...Next loop, it follows these steps:

1. It defines element as naming the first element in group (provided that there's at least one element).
2. It runs statements.
3. It tests to see whether element is the last element in group. If so, Visual Basic exits the loop.
4. It defines element as naming the next element in group.
5. It repeats steps 2 through 4.

The following Microsoft Excel example examines each cell in the current region for cell A1 on the worksheet named "Sheet3" and formats its contents as red if its value is less than – 1.

```
For Each c In Worksheets("sheet3").Range("a1").CurrentRegion.Cells
        If c.Value < -1 Then c.Font.ColorIndex = 3
Next c
```

The following Word example loops through all the revisions in the current selection and accepts each one.

```
 For Each myRev In Selection.Range.Revisions
     myRev.Accept
 Next myRev
```

The variable name after the Next statement — c in the Microsoft Excel example and myRev in the Word example — is optional, but it can make your code easier to read, especially if you have several nested For…Each loops.

Important   If you want to delete all the objects in a collection, use a For...Next loop instead of a For Each...Next loop. The following example deletes all the slides in the active PowerPoint presentation.

```
 Set allSlides = ActivePresentation.Slides
 For s = allSlides.Count To 1 Step -1
     allSlides.Item(s).Delete
 Next
```

The code in the following example, on the other hand, won't work (it will delete every other slide in the presentation).

```
 For Each s In ActivePresentation.Slides
     s.Delete
 Next
```

Keep the following restrictions in mind when using the For Each...Next statement:

For collections, element can only be a Variant variable, a generic Object variable, or a specific object type in a referenced object library. For arrays, element can only be a Variant variable.

You cannot use the For Each...Next statement with an array of user-defined types, because a Variant variable cannot contain a user-defined type.

Nesting Control Structures

You can place control structures inside other control structures; for instance, you can place an If...Then block within a For Each...Next loop within another If...Then block, and so on. A control structure placed inside another control structure is said to be nested.

The following example searches the range of cells you specify with an argument and counts the number of cells that match the value you specify.

```
Function CountValues(rangeToSearch, searchValue)
        If TypeName(rangeToSearch) <> "Range" Then
  MsgBox "You can search only a range of cells."
        Else
  For Each c in rangeToSearch.cells
          If c.Value = searchValue Then
    counter = counter + 1
          End If
  Next c
        End If
        CountValues = counter
End Function
```

Notice that the first End If statement closes the inner If...Then block and that the last End If statement closes the outer If...Then block. Likewise, in nested For...Next and For Each...Next loops, the Next statements automatically apply to the nearest prior For or For Each statement. Nested Do...Loop structures work in a similar fashion, with the innermost Loop statement matching the innermost Do statement.

Exiting Loops and Procedures

Usually, your macros will run through loops and procedures from beginning to end. There may be situations, however, in which leaving, or exiting, a loop or procedure earlier than normal can save you time by avoiding unnecessary repetition.

For example, if you're searching for a value in an array using a For...Next loop and you find the value the first time through the loop, there's no reason to search the rest of the array — you can stop repeating the loop and continue with the rest of the procedure immediately. If an error occurs in a procedure that makes the remainder of the procedure unnecessary, you can exit the procedure immediately. You can cut a control structure off early by using one of the Exit statements.

Although the Exit statements can be convenient, you should use them only when it's absolutely necessary and only as a response to an extraordinary condition (not in the normal flow of a loop or procedure). Overusing Exit statements can make your code difficult to read and debug.

