From Technologies to Solutions

Learn

# OpenOffice.org Spreadsheet Macro Programming

## OOoBasic and Calc Automation

A fast and friendly tutorial to writing macros and spreadsheet applications

Dr. Mark Alexander Bain

PACKT PUBL SH NG

# Learn OpenOffice.org Spreadsheet Macro Programming

## OOoBasic and Calc Automation

A fast and friendly tutorial to writing macros and spreadsheet applications

**Dr. Mark Alexander Bain**


BIRMINGHAM - MUMBAI

# Learn OpenOffice.org Spreadsheet Macro Programming

## OOoBasic and Calc Automation



First published: December 2006

Production Reference: 1041206

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 1-84719-097-9

www.packtpub.com

Cover Image by www.visionwt.com

# Credits

**Author**

Dr. Mark Alexander Bain

**Reviewer**

Andrew Pitonyak

**Development Editor**

David Barnes

**Technical Editors**

Divya Menon

Saurabh Singh

**Editorial Manager**

Dipali Chittar

**Project Manager**

Patricia Weir

**Project Coordinator**

Suneet Amrute

**Indexer**

Bhushan Pangaonkar

**Proofreader**

Chris Smith

**Layouts and Illustrations**

Shantanu Zagade

**Cover Design**

Shantanu Zagade

# About the Author

**Dr. Mark Alexander Bain** hasn't always been the leading authority on open-source software that you know him as now. Back in the late seventies he started work as a woodsman at Bowood Estates in Wiltshire. After that he spent a number of years working at Lowther Wildlife Park in Cumbria—it's not clear if his character made him suitable for looking after packs of wolves, or whether the experience made him the way he is now.

In the mid eighties there was a general downturn in the popularity of animal parks in the UK, and Mark found himself out of work with two young sons (Simon and Michael) but with a growing interest in programming. His wife had recently bought him the state-of-the-art Sinclair ZX 81, and it was she who suggested that he went to college to study computing.

Mark left college in 1989 and joined Vodafone—then a very small company—where he started writing programs using VAX/VMS. It was shortly after that, that he became addicted to something that was to drastically affect the rest of his life—Unix. His demise was further compounded when he was introduced to Oracle. After that there was no saving him. Over the next few years, Vodafone became the multinational company that it is now, and Mark progressed from Technician to Engineer, and from Engineer to Senior Engineer, and finally to Principal Engineer.

At the turn of the century, general ill health made Mark reconsider his career; and his wife again came to his rescue when she saw a job advert for a lecturer at the University of Central Lancashire. It was also she who suggested that he should think about writing.

Today Mark writes regularly for *Linux Format*, *Newsforge.com*, and *Linux Journal*. He's still teaching. And (apparently) he writes books as well.

In memory of my Father—he would've got a real real kick out of this.

Thanks to my Mother and my family for their continual (and continuing) support and encouragement.

Thanks also to Noel Power and Michael Meeks for help with the chapter on VBA, to Paul Hudson for introducing me to Noel and Michael, to Andrew Pitonyak—I'm pleased that I could teach something to even you, and to David Barnes for that email back in May 2006.

# About the Reviewer

**Andrew Pitonyak** is a Principal Research Scientist / Software Engineer for Battelle Memorial Institute. He has been using OpenOffice.org since it was StarOffice 5, and he is the author of "OpenOffice.org Macros Explained", "Andrew's Macro Document", and other OOo related documents (see `http://www.pitonyak.org`). He has a Master of Science degree in computer science and another in mathematics.

In his spare time Andrew is very involved in his church, is a trained Stephen Minister and a professional puppeteer, works on his house, and spends time with his wife and daughter. He is an NRA-certified firearms instructor, holds a General-class amateur radio license, and spends a lot of time working with his digital camera. You can reach Andrew at `andrew@pitonyak.org`.

# Table of Contents

# Preface

What would you say if I asked you to name the thing that had the greatest impact on Western Society in the second half of the 20th Century? Chances are you'd say the PC—the ubiquitous Personal Computer. But that's only half the story; it wasn't the PCs themselves that caused the revolution. After all, I got my first PC, a Sinclair ZX 81 back in 1981, and although it made an interesting hobby, it certainly wasn't life changing.

By the end of the 80's I was using something that anyone today would recognize as looking like a PC, but it was still very primitive. Apart from running a word processor called Lex-WP, it was really just an interface to VAX and Unix servers.

So, what was it that turned the PC from just a useful tool into the essential, number one requirement for any business? One answer is Excel—we can even put a date to the start of this revolution—November 1987.

After starting life as Multiplan, Excel became available to everyone who was running Microsoft Windows (and who had the money). Overnight, virtually every major business became addicted to the software; and Microsoft became the giant that we know and love today.

It's not really a surprise that Excel was so successful. It was an application with which you could organize your information to analyze and manipulate your data. You could even extend the basic functionality by using macros.

And that's pretty well how things remained for the rest of the century.

However, things were about to change.

In January 1998, a new term was introduced in a meeting at Palo Alto in California—open source. Then in 2000, Sun Microsystems informed the world that they were going to join the open-source community; so on 13th October, 2000, OpenOffice.org was born.

Today, the realm of the professional spreadsheet is not just limited to those that can afford it. Today even the smallest business or individual user can use Calc, and (as we'll see in this book) we can take the basic application and bend it to our own will.

Now that's a revolution.

# What This Book Covers

*Chapter 1* introduces you to the tools that you'll need in order to write your own macros. By the end of the chapter you'll have become acclimatized to Calc's development environment, and you'll know which buttons to press to make your life a little bit easier.

*Chapter 2* starts to make use of the basic building blocks that you'll need for your macros: Libraries, Modules, Subroutines, and Functions. By the end of the chapter you'll have your first macro up and running.

*Chapter 3* gives an overview of the objects that are built into Calc, and which we can make use of in order to create macros that perform quite complex operations; we'll see just how easy they are to use. We'll also see where to get further information on these objects.

*Chapter 4* is where we really get into writing macros. Here you'll learn how to manipulate the contents of one (or more) spreadsheets—and after all, that's what we're here for, isn't it?

*Chapter 5* looks at how we can format the data contained in our spreadsheet—it doesn't matter how accurate our data is, if all of the columns overlap each other making the contents impossible to read.

*Chapter 6* is an introduction to databases—how to access them, how to display the results of queries in a spreadsheet, and how to change the contents of the databases themselves.

*Chapter 7* explains how to make use of other documents (such as charts) within Calc, and how they can be sources of information; for instance, the contents of websites.

*Chapter 8* moves away from purely writing code, and shows how you can build a user interface—by building your own dialogs.

*Chapter 9* brings everything together. By the end of the chapter you'll be able to create and distribute a complete application.

*Chapter 10* takes a look into the future of Calc, and what to do if you're moving from Excel to Calc but don't want to have to rewrite all of your code.

# What You Need for This Book

You don't need to be a programmer to use this book, but you do need to be familiar with the concept of a program and how simple things like a loop might work. The book is compatible with StarBasic, the macro language for commercial version of OOo—StarOffice.

As you progress through the book, you'll find that some of the issues we deal with only relate to the most current versions of OOo. At the time of writing, all of the code in Chapters 2-9 works for version 2.0.4 on Windows and 2.0.2 on Linux. Chapter 10 is another story—that really is on the cutting edge, and for that you'll need Novell's version of OpenOffice.org.

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "Don't forget that `Main` must be the first macro in the module."

A block of code will be set as follows:

```
    Dim fname as String
    Dim sname as String
    Dim username as String
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
Sub click_cmd_view_symbols
  Dim oTxt_company as Object
  Dim oLstCompanySymbol as Object
  oTxt_company = oFinance_dialog.getControl("txt_company")
  oLstCompanySymbol = oFinance_dialog.getControl("lstCompanySymbol")
  oLstCompanySymbol.AddItem( oTxt_company.Text ,0)
End Sub
```

**New terms** and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

# Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

# Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

# Downloading the Example Code for the Book

Visit `http://www.packtpub.com/support` and select this book from the list of titles to download any example code or extra resources for this book. The files available for download will then be displayed.

The downloadable files contain instructions on how to use them.

# Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata have been verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

# Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

# 1
# Working with OOo's Basic IDE

You always know a thing is good if it's got a TLA (that's a Three Letter Abbreviation). By the end of this chapter you will be fluent in a double TLA—the OOo IDE. The **O**pen**O**ffice.**o**r**g** **I**ntegrated **D**evelopment **E**nvironment is our interface into the world of writing OpenOffice.org Calc macros.

If you already know your way around OOo, then jump on to Chapter 2 where we'll start looking at writing macros. If, however, you're still new to all of this, then spend some time learning about the IDE. This is where we'll do all of our work—and so in this chapter we'll spend our time getting acclimatized to it. We will see how to manage macros, how to navigate around the IDE, and how to start designing dialogs.

And by the way, by 'OOo's *Basic* IDE' I don't mean that it's low-level. By 'Basic' I mean that we'll be using OOo Basic as the programming language.

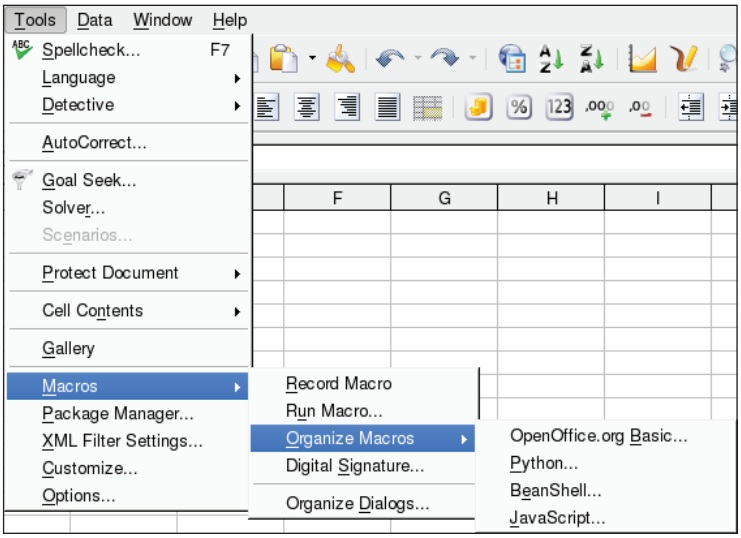
## Before We Start

As you're reading this chapter, you may want to follow the examples on your own PC. When you do, you'll probably see differences in the way that the screens look. Don't worry—that's because the look and feel varies according to the OS (Operating System) that you're using. For instance, here are the OOo start-up screens for Debian 3.1 and SUSE 10.1, respectively:


And a little advice—before you do anything else, make sure that you save your spreadsheet with a sensible name, something that is meaningful to the project that you're currently working on. In this case, we'll be look at a spreadsheet for Penguin P.I.—Private Investigator in the dark world between Windows and Linux.

# Accessing the OOo IDE

With our appropriately named Calc spreadsheet open, getting into the IDE (regardless of the OS that you're using) is simple enough:



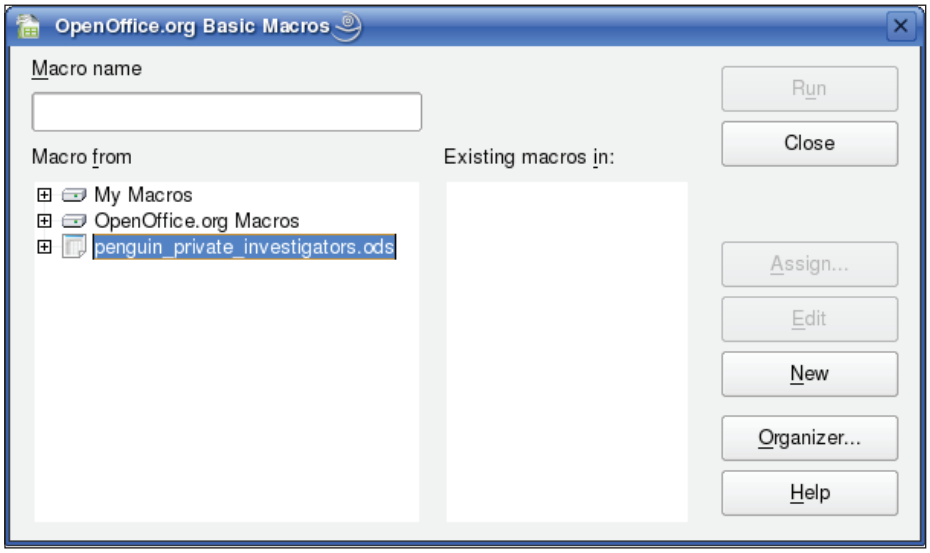
Start with creating a new Calc document and save it with the name `penguin_private_investigators.ods`. Next, use **Tools | Macros | Organize Macros | OpenOffice.org Basic...** to open the Basic IDE.

You'll notice that other than OpenOffice.org Basic, you actually have a choice of three languages in which you can work:

- Python (a common object-oriented programming language)
- BeanShell (a Java scripting language)
- JavaScript (the scripting language behind many web pages)

That's good, because it means that if you already have skills in one of these then you don't need to learn a new programming language. However, we will be only working in OpenOffice.org Basic, and so that's the option to choose from the menu.

As soon as you click on **OpenOffice.org Basic...**, you'll be presented with the **OpenOffice.org Basic Macros** dialog:



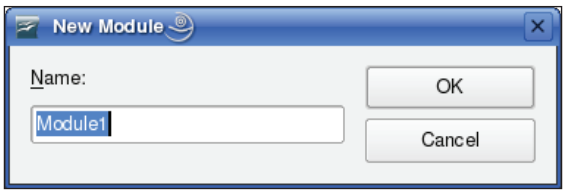

It's from here that you can choose where to create your first macro. However, you've got a choice of areas in which to create it. You can see that there are three groupings, and each one is used for a different purpose:

1. **My Macros**: If you want a macro to be used in all of your spreadsheets then store it here. This is useful for commonly used functionality, but don't forget —if you email your spreadsheet to someone else, then the macros won't work (because, of course, each user will have their own 'My Macros').

2. **OpenOffice.org Macros**: If you want to write a macro that's to be available to spreadsheets used by all users on your system, then store it here. 'OpenOffice.org Macros' is a system directory, so on Linux you will need root privileges to do this.
3. **penguin_private_investigations.ods**: (You will, of course, see the name of your own spreadsheet here.) If the macro is to be embedded in the spreadsheet, and not to be used elsewhere, then store it here. Use this storage method if you are going to email your spreadsheet to someone else to use, or if you're going to store it on a network drive.

So, at the moment, we're only really going to be interested either in using **My Macros** or for the macro to be embedded into the spreadsheet.

When you are ready, highlight either **My Macros** or your new spreadsheet, then click **New**. You'll now be presented with a **New Module** dialog box:



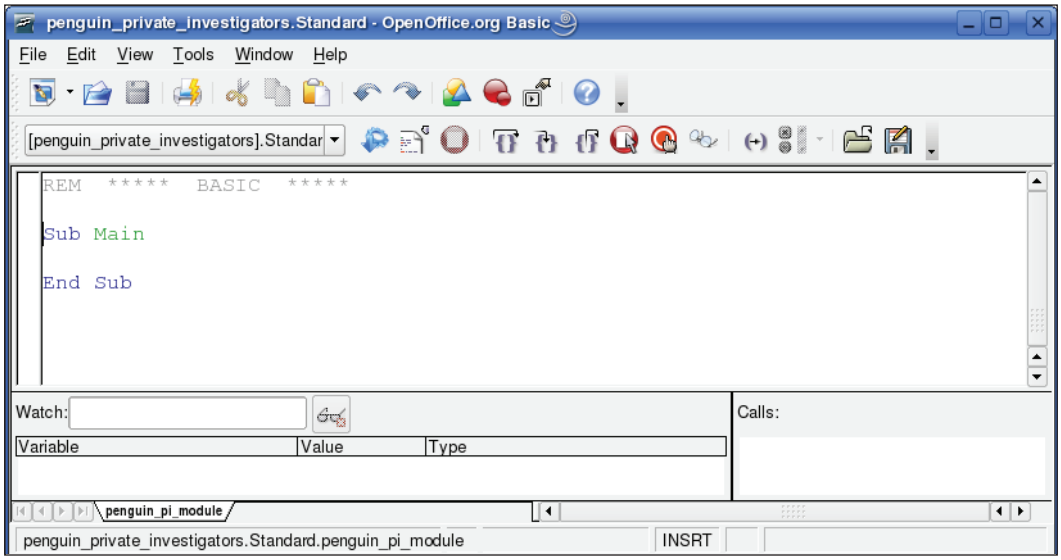

And what is a module? Quite simply—a module is a file in which you'll store all of your code (we'll learn more about the structures of modules and macros in Chapter 2).

