How to use variables in Excel sub-procedures in Visual Basic for Applications

Article ID: 843144 - View products that this article applies to.

INTRODUCTION

This article describes how to use variables in Microsoft Excel sub-procedures in Microsoft Visual Basic for Applications.

MORE INFORMATION

Variables in a sub procedure

A powerful feature of programming languages is the ability to store something in a variable so that the contents of the variable can be used or can be changed later in the procedure. This document discusses the following use of variables in Visual Basic:

- How variables are declared.
- The procedures and the projects that can use the variable.
- The lifetime of a variable.

Declare a variable in a macro

The simplest way to declare a variable in a macro is to use the **Dim** statement. The following line declares two variables, x and y, as Integers:

```
Dim x As Integer, y As Integer
```

With x and y specified as integers, you are telling Visual Basic to set aside sufficient memory for an integer variable (2 bytes each for x and y) and that the information that is stored in either x or y is a whole number between -32768 and 32767.

Note If you declare more than one variable by using a single **Dim** statement, you must specify the data type for each variable.

If you do not specify the data type for each variable, as in the following Visual Basic code, only the variable y is set up as an integer variable. The variable x will be a variant type:

```
Dim x, y As Integer
```

For additional information, see the "Variant data type" section.

To perform a variable test, follow these steps:

1. Save and close any open workbooks, and then open a new workbook.
2. Start the Visual Basic Editor (press ALT+F11).
3. On the **Insert** menu, click **Module**.
4. Type the following code:

   ```
   Sub Variable_Test() Dim x As Integer, y As Integer x = 10 y = 100 MsgBox "the value of x is " & x & _ Chr(13) & "the value of y is " & y End Sub
   ```

5. Run the **Variable_Test** macro. You receive the following message:

   the value of x is 10
   the value of y is 100

6. Click **OK**.
7. In the **Variable_Test** macro change the following line:

   ```
   x = 10
   ```

   to:

   ```
   x = "error"
   ```

8. Run the **Variable_Test** macro.

You will receive a run-time error because "error" is not an integer and you are trying to assign this string value to the integer variable x.

Data type summary

The following table lists common variable data types:

Data type	Storage size	Allowable Range
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,647
Double	8 bytes	-1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date	8 bytes	January 1, 100 to December 31, 9999
String	10 bytes + string length	

Variant data type

If you do not specify a data type when you declare a variable, or you do not declare a variable at all, Visual Basic automatically specifies the variant data type for this variable. The following are the advantages of variables that are declared as this data type:

- The variables can contain string, date, time, Boolean, or numeric values.
- The variables can convert the values that they contain automatically.

The disadvantage is that variant variables require at least 16 bytes of memory. 16 bytes of memory can be significant in large procedures or in complex modules.

To see how this works in the **Variable_Test** macro, follow these steps:

1. Change the code in the **Variable_Test** macro to:

   ```
   Sub Variable_Test() Dim x, y x = "string" y = 1.23 MsgBox "the value of x is " & x & _ Chr(13) & "the value of y is " & y End Sub
   ```

2. Run the **Variable_Test** macro.

You will not receive an error because you can assign anything to the variant variables x and y.

Note You can also leave out the following line and the macro will still work as the variables x and y are treated as Variant data types:

```
Dim x, y
```

Scope of a variable

When you declare a variable, it may or may not be seen by other macros in the same module, in other modules, or in other projects. This availability of a variable in modules is referred to as scope. The three types of scope are procedure-level, private module-level, and public module-level. The scope depends on how and where you declare your variable or variables.

Procedure-level scope

A variable with procedure-level scope is not seen outside the procedure where it is declared. If you set the value of a variable that has procedure-level scope, that variable's contents will not be seen by other macros.

To verify that a variable with procedure-level scope is not seen outside the procedure where it is declared, follow these steps:

1. Insert a new module into your project.
2. Type both of the following macros into this module:

   ```
   Sub Macro1() Dim x As Integer x = 10 MsgBox "x, as seen by Macro1 is " & x 'the next line runs Macro2 Macro2 End Sub Sub Macro2() MsgBox "x, as seen by Macro2 is " & x End Sub
   ```

3. Run **Macro1**.

   You receive the following message:

   > x, as seen by Macro1 is 10

4. Click **OK**.

   You receive the following message:

   > x, as seen by Macro2 is

5. Click **OK**.

Macro2 does not display a value for the variable x because the variable x is local to **Macro1**.

Private and public module-level scope

You can define variables in the declarations section of a module (at the top of a module, above all sub procedures), and set the scope of your variable by using the **Public** statement, the **Dim** statement, or the **Private** statement. If you put the **Public** statement in front of your variable, your variable will be available to all the macros in all the modules in the project. If you put either the **Dim** statement or the **Private** statement in front of your variable, your variable is available only to macros in the module where it is being declared.

To see the difference between the **Public** statement and the **Dim** statement, follow these steps:

1. Save and close any open workbooks and then open a new workbook.
2. Start the Visual Basic Editor.
3. Insert a module into your project.
4. Type the following code into this module:

   ```
   Public x As Integer Sub Macro_1a() x = 10 MsgBox x Macro_1b End Sub Sub Macro_1b() x = x * 2 MsgBox x Macro2 End Sub
   ```

5. Insert another module into your project.
6. Type the following code into this module:

   ```
   Sub Macro2() x = x * 3 MsgBox x End Sub
   ```

7. Run the **Macro_1a** macro in the first module.

   With the variable x declared as "Public x As Integer", all three macros in the project have access to the value of x. The first message box displays a value of 10. The second message box displays a value of 20 (because x is multiplied by 2 in **Macro_1b**). The third message box displays a value of 60 (because the value of x was changed to 20 in **Macro_1b** and then it was multiplied by 3 in **Macro2**).
8. Change the declaration line in the first module from:

