APPENDIX

B LibreOffice Basic

INTRODUCTION

LibreOffice BASIC (or LO Basic in short) is the accompanying programming interface that is coupled with LO Calc to perform more advanced automation. It has as its backbone the BASIC programming language and added elements to permit it to interact with the component objects of Calc (and also other LibreOffice suite applications). BASIC is the name of the programming language, and not that it is easy or elementary. To access the **Integrated Development Environment** (IDE), select **Tools/Macros/Organize Macros/LibreOffice Basic** or simply key **Alt + F11**. The LO IDE is an updated version of the OpenOffice.org IDE. Other than BASIC, the LO IDE supports other programming languages such as BeanShell, JavaScript, and Python. We will only focus on LO Basic here.

Once you invoked the IDE, a separate LO **Basic Macros** dialog as shown in *Figure B-1* will appear.

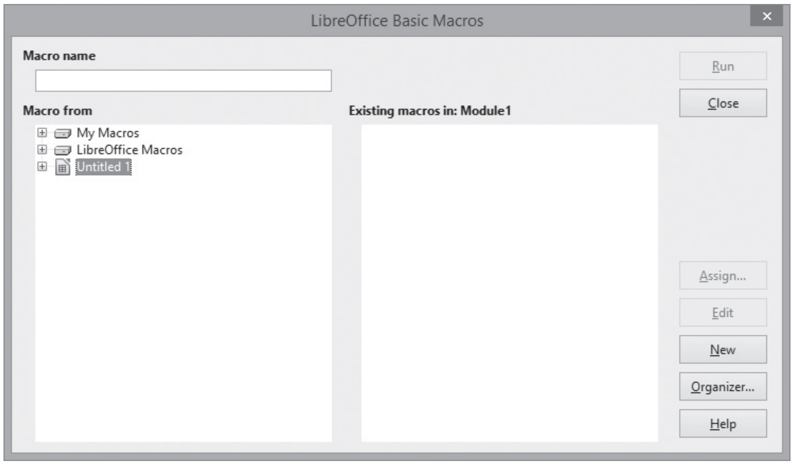


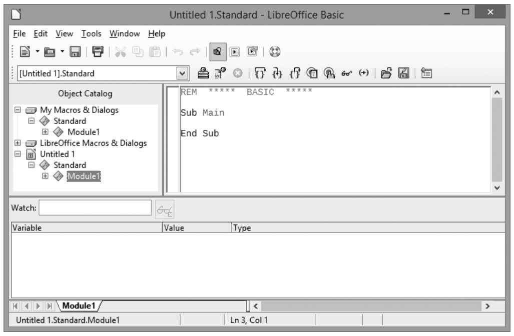
Figure B-1

LO Basic Macros Dialog

LO organizes the macros as follows:

- The highest level is the **Library Container**. Each library container holds at least one **Library** and each library holds **Modules**, if there are any. Each module contains macros, which can be viewed and edited in the spreadsheet software's IDE (also referred to as the LO Basic Editor), as shown in *Figure B-2*.
- The Calc application itself has two library containers: the **LibreOffice Macros** container holds all the macros that come with Calc, and the **My Macros** container has the macros that you have placed there. These macros are of course organized into libraries and modules as described above in these containers. The macros in OpenOffice.org Macros are utilities that you can use as study reference.
- **My Macros** library container comes by default with an empty **Standard** library. The macros in the **My Macros** modules stay with the LO application on your computer and are available whenever you open Calc. The LibreOffice Macros container goes a bit further in that the macros there are available to anyone using Calc on his or her computer.
- Every spreadsheet document is a library container. It contains by default a **Standard** library each. You can add more libraries. To do this, click **Organizer** in the **Basic Macros** dialog. In the **Organizer** dialog that now appears, select the **Library** tab and choose the library container you want to locate your library. Click **New**, provide the name of your library, and click **OK**. You can also similarly delete your created libraries. If you want to return to the Basic Macros dialog, click **Close**.
- In the Organizer dialog, you can add a module or dialog. Yes, you can make dialogs like those that you have been using thus far. To add a **module**, select the Module tab, click the library you want to locate your module, and click **New**. Enter next your module name into the dialog and click **OK**. To delete the module, click on the module and click **Delete**. You can similarly add or delete a dialog. Click **Close** to return to the Basic Macros dialog.
- In the Basic Macros dialog, select a library and click **New**. This will create a new macro in the first module of that library and **Basic Editor** will automatically open for you to edit this and other macros in the module. You can also move to other modules in the same library by clicking on the module tabs in the bottom-left corner of the IDE. To get back to the Basic Macros dialog from Basic Editor, select **Tools/ Macros/Organize Macros** from its menu.
- Another way to get to Basic Editor from the Basic Macros dialog is to select the module instead and click **Edit**. All this sounds rather confusing. The main thing to remember is that the Basic Macros dialog only allows you to interact with the macros while Organizer lets you interact with modules, dialogs, and libraries.

- Putting macros in the spreadsheet document's library container, as oppose to the LibreOffice Macros and My Macros containers of the application, means that your macros there are available with the spreadsheet document wherever it goes. This is particularly important if you intend to pass your spreadsheet document to another person or want to be able to work on it on another computer.



```
REM  *****  BASIC  *****

Sub Main

End Sub
```

Figure B-2

LO Basic Editor and Integrated Development Environment

Let us now review the controls in the Basic Editor:

- In the top panel, you will see the **Code Window**. This displays the macro codes in the active module. Basic Editor would display at least a Main subroutine. This is to show you the format of a subroutine. You can rename this subroutine and write codes in it. We will discuss more about writing codes later.
- Just below the main menu, there is a row of buttons that you can use to support your macro development work. The first button is the **Compile** button. As you amend your code, you may make syntax or "grammatical" mistakes in using LO Basic and Calc objects. Click **Compile** to request the IDE to compile your code and surface the syntax errors.
- The IDE provides an **Object Catalog** as a reference for programmers. Click **Object Catalog** to list all the objects at your disposal. As you review or type your macro codes, you can select a key word and key **F1** to get help on it.
- To run your macro, click **Run**. This will run the first macro in your module. If you are writing a few macros in the module, you can shift each in turn to the top and test them one at a time. After you are done, you can rearrange them to the final order.
- Before you run the macro you have written or edited, it will be wise to step through it first. This is to run the macro one line at a time, as if in slow motion. To do this, just click **Step Into**. A marker on the left of the code indicates which code line it is at.
- A macro may call another macro. If you step into a line with a macro call, it will take you into the called macro. You can skip this line when you come to it by clicking **Step Over** to stay in your current macro. After you are satisfied with your code review and testing, you can click **Step Out** to run the remaining lines of code and finish the test run.