At this point, the best bit advice I can give you is—*Don't Click 'OK'*. Well, not yet anyway. Why? Because you need to break a bad habit before it starts. If you save this module as 'Module1' then the next will be 'Module2', then 'Module3', and so on. Well, you get the picture don't you? At the moment we have no modules, but imagine that we've got 5, or 10, or 100, or imagine that you come back to your modules in six months time or a year. How can you possibly remember what each module is to be used for?

So make your life easier by naming the module sensibly (just as you did with the spreadsheet):

Now you can click **OK**, and you'll have a module with a name that has meaning, whenever you come back to it.

At last! We're finally looking at the OOo Basic editor:




It's in this window that we'll be spending most of our time, because this is where you'll write all of your macros. Notice one interesting thing: OOo kindly creates a macro for you—`Main`. You can remove this if you wish; it just shows you the format that OOo expects.

# Controls in IDE

It's tempting to just jump straight in and start writing your code. However, it will be worth your while spending some time looking at the controls of the IDE and seeing how they can help you develop even better macros. So that's what we're going to do next.

If there are only two buttons that you ever use in the IDE, they are these:

**The Save button**: Use it a lot. Don't come crying to me when you've just written 500 lines of code, accidentally introduced an infinite loop, crashed the OOo IDE, and lost all of your work—just because you didn't press the save button (of course, if you don't use the button make sure that you use: **File | Save**).

**The Run button:** Obviously once you've written some code, you'll want to see it in action. If you click this button, then OOo will run the *first* macro in the module.

If you remember, we saw that screens may not always look the same—depending on whether you're Windows or Linux, and which version of Linux you're using. This is one of those cases. If you don't recognize the run button above, then your one may look like the following:

If all goes well, then those are the only two buttons you'll need. Just write the code, save it, and run it. Unfortunately this is the real world, and most people will tell you Sod's law—If anything can go wrong, it will go wrong. There are also those that will add an addendum—If anything *can't* go wrong, it will go wrong. And then there are a lot of people who think that all the others are being just a little bit too optimistic.

If things do start to go wrong, then you may find that the **Stop button** comes in handy. Clicking this button will stop the macro from running and return to edit mode in the Basic editor window:

Occasionally you may find that you have to amend your code, but you don't want to actually run the macro (let's say, for instance, it's a macro that writes information to a database). If this is the case, then you can use the **Compile button**. This will check the syntax of the macro without running it.

You may find that the button may look different depending on your system, for example, we've just seen the Compile button for Linux, so here's the button for Windows:

Having written the macro and compiled it, things may still not be running as you think they should. Fortunately, the OOo IDE can help you analyze exactly what's going on.

If things aren't running exactly as you expected, or even if you just want to see what's going on inside your macro, then the IDE has the tools to help you. Three buttons that you'll find very useful are:

**Step Into**: This allows you to run the code one line at a time so that you can see exactly what's going on

**Step Over**: As you're stepping through the code, you may come across a call to another macro. If you don't want to step through that macro and just want to run it as a single command, then use the Step over button.

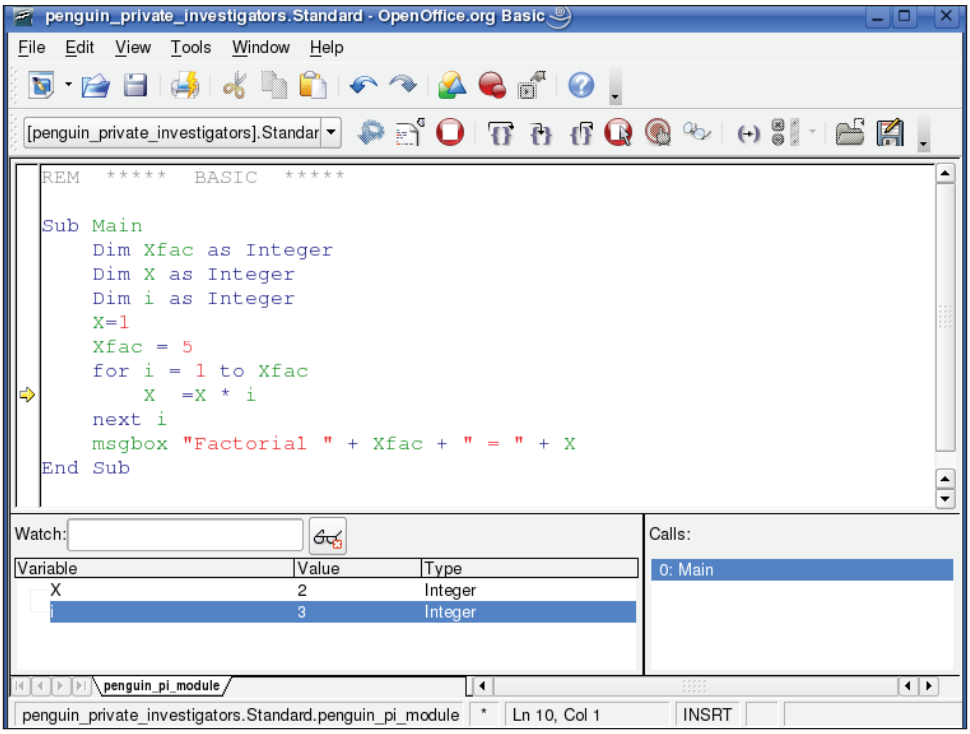
**Step Out**: When you've stepped through all of the code that you want to see running, then use this button to end and let the current macro that you're in finish running normally.

You'll find these buttons extremely useful as you try to understand what's going on with your macros, especially if something is going on that you don't quite understand. You can gain even more understanding by using another button.

**The Watch button**: As you step through the macro the values of variables in the code are going to change. To see these changes all you have to do is select the variable (in code window), click the watch button, and then step through the code, and watch the values as they are displayed in the watch window.

Now we're able to see exactly what's going on in the macro:

- The marker on the left-hand side tells you exactly where your are in the code.
- The watch window at the bottom left shows you the current value of your selected variables.
- The window at the bottom right gives you information on which macros are being called; this can be particularly useful if one macro calls another.



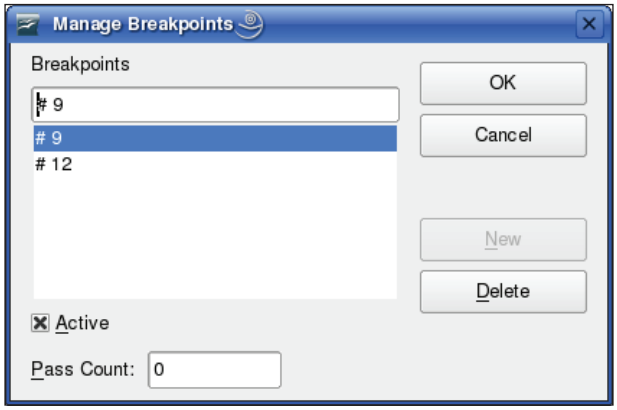

"Hang on", I hear you say, "What happens if I've got 100 lines of code, or 1000? I don't want to have to step through every one of them to see the status of a variable." Quite right too! This is where we can make use of the **Breakpoints button**.

Select the line of code in the macro that you want to monitor. Click the Breakpoint button. You will see a red dot appear in the left-hand border (or you can double-click in the left hand border to toggle a break point on or off). The **Breakpoint button** looks like the following:

Now you can click the Run button—the macro will start working, but will pause on the line where you placed the breakpoint, and you can view the Watch window and the Calls window to see what's going on. When you're happy, you can click the Run button again and the macro will resume its operation.

You're not just limited to one breakpoint, you can mark as many lines as you need. If you do use a number of them, then it can become laborious to scroll through all of the code to see where they are. To make life easier use the **Manage Breakpoints** dialog, after clicking the following button:

The **Manage Breakpoints** dialog will tell you the line numbers on which you have breakpoints, and you can activate, deactivate, or delete them.




So far in this chapter we've covered all of the crucial elements of the OOo Basic editor part of the IDE. We've seen what each part of the screen is for, and we've also seen how to use the key buttons. You may now feel that you want to start actually writing macros. If that's the case, then you're ready for Chapter 2.

However, there are still some more elements of the IDE that you'll find useful, and are there to make your life easier.

# Navigating around the IDE

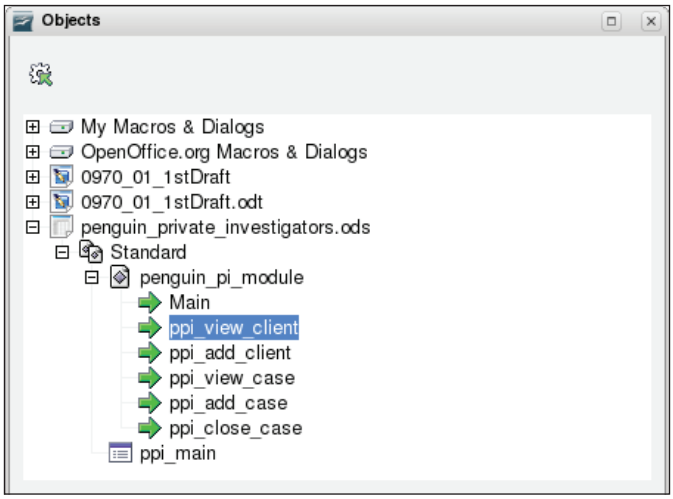
In all probability you'll find that before long you've written dozens of macros. If you're writing a more complex application, then this may even run into hundreds of macros. When it comes to maintaining the macros, you can make this easier by organizing your layout. For example, you could write the macros in alphabetical order or group them according to functionality. But come on—scrolling up and down the screen is not the most efficient way of finding the macro that you want.

So is there an easier way for us to move from macro to macro? Of course there is. In fact there are a few ways, and we'll look at them now.

## The Object Catalog

The first thing that you'll need to do is find the Object Catalog icon on the IDE toolbar:

If you've got the Object Catalog open, then you just have to click on the macro that you want to edit and OOo will move to it in the Basic IDE.



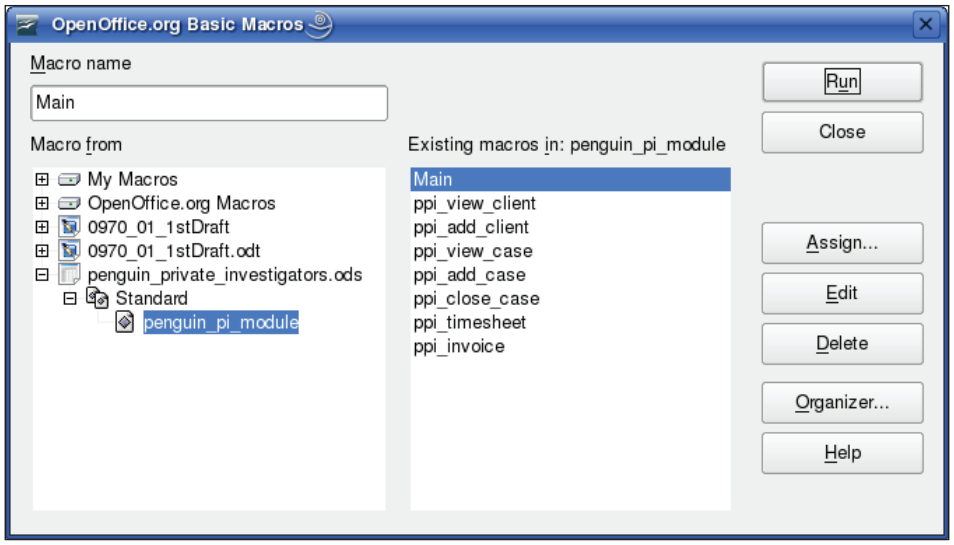

You can keep this window open while you're working; however, you will notice that any new macros that you write will not automatically appear in it. In order for the new macro to appear just minimize the module (**penguin_pi_module** in the above example) by clicking the minus sign. Expand it again (you'll see a plus sign next to the module name) and then the new macro name will appear in the window.

## Select Macro

The next button to find on the IDE gives us access to the **Select Macro** dialog. You'll need to find an icon that looks like:

If you can't see the icon above, then have a look for:

Once you've found the right button you'll be able to call up the **OpenOffice.org Basic Macros** dialog:

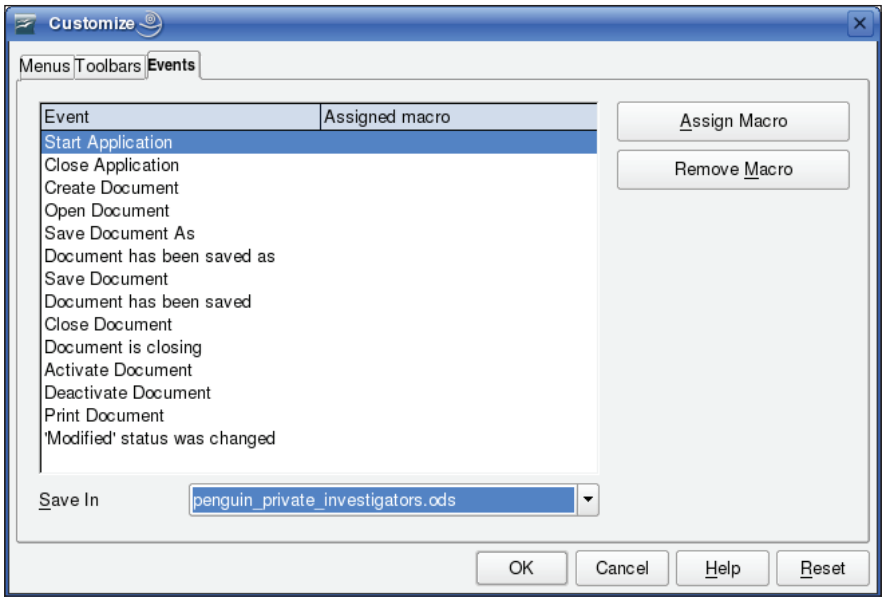



You may be thinking that this screen looks familiar. In fact, if you care to look back to the start of this chapter, then you'll see something very similar—it looks just like the very first dialog that we came across (if you remember, we accessed it from the menus by clicking on **Tools | Macros | Organize Macros | OpenOffice.org Basic**). There are, however, some key differences:

- The **Existing macros in** box is now populated with the macros that you've written.
- The **Run**, **Assign**, and **Edit** buttons are now enabled.
- There's no **New** button; it's been replaced by **Delete.**

So is this the same screen? Well, yes it is, and you can prove that to yourself by clicking on **My Macros**—immediately the buttons will change so that the screen looks like the first one that we saw. Click back on to our module and the buttons will change back again.

