Visual Basic - Modules and Procedures

Introduction

A *procedure* is a unit of code enclosed either between the **Sub** and **End Sub** statements or between the **Function** and **End Function** statements. A procedure should accomplish a simple well-defined task.

Modules are workbook sheets that contain code. Each module can contain a *declarations* section followed by procedures. There may be many procedures in a module, but preferably the procedures should have some common theme.

The *declaration* section can contain

- *Option statements* which set module-level options and must appear before any procedure.
- User defined type definitions
- Declarations any variable or constant declarations that are not contained within a procedure must be placed before any procedures in the module.

Except for the three classes above, ALL code in VB must be contained within procedures.

Procedures that access common data or provide a coherent set of actions can be grouped into the same module. Modules can be grouped into a single workbook, (or over several linked workbooks but that's too technical for this course).

Creating procedures

When you create a procedure you must decide

- how widely accessible you want it to be,
- what to name it,
- what task it will perform,
- what types of data it will accept and return.

Specifying scope

The term *scope* means the parts of the programme from which a variable, constant or procedure is accessible or "visible". VB procedures can have either private or public scope. A procedure with private scope is visible only to other procedures in the same module; one with public scope is visible to all procedures in all modules in a workbook. Unless specified otherwise, procedures have public scope. To make a procedure private we use the **Private** keyword,

Private Sub Savings() MsgBox "A private procedure" End Sub

A private procedure cannot run as a stand-alone procedure; it can only be called from another procedure.

Naming procedures

A procedure name must

- begin with a letter,
- not contain periods, mathematical or comparison operators,
- not exceed 255 characters,
- be unique within its scope
- not be the name of an existing VB method, function or property.

Types of procedure

There are two types: **Sub** procedures and **Function** procedures.

A **Sub** procedure is enclosed between **Sub** and **End Sub** statements and performs a task but does not return a value. It may modify an Excel spreadsheet.

A **Function** procedure is enclosed between **Function** and **End Function**. It also performs some task but must also return a value. A function procedure is analogous to a mathematical function. It may not modify a spreadsheet, for example by deleting rows.

For a function to return a value it must include a function assignment statement that gives a value to the name of the function. For example

```
Function ConeSurface(radius, height)
` calculates the surface area of a right-cone
' given its radius and height
Const Pi = 3.14159
coneBase = Pi * radius ^ 2
coneSide = Pi * radius * _
Sqr(radius ^ 2 + height ^ 2)
ConeSurface = coneSide + coneBase
End Function
```

End Function

The information for the function to perform its task (radius and height) is passed as arguments.

This function may be used in an expression such as

surfaceCone = ConeSurface(3,10)

in another procedure and also as a function in a workbook in which it is defined. In a cell you may type the formula =ConeSurface(3,10), or, for example, in cell C1 put =ConeSurface(A1,B1) if the radius and height data are stored in cells A1 and B1 respectively.

[The function can be used in a workbook in which it is not defined provided you precede the function with the name of the workbook in which it is defined (Conebook.xls) and which must also be open, i.e.

=CONEBOOK.XLS!ConeSurface(3,10) .]

Calling procedures

There are two ways to call a **Sub** procedure

- 1. via a **Call** syntax with the arguments within brackets, e.g. Call MyProc(arg1, arg2)
- 2. without call and without brackets, e.g. MyProc arg1, arg2

Functions are called in the same way as built-in functions, =sin(x), e.g.

```
X = ConeSurface(radius, height)
```

Arguments

When you use an argument you can specify

- the data type,
- whether the procedure can change the argument's value
- whether an argument is required or is optional

The arguments are included in brackets after the procedure name in its definition, e.g. Sub UpdateRecords(ownerID, ownerName) is called in the following extract

```
Dim newID As Integer, newName As String
    newId = 1467
    newName = "Joe Green"
    UpdateRecords newID, newName
```

Note that the name of the variable passed from the calling procedure does NOT have to be the same as the name of the argument declared in the definition of the called procedure. If an array is passed as an argument, you include empty brackets after the array name. Since the dimensions of the array are not defined, the procedure can accept arrays of any size, e.g. Sub CustomerList(customerArray()).

