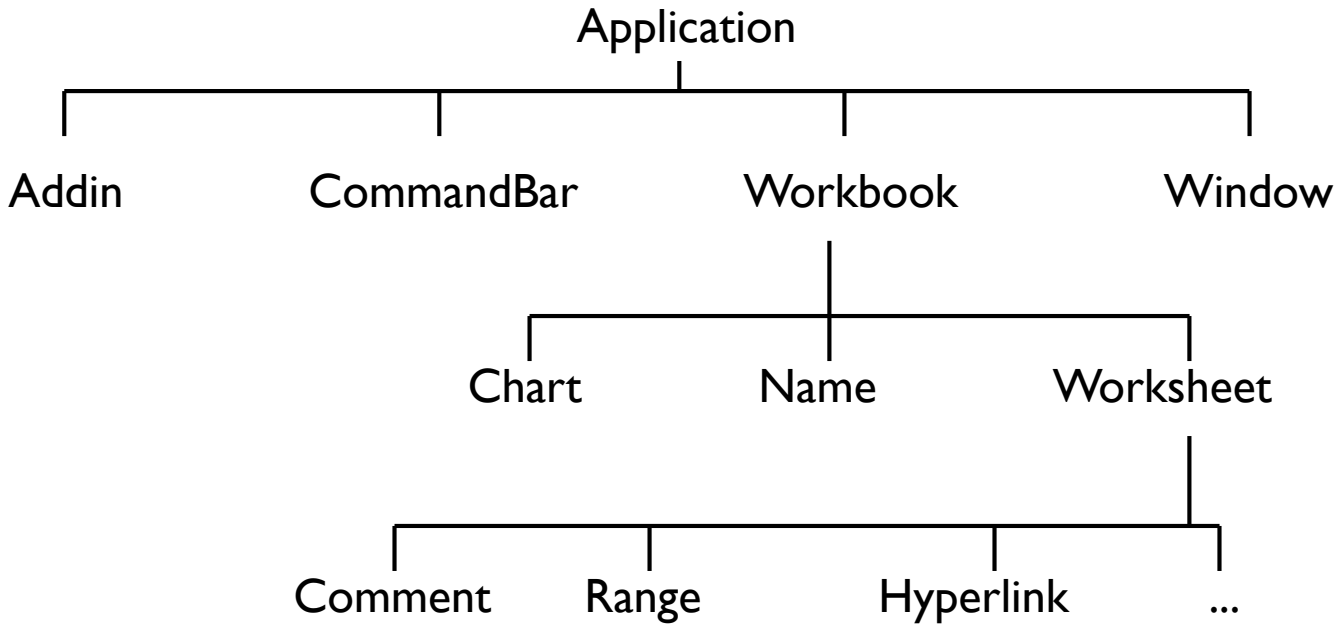
Excel
VBA programming
Visual Basic for Applications

Herve Hocquard

http://www.labri.fr/perso/hocquard

THE OBJECT MODEL IN VBA

- An object consists of properties and methods associated with it.
- The existing objects are constituted in hierarchy (composition relation).



- **Objects:** VBA manipulates objects contained in its host application. (In this case, Excel is the host application.) Excel provides you with more than 100 classes of objects to manipulate.

  Examples of objects include a workbook, a worksheet, a range on a worksheet, a chart, and a shape. Many more objects are at your disposal, and you can use VBA code to manipulate them. Object classes are arranged in a hierarchy.

- **Objects** also can act as containers for other objects. For example, Excel is an object called **Application**, and it contains other objects, such as **Workbook objects**. The **Workbook object** contains other objects, such as **Worksheet objects** and **Chart objects**. A **Worksheet object** contains objects such as **Range objects**, **PivotTable objects**, and so on. The arrangement of these objects is referred to as Excel's object model.

- **Collections.** A key concept in VBA programming is collections. A collection is a group of objects of the same class, and a collection is itself an object. As I note earlier, Workbooks is a collection of all Workbook objects currently open. Worksheets is a collection of all Worksheet objects in a particular Workbook object.