You'll probably be able to guess what most of the buttons do (such as **Run**, **Edit**, and **Delete**), but what about **Assign...**? If you click the **Assign...** button, then a new window will open:

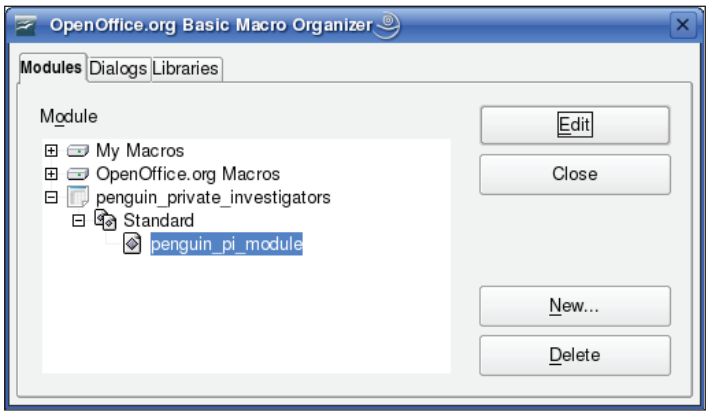



You'll find this is a very useful screen; with it you can assign any of your macros to menu items, toolbar buttons, or even window events. For instance, by using this you could automatically run a macro every time that you open your spreadsheet.

If you close this window (for now) and return to the **OpenOffice.org Basic Macros** dialog, then there's another button that you may be wondering about—**Organizer...**. Click it and you'll see the screen for the **OpenOffice.org Basic Macro Organizer.**

## The OpenOffice.org Basic Macro Organizer

A dialog that you'll see a lot of is the Basic Macro organizer:



You can also get to this screen from a couple of other places as well:

- There's a button on the Basic Editor toolbar. The icon varies from system to system, but look for one of these two:

- There's a menu selection on the spreadsheet—click on **Tools | Macros | Organize Dialogs...**.

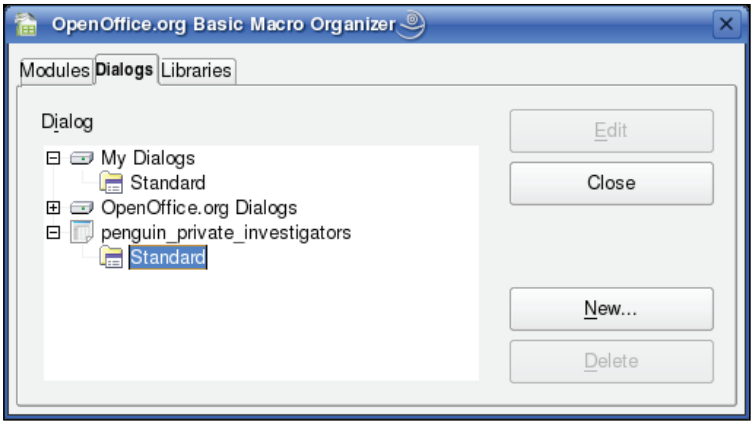
If you look at the form you'll see that it has three tabs—**Modules**, **Dialogs**, and **Libraries**. We'll be looking at **Modules** and **Libraries** in Chapter 2, but for the time being just remember:

- Libraries are storage areas for grouping modules and dialogs.
- Modules are storage areas for grouping macros.
- Dialogs are used to create buttons, combo-boxes, and all the normal things that you would expect in any GUI (Graphical User Interface).

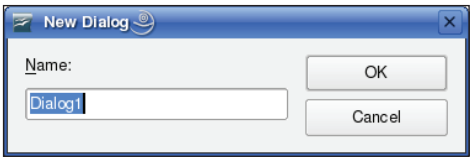
# Designing Dialogs with the IDE

You can create a dialog to provide a user interface for your macros. We'll be looking at dialogs in depth in Chapter 8, but for now we'll see how we can create dialogs using the IDE:

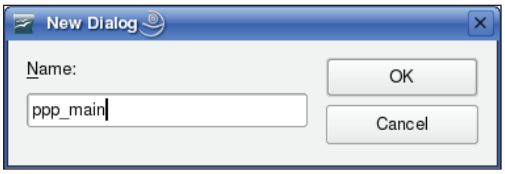
- Open up the OOo Basic Macro Organizer dialog box (you can open it by any of the methods that we've already discussed), and select the **Dialogs** tab.



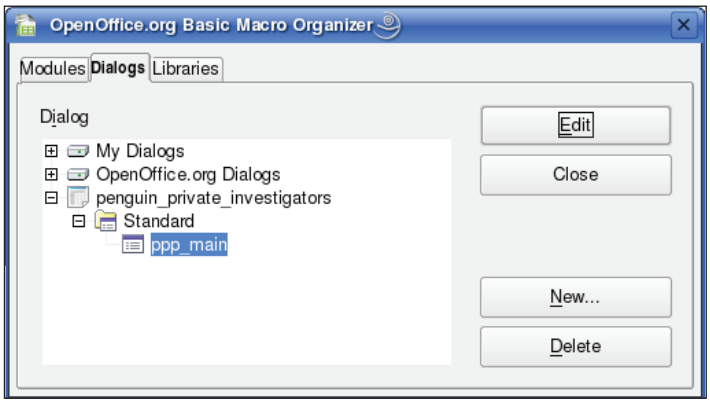
- Next select the library in which you want to store the dialog. In the example on the previous page you can see that we're using the **Standard** library in our `penguin_private_investigators` spreadsheet. Do that and you're ready to click **New** to create the dialog.




- If you cast your mind back to when we created our module, then you'll remember that we didn't use the default name that OOo gave us. This is just the same. Using the default 'Dialog1', 'Dialog2', etc. will only lead to confusion later on. So, we will choose a sensible name—something that has some meaning for the project:

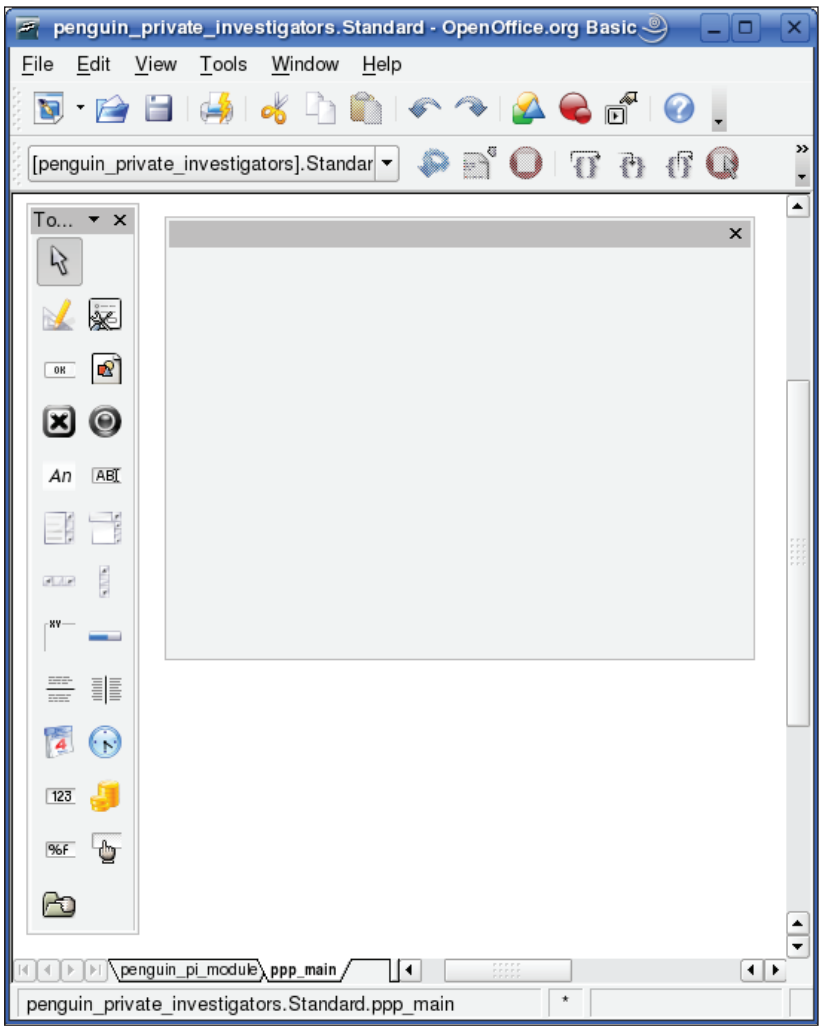



- Having chosen a suitable name you can click **OK**, and OOo will create the new dialog for you:



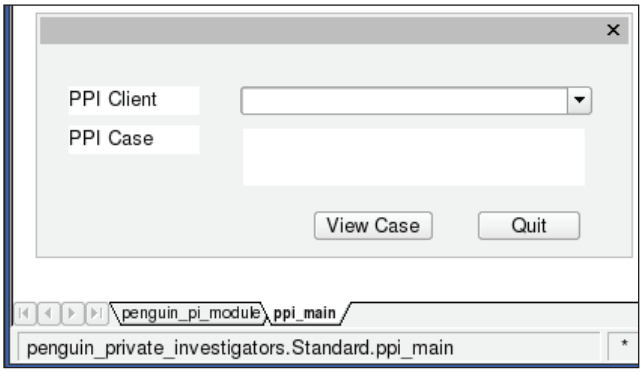

- You'll see that OOo will take you back to the Organizer dialog box and not to the newly created dialog. This means that you can create all the dialogs that you're going to need before you continue. Once you've created all the ones that you want, or if you're only going to create one at the moment, then click on **Edit** to continue.

- If you stop to look at the Macro Editor, then you'll realize that it is actually the same Basic editor that we've already been working with. In fact, if you look at the bottom of the screen, then you'll see a tab for the module that we've already created, and you can easily switch between the two. This is important because we have to write the code as a macro and then assign the macro to an object in the dialog (such as a button, combo-box, etc.). Technically speaking, you can configure Controls to call your macros when Events occur.

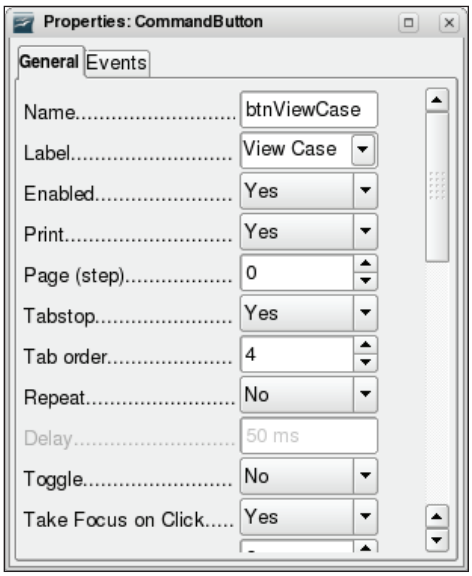


- It's easy to design your new dialog, just add the objects from the Toolbox toolbar. (Click on the one that you want and then use the mouse to drag it to the correct position on the dialog itself. Don't forget, if the Toolbox toolbar is not displayed, you can use **View | Toolbars | Toolbox**.) If you've drawn an object (such as a button), and then decide that it's in the wrong place, or too big, or too small, then just click on it and you can resize or reposition it. If you decide that you don't need it at all, then click on it and press the *Delete* key on your keyboard.

With very little effort you'll end up with a professional looking dialog:

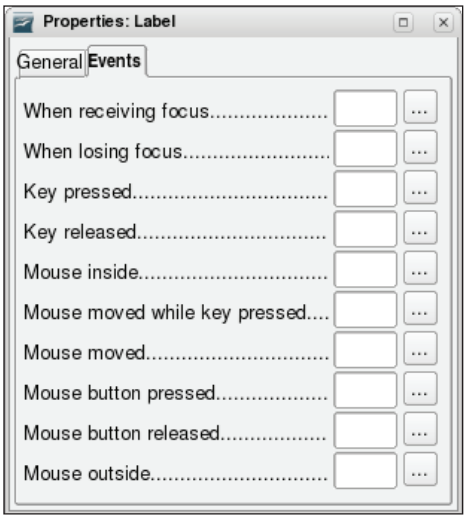



"But mine doesn't look like that", I hear you cry; "All the buttons and labels display things like CommandButton1 and Label1". That's soon remedied—just use your mouse to right-click on the object and then select **Properties**:



Once you're in the Properties screen, edit the **Label** field to change the text that you want to be displayed on the label or button. And while you're here, change the name of the object to something more meaningful—just as we don't want modules called Module1, Module2, or dialogs called Dialog1, Dialog2 nor do we want loads of buttons all called CommandButton1 or CommandButton2. For instance, if we're going to use a button to view all of Penguin P.I. cases, then name it **btnViewCase** or maybe **cmdViewCase**.

Now, you may have noticed that there is a second tab called **Events**:




If you're desperate to know more about working with dialogs, and already have some knowledge of macros, then you may now want to move on to Chapter 8. There we'll see how to use this dialog to assign macros to any event on the dialog that you've designed here; for example running a macro when you click on a button.

However, if not then it's time to move on to Chapter 2 where we'll look in depth at how to use the OOo Basic IDE to write macros.

# Summary

In this first chapter, we've had a look at all of the elements that make up the OOo IDE, and how to call them up. If you've just opened a spreadsheet:

- For the OOo Basic Editor click on **Tools | Macros | Organize Macros | OpenOffice.org Basic...**
- For the OOo Basic Organizer click on **Tools | Macros | Organize Macros | Organize Dialogs...**

Remember that you can store modules in one of three areas—My Macros, OpenOffice.org Macros, or embedded in the spreadsheet.

We have seen different useful buttons in the OOo Basic editor toolbar. We have also seen how to navigate around the IDE using the Object Catalog, Select Macro, and the OOo Organizer (remember there may be an alternative icon).

The OOo Basic editor is used for both writing macros and designing dialogs. If you are using the OOo Basic editor to design a dialog, then change the text on an object (such as a button or label) by using the properties window. You can also use the properties window to assign macros to the objects.

We've now covered the OOo IDE, and you're now able to navigate around it and use each of the elements in it. In Chapter 2 we'll be using the IDE some more when we look at libraries, modules, subroutines, and functions.

# 2
# Libraries, Modules, Subroutines, and Functions

In Chapter 1, we learned about the OOo IDE, how to navigate around it, and how to use the different elements built into it. In this chapter we'll carry on using the IDE, but now we'll be learning:

- How to write macros
- The difference between subroutines and functions
- Using variables in macros
- How to make the best use of modules and libraries

By the end of Chapter 2, you'll be able to start adding your own custom functionality into OpenOffice.org Calc.

## Using Libraries

Before we start writing macros, we'll have a look at libraries—you'll recall from Chapter 1 that macros are stored in modules, and modules themselves are stored in **libraries**. Libraries can be in one of the three areas:

- My Macros: The location common to all of your own macros
- OpenOffice.org Macros: The system-wide location for macros
- Embedded in a spreadsheet: Accessible by a single spreadsheet

You may also remember that when we came to create a module, we didn't have to create a library; OOo did that for us, setting up one called **Standard**. This is a mandatory library, and the OOo IDE will not let you delete it or rename it. Unless you instruct the IDE otherwise, all of your new modules will be placed in Standard.