- To run multiple lines of code, faster than one step at a time, you can introduce breakpoints to it. Move your cursor to a critical line of code and click **BreakPoint On/Off**. You can continue to mark as many breakpoints as you desire. You can switch each of them off by repeating the same steps. With breakpoints introduced, when you run your code, it will pause at the next breakpoint.
- The values of variables in your macro change as they are processed through the code. To better understand what is happening in your macro, you would want to watch the variable values during each stage of the processing. Select a variable that you want to watch and click **Enable Watch**. You can keep doing this for as many variables as you like, one at a time.
- In the bottom-left corner of the IDE, you will find the **Watch Window**. As you run or step through your macro, the variables and their current values will be displayed there. The window in the bottom-right corner shows which macro is currently running and the hierarchy of macros being called.
- When you edit and test your macro, remember to use the **Save** button often to save your work. Sometimes, for example, when the macro runs into an infinite loop and crashes the IDE, all of your work may be irrecoverably lost.

FURTHER REFERENCES

- Leong, 2014. LibreOffice and OpenOffice Basic Web Resources
  - http://isotope.unisim.edu.sg/users/tyleong/SpreadsheetModeling.htm#OOBasic
  - http://dl.dropboxusercontent.com/u/19228704/SpreadsheetModeling.htm#OOBasic

LO BASIC PRIMER

For non-BASIC programmers, macro recording is the fastest way to perform simple automation within Calc. It is the simplest form of programming in that you do not need to know how to write a single line of code. The codes will be automatically created. All you have to do is to record your keyboard and mouse actions as you work on a spreadsheet. The saved sequence of commands representing the executed keystrokes and mouse movement is a macro. It is also alternatively called a program, though this is generally codes that are directly written by a programmer. The recorded macro will be coded as a **Sub procedure** (or better known as subroutine) in a module. The actions, once recorded, can be repeated by running the macro.

However, in many cases, superfluous garbage codes are recorded as well. There are also some actions that cannot be coded by recording alone. Therefore, simple editing is sometimes needed to make the recorded macro work better. We will cover more on LO Basic programming in later sections when we discuss editing and writing the codes. For now, it is sufficient to know that macro recording is an extremely convenient way to capture macro codes that interact with Calc, saving users the need to think hard about how to code these spreadsheet actions.

Excel/VBA macro codes can also be similarly recorded. The code captures the resultant manipulations of Excel objects and therefore can be generalized with manual editing and addition of new lines of codes. The code recorded in LO Calc/Basic is however of a different nature. It records instead the user's interactions with the spreadsheet application (also known as the **Dispatch Framework**) and not the resultant manipulations of spreadsheet objects. The recorded LO macro codes, not quite in the LO Basic language, are as such not really easy to generalize and edit by programmers.

If you are an experienced Excel/VBA user, open your spreadsheet document (which can be in either Excel or LO format) in Excel and record your macros there. Remember to save it as an *xls* or *xlsm* file and close it. Open this Excel file in LO Calc and save the file in *ods* format. In the conversion, the LO spreadsheet software adds a *ThisWorkbook* module and also one module for every worksheet, among a few other such things as well. You will find an `Option VBASupport 1` statement at the top line of every module. This enables your VBA codes to work in the LO Basic environment. You may want to test if your macros still work since the VBA support for Calc, now progressively being implemented in their version releases, is not complete, and may not yet cover the more advanced features you are using.

Nothing beats learning from interesting and practical examples. There are many examples in the exercises and tools in this book. Look out for those Excel 2007 format workbooks with the *xlsm* extension in their file names and find their corresponding *ods* Calc files. Particularly worth an extra mention are the completed projects and tools in Chapter 9, especially the *Useful Macros* tool.

NOTES

Note 16: MACRO RECORDING Let us now do a simple macro recording and then attempt to read and understand the codes created. To enable macro recording, select **Tools/Options/LibreOffice/Advanced** in Microsoft Windows or **LibreOffice Preferences/ LibreOffice/Advanced** in MacOS.

Example: Record a macro to do automatic copying and pasting of values

1. In a new spreadsheet document, select **Tools/Macros/Record Macro**. Once selected, whatever actions you do with your mouse or keyboard, such as selecting a cell or scrolling up and down the worksheet, will be recorded as codes. A **Record M** ... dialog will appear when macro recording starts. It has a sole **Stop Recording** button. Unlike Excel, there is no **Relative Reference** button on this or in the main menu. How two different referencing effects are achieved will be discussed in the next step. Should you accidentally click the dialog-close (**X**) button in the upper right-hand corner, the dialog box will disappear and the recording process terminates without any negative effect.
2. Click cell B2 and then select cells B2:B5. Right-click and select **Copy** in the pop-up side menu (as shown in *Figure B-3*). This move is an absolute referenced move. If you use the keyboard arrows to reach cell B2, the recorded moves in the LO Calc macro will be relative referenced moves.
3. Select cell C2 by clicking the mouse on the cell, right-click, and then select **Paste**.
4. Click **Stop Recording** to end the process and the **Basic Macros** dialog as shown in *Figure B-4* will appear. Select the library you want your macro to be in, enter the macro name as *Macro1*, and click **Save**. Finally, click **Close** to close the Basic Macros dialog and end the process.

Once recorded, the macro can be used over and over again. To run the macro, select **Tools/Macros/Run Macro**, select the macro to run, and click **Run**.

Next, let us review and understand the codes you have recorded.

1. Key **Alt** + **F11** to launch the **Basic Editor** window.
2. Select **Module1** to show the codes in the code window, as shown in *Figure B-5*.
3. The macro begins with `Sub Macro1` and ends with `End  Sub`. These mark the beginning and the end of the subroutine.
4. Within the subroutine,
   a. Lines 2 to 10 = Setup the macro and declare the variables.
   b. Lines 11 to 24 = Cell range B2:B5 is selected.
   c. Lines 25 to 27 = The selection is copied.
   d. Lines 28 to 34 = Cell C2 is selected.
   e. Lines 35 to 37 = The copied selection is pasted with its upper-left corner in the selected cell in the active sheet.