  You can work with an entire collection of objects or with an individual object in a collection. To reference a single object from a collection, you put the object's name or index number in parentheses after the name of the collection, like this: Worksheets("Sheet1")

If Sheet1 is the first worksheet in the collection, you could also use the following reference: Worksheets(1)

You refer to the second worksheet in a Workbook as Worksheets(2), and so on.

There is also a collection called Sheets, which is made up of all sheets in a workbook, whether they're worksheets or chart sheets. If Sheet1 is the first sheet in the workbook, you can reference it as follows: Sheets(1).

- **Object hierarchy.** When you refer to an object, you specify its position in the object hierarchy by using a period (also known as a dot) as a separator between the container and the member. For example, you can refer to a workbook named Book1.xlsx as Application.Workbooks(''Book1.xlsx'')
  - This code refers to the Book1.xlsx workbook in the Workbooks collection. The Workbooks collection is contained in the Excel Application object. Extending this type of referencing to another level, you can refer to Sheet1 in Book1 as Application.Workbooks(''Book1.xlsx'').Worksheets(''Sheet1'')
  - You can take it to still another level and refer to a specific cell as follows:

    Application.Workbooks(''Book1.xlsx'').Worksheets(''Sheet1'').Range(''A1'')

Most of the time, however, you can omit the Application object in your references because it is assumed.

- **Active objects.** If you omit a specific reference to an object, Excel uses the active objects. If Book1 is the active workbook, the preceding reference can be simplified as If the Book1 object is the active workbook, you can even omit that object reference and use this:

  ```
  Worksheets("Sheet1").Range("A1")
  ```

  And — I think you know where I'm going with this — if Sheet1 is the active worksheet, you can use an even simpler expression: `Range("A1")`.

Contrary to what you might expect, Excel doesn't have an object that refers to an individual cell that is called **Cell**. A single cell is simply a **Range** object that happens to consist of just one element.

Simply referring to objects (as in these examples) doesn't do anything. To perform anything meaningful, you must read or modify an object's properties or specify a method to be used with an object.

- **Objects properties.** Every object has properties. A property can be thought of as a setting for an object. For example, a **Range** object has properties such as Value and Address. A **Chart** object has properties such as HasTitle and Type. You can use VBA to determine object properties and also to change them. Some properties are read-only properties and can't be changed by using VBA.

  You refer to properties by combining the object with the property, separated by a dot. For example, you can refer to the value in cell A1 on Sheet1 as

```
Worksheets(''Sheet1'').Range(''A1'').Value
```

You can write VBA code to display the Value property or write VBA code to set the Value property to a specific value. Here's a procedure that uses the VBA MsgBox function to pop up a box that displays the value in cell A1 on Sheet1 of the active workbook:

```
Sub ShowValue()
    Msgbox Worksheets(''Sheet1'').Range(''A1'').Value
End Sub
```

The code in the preceding example displays the current setting of the Value property of a specific cell: cell A1 on a worksheet named Sheet1 in the active workbook. Note that if the active workbook doesn't have a sheet named Sheet1, the macro generates an error.

What if you want to change the Value property? The following procedure changes the value displayed in cell A1 by changing the cell's Value property:

```
Sub ChangeValue()
    Msgbox Worksheets(''Sheet1'').Range(''A1'').Value=123.45
End Sub
```

After executing this routine, cell A1 on Sheet1 has the value 123.45.

You may want to enter these procedures in a module and experiment with them.

Keep in mind that you can read the Value property only for a single-cell Range object. However, your code can write to the Value property for a multicell Range object. In the following statements, the first one is valid, and the second one is not:

```
Range(''A1:C12'').Value = 99
MsgBox Range(''A1:C12'').Value
```

Most objects have a default property. For a Range object, the default property is Value. Therefore, you can omit the .Value part from the preceding code, and it has the same effect. However, it's usually considered good programming practice to include the property in your code, even if it is the default property.

The statement that follows accesses the HasFormula and the Formula properties of a Range object:

```
If Range(''A1'').HasFormula Then MsgBox Range(''A1'').Formula
```

I use an If-Then construct to display a message box conditionally: If the cell has a formula, then display the formula by accessing the Formula property. If cell A1 doesn't have a formula, nothing happens.