Now, you may think that this is all that you need to know about libraries and that you're quite happy with using the OOo's Standard library. However, before we move on, there are a couple of reasons why you might consider using your own libraries and not the default one:

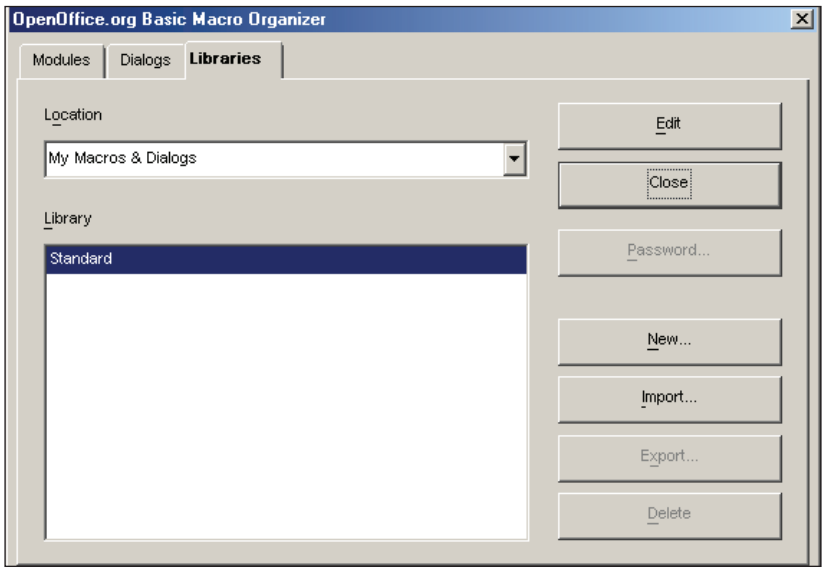
1. You want to manage your modules more effectively.
2. You work in a multi-user environment.

We'll have a look at both of those points in a bit more detail.

# Managing Modules using Libraries

Let's think about our example company—Penguin P.I.-Private Investigator in the dark world between Windows and Linux. When its founder, Pygoscelis P. Ellsworthy, started his business, he was able to work with a single spreadsheet (`penguin_private_investigators.ods`). Pygoscelis wrote a number of macros and carefully placed them in modules according to their purpose, and with the modules placed in the Standard library. However, he soon found that these modules didn't really fit together; some were specifically for his field operative—the famous femme fatale Korora Blue. Others were specifically for his office clerk—Sphen Dermersus. Pygoscelis realized that the solution was simple—split the modules into different libraries.

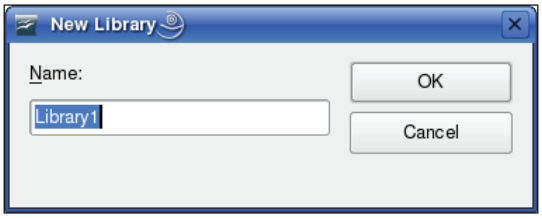
If, like Pygoscelis, you wish to make use of custom libraries, then start by going up the OOo Basic Organizer (the easiest way is to open up a spreadsheet and then click on **Tools | Macros | Organize Macros | Organize Dialogs**), and select the **Libraries** tab.



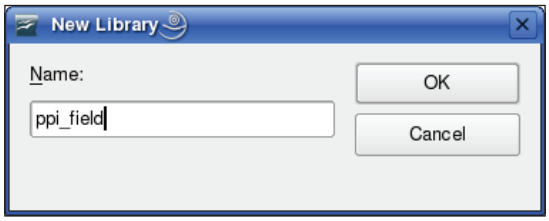
The dialog may look slightly different compared to the screenshot opposite; this may vary according to the flavor of Linux that you're using and the particular version of OpenOffice.org that you've got installed. The key difference that you may see is that you won't see the **Import** button, instead you'll see an **Append** button; but don't worry they do the same job.

Whichever style you've got just make certain that the **Location** shows **My Macros & Dialogues.** And, as you may expect, only the **Standard** library exists at the moment.

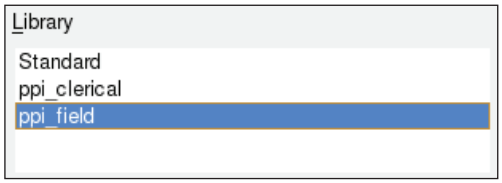
So, to create your own library, just click **New** and you get the following dialog box:



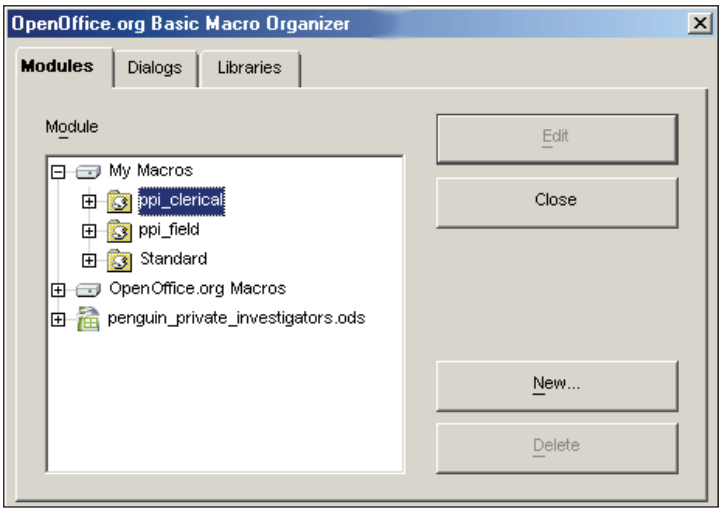

We've seen this type of box before (when we created new modules and new dialogs) and so you know not to use the default name, but instead change the name to something more appropriate:



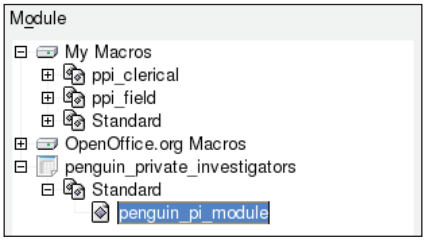

Within a few minutes you'll have all of the libraries that you need for your project:



Now you're ready to create modules in your new library:




If you're worried about modules that you've already created, just expand the location of the existing module, and then use the mouse to drag-and-drop it into your new library:




Having seen how useful libraries are when it comes to managing modules, we can now have a look at why libraries become even more important in a multi-user environment.

# Using Libraries in a Multi-User Environment

Let's look at one particular Friday in the Penguin PI agency. Pygoscelis was (as always) very busy. As soon as Korora and Sphen arrived, he called them into his private office.

"Listen" he said, "this case is about to break. I'm meeting a contact later today—he's got a disk with data that we need. However, it requires analyzing. You'll have to write the macros to do that, and I'll run them tonight. Save them into your **My**

**Macros** area and then I can import them into my spreadsheet." With that he put on his false beard and left for his appointment.

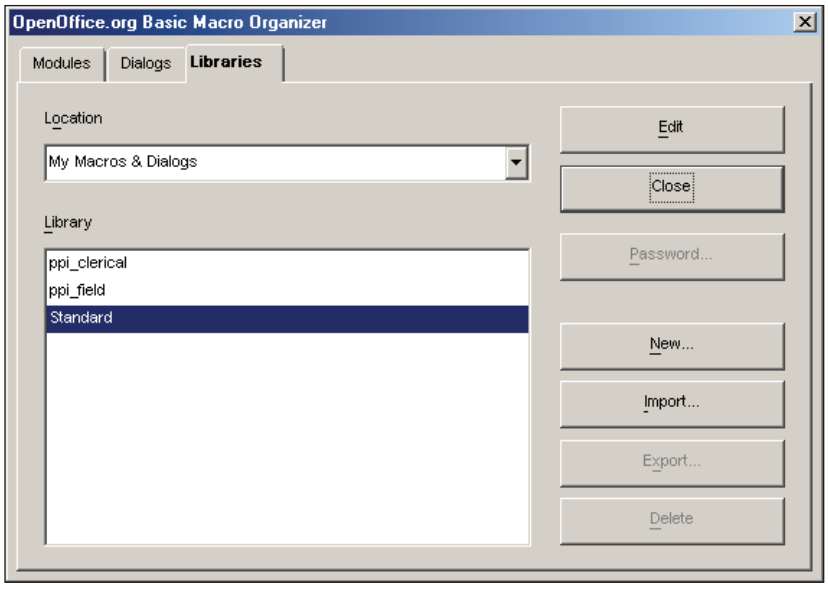
Korora and Sphen spent the whole day working on their macros. Everything worked perfectly, and at the end of the day each left a note on Pygoscelis's desk telling him that everything had been done. Then they both headed off for the weekend.

Later that night, Pygoscelis rushed back into the office. He knew that he was being followed, and had only minutes to run the macros. He read the note that Korora had left him and saw that she'd created a library called `ppi_field_operations`. He quickly loaded it. Outside a car screeched to a halt.

Next, he tried to import Sphen's macros. However, to his horror he found that Sphen had used the **Standard** library; he was unable to import it. Then he heard a sound, looking up he saw the handle of the door slowly turning...

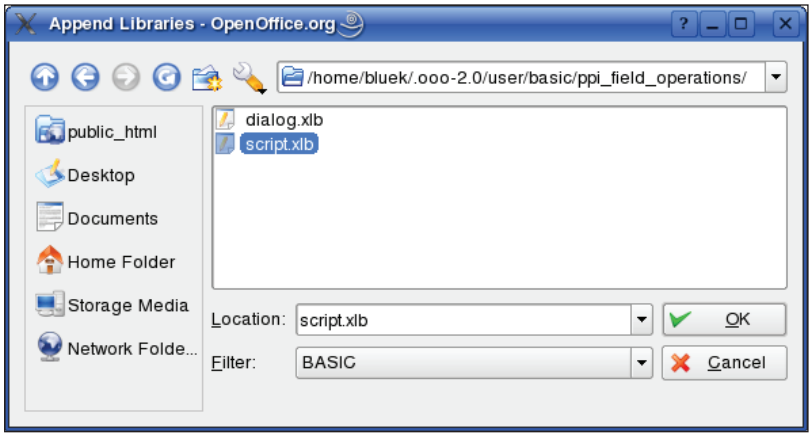
I'm sure that you get the moral of the story immediately—you can import a named library, but the standard one is going to cause you problems.

The process of loading someone else's library is actually very easy: Go to the **Libraries** tab of the organizer and click **Import** (don't forget you may see **Append** instead of **Import**):




It's worth thinking about the word **Import** for a moment. OOo will import the library you select into the area you choose (for example **My Macros** or the spreadsheet itself).

As soon as you click the **Import** button, OOo will present you with a dialog box. You can use this to move to the location of the library that you're going to import; you must of course have read rights on the directory (or folder) where the library is stored.




You'll notice that the library is actually a directory containing two files:

- `dialog.xlb`: containing dialog indexes
- `script.xlb`: containing module indexes

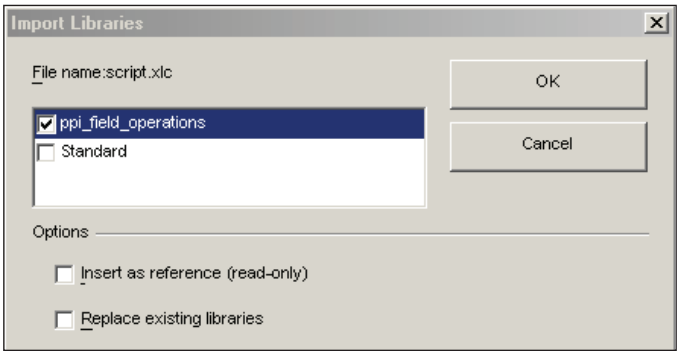
If you look in the directories, then you'll also find one or more `xba` files—these are the module definition files and contain the actual macros themselves. If you examine any of the files, then you'll find that they are written in **XML** (**eXtensible Markup Language**). However, don't edit them yourself, leave that to the automatic processes in OOo.

At this stage you might be experiencing a little bit of a problem: you can't find the OOo directory at all. That's because the location where OOo stores it files varies according to the system that you're using. For example:

- Suse 10.1: `/home/<user>/.ooo-2`
- Debian 3.1: `/home/<user>/.openoffice.org2`
- Windows XP:
  `Documents and Settings/<user>/ApplicationData/OpenOffice.org`

If none of these look familiar, then you'll need to search your system for the 'basic' directory or for the name of the library. However, don't forget that if the library belongs to another user, then they need to give you read-only access to it. Without that the library will not be available to you.

Once you do find the library and the owner has given you read access to it, then select the relevant `xlb` file and click **OK**.




You've now got the choice of *how* you want to insert the library. You can either:

1 Import a copy of the library: Useful if you want to make your own changes to it. However, don't forget that the original will remain unaffected.

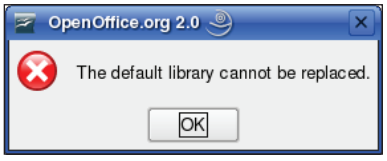
2. Create a link to the library: This gives you read-only access to the library, meaning that you can run the macros but can't make any changes to them.

You also have the option to overwrite existing libraries. Be very careful with this, if you have a library with the same name as the one that you're trying to import, then you'll destroy any code already in your existing library.

This is of course the problem that Pygoscelis had with Sphen's library; both Pygoselis and Sphen had a library named **Standard** in their **My Macros** area.

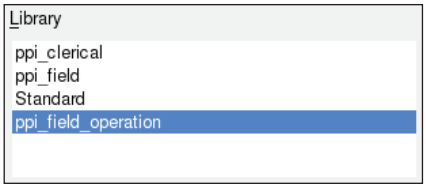
If you try to append the library, then OOo will tell you:


And if you persist and try to overwrite the **Standard** library, then you'll get a polite refusal:

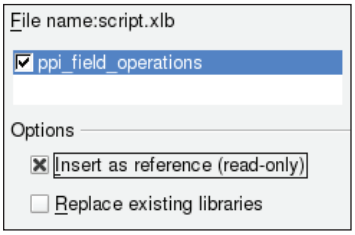


However, do be warned: OOo *will* allow you to overwrite custom libraries, so be careful when importing.

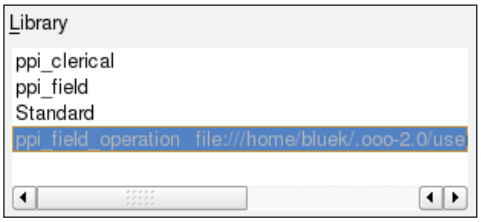
When you have decided to use the **Import** button to fully import a library, you will see it in the Organizer Library tab:




As we've already seen you can import a library as a link by marking the **Insert as reference** checkbox:




This time when you look at the Organizer, you'll see that location of the new library is shown by the library name:




The big advantage of using a reference library is that you have access to other developers' macros, greatly extending the functionality of your own spreadsheets.

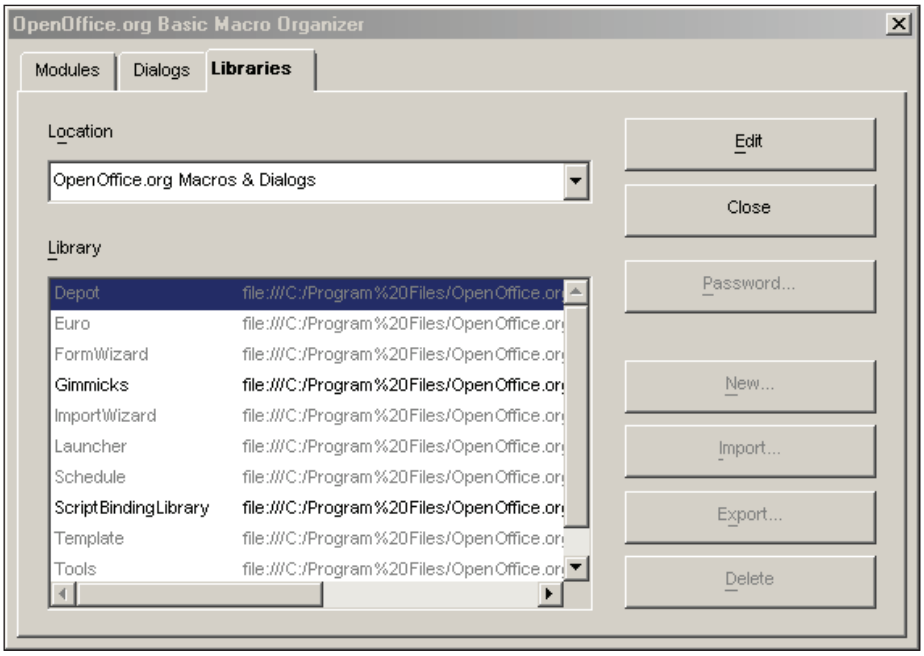
There is, of course, a disadvantage: you can't guarantee that any referenced libraries will not be modified or even deleted. If you do rely on such a library, then you should consider moving it into the OpenOffice.org Macros area.