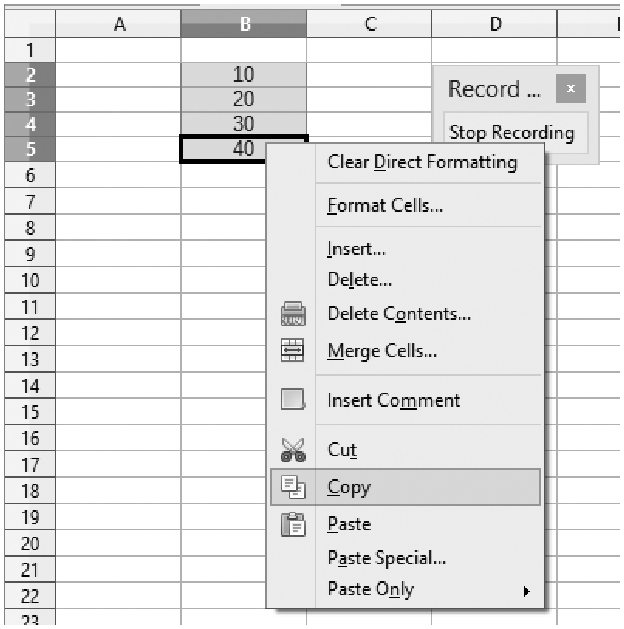


Figure B-3 Step 2 of Macro Recording

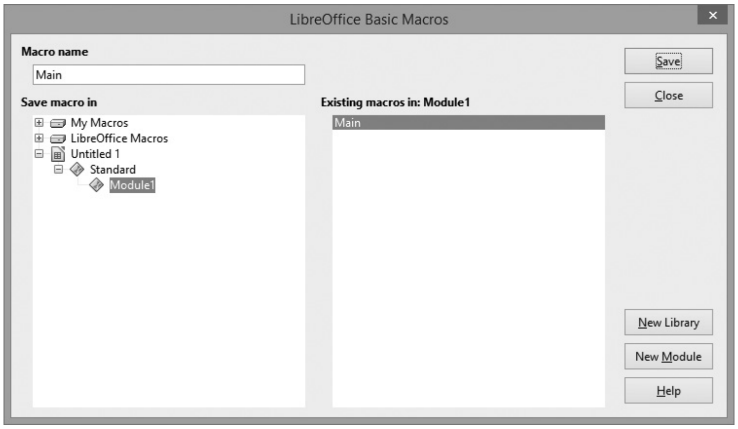
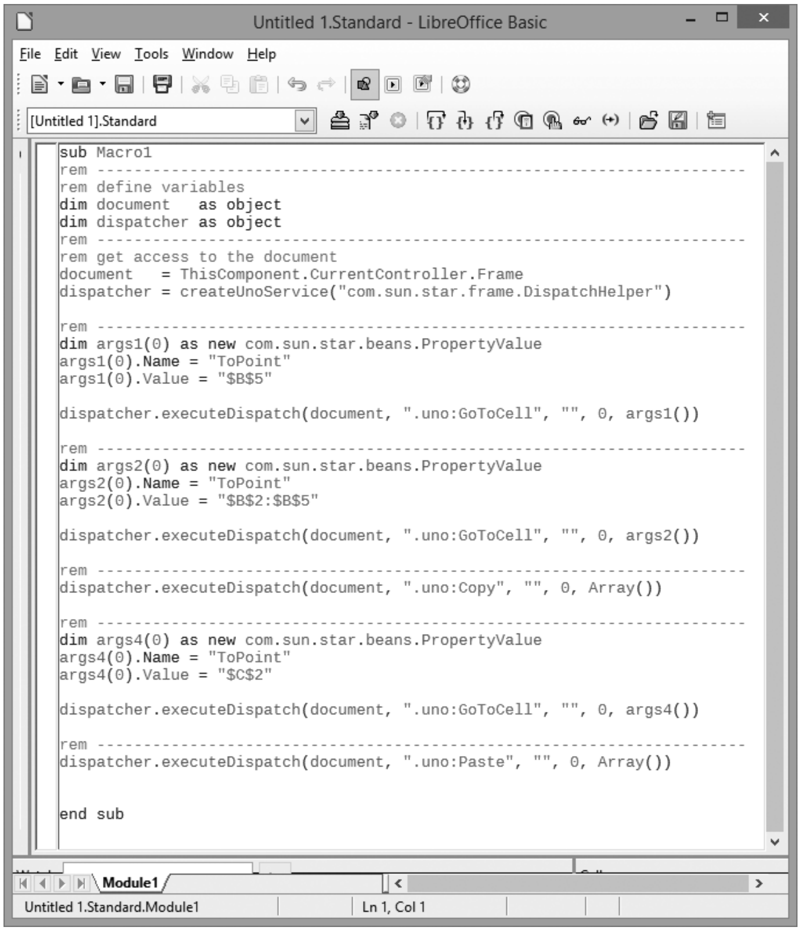


Figure B-4 Basic Macros Dialog

It is not easy to understand the codes generated. One way to speed up your learning of recorded macro code is to create mental connections between the mouse and keystroke actions you made and the recorded lines of code. As an additional check, you can reverse the process and **Step Into** the code, making it run one line at a time. To do this, ensure that your macro is the first in the module and select from the Basic Editor main menu **Debug/Step Into** (or key **F8**). Keep keying F8 to see how the marker moves from one line code to another, while watching the actions played out in the worksheet.

Figure B-5

Codes for Macro1


Untitled 1.Standard - LibreOffice Basic
File Edit View Tools Window Help
[Untitled 1].Standard
Module1
Untitled 1.Standard.Module1
Ln 1, Col 1

```
sub Macro1
rem ----------------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------------------------
dim args1(0) as new com.sun.star.beans.PropertyValue
args1(0).Name = "ToPoint"
args1(0).Value = "$B$5"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args1())

rem ----------------------------------------------------------------------
dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).Name = "ToPoint"
args2(0).Value = "$B$2:$B$5"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args2())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())

rem ----------------------------------------------------------------------
dim args4(0) as new com.sun.star.beans.PropertyValue
args4(0).Name = "ToPoint"
args4(0).Value = "$C$2"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args4())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Paste", "", 0, Array())


end sub
```

Absolute and Relative Referencing

We have covered absolute and relative cell referencing in *Note 3* of Appendix A. When you copy and paste a spreadsheet formula from one cell to another, it automatically adjusts the cell references in the formula unless they have been made **Absolute** using the $ sign. There is a similar concept in macro recording. Whenever you select a cell directly with your mouse, that selection would refer to that cell and is not interpreted as some relative movement from the last active cell. If you use the keyboard arrows instead, the moves recorded will be relative to the last position your cursor was at. You can record the same macro as in the earlier example, but this time, select cell B2 before recording the macro and use the keyboard arrows instead of mouse click to move the cursor. Study the difference between the two approaches.

1. Select **Tools/Macros/Record Macro**.
2. Select cells B2:B5 by holding down the **Shift** key and then using the down-arrow key. Right-click and select **Copy**.

3. Key the up-arrow a few times to get from cell B5 to B2, key the right-arrow to select cell C2, right-click, and then select **Paste**.
4. Click **Stop Recording** to end the process and the Basic Macros dialog will appear. Select the library you want your macro to be in, enter the macro name as *Macro2*, and click **OK**. Finally, click **Close** to close the Basic Macros dialog and end the process.