Also, there may be a better way to skip portions of your macro. For instance, instead of using an Exit statement inside a For...Next loop while searching for a value in an array, you could use a Do…Loop to search the array only while an incremented index value is smaller than the array's upper bound and a Boolean variable value is False, as shown in the following example. When you find the array value, setting the Boolean value to True causes the loop to stop.

```
i = LBound(searchArray)
ub = UBound(searchArray)
foundIt = False
Do
        If searchArray(i) = findThis Then foundIt = True
        i = i + 1
Loop While i <= ub And Not foundIt
```

You use the Exit Do statement to exit directly from a Do…Loop, and you use the Exit For statement to exit directly from a For loop, as shown in the following example.

```
For Each c in rangeToSearch
        If c.Value = searchValue Then
  found = True
  Exit For
        End If
Next
```

You use the Exit Sub and Exit Function statements to exit a procedure. The following example demonstrates the use of Exit Function.

```
For Each c in rangeToSearch
        If c.Value = searchValue Then
  counter = counter + 1
        ElseIf c.Value = "Bad Data" Then
  countValues = Null
  Exit Function   'Stop testing and exit immediately.
        End If
Next c
```

```
Sub LearnIfThen()
    Dim answer As String
    answer = InputBox("Yes or no?")
    If answer = "no" Then MsgBox "I'm sorry, you lose"
    If answer = "yes" Then
        MsgBox "Congratulations! - you pass!"
    End If
End Sub


Sub LearnIfThenElse()
    Dim Age As Integer
    Age = ActiveWorkbook.Worksheets(1).Range("B1").value
    If Age < 16 Then
        MsgBox "You are too young to drive"
    Else
        MsgBox "You may drive (but not MY car!)"
   End If
End Sub


Sub LearnIfThenElseIf()
    Dim grade As Integer
    grade = InputBox("Please enter number grade: ")
    If grade < 65 Then
        MsgBox "Sorry - you failed"
    ElseIf grade < 80 Then
        MsgBox "You got a C on the test"
    ElseIf grade < 90 Then
        MsgBox "You got a B on the test"
    ElseIf grade < 100 Then
        MsgBox "You got an A on the test - congratulations!"
    Else  ' If grade >= 100
        MsgBox "Congratulations! - you got an A+"
    End If
End Sub


Sub LearnSelectCase()
    Dim FootballGames As Integer
    FootballGames = InputBox("How many football games will we win this year? ")
    Select Case FootballGames
        Case Is < 8
           MsgBox "Pessimist!"
        Case 8
           MsgBox "Eight games - not bad"
        Case 9
           MsgBox "Nine games - pretty good"
        Case 10
           MsgBox "Ten games - great!"
        Case 11
           MsgBox "Eleven games - Conference Champions!"
        Case Else
           MsgBox "National Championship!!!!"
    End Select
End Sub
```

```
Sub LearnFor()
    Dim randomnumber As Integer
    For i = 1 To 4
        randomnumber = Int(56 * Rnd)
        Worksheets("Sheet1").Range("A1").Select
        Worksheets("Sheet1").Cells(i, 2).Interior.ColorIndex = randomnumber
    Next
End Sub


Sub LearnNestedFor()
    Dim randomnumber As Integer
    For i = 1 To 4
      For j = 1 To 5
        randomnumber = Int(56 * Rnd)
        Worksheets("Sheet1").Range("A1").Select
        Worksheets("Sheet1").Cells(i + 5, j).Interior.ColorIndex = randomnumber
      Next
    Next
End Sub


Sub LearnForEach()
    Dim wkbk As Workbook
    Dim wksht As Worksheet
    Set wkbk = ActiveWorkbook
    For Each wksht In wkbk.Worksheets
      MsgBox wksht.Name & " = Sheet" & wksht.Index
    Next wksht
End Sub


Sub LearnDoWhile()
    Dim value As String
    value = InputBox("Do you want to continue? ")
    Do While value = "yes"
      MsgBox "Good! - let's try again"
      value = InputBox("do you want to continue? ")
    Loop
End Sub


Sub LearnDo()
    Dim value As String
    value = InputBox("Do you want to continue? ")
    Do
        MsgBox "Good! - let's try again, regardless..."
        value = InputBox("Do you want to continue? ")
    Loop While value = "yes"
End Sub
```