   ```
   Public x As Integer
   ```

   to:

   ```
   Dim x As Integer
   ```

9. Run the **Macro_1a** macro.

   With the variable x declared as "Dim x As Integer", only the macros in the first module have access to the value of x. So the first message box displays a value of 10, the second message box displays a value of 20, (because x is multiplied by 2 in **Macro_1b**) and the third message box displays a value of 0 (because **Macro2** does not see the value of x and the uninitialized value of zero is used by **Macro 2**).
10. Change the declaration line in the first module from:

    ```
    Dim x As Integer
    ```

    to:

    ```
    Private x As Integer
    ```

11. Run the **Macro_1a** macro.

The same message boxes are displayed by using the **Private** statement scope as they were using the **Dim** statement. The variable x has the same scope, private to the module where it is declared.

Note If you want the scope of your variable to be limited to the module where it is declared, use the **Private** statement instead of the **Dim** statement. They both achieve the same effect, but the scope is clearer when you read the code if you use the **Private** statement.

Lifetime of a variable

The time during which a variable retains its value is known as its lifetime. The value of a variable may change over its lifetime but it will retain a value. Also, when a variable loses scope, it no longer has a value.

Initialize the value of a variable

When you run a macro, all the variables are initialized to a value. A numeric variable is initialized to zero, a variable length string is initialized to a zero-length string (""), and a fixed length string is filled with the ASCII code 0. Variant variables are initialized to Empty. An Empty variable is represented by a zero in a numeric context and a zero-length string ("") in a string context.

Procedure-level variables

If you have a variable that is declared in a macro by using the **Dim** statement, the variable retains its value as long as the macro is running. If this macro calls other macros, the value of the variable is retained (not available to the other macros though) as long as these other macros are also running.

To demonstrate how procedure-level variables work, follow these steps:

1. Insert a new module into your project.
2. Type both of the following macros into this module:

```
Sub Macro1() 'set x as a procedure level variable Dim x As Integer MsgBox "the initialized value of x is " & x x = 10 MsgBox "x is " & x 'the next line runs Macro2 Macro2 MsgBox "x is still " & x End Sub Sub Macro2() MsgBox "x, as seen by Macro2 is " & x End Sub
```

3. Run **Macro1**.

   You receive the following message:

   the initialized value of x is 0

4. Click **OK**.

   You receive the following message:

   x is 10

5. Click **OK**.

   You receive the following message:

   x, as seen by Macro2 is

6. Click **OK**.

   Macro2 does not display a value for the variable x because the variable x is local to **Macro1**. You receive the following message:

   x is still 10

7. Click **OK**.
8. Run **Macro1**.

You receive the same messages that are described in steps 3 through 6 because as soon as **Macro1** stopped running in Step 6, the value of the variable x was lost. Therefore, when you rerun **Macro1** in Step 7, the first message shows the value of x as zero (the initialized value).

Static keyword

If a procedure-level variable is declared by using the Static keyword, the variable retains its value until your project is reset. Therefore, if you have a static variable, the next time that you call your procedure, the static variable is initialized to its last value.

To see how the Static keyword works, follow these steps:

1. Change the code in **Macro1** to:

```
Sub Macro1() 'set x as a procedure level variable Static x As Integer MsgBox "the initialized value of x is " & x x = x + 10 MsgBox "x is " & x End Sub
```

2. Run **Macro1**.

   You receive the following message:

   the initialized value of x is 0

3. Click **OK**.

   You receive the following message:

   x is 10

4. Click **OK**.
5. Run **Macro1**.

   You receive the following message:

   the initialized value of x is 10

6. Click **OK**.

   You receive the following message:

   x is 20

7. Click **OK**.

The values that appear in the messages are different the second time because the variable x is declared as a static variable and the variable retains its value after you run **Macro1** the first time.

Note If you have a module-level variable, its lifetime is the same as if it were a static procedure-level variable.

To verify the lifetime of a module-level variable, follow these steps:

1. Change the code in the module that contains **Macro1** to the following:

```
Dim x As Integer 'create a module-level variable Sub Macro1() MsgBox "the initialized value of x is " & x x = x + 10 MsgBox "x is " & x End Sub
```

2. Run **Macro1**.

   You receive the following message:

   the initialized value of x is 0

3. Click **OK**.

   You receive the following message:

   x is 10

4. Click **OK**.
5. Run **Macro1**.

   You receive the following message:

   the initialized value of x is 10

6. Click **OK**.

   You receive the following message:

   x is 20

7. Click **OK**.

The values that appear in the messages are different the second time because the variable x is declared as a static variable and it retains its value after you run **Macro1** the first time.

Reset a project to reset variables

If you want to reset the value for a static variable or for a module-level variable, click the **Reset** button on the **Standard** toolbar, or click **Reset** on the **Run** menu.

If you do this for the **Macro1** project and then rerun **Macro1**, the value of the variable x is initialized back to zero and you receive the first message:

the initialized value of x is 0

For more information, click the following article number to view the article in the Microsoft Knowledge Base:
843145 (http://support.microsoft.com/kb/843145/) Description of Excel sub-procedures in Visual Basic for Applications (Arrays)

back to the top

Properties

Article ID: 843144 - Last Review: September 19, 2011 - Revision: 4.0

APPLIES TO

- Microsoft Office Excel 2007
- Microsoft Excel 2002 Standard Edition
- Microsoft Excel 2000 Standard Edition
- Microsoft Excel 97 Standard Edition

Keywords: kbvba kbprogramming kbinfo KB843144