   Let us get back to **Basic Editor** and read the codes for Macro2:

```
sub Macro2
rem ----------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame

dispatcher =
   createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------

dim args1(1) as new com.sun.star.beans.PropertyValue
args1(0).Name = "By"
args1(0).Value = 1
args1(1).Name = "Sel"
args1(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoDown", "", 0,
   args1())
rem ----------------------------------------------------

dim args2(1) as new com.sun.star.beans.PropertyValue
args2(0).Name = "By"
args2(0).Value = 1
args2(1).Name = "Sel"
args2(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoDown", "", 0,
   args2())
```

```
rem -------------------------------------------------------
dim args3(1) as new com.sun.star.beans.PropertyValue
args3(0).Name = "By"
args3(0).Value = 1
args3(1).Name = "Sel"
args3(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoDown", "", 0,
   args3())

rem -------------------------------------------------------
dim args4(1) as new com.sun.star.beans.PropertyValue
args4(0).Name = "By"
args4(0).Value = 1
args4(1).Name = "Sel"
args4(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoDown", "", 0, args4())

rem -------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Copy", "", 0,
   Array())

rem -------------------------------------------------------
dim args6(1) as new com.sun.star.beans.PropertyValue
args6(0).Name = "By"
args6(0).Value = 1
args6(1).Name = "Sel"
args6(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoUp", "", 0,
   args6())

 rem -------------------------------------------------------
 dim args7(1) as new com.sun.star.beans.PropertyValue
 args7(0).Name = "By"
 args7(0).Value = 1
 args7(1).Name = "Sel"
 args7(1).Value = false
```

```
dispatcher.executeDispatch(document, ".uno:GoUp", "", 0,
   args7())

rem ----------------------------------------------------
dim args8(1) as new com.sun.star.beans.PropertyValue
args8(0).Name = "By"
args8(0).Value = 1
args8(1).Name = "Sel"
args8(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoUp", "", 0, args8())

rem ----------------------------------------------------
dim args9(1) as new com.sun.star.beans.PropertyValue
args9(0).Name = "By"
args9(0).Value = 1
args9(1).Name = "Sel"
args9(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoUp", "", 0,
   args9())

rem ----------------------------------------------------
dim args10(1) as new com.sun.star.beans.PropertyValue
args10(0).Name = "By"
args10(0).Value = 1
args10(1).Name = "Sel"
args10(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoRight", "",
   0, args10())

rem ----------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Paste", "", 0,
   Array())
end sub
```

- Lines 11 to 46 = A column of four cells from the current active cell is selected. Since cell B2 was selected before the macro was recorded, this means cells B2:B5 is selected.
- Lines 47 to 49 = The selection is copied.

- Lines 50 to 95 = A relative offset of three rows up and one column to the right from the last active cell B5 is selected. In Excel, the active cell after the last selection would be cell B2 and not B5.
- Lines 96 to 98 = The copied selection is pasted into the selected cell.

Relative referencing provides greater flexibility as new active cells (to be selected before the macro is run) can be any cell in any worksheet. For this demonstrated macro, it means that we can copy and paste any column of four cells to its immediate right.

The similar macro codes for the two (absolute and relative referenced) cases in Excel VBA would read:

```
Sub  Macro1()
  Range("B2:B5").Select
  Selection.Copy
  Range("C2").Select
  ActiveSheet.Paste
End  Sub

Sub  Macro2()
  ActiveCell.Range("A1:A4").Select
  Selection.Copy
  ActiveCell.Offset(0,1).Range("A1").Select
  ActiveSheet.Paste
End  Sub
```

These codes will also run in LO Basic if the additional `Option VBASupport 1` statement is added to the top of the module.

Quick Tip

- After each copy and paste sequence of operations, Calc leaves behind highlighted cell range selections in the worksheet. It is a good housekeeping practice to end your macro with the cursor placed in the appropriate cell and clear away all such unwarranted distractions. In the worksheet, just point your mouse to the end cell location and key **Esc**.
- The corresponding macro statements at the end of your Sub are as follows:

```
dispatcher=createUnoService("com.sun.star.frame.DispatchHelper")
oDoc = ThisComponent.CurrentController.Frame

dim args(0) as new com.sun.star.beans.PropertyValue
args(0).Name = "ToPoint"
args(0).Value = "$A$1"

dispatcher.executeDispatch(oDoc,".uno:GoToCell","",0,args())
```

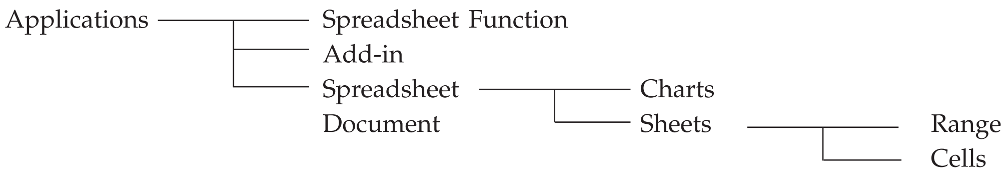
- Alternatively, without using the **Dispatch Framework**, the macro statements to signal the end of your Sub are as follows:

```
oSheet = ThisComponent.Sheets.getByName("Home")
ThisComponent.CurrentController.setActiveSheet(oSheet)

oCell = oSheet.getCellRangeByName("A1")
ThisComponent.CurrentController.select(oCell)
```

Note 17: LO BASIC GRAMMAR So far all the programming we have attempted is by macro recording, letting Calc automatically generate the codes. In order to perform more complex automation, we need to understand the LO Basic language and Calc objects a little more so that we can directly work with spreadsheet objects and not just the **Dispatch Framework** employed in the recorded codes.

LO Basic is an object-oriented programming language. Thus, it is useful to first understand the hierarchy of objects to better use the properties, methods, and events of the objects. The hierarchy of Calc objects is given as follows:



Properties

Each object has its own set of **Properties** that describes it. Some of the properties of the **Sheet** object include row height, column width, and font.

You can set the width of columns A to C to the respective dimensions using the code below:

```
oSheet = ThisComponent.CurrentController.ActiveSheet
oSheet.Columns(0).width = 1000  'Sets column A width = 1.0 cm
oSheet.Columns(1).width = 1500  'Sets column B width = 1.5 cm
oSheet.Columns(2).width = 2000  'Sets column C width = 2.0 cm
```

In general, any property can be set using the syntax below:

Object.Property = Value

Methods

Each object also has its own associated set of **Methods**, which are actions that can be performed on it. For example, one of the methods for the range object is **ClearContents**. You can clear the contents of a selected range E4:G7 using the code below:

```
oSheet = ThisComponent.CurrentController.ActiveSheet
oSheet.getCellByName("E4:G7").clearContents( _
 com.sun.star.sheet.CellFlags.VALUE _
 +com.sun.star.sheet.CellFlags.STRING _
 +com.sun.star.sheet.CellFlags.DATETIME)
```

In general, any method can be activated using the syntax below:

```
Object.Method
```

If you have been working with Excel VBA, here is a general difference between that and LO Basic which you should note. When an Excel object (e.g., cell range) is referenced in VBA, this object, unless explicitly coded, is assumed to be in the currently active Excel container (e.g., the active workbook and active worksheet). In LO Basic, each reference to a Calc object must be fully qualified; you have to specify the spreadsheet document and sheet explicitly as shown below:

```
oDoc = ThisComponent
oSheet = oDoc.CurrentController.ActiveSheet
numCars = oSheet.getCellRangeByName("C7").Value
```