Arguments have the same scope as variables declared within the procedure. They are not accessible outside the procedure in whose argument list they are declared. By default, procedure arguments have the Variant data type (see later notes). You can explicitly declare other data types using the As keyword in the declaration, e.g. Function ReverseText(text As String, n As Integer) ' reverses the first n characters in "text" Dim temp As String, i As Integer ' check on n compared to number of characters in "text" If n > Len(text) Then n = Len(text) ' fill "temp" with characters reversed For i = n To 1 Step -1 temp = temp & Mid(text, i, 1) Next i ReverseText = temp & Right(text, Len(text) - n) End Function

Passing arguments by Reference or by Value

This discussion is quite technical and initially can be ignored. It is, however, one cause of errors in programmes and so could be looked at on a second pass through the material.

If you pass an argument by reference when calling a procedure it has access to the actual variable in memory and so can change its value. This is the default mode in VB. The statement Sub Pass(ByRef passedVariable As Integer) has the same effect as the statement Sub Pass(passedVariable As Integer).

If you pass an argument by value when calling a procedure the called procedure only gets a COPY of the variable. Hence if the called procedure changes its value it only affects the copy and the original variable is unchanged. Passing variables by value therefore protects the original data from being changed accidentally by the called procedure. Compare the two cases below. TestArgs calls the procedure AddTen with the arguments passed by reference or by value. The numerical outcomes are very different!

```
Sub TestArgs()
     Dim varToPass As Integer
     varToPass = 4
     AddTen varToPass
     MsgBox varToPass ' this displays 14
End Sub
Sub Addten(ByRef passedVar As Integer)
     passedVar = passedVar + 10
     Msqbox passedVar
                         ' this also displays 14
End Sub
When TestArgs is run the message box display the number 14.
If on the other hand we had passed the argument by value and defined
Sub AddTen(ByVal passedVar As Integer)
     passedVar = passedVar + 10
     Msqbox passedVar
                        ` this displays 14
```

End Sub

we would find that the message box in TestArgs would display the number 4, not 14 because AddTen only received a copy of the argument passedVar and could not change the original. Note that passedVar is NOT available in TestArgs in either case.

If a data type is specified for an argument passed by reference you must pass a value of that type. On the other hand if the argument is passed by value, you need not pass a value of the same data type as the argument.

There is one error that is produced by sloppy typing. If the argument is to be passed by reference then the statements

```
Call AddTen(passedVar)
or
```

AddTen passedVar

will do so. However if the Call is inadvertently omitted from the first syntax so that you write

AddTen(passedVar),

then the argument is passed by value and the outcome is different as in the previous example. Beware!

Optional arguments

The **Optional** keyword can be used in the argument list to specify that some of the arguments to the procedure are optional. If used, all subsequent arguments in the list must be optional. (Optional arguments must be of the Variant type). For example Sub List(startRow As Integer, startCol As Integer, _

```
Optional newText, Optional oldText)
When this procedure is called we could use
List 2, 4, False, True
List 2, 4, , True
```

Named arguments

Many built-in functions and methods take several optional arguments. Commas would be needed for all the missing ones and typing errors are liable to occur. You can use named arguments instead. VB associates argument names with the procedure. So the List procedure could be called in the following way. List newText:= False, startRow:= 2, startCol:= 4 Note the syntax := after each argument name, and also that the named arguments can

be given in any order.

Indefinite number of arguments

Usually the number of arguments in the calling of a procedure must be the same as in the procedures definition. The use of the **ParamArray** keyword allows you to specify that the procedure will accept an arbitrary number of arguments. (The argument used with **ParamArray** must be an array of **Variant** variables). The **ParamArray** keyword can only be used with the last argument in the list. The following procedure will work with any number of arguments.

```
Function AddIntegers(ParamArray intNumbers())
    Dim j As Integer, y As Variant
    For Each y In intNumbers
        x = x + y
    Next y
    AddIntegers = x
End Function
This procedure could be called by AddIntegers(1, 3, 7) where three
arguments are supplied or by AddIntegers(1, 3, 7, 9, 11, 13, 15)if
```

seven are given.

Exiting procedures

Procedures usually run from the first statement to the last. Occasionally you may need to exit earlier, e.g. an error has occured in the procedure that makes it necessary to quit. Use **Exit Sub** and **Exit Function** to exit the procedure in a controlled manner. **Exit** statements are convenient but should not be overused as it makes the code difficult to read and debug.