# Adding a Library to the OpenOffice.org Macros Area

If you've got a library (either your own or someone else's) that you want a number of people to use, then you can (as we've seen) append it into the **My Macros** library container. This works well, but there are a couple of disadvantages:

1. If you fully import the library, then you end up with multiple copies of each library. The question will then arise at some point "Which is the correct version?"
2. If you append the library as link, then it will be subject to change or even deletion without your control.

The obvious solution is to append it into the OpenOffice.org Macros area. However, if you try to do that via the Organizer, then you'll find that the **Import** button is disabled:




Having tried that, it may occur to you that it would be wrong for just anyone to be able to add files into such an important area, and this should actually be left to a system administrator. However, even if you log on with administrator rights on Windows or as root on Linux, you'll find exactly the same thing—the **Import** button is disabled.

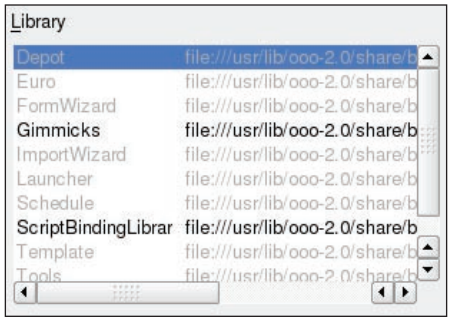
In fact this is a task that must be done *outside* of the OpenOffice.org application while it is *not* running.

Before moving on, let's review what we've already learned about the structure of libraries:

1. A library is stored as a directory.
2. The directory contains two `xlb` files for indexing the modules and dialogs.
3. The directory contains one or more `xba` files; these are the module definition files and contain the macros (or dialogs).

So the solution seems obvious enough; just copy a user's library into wherever OOo stores its own commonly accessed libraries.

However, first you need to find the location where OOo stores the libraries that you can access via Openoffice.org Macros. From what we've seen already, you can probably guess that this location will vary according to the particular operating system that you're using. This time you won't need to search around the system to find it, just look in the **Libraries** tab of the Organizer. There you can see the link details for each of the global libraries:




Having found where to place the library, all you have to do is move to the directory:

```
cd /usr/lib/ooo-2.0/share/basic
```

Then copy the library into the correct location. Remember that you will have to have system administrator authority to do this:

```
cp -r ~bluek/.ooo-2.0/user/basic/ppi_field_operations/ .
```

That's all it takes to set the library up; however, you won't be able to access it from your own account . If you fire up OOo Calc and have a look at the Organizer, you won't be able to see the new library—not quite yet.

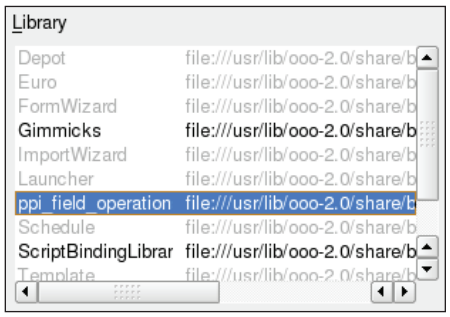
The list of OOo libraries that you *can* see is stored in your user account under the OpenOffice.org directory. You'll need to edit this `script.xlc` file and it'll be located at somewhere like:

```
/home/ellsworthyp/.ooo-2.0/user/basic/script.xlc
```

When you edit the file using your favorite editor (notepad, nano, emacs, etc.), then you'll see that it contains the list of links to libraries, using the XML format. You just have to add a line telling OOo where to find the new library:

```
<library:library library:name="ppi_field_operations"
 xlink:href="$(INST)/share/basic/ppi_field_operations/script.xlb/"
 xlink:type="simple" library:link="true" library:readonly="false"/>
```

Now when you restart Calc you'll find that the library is visible in the Organizer:




With this you are now able to:

- Create custom libraries
- Make use of other users' custom libraries
- Make a library accessible to any user

And Pygoscelis? Suddenly he realized that Sphen's **Standard** library was a directory with XML files. Quickly he saved it to disk, shoved it in his pocket, and headed for the window. As his office door burst open, he was already at the bottom of the fire-escape—this was not over yet, not by a long way.

# Using Modules

Having just dealt with Libraries, we'll recap what we learned about modules in Chapter 1:

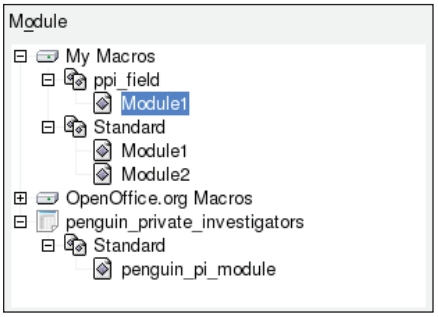
- One or more modules can be stored in a library.
- Modules can be one of two types—script (containing macros) or dialog.

- You have to create a module before you can write any macros.
- When you create a module make sure that you give it an appropriate name; names like 'Module1', 'Module2', or 'Module3' aren't of any use to anyone.

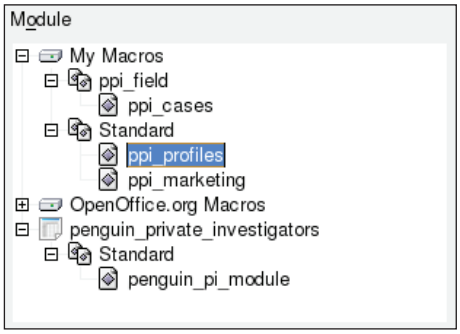
We've also learned something new about modules while we've been discussing libraries:

- Libraries are actually directories that contain two key files—`script.xlb` and `dialog.xlb`. These are the indexes to all of the modules in the library.
- Modules are actually `xba` files in the library directory. These contain the dialog definitions or the code for macros.

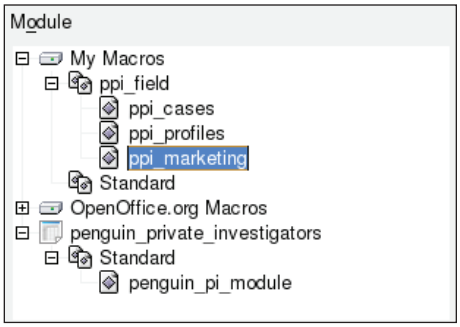
We're nearly ready to start writing macros, but there are a couple of points to consider before moving on. The first is renaming modules—you may not always name a module correctly, or you may decide that it's name isn't appropriate:




Use the Organizer to change the name of a module. Just click on the module name (you may have to click a couple of times), and then you can change the text to whatever is suitable for your project:



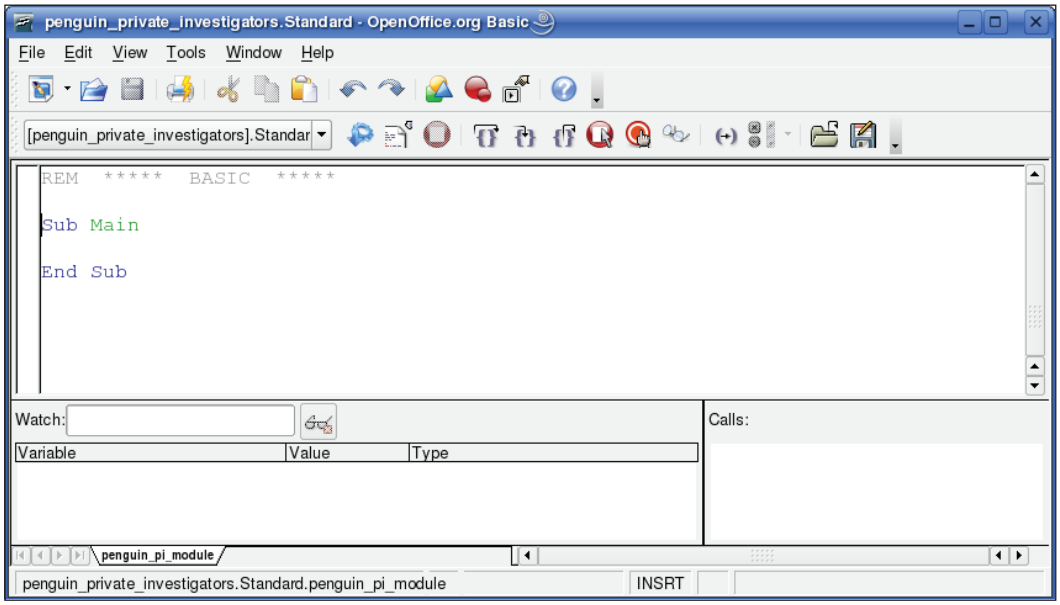
While you're looking at your modules, you may decide that they are in the wrong library, for example, if you've created them all in **Standard** instead of a custom library. In such a case, you can drag-and-drop modules from one library to another:




You should now be confident in managing and using your libraries and modules. All you have to do is start writing macros, and so that's exactly what we're going to do next.

# Writing Macros

You'll remember from Chapter 1 that we use the OOo Basic Editor to work with macros:



It's here that we can edit, run, save, and debug the code. You'll also remember that a macro is automatically created for us (**Main**). This is the macro that OOo will run when we click the **Run** button (as long as you keep **Main** as the *first* macro in the module).

Now, there's something that you may be wondering about in the definition of the macro. Why does it say:

```
Sub Main
End Sub
```

and not:

```
Macro Main
End Macro
```

That's because *macro* is actually the generic name for **Subroutines** and **Functions**, and it's these that we're going to write. Let's start by having a look at subroutines.

# Writing Subroutines

If you look at **Main**, then you can see the structure of a typical subroutine:

- Start the subroutine with the `Sub` statement.
- Give the subroutine a unique name.
- End the subroutine with an `End Sub` statement.

So, to create a new subroutine called `ppi_add_user`, you would write:

```
Sub ppi_add_user
End Sub
```

OK, we've got a subroutine, but it hasn't got any functionality yet. So that's the next thing to do:

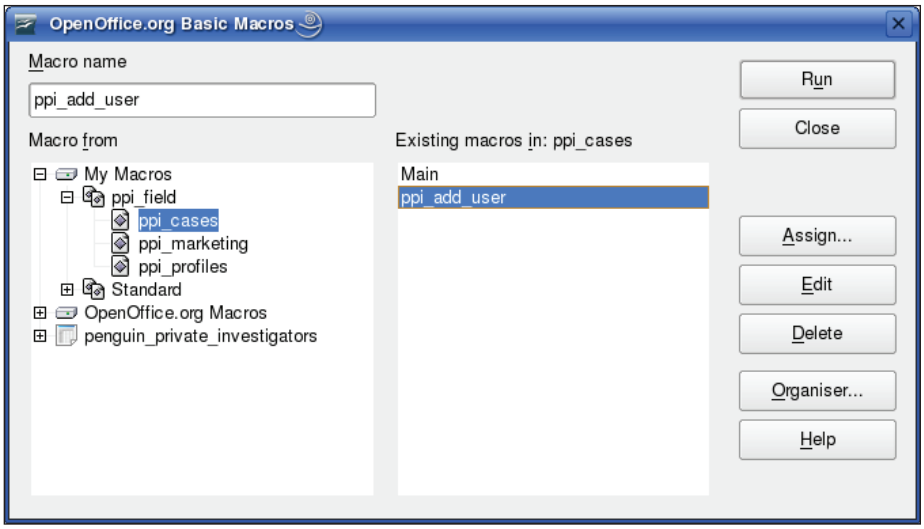
```
Sub ppi_add_user
    Dim fname as String      ' First name
    Dim sname as String      ' Surname (Last name)
    Dim username as String   ' User Name
    fname = "Fred"
    sname = "Smith"
    username =lcase(sname+mid(fname,1,1))
    msgbox "User Name for " & fname & " " & sname _
    & " is " & username, 0 ,"User Name"
End Sub
```

There are two ways in which you can see this in operation, and it really comes down to personal choice. The first (and the way that I prefer to do it) is to modify the `Main` subroutine:

```
Sub Main
    ppi_add_user
End Sub
```

And then click the **Run** button (don't forget that `Main` must be the *first* macro in the module).

The second way is to display the Select Macro window, select the macro, and then click on **Run**:




Whichever way you prefer the end result will be a message box:


Having seen the macro (or subroutine) in action, it's worth spending a little time analyzing the code itself—by doing this, we'll see the fundamental operations that any subroutines will need to carry out.

## Declare Variables

This first thing that you need to do (in this or any other subroutine) is to declare the variables that are going to be used:

```
Dim fname as String
Dim sname as String
Dim username as String
```

The `Dim` statement is used to declare variables. In this case, three have been created, each of type `String`.

## Assign Values to the Variables

Once you've declared a variable, then you need to assign a value to it:

```
fname = "Fred"
sname = "Smith"
```

## Do the Work!

Having declared the variables that you need and set their values, you can now get the subroutine to carry out the job that you need:

```
username =lcase(sname+mid(fname,1,1))
msgbox "User Name for " + fname + " " + sname _
+ " is " + username, 0 ,"User Name"
```

In this example we're creating a user name by taking the first letter from the variable `fname` and appending it on to the end of the variable `sname`. The result is then displayed in a message box.

## Inputting Variables

Now you're probably thinking that very few of the clients are likely to be called `Fred Smith` and so the subroutine is of limited use. Fortunately, we can rewrite the subroutine so that it can accept any surname and any first name:

```
Sub Main
    ppi_add_user("Fred", "Smith")
End Sub
Sub  ppi_add_user (fname as String, sname as String)
    Dim username as String
    username =lcase(sname+mid(fname,1,1))
    msgbox "User Name for " + fname + " " + sname _
    + " is " + username, 0 ,"User Name"
End Sub
```

# Writing Functions

So what's the difference between a subroutine and a function? Well, very little actually. In fact the *only* difference is that the function returns a *result*—something that you can load into a variable. We've already seen a couple being used, like `lcase` and `mid` in our subroutine.

To understand this better, let's convert the `ppi_add_user` from a subroutine into a function:

```
Sub Main
    Dim fname as String
    Dim sname as String
    Dim username as String
    fname = "Fred"
    sname = "Smith"
    username = ppi_add_user(fname, sname)
    msgbox "User Name for " & fname & " " & sname _
    & " is " & username, 0 ,"User Name"
End Sub
Function ppi_add_user (fname as String, sname as String) as String
    ppi_add_user = lcase(sname+mid(fname,1,1))
End Function
```

There are two key things to take note of:

1. The function has a variable type; this is what is returned from the function when you run it.
2. The function just does the core work; everything else is moved up to the `Main` level.