If you want to get to another spreadsheet document and sheet, this can be done as shown below:

```
'Open your spreadsheet document ----
oSheet = ThisComponent.Sheets.getByName("Sheet1")
ThisComponent.CurrentController.setActiveSheet(oSheet)
numCars = oSheet.getCellRangeByName("C7").Value
'Close your spreadsheet document ----
```

You may need to remember first the active spreadsheet document, sheet, and cell, and return there after you are done so that it would not affect other macros. This is done as follows:

```
oActiveSheet = ThisComponent.CurrentController.ActiveSheet
oActiveCell = ThisComponent.CurrentSelection

'The earlier four lines of code here. ----

ThisComponent.CurrentController.setActiveSheet(oActiveSheet)
ThisComponent.CurrentController.select(oActiveCell)
```

It may be more efficient not to have to move the cursor around the spreadsheet documents and sheets and directly work with Calc objects as follows:

```
numCars =
ThisComponent.Sheets("Sheet1").getCellRangeByName("C7").Value
```

Quick Tip 1

The ′ sign is the shorthand for `REM`, the remark or comment key word. All statements that appear after the sign or the keyword will be regarded as mere comments and not code. You can sprinkle your macro with comments to make it easier to understand and read. The other best practice is to indent lines when they are part of another programming structure. In this case, you can see that all statements between `Sub` and `End  Sub` are indented to show that they are part of the subroutine. You can observe how they are used in other examples later in the chapter.

Similarly, when you record macros, values can only be transferred from one set of cells to another using **Copy** and **PasteSpecial** as shown in *Note 16*. When you edit the macro, rewrite it in LO Basic to work directly with Calc objects as shown below.

```
oSheet = ThisComponent.CurrentController.ActiveSheet
oRange1 = oSheet.getCellRangeByName("B2:B5")
oRange2 = oSheet.getCellRangeByName("C2:C5")
oRange2.DataArray = oRange1.DataArray
```

The above transfers values between two cell ranges. When there are only single cells involved, the code is even simpler:

```
oSheet.getCellRangeByName("C2").Value = _
  oSheet.getCellRangeByName("B2").Value
```

The underscore (_) is used to break a long code statement into separate rows for easier reading.

Other than transferring data values between cells, you can set formulas in any cell in the spreadsheet as demonstrated below:

```
oSheet.getCellRangeByName("A12").Formula = "=SUM(A2:A11)"
```

When the choice of cell uses depends on other values, use the getCellByPosition function as follows:

Example 1

```
r = 2 : c = 1
oSheet.getCellRangeByName("C3").Value = _
  oSheet.getCellByPosition(c,r).Value 'Referring to cell B3
```

Example 2

```
r = 4 : c = 3
oRange = oSheet.getCellRangeByName("L10:P20")
result = oRange.getCellByPosition(c,r).Value 'Referring to cell O14
```

The colon (:) is used to separate two statements so that they can be put on a single line. Conversely, the underscore (_) had been used to break a long code statement into separate rows for easier reading. You should be reminded that Calc uses (c, r) and not the more conventional (r, c) scheme and it starts counting from 0. Therefore in *Example 1* where the referenced context is the whole sheet, r = 0 refers to row 1, r = 1 refers to row 2, ... and c = 0 refers to column A, c = 1 refers to column B, etc. In *Example 2*, the context is the given range L10:P20. Here, r = 0 refers to row 10, r = 1 refers to row 11, ... and c = 0 refers to column L, c = 1 refers to column M, etc.

Sub and Function Procedures

A **Sub** procedure (or subroutine) is a set of codes which when executed performs a series of spreadsheet actions. Each recorded macro is a Sub. You will now learn how to write one. Insert a new **Module** by selecting **Tools/Macros/Organize Macro/OpenOffice Basic** in Calc or the **Basic Editor** to get to the **Basic Macros**. In this dialog, click **Organizer**. In the **Organizer** dialog, select the **Module** tab, select the library you want to locate the module, click **New**, provide the name of the module and click **OK**. Back in the Basic Macros dialog, select the module and click **Edit**. In the code window, clear away all other codes and type the following:

```
Sub SayHi
   MsgBox("Hi!")
End Sub
```

This is a simple Sub to prompt a greeting. To run it in the Basic Editor, click **Run BASIC.** Having fun yet?

A **Function** procedure is also a set of codes. However, its primary purpose is to return a result computed with the inputs offered to it. You may have used some of the given spreadsheet functions like **AVERAGE** and **SUM**, and soon you will be able to create more useful functions of your own. Let us begin by writing a simple function to compute the cube root of a number. In order for the function to return the computed value when the function is used, the result variable must bear the function's name, which in this case is *CubeRoot*.

In an empty space in the module you have just inserted, type in the following:

```
Function  CubeRoot(number)
   CubeRoot  =  number^(1/3)
End  Function
```

With this done, you can use it in the worksheet. Key in formula =CubeRoot(8) into a cell and see the number 2 appearing in it. All worksheets in the same spreadsheet document can use this function. The function you have created can also be used by other Sub and Function procedures in your spreadsheet document.

Though a Sub procedure does not directly return computed results, data values can be passed indirectly to and from it through the variable arguments specified within the brackets next to its name, or cells in the worksheets. A subroutine with arguments can only be called by another subroutine and not run from Calc directly as a macro since there is no way to pass the argument values to it that way. We will leave this as a future topic for you to explore on your own.

Variables and Declaration

In many programming software languages, variables to hold values, whether entered or computed, need to be defined first. Each variable must be of a data type, namely integer, real, text, Boolean, array, or object. Calc does not force us to declare all variables before use. Instead, it automatically creates a **Variant** variable type for each variable with type not declared. Although convenient, this implies that more memory storage is set aside for such variables. This bad programming practice is strongly discouraged.

To declare a variable to be of a certain data type, you use the following syntax:

```
Dim variableName As dataType
```

The full list of various data types and their details are given below:

Data Type	Storage Size	Range Values
Byte	1 byte	0 to 255
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
Single	4 bytes	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double	8 bytes	–1.79769313486231E308 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
Decimal	14 bytes	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point; +/–7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/–0.0000000000000000000000000001
Date	8 bytes	1 January 0100 to 31 December 9999
Object	4 bytes	Any object reference
String (fixed length)	Length of string	1 to approximately 65,535 characters
String (variable length)	10 bytes + string length	0 to approximately 2 billion
Variant (with numbers)	16 bytes	Any numerical value up to the range of a double
Variant (with characters)	22 bytes + string length	Same range as a variable string length

Source: Table adapted from Calc *Help*.

Here are some general rules you should follow when declaring variables.

- A variable declared in a procedure is only meaningful for that procedure (i.e., as a **Local** variable). Another variable of the same name declared within another procedure will be recognized as a different local variable applicable to that procedure only.
- Variables that are to be shared by all procedures within a module should be declared before the first procedure in the module.
- Variables that are shared by procedures in all modules and sheets in a project should be declared using **Public** in place of **Dim**. The declaration should be made before the first procedure in any module in the project. Sometimes, the **Const** keyword is also added after Public to declare the variable as unchanging and permit a value to be assigned to it at the declaration.

