2 Defining Variables

- **Variables**
  - Dim Statement
  - Option Explicit
  - Variable Names
    - Standard Naming Conventions
  - Data Types and Dim Examples
    - Byte
    - Boolean
    - Integer
    - Long
    - Currency
    - Single
    - Double
    - Date
    - String (for variable-length strings)
    - String * length (for fixed-length strings)
    - Object
    - Variant
    - user-defined type
    - object type
  - AutoComplete
  - Indenting Code
    - Indenting several lines of code
    - Removing an indent
  - TypeName Function
- **Arrays**
  - Option Base Statement
  - Array Examples

Objective: Understand variables and Data Types; AutoComplete; indent code; Option Explicit and Option Base statements

Variables

When you are writing programs, you want to be able to keep track of values in code so you can use them later. For instance, let's say you want to write a program to ask the user their name. You can create a variable to hold the name and prompt the user for its value.

"Declaring variables" means that you are going to tell Excel what variables you are going to use and (optionally, but suggested) what type of data those variables are going to have.

To declare a variable will be used in the program code, the **Dim** (dimension) statement is used.

Dim Statement

Dim **[**WithEvents**]** ***varname*****[([*subscripts*])]** **[**As **[**New**]** ***type*****]** **[**, **[**WithEvents**]** ***varname*****[([*subscripts*])]** **[**As **[**New**]** ***type*****]]** . . .

The most common form of the **Dim** statement is:

Dim varname As type

The following statement declares a variable named **mName** as a string, which means a sequence of characters.

```
Dim mUserName As String
```

If you don't declare your variable, Excel will automatically create it the first time you use it in your code. If you don't require variable declaration, and just type variable names in code, you may make a typo. For instance:

```
mUserName = "Cindy"

'lots of code lines

MsgBox mUser_Name
```

The problem is that these two variable names are not the same and you may not catch it. Spelling errors are difficult to find. When **mUser_Name** is used in the code, Excel will create another variable and assign it a value of **Empty**. You can eliminate this problem by *requiring* variable declaration. To do this, use the **Option Explicit** statement. Then, Excel will find this error for you when you compile the code.

Not declaring variables is a very bad habit.

Option Explicit

To require variable declaration in a module sheet, the following line can be typed at the top of the sheet:

```
Option Explicit
```

The **Option Explicit** statement must be at the top of the code sheet before any global variable declarations and any procedures.

To set the default value to require variable declaration (which is a good idea), from a module window, choose

Tools, Options… Editor Tab
and check Require Variable Declaration

the next time you create a module sheet, **Option Explicit** will be written at the top. Alternately, you can write **Option Explicit** yourself – it must be above any procedures.

If you do not require variable declaration and you use variables that are not defined, typos will be hard to find since the debugger won't catch them.

Variable Names

A variable name represents a storage location that can contain data and be modified during program execution.

Variable names must:

- begin with an alphabetic character
- be unique within the same scope (for instance, you cannot create two variables with the same name in a procedure)
- be from 1 to 255 characters long
- not contain embedded periods, type-declaration characters, spaces or other illegal characters
- not be reserved words.

It is best to create variable names that are:

- short yet descriptive
- composed of:
  alphabetic characters
  numeric characters
  underscore characters _
- Mixed Case
  If you dimension a variable as MixedCase and then later type that variable in all lowercase, Excel will adjust the case to the way you declared it when you move off the line.

To avoid accidentally using a reserved word as a variable name, I like to start memory variables with a lowercase "m" and array variable with "a". For instance, **mUserName** for a variable that will hold the user name, **aDaysofWeek()** for an array containing the names of days.

Standard Naming Conventions

If you follow the standard naming convention for variables, 3 character prefixes such as "**str**" for string, "**int**" for integer, "**lng**" for long, "**cur**" for currency, are used. The reason I don't like this method is that you have to visually strip too many characters off the variable name to determine what it is. If you know your data well and use logical names, you will know what the type is.

Also, by convention, variables are dimensioned at the top of your code, so you can easily refer back to them.

Data Types and Dim Examples

These examples show how the **Dim** statement used to declare variables. Several declarations can be included on one line and separated by commas.