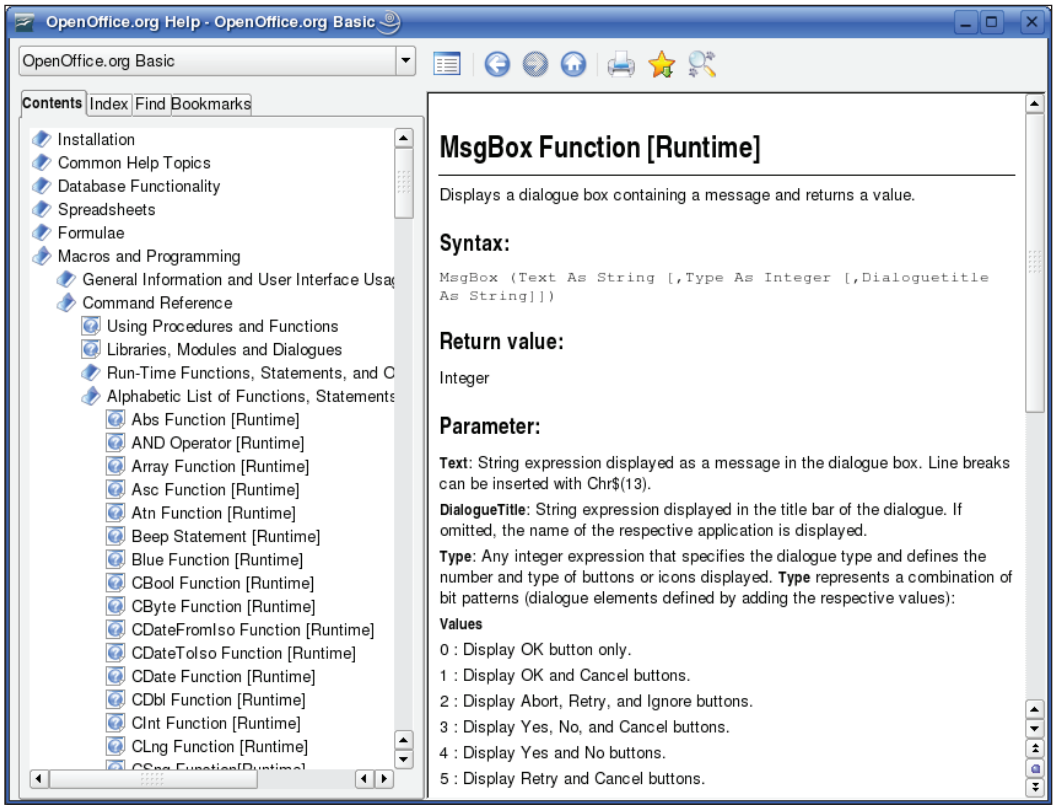
# Getting more Information

You'll have noticed that we've not gone into a great amount of detail here. For instance:

- We've seen how to declare variables, but not what the different variable types are.
- We've seen the functions `lcase`, `mid`, and `msgbox` in action, but there are no technical details on them.

Quite simply, we haven't got enough room here to discuss all of the functionality that is built into OOo and that's available to us. Fortunately, the IDE has all the information that you require. If you need to know more, then highlight a keyword

(for example, `Dim` or `msgbox`) in the Basic Editor, and then press *F1*. You will see a screen similar to the following screenshot:




You'll find that there is a complete listing of all of the OOo Basic functions that you can use in your macros.

# Subroutines and Functions in Different Libraries

Finally, since we've gone through a lot of trouble creating libraries, let's look at using them. Each library contains modules and each module contains macros (and/or dialogs). You may, therefore, be wondering how you access each subroutine or function. The answer is that OOo doesn't really care where the macros are; as long as you have an access to a library, you have access to all of the macros that it contains.

So, in the examples that we've already looked at, the `Main` subroutine and the `ppi_add_user` function can be in completely different modules or libraries.

However, this does mean that you need to be careful with the naming of your macros. For example, if you have two modules each containing a macro called `ppi_add_user`, then you've got potential conflict—how can you guarantee which one OOo is going to use?

If you're concerned about such a situation, then you can tell OOo to use a specific macro by including the library and module names:

```
username = ppi_field.ppi_cases.ppi_add_user(fname, sname)
```

With that you're able to create custom libraries, write macros, and make use of the macros wherever you need them.

# Summary

In this chapter we've covered creating and using libraries, modules, subroutines, and functions. We have seen how to use the Organizer to create your own custom libraries rather than using the **Standard** library, how to give your library a useful name, and how to make use of other people's libraries by using the **Insert** button. We have also cover adding a library to the OpenOffice.org Macros area by copying a user's library into OOo's main area (remember to modify the `script.xlc` file once you've done that).

This chapter also taught you how to store modules in libraries, give each module a relevant name, try to group similar modules in the same library, rename modules and move them between libraries using the Organizer. You have learned what subroutines and functions are, how to declare variables using the `Dim` statement, how to use Calc's help to find out the use of any of the OOo's inbuilt functions and variable types.

In Chapter 3, we'll start to learn how to get the best out of Calc by using the OOo Object Model.

# 3
# The OOo Object Model

Pygoscelis didn't look back; he just ran. He didn't look back as he saw a gun sticking out of his office window. He ran first down one alley, then down another, and then dived into the blackness of a doorway.

Standing in the darkest corner, he forced his breath to calm. He checked his inside coat pocket, just to make sure that the disk was safe. Relieved he relaxed, and started to consider his options. He needed help and he needed to be objective above all.

And that's exactly what we're going to be doing in this chapter. We're go to be objective—that's to say that we're going to start looking at OpenOffice.org's **Universal Network Objects** usually referred to as **UNO**s. UNOs are the platform- and language-independent objects that make up OOo Calc, and which we can make use of in order to make truly powerful macros, utilizing every aspect of the OOo environment.

Now this chapter is split broadly into two sections:

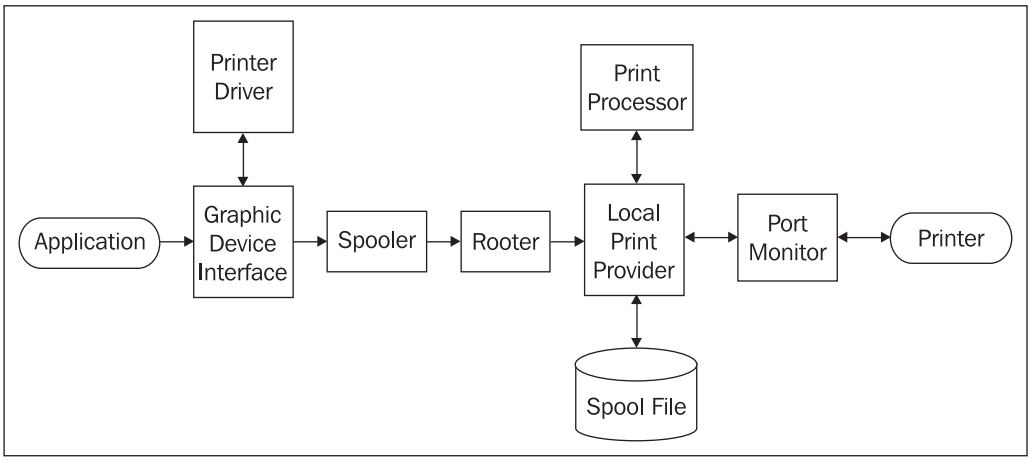
- The first section is practical, and will show you how to access UNOs and to carry out some basic, but essential, activities.
- The second section has no practical work for you to do; it's all reference material. Therefore I'd be surprised if you read it from end to end in one go. However, you will find it useful to come back to it from time to time as we look at more advanced macros in the later chapters.

That said, it is important that you have at least read enough of this chapter so that you:

- Understand the OOo Object Model
- Understand why UNOs are so useful
- Know how to access and use the UNO objects
- Know where to find any more information that you need

# Why be Interested in UNOs?

How do you print a Calc document? The easy answer is: Just press the print button. Not difficult. However, the background process that you've set in motion is complicated.



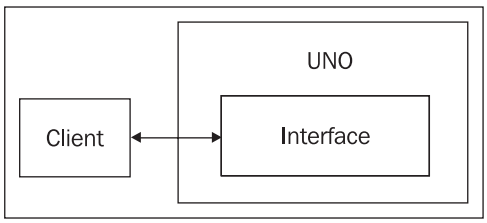

So, without that print button you'd need to have quite a bit of in-depth knowledge about all the system interactions that need to be carried out just to produce a single piece of paper.

But you don't have to worry about all of that when you're writing macros, you just need to know which UNO will do all of the work for you.

# Overview of the OOo Object Model

We've already learned that we're going to be working with UNOs—OpenOffice.org's Universal Network Objects—and to understand them better, we're actually going to start at the bottom and work our way upwards.

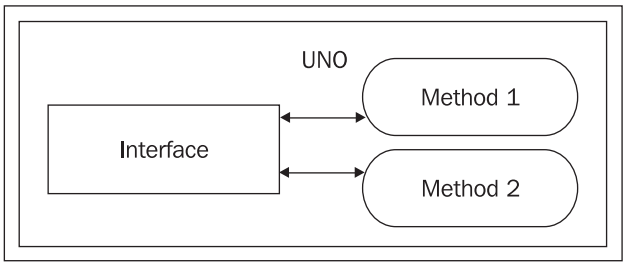
At its simplest level, we've got a client (your macro) that interfaces with, well, with an interface:



# The Interface

Each interface is simply made up of a set of one or more methods, and you can use these methods to:

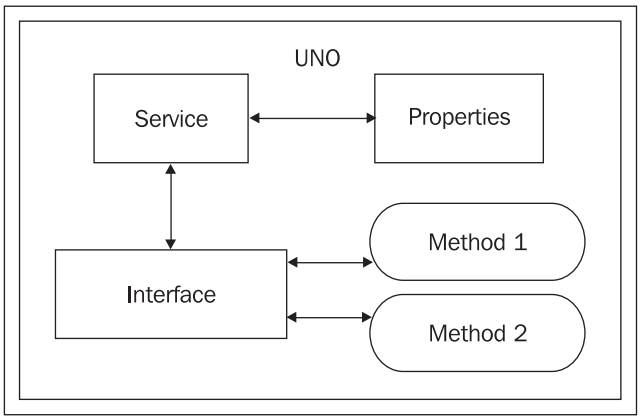
- Get or set parameters
- Control the operation of any functionality that the interface defines




Every interface is contained in a service.

# The Service

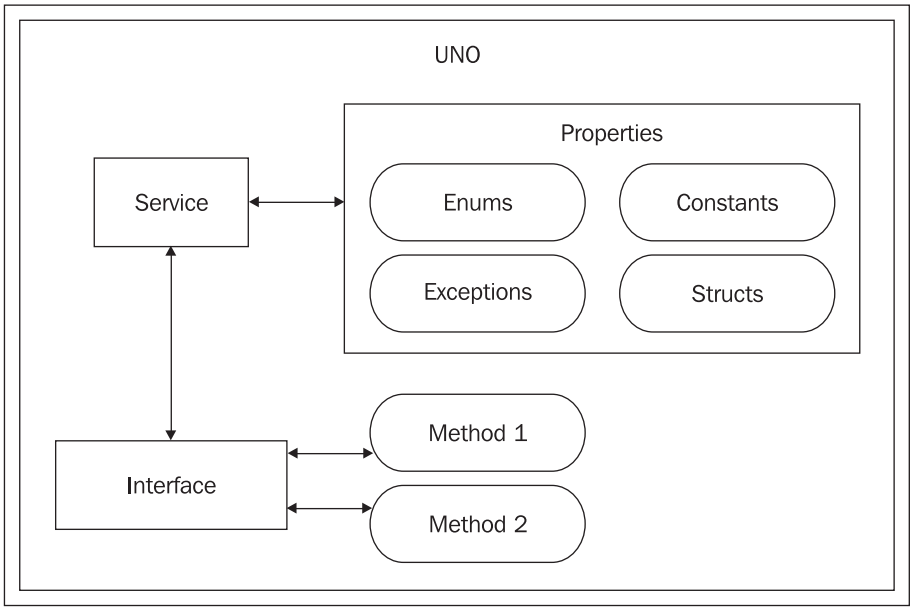
A service is a UNO component—the building block of OOo Calc. Each service consists of one or more interfaces and it has a set of types associated with it (and yes, you would be right it thinking that the interface is a type as well):




You'll find that there are four types associated with the services:

- Constants
- Enums

- Exceptions
- Structs




It'll be obvious to you what three of the property types are used for—constants, exceptions, and structs. But what about enums? And no, they're nothing to do with food additives. An enum is a set of numeric values, grouped together according to their use.

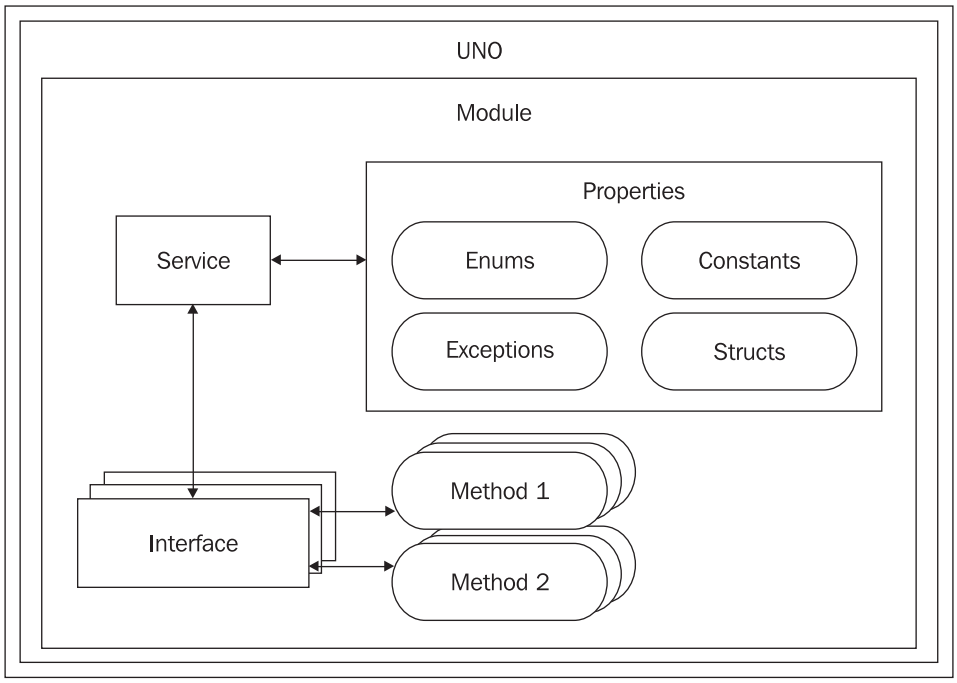
So, we've seen that:

- An interface consists of one or more methods
- A service consists of one or more interfaces
- A set of properties (constants, enums, exceptions, and structs) is associated with the service

You'll find that similar services and their properties are grouped together into modules.

# The Module

UNO services are grouped hierarchically into modules:




We've now moved all the way from the most basic level of the UNO right up to the top level, well nearly. Some of the modules that we'll use will be nested inside another module. In fact all modules that we'll use are nested in one central module—`com.sun.star`.

We've now completed the general overview of the OpenOffice.org Object Model. You should now at least have a basic understanding of what a UNO is. We'll be using UNOs and UNO services throughout the rest of the book, and so if you want to see them in action, then carry on reading.

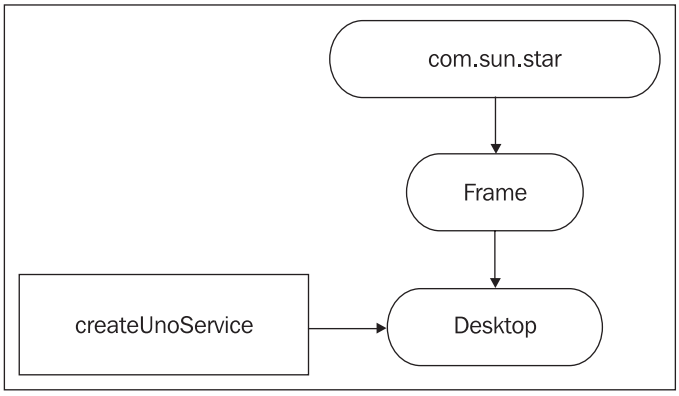
# Starting to Work with UNOs

I'm sure you'll agree that the first step towards complete automation is the opening and closing of a spreadsheet, so let's start by seeing how OOo's UNOs can help us do that.