The Formula property is a read-write property for only single-cell Range objects. For multicell Range objects, it's write-only. The following statement enters a formula into a range of cells:

```
Range(''A1:D12'').Formula = ''=RAND()*100''
```

- **VBA variables.** You can assign values to VBA variables. Think of a variable as a name that you can use to store a particular value. To assign the value in cell A1 on Sheet1 to a variable called Interest, use the following VBA statement:

```
Interest = Worksheets(''Sheet1'').Range(''A1'').Value
```

- **Object methods.** In addition to properties, objects also have methods. A method is an action that you perform with an object. Here's a simple example that uses the Clear method on a Range object. After you execute this procedure, A1:C3 on Sheet1 is empty and all cell formatting is removed.

```
Sub ZapRange()
    Worksheets (''Sheet1'').Range(''A1:C3'').Clear
End Sub
```

If you'd like to delete the values in a range but keep the formatting, use the **ClearContents** method of the Range object.

```
Range(''A1:C3'').ClearContents
```

Most methods also take arguments to define the action further. Here's an example that copies cell A1 to cell B1 on the active sheet by using the Copy method of the Range object. In this example, the Copy method has one argument (the destination of the copy).

```
Sub CopyOne()
    Range(''A1'').Copy Range(''B1'')
End Sub
```

An issue that often leads to confusion among new VBA programmers concerns arguments for methods and properties. Some methods use arguments to further clarify the action to be taken, and some properties use arguments to further specify the property value. In some cases, one or more of the arguments are optional.

If a method uses arguments, place the arguments after the name of the method, separated by commas.

If the method uses optional arguments, you can insert blank placeholders for the optional arguments. Later in this sidebar, I show you how to insert these placeholders.

Consider the **Protect method** for a workbook object. Check the Help system, and you'll find that the Protect method takes three arguments: password, structure, and windows. These arguments correspond to the options in the Protect Structure and Windows dialog box.

If you want to protect a workbook named MyBook.xlsx, for example, you might use a statement like this:

```
Workbooks(''MyBook.xlsx'').Protect ''xyzzy'', True, False
```

In this case, the workbook is protected with a password (argument 1). Its structure is protected (argument 2) but not its windows (argument 3).

If you don't want to assign a password, you can use a statement like this:

```
Workbooks(''MyBook.xlsx'').Protect , True, False
```

Note that the first argument is omitted and that I specified the placeholder by using a comma.

Another approach, which makes your code more readable, is to use named arguments. Here's an example of how you use named arguments for the preceding example:

```
Workbooks(''MyBook.xlsx'').Protect Structure:=True, Windows:=False
```

Using named arguments is a good idea, especially for methods that have many optional arguments and also when you need to use only a few of them. When you use named arguments, you don't need to use a placeholder for missing arguments.

For properties (and methods) that return a value, you must use parentheses around the arguments. For example, the Address property of a Range object takes five optional arguments. Because the Address property returns a value, the following statement isn't valid because the parentheses are omitted:

```
MsgBox Range("A1").Address False  'invalid
```

The proper syntax for such a statement requires parentheses, as follows:

```
MsgBox Range("A1").Address(False)
```

You can also write the statement using a named argument:

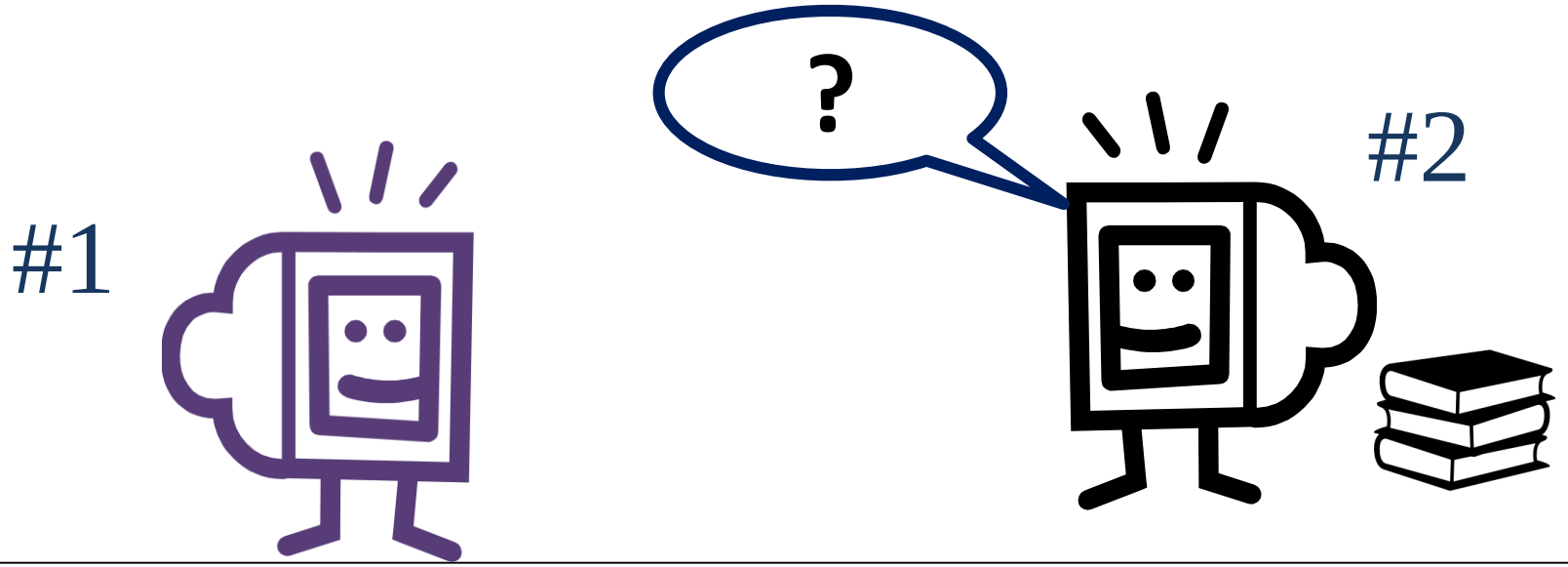
```
MsgBox Range("A1").Address(rowAbsolute:=False)
```

These nuances will become clearer as you gain more experience with VBA.

- **Standard programming constructs:** VBA also includes many constructs found in modern programming languages, including arrays, conditional statements, and loops.

- **Events:** Some objects recognize specific events, and you can write VBA code that is executed when the event occurs. For example, opening a workbook triggers a Workbook_Open event. Changing a cell in a worksheet triggers a Worksheet_Change event.

- Believe it or not, the preceding section pretty much summarizes what VBA is all about and how it works with Excel. Now you just need to learn the details…



In this analogy, I compare Excel with a fast-food restaurant chain.

The basic unit of Excel is a Workbook object.

In a fast-food chain, the basic unit is an individual restaurant.

With Excel, you can add workbooks and close workbooks, and the set of all the open workbooks is known as Workbooks (a collection of Workbook objects).

Similarly, the management of a fast-food chain can add restaurants and close restaurants — and all the restaurants in the chain can be viewed as the Restaurants collection — a collection of Restaurant objects.

An Excel workbook is an object, but it also contains other objects, such as worksheets, charts, VBA modules, and so on. Furthermore, each object in a workbook can contain its own objects. For example, a Worksheet object can contain Range objects, PivotTable objects, Shape objects, and so on.

Continuing with the analogy, a fast-food restaurant (like a workbook) contains objects, such as the Kitchen, DiningArea, and Tables (a collection). Furthermore, management can add or remove objects from the Restaurant object. For example, management can add more tables to the Tables collection. Each of these objects can contain other objects. For example, the Kitchen object has a Stove object, a VentilationFan object, a Chef object, a Sink object, and so on.

So far, so good. This analogy seems to work. Let's see whether I can take it further.

Excel objects have properties. For example, a Range object has properties such as Value and Name, and a Shape object has properties such as Width and Height.