Data types of variables may be:

Byte

positive whole numbers
range: 0-255
size: single, unsigned 8-bit (1-byte) numbers.

```
Dim mByte As Byte
```

Boolean

two possible values
range: True (-1) or False (0)
size: 16-bit (2-byte) numbers

```
Dim IsDone as Boolean
```

Integer

whole numbers
range: -32,768 to 32,767
size: 16-bit (2-byte) numbers
type-declaration character: %

```
Dim mCounter As Integer, MyInteger%
```

Long

whole numbers
range: -2,147,483,648 to 2,147,483,647
size: 32-bit (4-byte) numbers
type-declaration character: &

```
Dim mNumberOfRows As Long, MyLong&
```

Currency

numbers: 15 digits before decimal, 4 digits after
use: where accuracy is particularly important
range: -922,337,203,685,477.5808 to
922,337,203,685,477.5807
size: 32-bit (4-byte) numbers
type-declaration character: @

```
Dim mAmount As Currency, Salary@
```

Single

single-precision floating-point numbers
range: -3.402823E38 to -1.401298E-45 for negative values, and 1.401298E-45 to 3.402823E38 for positive values
size: 32-bit (4-byte) numbers
type-declaration character: !

```
Dim mQuantity As Single, MySingle!
```

Double

double -precision floating-point numbers
range: -1.79769313486231E308 to -4.94065645841247E-324 for negative values, and 4.94065645841247E-324 to 1.79769313486232E308 for positive values
size: 64-bit (8-byte) numbers
type-declaration character: #

```
Dim mWeight As Double, MyDouble#
```

Date

dates and times as a real number where the integer portion represents date and the decimal portion represents time (value to the left of the decimal represents a date, and the value to the right of the decimal represents a time)
range: 1/1/100 to 12/31/9999
size: 64-bit (8-byte) numbers

```
Dim mBirthDate As Date, mTimeIn As Date
```

Here is something fun:

1. Type **1** into cell A1 and format the cell to be a date – you should see 1/1/1900.
2. Type your birth date into cell A2
3. In cell A3, press Ctrl ; to enter today's date
4. since dates are stored as numbers, we can subtract one from another. In cell A4, enter
   =A3 – A2
5. Right Click on cell A4 and choose Format Cells… from the shortcut menu
6. on the Number Tab, choose "Custom" and enter **#,##0** for the format code in the **Type** box

This is how many days you have been alive.

String (for variable-length strings)

sequence of contiguous characters that represent the characters themselves rather than their numeric values. A String can include letters, numbers, spaces, and punctuation.
size: dynamic strings from 0 to approximately 2 billion characters
type-declaration character: $

```
Dim mUserName As string, mCity$
```

String * length (for fixed-length strings)

size: fixed-length strings from 0 to approximately 63K characters

```
Dim mZipCode As String*10
```

Object

represents any Object reference
size: 32-bit (4-byte) addresses that refer to objects

Variant

numbers, strings, dates, arrays, user-defined types, etc. as well as the special values **Empty** and **Null**
size: 16 bytes for Decimal
22 bytes (plus string length) for character storage

All variables are treated as **Variant** data types if not explicitly declared as some other data type. The **VarType** function defines how the data in a **Variant** is treated.

If a variable is dimensioned *without* declaring the data type, the **Variant** type is assumed. Variables of **Variant** data type take more room to store and are, therefore, not as efficient.

The first is a **Variant** because it is not specified and the second is a **Variant** because it is declared to be a **Variant**.

```
Dim AnyValue, MyValue As Variant
```

When you use a **Variant** type, which can hold any type of data -- numbers, strings, arrays, etc – you have flexibility to be lazy, and that also degrades performance. In order to work with a **Variant**, Excel has to figure out what type of data it has, keep track of that in another place, as well as perform conversions, whereas if you specify a data type, Visual Basic can compile the code more efficiently.

Another reason to specifically declare the data type is so that the AutoComplete feature of Visual Basic can list only things that apply specifically to that data type.

user-defined type

can contain one or more elements of any data type and are defined using the **Type** statement. Arrays of user-defined and other data types are created using the **Dim** statement. Arrays of any type can be included within user-defined types.

object type

represents objects such as Application, File, Range, Workbook, and Sheet that are exposed by an application through Automation. Use the **Object Browser** (F2 from module window) or refer to the application's documentation for a complete listing of available objects.

Unlike regular variables, which can be assigned using the equal sign (=), object variables must be assigned with the **SET** statement. Also, they must be specifically released from memory when you are done by **SET**ting them to **NOTHING**.

```
Sub ObjectDataTypes()

   Dim Sht As Worksheet, Wb As Workbook

   Set Sht = Application.ActiveSheet

   Set Wb = Application.ActiveWorkbook

   MsgBox Sht.Name, , "Activesheet Name"

   MsgBox Wb.Name, , "ActiveWorkbook Name"

   Set Sht = Nothing

   Set Wb = Nothing

End Sub
```

The **Application** object represents Excel.

Using **Object** as the data type for an object is like using a variant. It is inefficient and degrades performance.

Use

```
Dim Sht As Worksheet
```

instead of

```
Dim Sht As Object
```

It is important to develop good habits when you start coding. Just because you are allowed to not declare variables and data types doesn't mean that you should.