Quick Tip 2

- To force yourself to declare all variables used, put `Option Explicit` as the first statement at the very top of your module. With this, VBE will prompt an error whenever an undeclared variable is present.

- ○ Always choose the most suitable data type, one that uses the smallest number of bytes for the variable.

Here is a simple example to help you understand better.

```
Public x as Integer
Public Const gravity as Single = 9.8
Dim y as Long


Sub Mysub()
Dim z as Single
Static k as Integer
k = k + 1
...
End Sub
```

- *x* is declared as an integer variable using the **Public** keyword and can therefore be used by procedures in all modules.
- *gravity* is declared using the **Public Const** keywords as a constant parameter, with the value of 9.8 and can be used by procedures in all modules.
- *y* can be used by all procedures within this module because *y* is declared before the first procedure in the module.
- *z* can only be used within procedure *Mysub*.
- *k* is a static variable declared within the procedure, which means that it will retain its value even when current call of the procedure ends. This value is then used by the same procedure the next time it is called. This is useful, for example, when you need to track the number of times the procedure is run.

Quick Tip 3

For a typical declaration statement like `Dim p, q, r As Integer`, only *r* is declared as an integer, while *p* and *q* are variants. This is a common mistake among BASIC programmers.

Declaring Arrays

An array, more commonly known as a matrix, is a group of variables sharing a common name. Arrays can be 1-dimensional or multi-dimensional.

An example of a 1-dimensional array declared to store the identification number of 200 compact disks is:

```
Dim CD_ID(1 to 200) As Integer
```

An example of a 2-dimensional array declared to store the identification number of 1,000 compact disks is:

```
Dim CD_ID(1 to 10, 1 to 100) As integer
```

An array may be declared as a dynamic array which does not have a preset size. Its size can be set later in the procedure using the **ReDim** statement. A simple example is given below:

```
Dim MyArray( ) As Single
Dim ASize As Integer
ASize = oSheet.getCellRangeByName("A1").Value
ReDim MyArray(ASize)
```

In the above, *MyArray* is first declared with no size specified. Its size is read from cell A1. This array is then sized according to the value in cell A1.

Note 18: MORE PROGRAMMING Up until this point, you can write simple LO Basic programs in which all the lines of code are sequentially executed. However, there will be many instances where we would like the program to skip some steps or go directly to one set of steps or another, on satisfying or not satisfying a test condition, respectively. Here are some examples of the most useful ones.

The **GoTo** statement is used when you wish the program to go directly to the start of another block of codes. An example is given below:

```
Sub GotoDemo()
  Rating = InputBox("Enter rating (1 or 2): ")
  If Rating = 1 Then GoTo Ans1
  MsgBox("You have entered 2.")
  Exit Sub
Ans1:
  MsgBox ("You have entered 1.")
End Sub
```

Ans1 here is the label of a line location and it must be suffixed by the : sign. Serious programmers dislike using the GoTo statement because it makes the program unstructured and therefore difficult to follow.

If-Then-Else is one of the most useful statements, which allows the program to execute alternative codes depending on whether the test condition results in a TRUE or FALSE. If the result is TRUE, the codes following **Then** (up to the line containing the **Else** keyword if it exists, or up to **End If** if it does not) will be executed. Otherwise, the codes in the lines following **Else** (up to **End If**) will be executed.

```
If testCondition Then
  doSomethingWhenTrue
Else
  doSomethingWhenFalse
End If
```

An example is given below:

```
Sub IfThenElseDemo()
  Rating = InputBox("Enter rating (1 or 2): ")
  If rating = 1 Then
    MsgBox ("You have entered 1.")
  Else
    MsgBox ("You have entered 2.")
  End If
End Sub
```

Select-Case is useful when the test condition can result in more than two alternatives, thus requiring more paths for the codes to continue the operation.

```
Select Case variableName
  Case value1
    Statement set 1
  Case value2
    Statement set 2
  Case value3
    Statement set 3
End Select
```

An example is given below:

```
Sub SelectCaseDemo()
  Rating = InputBox("Enter rating 1, 2 or 3:")
  Select Case Rating
  Case 1
    MsgBox ("You have entered 1.")
    etc …
  Case 2
   MsgBox ("You have entered 2.")
   etc …
  Case 3
   MsgBox ("You have entered 3.")
   etc …
  End Select
End Sub
```

When you need the program to loop through a set of codes for some number of times, the **For-Next** statement will be very handy. The looping is controlled by a *counter* that will go from a *start* number to an *end* number, increasing by a *stepSize* after each execution of the loop. When not declared, the default step size is 1.

```
For counter = start to end [Step stepSize]
  statements …
Next counter
```

An example is given below:

```
Sub ForNextDemo()
  Dim j As Integer
  For j = 1 to 10
   MsgBox("Hi")
  Next j
End Sub
```

The program above displays the message "Hi" 10 times, using *j* as the counter.

A **Do-While** statement is useful when you need the program to loop through a set of codes until a test condition returns a FALSE. Do-While can be used in two slightly different methods.

Method 1: This method tests the condition first and executes the statement when the condition is tested TRUE. The program ends immediately when the test condition results in a FALSE.

```
Do While testCondition
  statements  …
Loop
```

Method 2: This alternative method executes the statement first and then tests the condition. Only when the condition is tested TRUE will the next iteration be executed. Similarly, the program ends when the test condition returns a FALSE. The main difference is that the codes in the loop in method 2 will be executed at least once, whereas this may be by-passed completely in method 1.

```
Do
  statements …
Loop While testCondition
```

The corresponding examples are as follows:

```
Sub DoWhileDemoMethod1()
  Dim j As Integer
    j = 1
  Do While j < 10
    MsgBox ("Hi")
    j = j + 1
  Loop
End Sub
```

Method 2:

```
Sub DoWhileDemoMethod2()
  Dim j As Integer
    j = 1
  Do
    MsgBox ("Hi")
    j = j + 1
  Loop While j < 10
End Sub
```

A **Do-Until** statement is similar to a Do-While statement except the former executes the codes until the test condition becomes TRUE. Again, there are two methods to program a Do-Until statement.

Method 1: This method tests the condition first and executes the codes when the test condition returns a FALSE. The program ends when the test condition returns a TRUE.

```
Do  Until  testCondition
  statements  …
Loop
```

Method 2: This method executes the codes once and then tests the condition. The program continues to loop as long as the condition returns a FALSE, and ends when it returns a TRUE.

```
Do
  statements …
Loop Until testCondition
```

Their corresponding examples are given below.

```
Sub DoUntilDemoMethod1()
  Dim j As Integer
    j = 1
  Do Until j = 10
    MsgBox ("Hi")
    j = j + 1
  Loop
End Sub

Sub DoUntilDemoMethod2()
  Dim j As Integer
    j = 1
  Do
    MsgBox ("Hi")
    j = j + 1
  Loop Until j = 10
End Sub
```

Quick Tip

- Your computer screen may flicker during the running of a macro by virtue of its speed.
- To reduce the flicker and also to speed up the macro, stop Calc from updating the screen by putting `ThisComponent.LockControllers` as one of the first statements in your Sub.
- You can reinstate the default option by putting `ThisComponent.UnlockControllers` as one of the last statements in your Sub. Do not forget to put this in your macro to reinstate the default option if it has been changed. Your worksheet cells may otherwise not refresh so dynamically in Calc.
- It is very important to reinstate `ThisComponent.UnlockControllers` at some point before the macro stop running and you are back in the spreadsheet. In Calc 4 and later versions especially, it will most likely affect the responsiveness of your spreadsheet. Calculations and Conditional Formatting in particular will not be updated with new inputs.