Not surprisingly, objects in a fast-food restaurant also have properties. The Stove object, for example, has properties such as Temperature and NumberofBurners. The VentilationFan object has its own set of properties (TurnedOn, RPM, and so on).

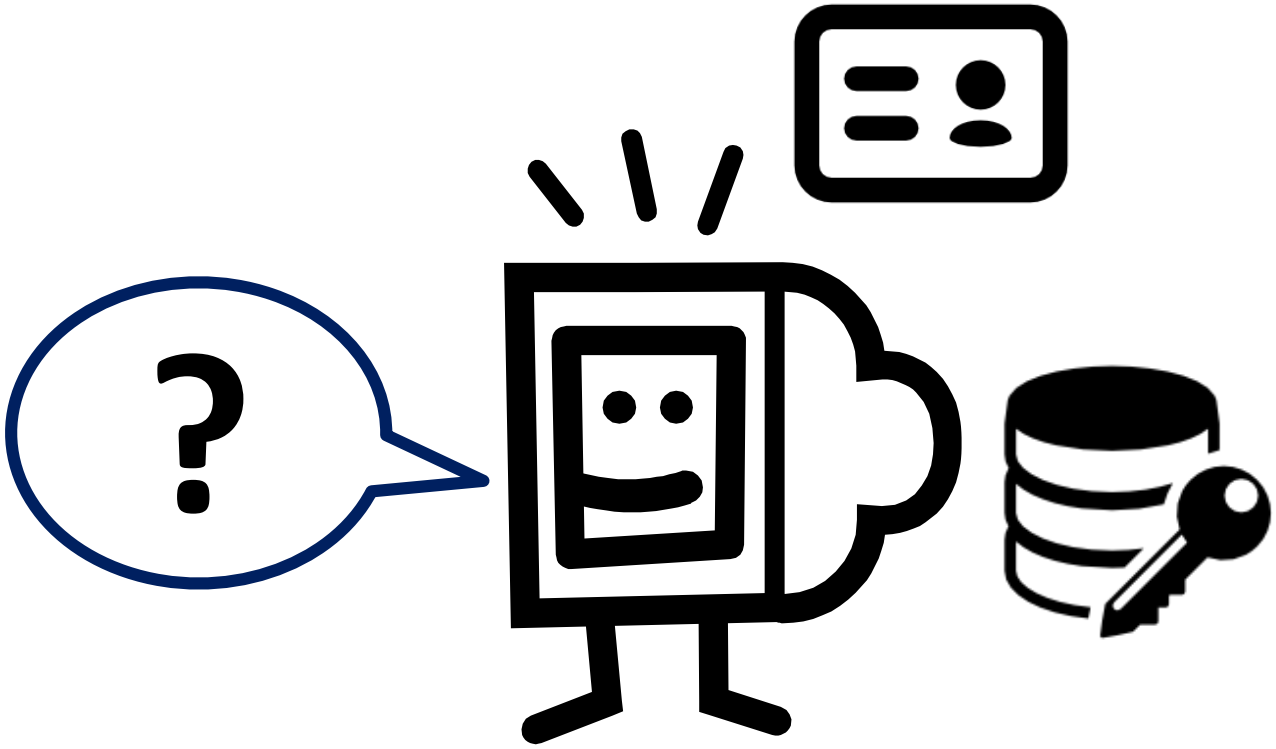
Besides properties, Excel's objects also have methods, which perform operations on objects. For example, the ClearContents method erases the contents of a Range object.

An object in a fast-food restaurant also has methods. You can easily envision a ChangeThermostat method for a Stove object, or a SwitchOn method for a VentilationFan object.

With Excel, methods sometimes change an object's properties. The ClearContents method for a Range object changes the Range Value property.

Similarly, the ChangeThermostat method on a Stove object affects its Temperature property.

With VBA, you can write procedures to manipulate Excel's objects. In a fast-food restaurant, the management can give orders to manipulate the objects in the restaurants. ("Turn on the stove and switch the ventilation fan to high.")



To be continued with VBA Language…

Thank you

Herve Hocquard(hocquard@labri.fr)

http://www.labri.fr/perso/hocquard/Teaching.html