# Opening and Closing Spreadsheets Automatically

There are a couple of things that you need to know before you can use a service:

- Each service is stored in a module.
- All modules are stored within a central module—`com.sun.star`.
- A service is created by using the `createUnoService` function.
- We're going to be using the `Desktop` service, and this can be found in the `frame` module:




So, let's look at using a UNO in a very simple subroutine that opens and closes a file:

```
Sub OpenAndClose
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oUrl as String
  oDesk = createUnoService ("com.sun.star.frame.Desktop")
  oUrl = "private:factory/scalc"
  oDoc = oDesk.loadComponentFromURL (oUrl, "_blank", 0, Array() )
  oDoc.close(true)
End Sub
```

OK, all that happens if you run this is that a blank Calc sheet is displayed on your screen for a moment; not terribly useful, but it does show that you now have control of your spreadsheets.

What about existing files? All you have to do is change `"private:factory/scalc"` to the name of the spreadsheet that you want to open. Well, nearly. As you've probably noticed the command for actually opening the spreadsheet is

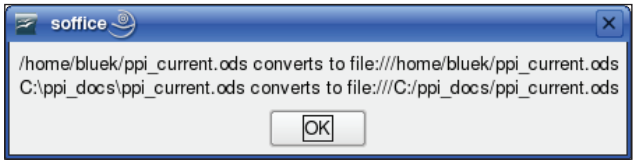
`loadComponentFromURL`—notice the *FromUrl*. Therefore the funtion expects an input, something like:

- `file:///home/bluek/ppi_current.ods` for Linux
- `file:///C:/ppi_docs/pi_current.ods` for Windows

However, don't worry, you don't have to carry out this conversion yourself, just use the `convertToUrl` function:

```
Sub exampleConversion
  Dim f1 as String
  Dim f2 as String

  f1 = "/home/bluek/ppi_current.ods"
  f2 = "C:\ppi_docs\ppi_current.ods"

  msgBox _
    f1 & " converts to " & convertToUrl (f1) & chr (10) & _
    f2 & " converts to " & convertToUrl (f2)
End Sub
```

If you run this code, then you'll see a message box showing the original file names and their conversions:




Now, if you're anything like me, then you'll have taken this information; modified the `OpenAndClose` subroutine, run it and then seen the error:


The message looks confusing, but all it means is that the macro can't find the file that you've input, probably because it doesn't exist yet. The solution is simple, amend the subroutine so that it:

- Checks to see if the file exists
- Creates a new spreadsheet if the requested one doesn't exist
- Saves the spreadsheet before it closes

So we can easily make these changes to our `OpenAndClose` macro:

```
Sub OpenAndClose
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oFile as String
  Dim oUrl as String
  Dim oUrlTemp as String

  oDesk = createUnoService ("com.sun.star.frame.Desktop")
  'Change the file name to one that you can write to
  oFile = "/home/bluek/ppi_current.ods"
  oUrl = convertToUrl (oFile)

  'Check that the file exists. If it doesn't then use blank spreadsheet
  If fileExists (oFile) Then
    oUrlTemp = oUrl
  Else
    oUrlTemp = "private:factory/scalc"
  End If

  oDoc = oDesk.loadComponentFromURL (oUrlTemp, "_blank", 0, Array() )

  oDoc.storeAsUrl (oUrl, Array()) 'Save the file
  oDoc.close(true)
End Sub
```

If you run the macro again, then you won't see any errors; the spreadsheet will just open and then close. The only big difference is that if your spreadsheet did not originally exist, then the macro will create it for you.

Of course, if you stop to think about it, this isn't a terribly efficient way of writing the code. You don't want to put in this check for every spreadsheet that you open, do you? By putting the file-opening section into a function, you make the code usable elsewhere (meaning that in the long run you'll have to write less code) and it simplifies the subroutine that you've already got.

There's something else that we can do to make our lives easier. We've been creating the Desktop service ourselves, but in fact this is done automatically when Calc opens, and the object that is created is called `StartDeskTop`. This means that instead of:

```
    oDesk = createUnoService ("com.sun.star.frame.Desktop")
    oDoc = oDesk.loadComponentFromURL(oUrlTemp, "_blank", 0, Array())
```

We can just use:

```
    oDoc = starDeskTop.loadComponentFromURL(oUrlTemp, _
           "_blank", 0, Array())
```

The result is:

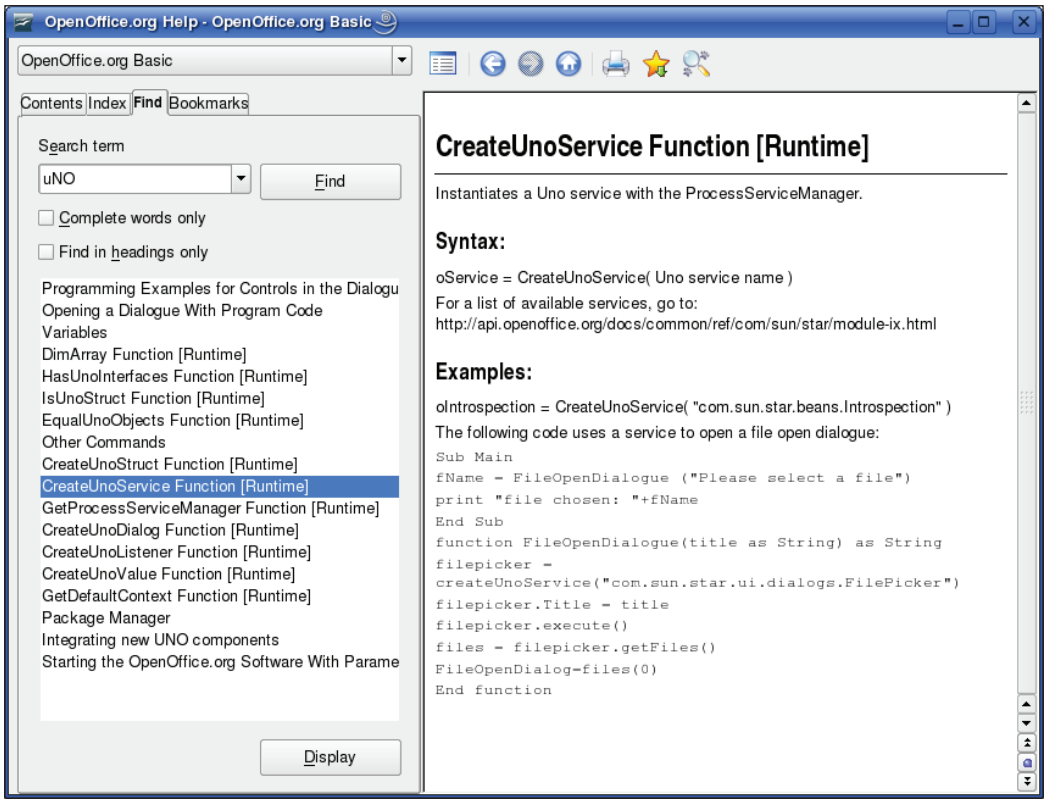
```
Function openSpreadSheet (iFile as String) as Object
  Dim oUrl as String

  If fileExists (iFile) Then
    oUrl = convertToUrl (iFile)
  Else
    oUrl = "private:factory/scalc"
  End If

  openSpreadSheet = starDeskTop.loadComponentFromURL _
    (oUrl, "_blank", 0, Array() )
End Function

Sub OpenAndClose
  Dim oDoc as Object
  Dim oFile as String
  Dim oUrl as String
  Dim oUrlTemp as String

  oFile = "/home/bluek/ppi_current.ods" 'The file you want to open
  oUrl = convertToUrl (oFile)

  oDoc = openSpreadSheet(oFile)
  oDoc.storeAsUrl(oUrl, Array())
  oDoc.close(true)
End Sub
```

Now that you can open and close a spreadsheet, the next obvious thing to consider is actually changing its contents and that's what we will do in Chapter 4, *Using Macros with Spreadsheets*.

However, in the meantime, we'll have a look at where we can find additional information about UNO's.

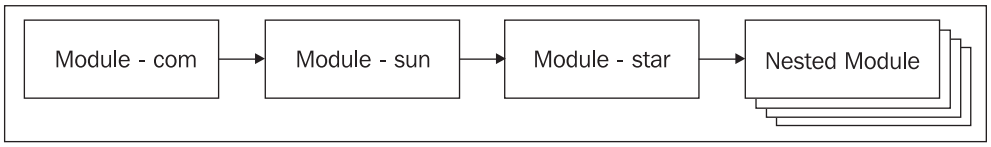
# Online Reference Material

You may be wondering why we're interested in online reference material: what about Calc's built-in help system? Well, you will find it useful for general information on how to use UNOs, but it won't tell you anything about using specific UNOs:

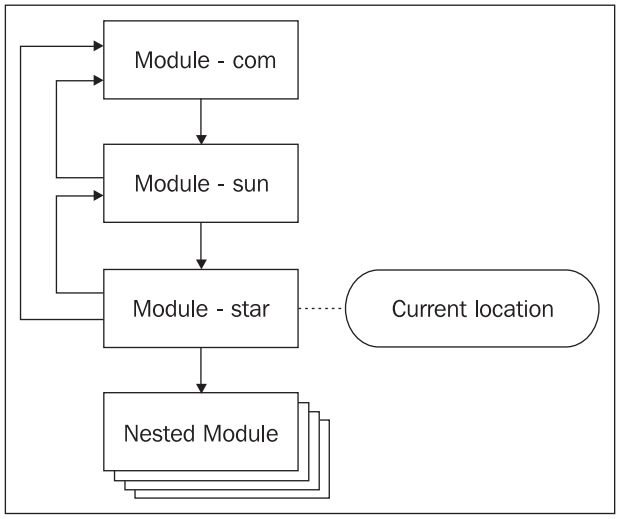



We need to look at OpenOffice.org's online information to learn about the actual UNOs that are available. So it's internet-browser time:


As soon as you look at the web page you'll realize that it contains all of the nested modules for `com.sun.star`. In addition to this you may surmise (quite rightly) that the website structure matches the top-level structure of the OOo object model that we've been discussing in this chapter:

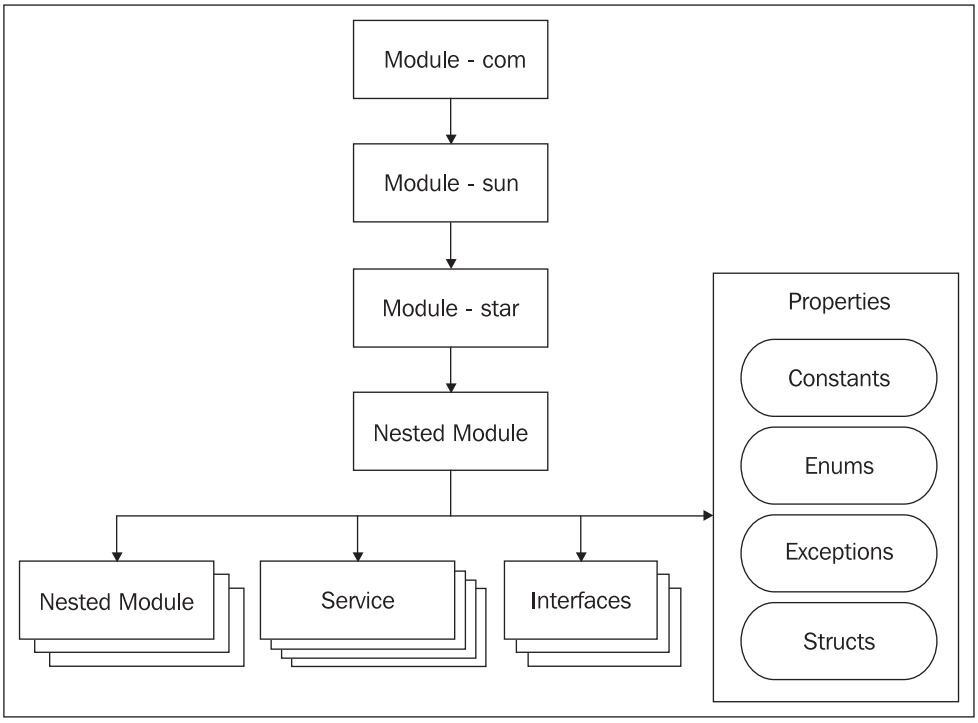



In fact, you'll find that the website will allow you to move to any point in the structure upwards from the current location and to the next position downwards:




If you click on any of the nested modules, you'll see all of the elements that we would expect from the object model:

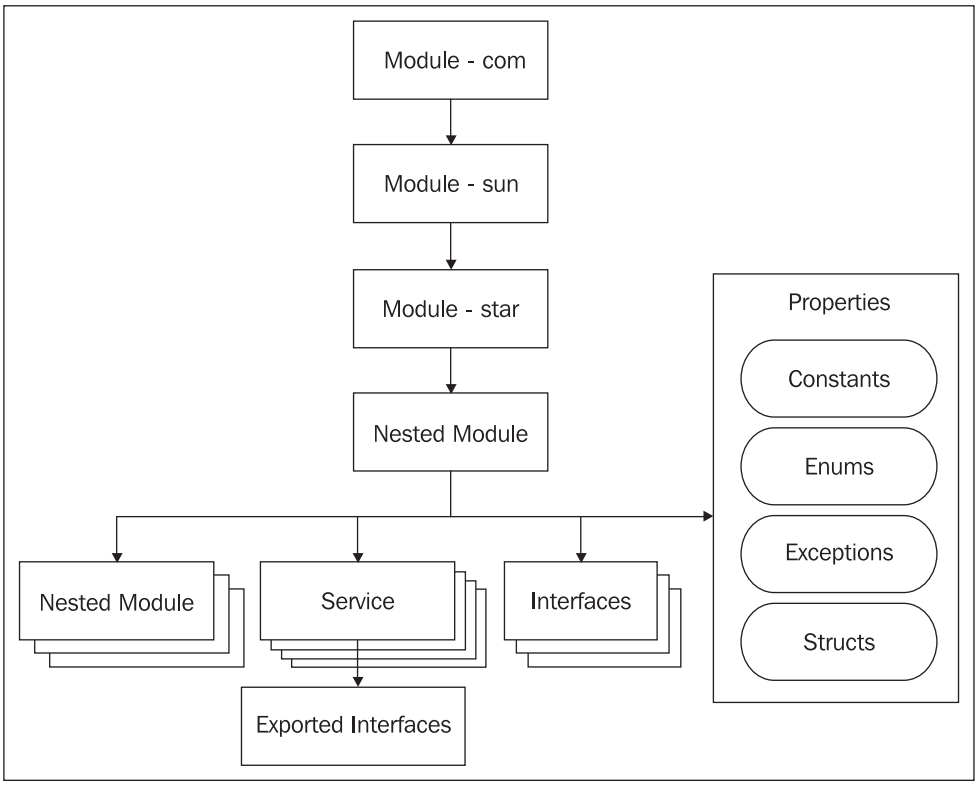
- Services
- Interfaces
- Constants
- Enums
- Exceptions
- Structs




You may find that some of the modules also contain:

- Further nested modules
- A list of interfaces used by the services in the module