Note 19: EXTENSION AND ADD-INS **Add-in** functions and operations are **extensions** that can be included in Calc. There are also other kinds of extensions that will extend the LO Basic environment to give you extra features.

Solver and **Analysis Toolpak** are add-ins in Excel. You no longer really need the tool pack and Calc has a built-in Solver.

Quick Tip

- Some spreadsheet operations will prompt dialog boxes for you to select your response. To avoid such incidences during a macro run, choose the default option and add the following statement to your Sub before the statement that causes the pop-up:

```
ThisComponent.addActionLock
```

- It is a good practice to set it back to the default option by putting, at the next earliest possibility, the next statement:

```
ThisComponent.removeActionLock
```

Note 20: AUTOMATIC PROCEDURES AND EVENTS **Event handler** procedures are programs that are activated by interactive actions or events. Every object should have their associated events, such as **Change, Activate**, **BeforeRightClick,** and **SelectionChange**. In LO Basic, it appears that these can only be accessed using the UNO (Universal Network Objects), a topic beyond the scope of this book. The Calc application and spreadsheet documents' actions such as **Start Application**, **Close Application, Open Document, Close Document, Activate Document**, and **Deactivate Document,** on the other hand, can be assigned to macros in Calc itself.

To automatically execute a macro when an Excel workbook is opened, you include **Sub Workbook_Open** in T*hisWorkbook* in the VBA macros. An example of the macro codes that you can put in the procedure is shown below:

```
Private Sub Workbook_Open()
  Sheets("Home").Select
  Range("A1").Select
End Sub
```

The equivalent Sub procedure in LO Basic would be

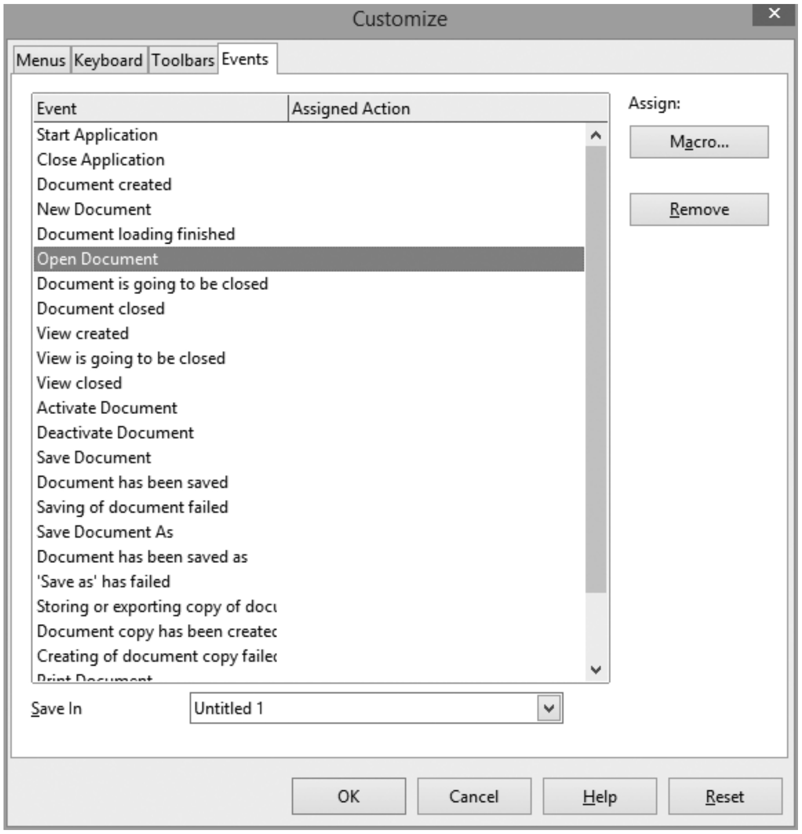
```
Sub SpreadsheetDocumentOpen
    oSheet = ThisComponent.Sheets.getByName("Home")
    ThisComponent.CurrentController.setActiveSheet(oSheet)

    oCell = oSheet.getCellRangeByName("A1")
    ThisComponent.CurrentController.select(oCell)
End Sub
```

In LO Calc Basic, you must assign a macro (Sub of any name such as the one above) to the **Open Document** event by selecting **Tools/Customize/Events**. This is a spreadsheet document event. To do the same as an application event, which means the macro will run no matter what document is being opened, you select **Tools/Macros/Organize Macros/ OpenOffice Basic/Assign**.... In the **Customize** dialog as shown in *Figure B-7*, select the **Event** tab and then the **Open Document** event. Click **Assign: Macro**, select the macro, and click **OK**. Since this an LO application level event, the macros you select can only come from the My Macros or OpenOffice.org Macros library containers. The dialog will only show these two library containers. Similarly, to remove the event assignment, click **Assign: Remove** in the **Customize** dialog. There are also other events listed in the Customize dialog that you can use.

Figure B-7

Customize Dialog



Event	Assigned Action
Start Application	
Close Application	
Document created	
New Document	
Document loading finished	
Open Document	
Document is going to be closed	
Document closed	
View created	
View is going to be closed	
View closed	
Activate Document	
Deactivate Document	
Save Document	
Document has been saved	
Saving of document failed	
Save Document As	
Document has been saved as	
'Save as' has failed	
Storing or exporting copy of doc	
Document copy has been create	
Creating of document copy faile	
Print Document	

Save In: Untitled 1

Whereas Excel/VBA has built-in events also for worksheets and other lower level actions, there is none given in LO Basic. For example, Excel's **Worksheet_Activate** event subroutine automatically runs each time the worksheet that has this subroutine is activated. To do the same in Calc, you will have to use the more general LO **Listener** event activation feature. That is, you must create your own *Worksheet_Activate* event subroutine.

An LO **Listener** is a process that runs in the background, waiting for a particular event to happen. To create an LO listener, you must

- Define a global listener object.
- Write a subroutine that starts the listener (if required, another subroutine to end it).
- Write a subroutine that the listener will run when the event is activated.