AutoComplete

After you type a keyword and the period (.), Excel will prompt you with a dropdown list of acceptable values. As you continue typing, the highlighted entry on the list will change to the first choice that matches the characters you have entered.

You can choose the highlighted choice by pressing Tab.

The shortcut key for the AutoComplete list is Ctrl Spacebar

Indenting Code

It is a good practice to *indent* code to make it easier to read. If you press the Tab key before you type anything, the code will indent to the next tab stop.

Indenting several lines of code

You can highlight several lines of code and press Tab to move all the highlighted lines one tab stop to the right.

Removing an indent

Press Shift Tab to move the indent to the left.

TypeName Function

The **TypeName** Function returns a **String** that provides information about a variable.

TypeName(*varname*)

Here is an example that uses the **TypeName** function:

```
Sub LearnTypeNameFunction()

   Dim mSingle As Single, mCurrency As Currency, mInteger As Integer

   MsgBox TypeName(mSingle), , "variable: mSingle"

   MsgBox TypeName(mCurrency), , "variable: mCurrency"

   MsgBox TypeName(mInteger), , "variable: mInteger"

End Sub
```

Declaring multiple variables on the same line

Even though you may declare multiple variables on the same line of code, each must have its type explicitly declared.

```
Dim mString1, mString2, mString3 As String
```

does not mean that **mString1** and **mString2** will be strings – they won't! They will be variants. You need to do this:

```
Dim mString1 As String, mString2 As String, mString3 As String
```

Another recommendation is that you group your variables together. When you put multiple variable declarations on the same line, it helps if they relate to each other.

```
   Dim mCustomerName As String, mInvoiceNum As String, mAmount As Currency

   Dim mHeaderFile As String, mFileNumber As Long
```

Another consideration is not to let your lines of code extend past the screen boundaries.

Arrays

When you have several values for "one" variable, such as days of the week, you can declare an **Array**. The lower bound for array subscripts is 0 unless explicitly defined.

Option Base Statement

The default lower bound can be overridden at the module level using the **Option Base** statement at the top of the module sheet before any procedures.

```
Option Base  0
```

OR

```
Option Base  1
```

Since 0 is the default starting value for the bounds of an array, it is not necessary to include the **Option Base** statement unless you want to override the default and make it 1.

Using **Option Base 1** makes it easier to work with your code since the array counters will start at 1 instead of 0.

Array Examples

Use the **Option Base 1** statement at the top of the module for the lower bound of arrays to start at 1.

```
Option Base 1
```

```
Sub DayArray()

   Dim i As Integer, aWeek, mDay As String

   aWeek = Array("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")

   i = InputBox("Enter a number from 1 to 7", "Day of Week")

   mDay = aWeek(i)

   MsgBox "You have chosen " & mDay, , i

End Sub
```

It is common to use

i
j
k

to represent integers. This is probably leftover from the mainframe days when the first letter of a variable determined its type.

An alternative to using the **Option Base** statement is to explicitly define an array to start at 1. You can also define and assign array elements like this:

```
Sub DayArray2()

   Dim i As Integer, aWeek(1 To 7) As String, mDay As String

   aWeek(1) = "Sun"

   aWeek(2) = "Mon"

   aWeek(3) = "Tue"

   aWeek(4) = "Wed"

   aWeek(5) = "Thu"

   aWeek(6) = "Fri"

   aWeek(7) = "Sat"

   i = InputBox("Enter a number from 1 to 7", "Day of Week")

   mDay = aWeek(i)

   MsgBox "You have chosen " & mDay, , i

End Sub
```

The arrays we have been using are called one-dimensional arrays. In these examples, "**a**" has been used as a prefix character for array variable names.

You can have arrays with more dimensions. An Excel spreadsheet is an example of a two-dimensional array. Each cell is an intersection between a row and a column and can be addressed as:

cells(RowNumber, ColumnNumber)

Assume you have a 10-row lookup table in your spreadsheet from column A to Column E (5th column) .

aCellValue is a two-dimensional array of integers.

```
Dim aCellValue (10, 5) As Integer
```

The bounds of an array can be anything you specify.

Lets say you are tracking production by

1. day of week (1 to 7)
2. department (10 to 15)
3. shift (1 to 2)

aProduction is defined as a three-dimensional array of double-precision numbers with explicit bounds.

```
Dim aProduction(1 To 7,  10 To 15,  1 To 2) As Double
```

Sometimes you do not know how many elements you will need in an array when you are dimensioning it. You can define a dynamic array and then use the **ReDim** statement to change the number of elements – that will be discussed later in this course.

This page left blank intentionally.