The specific code examples are shown below:

Example 1 (similar to Excel/VBA's Worksheet_Activate event subroutine)

```
Global oListener As Object

Sub AddListener
   listenerName = "com.sun.star.beans.XPropertyChangeListener"
   oListener = createUnoListener("OOo_", listenerName)

   oCurControl = ThisComponent.CurrentController
   oCurControl.addPropertyChangeListener("ActiveSheet",oListener)
End Sub

Sub RemoveListener
    oCurControl = ThisComponent.CurrentController
    oCurControl.removePropertyChangeListener("ActiveSheet",oListener)
End Sub

Sub OOo_PropertyChange(oEvent)
    Msgbox ("Property Change Listener is working.")
    'More statements to be provided by you
End Sub
```

Example 2 (similar to Excel/VBA's **Worksheet_Change** and **Worksheet_Calculate** event subroutines)

```
Global oListener As Object

Sub AddListener
  listenerName = "com.sun.star.chart.XChartDataChangeEventListener"
  oListener = CreateUnoListener("OOo_", listenerName)
```

```
      oSheet  =  ThisComponent.Sheets.getByName("Sheet1")
      oRange  =  oSheet.getCellRangeByName("E5")
      oRange.addChartDataChangeEventListener(oListener)
   End  Sub

   Sub  OOo_ChartDataChanged
      oSheet  =  ThisComponent.Sheets.getByName("Sheet1")
      oCell  =  oSheet.getCellRangeByName("E5")

      nValue  =  oCell.Value
      Msgbox  "Value  ="  &nValue
      Msgbox  "Data  Change  Listener  is  working"
      'More  statements  to  be  provided  by  you
   End  Sub
```

It is most crucial that the first parameter passed (in this case, the string *OOo_*) to the createUnoListener method be used as the prefix for the name of the subroutine that the listener is to run. The subroutine's name must end with specific listener reserved names such as *PropertyChange and ChartDataChanged*. This thus gives us subroutines *OOo_PropertyChange* and *OOo_ChartDataChanged*. The *oListener* is the listening object and *oEvent* (an OOo reserved object name) is the response object to be passed back to triggered subroutine for it to retrieve any event property.

The *AddListener* subroutine can be automatically activated by adding into the *SpreadsheetDocumentOpen* subroutine, explained earlier, the following line of code:

```
   AddListener
```

If there are more listeners, their *Add* subroutines can be named as *AddListener1*, *AddListener2*, and so on. These can then be selectively added to the *SpreadsheetDocumentOpen or WorkbookOpen* subroutine as well.

Do not forget to run the *RemoveListener* subroutine when the Listener is no longer needed. In our testing, we found that the *Remove* subroutines generally do not work. The listeners are however automatically removed when the workbook is closed and are not found to be listening when the workbook is next opened. This is, of course, unless they are automatically invoked by the *SpreadsheetDocumentOpen* subroutine.

The macro code you want to run upon listener event activation may depend on which worksheet (or cell) is changed, selected, or activated. So you may have to check if the active sheet (or cell) is indeed the targeted one, as shown by the examples below:

Example 1

```
oActiveSheet = ThisComponent.CurrentController.ActiveSheet
oActiveCell = ThisComponent.getCurrentSelection

If oActiveSheet.Name <> “Model” Then Exit Sub
```

Example 2

```
oActiveSheet = ThisComponent.CurrentController.ActiveSheet
oActiveCell = ThisComponent.getCurrentSelection

Select Case oActiveSheet.Name
Case "Sheet1"
   oRange1 = oSheet.getCellRangeByName("C4:F5")
   If Intersect(oActiveCell, oRange1) Is Nothing Then
         Exit Sub
   Else
      Statement set 1
   End If

Case "Sheet2"
   If oActiveCell = oSheet.getCellRangeByName("L9") Then
      Statement set 2
   Else
      Statement set 3
   End If

Case "Sheet3"
      Statement set 4
End Select
```

Before you do anything that ambitious, try testing with simple 1-line code such as

```
Sub OOo_PropertyChange(oEvent)
 Msgbox (“This listener is working:” & str(oEvent.Source.Value)
End Sub
```

There are possibly other types of listeners available. You may want to find out their names and what they do. Listener events can be difficult to implement and they can do damage to your workbook if not properly done. The two examples above are collated from what we found in the generally available resources and are by no means well tested. Do proceed with caution.

Note 21: RUN-TIME ERROR HANDLING Imagine that you wrote a simple subroutine to compute the square root of a user-input value. In order to ensure that the user has entered a positive value, you need to test the input value before executing the computation. You test by checking if the value is positive and if it is numeric. Alternatively, you can use a general error handling statement to trap all possible errors whenever they occur.

```
Sub SquareRootDemo()
    On Error Goto BadEntry
    Num = InputBox ("Enter a value: ")
    If Num = "" Then Exit Sub
    ThisComponent.getCurrentSelection.Value = Sqrt(Num)
    Exit Sub
BadEntry:
    MsgBox ("Make sure you enter a positive numeric value")
End Sub
```

This example allows the subroutine to proceed straight to the error message whenever the user input threatens to trigger a computation error. The `Exit Sub` statement jumps to the end of the subroutine when it successfully completes its computation.

Note 22: SPREADSHEET PROGRAMMING APPROACH There are really four approaches to programming a spreadsheet. The first is to use only spreadsheet functions and features (this topic is covered in Appendix A). Since its inception, spreadsheet application software has come a long way. Features that were once only available in programming languages are now present and regularly used in spreadsheets. The basic ones permit one variable (as represented by a cell) to take values from other variables, use of If-Then-Else logical branching, and multiple-stage computations with relative cell referencing. More recently, there are random variables, iterative, or recursive computations, lookups, and automated computation (i.e., loops in the form of Multiple Operations). Working on a spreadsheet workbook is really programming work, though many do not see it as such.

The second approach is to record mouse and keyboard actions as macros, and run these macros as automated steps in the spreadsheet operations. The steps are visible to the user by default, though it can be masked away to speed up operations. All the calculations are done in the sheets and so the user can vividly review the interactions between variables.

The third approach is to extend the abilities of macros by adding LO Basic codes to do what mouse and keyboard actions on the sheets cannot achieve. In addition, the recorded macros can be tidied up and made more efficient, for example, by removing sheet selections, cell selections, and copy-paste operations, replacing them with codes that work directly with Calc objects.

The fourth and final approach is to write subroutines and functions using the LO Basic language, with minimal use of spreadsheet features, other than to read data and write results. This is no different than normal computer programming, except now the sheets become data storage and reporting pages. The computations are all done in the lines of LO Basic codes and therefore the user must be able to read the computer language to understand, debug, and maintain the codes.

We prefer the third approach since the "computer program" in spreadsheets plus macros is already extremely powerful. On top of that, it is transparent and dynamic. This means that you can build a spreadsheet model with nontechnical people and its results are immediately responsive to changes in input values. Transparency, dynamism, and ease of use are the key strengths of spreadsheets; no other analytical software comes close to matching spreadsheets. No other software would be as readily accessible to novices and experts alike for situational exploration and problem discovery. And to beat that, the work done in these first steps can be further extended into user-friendly solutions, data and solution analyses, and management reports.