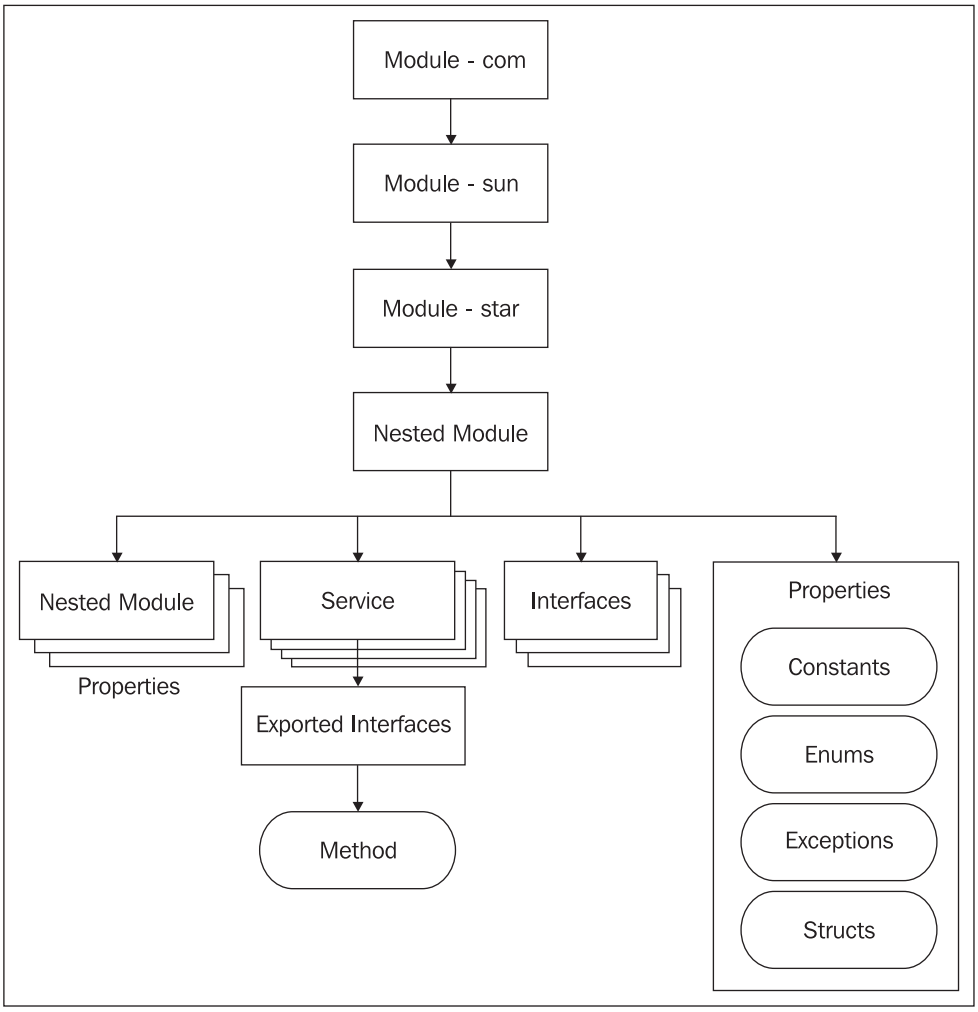
We can follow the structure by clicking on any one of the services in the module.




Up until now we've only been dealing with the general structure of the UNO; however, you'll find that the website will now start to give you much more in-depth information:

- Detailed description of the interfaces.
- Detailed description of the properties that can be used with the service.
- Links to related services.
- Links to related developer pages. These can be useful when working out how to use an interface. However, the examples do tend to be for C++ and Java rather than Basic.

Finally, you can click on the link to one of the interfaces to complete your tour of the website.




This last page will give you:

- A list of all of the methods available to the interface
- Detailed descriptions of how to use each method
- Links to associated developer pages for examples of using the methods

So, we've now seen that we can use the OpenOffice.org website to:

- Understand the structure of each UNO
- Learn how to use the methods built into each UNO
- Access developer pages to see examples of using the UNOs

The next step is to look at a real UNO and to build up a picture of it.

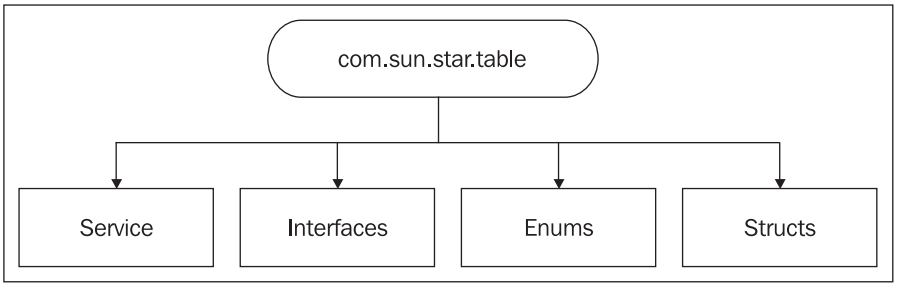
# A Real Example: Using the Table UNO to Access a Cell

We've only discussed generalities so far. Next we'll look at one of the UNOs that we'll be using on a day-to-day basis as we write macros: the table. Why look at the the table? Quite simply because that's all a worksheet is—a table. Learn to control the table and you can control the worksheet. In particular, we'll look at how the table UNO can be used to access a cell in the worksheet.

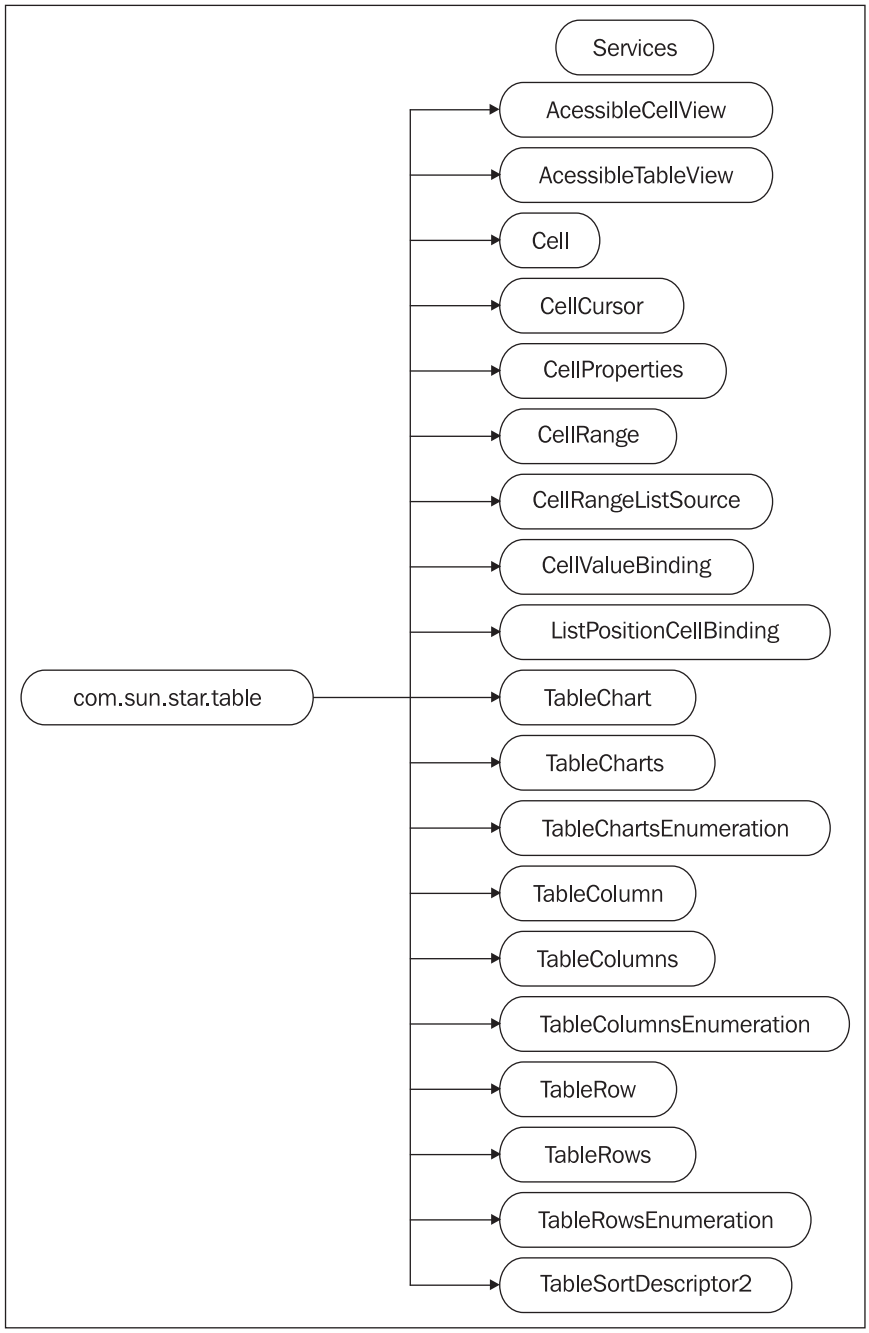
Our starting point is, of course, the OpenOffice.org website `http://api.openoffice.org/docs/common/ref/com/sun/star/module-ix.html` and the module `com.sun.star`.

From there we can follow the link to the to the `table` module:

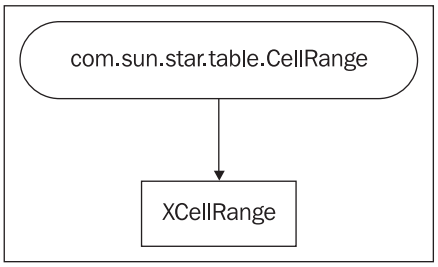
From the `table` module web page, we can see the contents of the module:



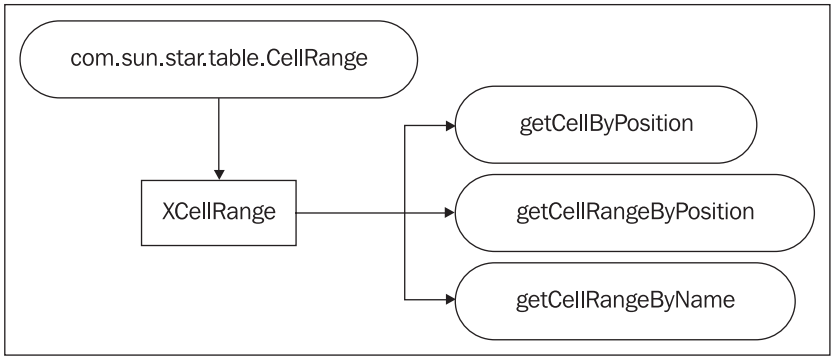
However, at the moment we're only interested in the services:



As you can see, there are quite a number of table-related Services, but we're only going to be looking at the `CellRange` service, and its `XCellRange` interface:




When we look on the interface page we can see that there are three methods for accessing the cells:



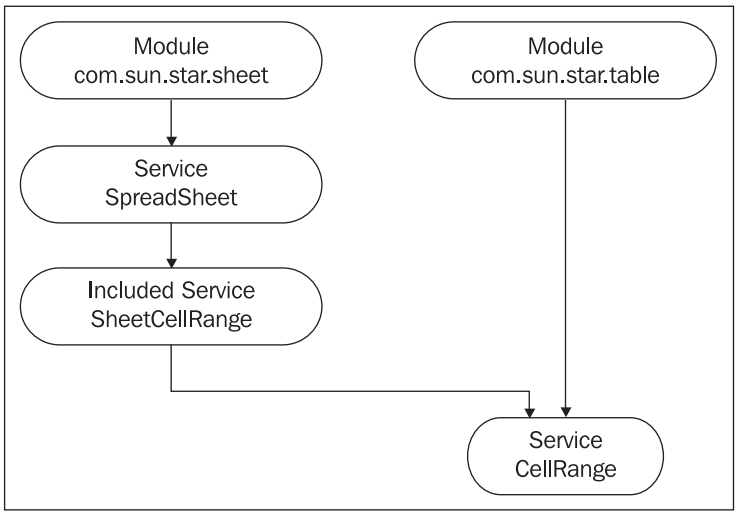

We'll see a number of macros that use these methods when we move on to Chapter 4. For now it's just a matter of getting used to the structure of the UNO.

# Services within Services

We've seen that we can use the `getCellRangeByName` method through the `CellRange` service in module `com.sun.star.table`, and you may be wondering if you have to call up this service every time that you what to use the method. The easy answer is 'yes, you need to'. However, sometimes this is done automatically for you.

Quite often you'll find that a service will have an included service (and of course this is the whole point of using a component model). It means that any method only has to be written once, and then reused wherever it is necessary.

So where else is `CellRange` used? In fact it is, as you would expect, re-used in the `spreadsheet` service:




So this means that if you use the `spreadsheet` service, then you automatically have access to the `CellRange` service.

# Finding Included Services

If you want to know if a service is available elsewhere as an included service, then there are two ways of going about it:

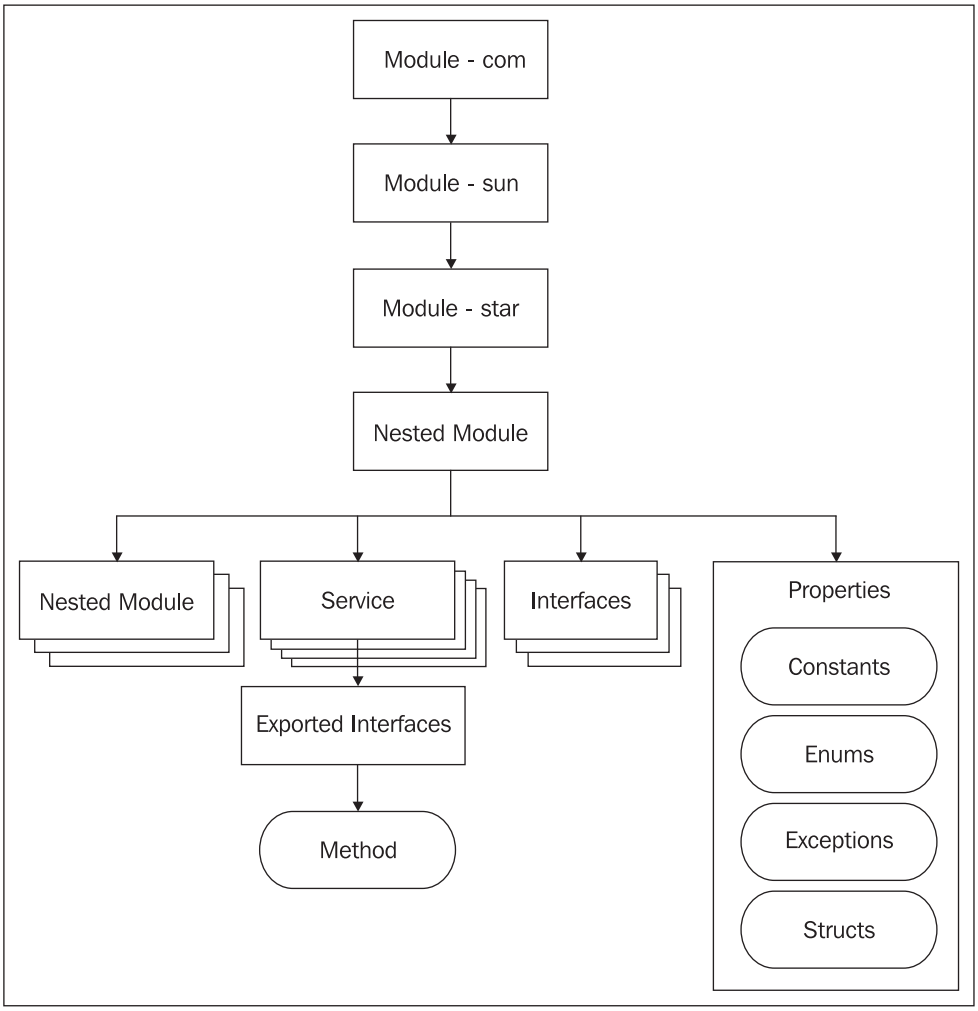
1. Look through the documentation for the service that you are going to be using and see if it includes any other services (the one that you want access to).
2. Go to the documentation for the service that you want to include and see if it has a 'Use' link enabled.

In the case of the `CellRange` service, you'll find that the link is enabled and that the service is used in:

- `com.sun.star.sheet.SheetCellCursor`
- `com.sun.star.sheet.SheetCellRange`
- `com.sun.star.sheet.Spreadsheet`

# List of Everything You Want to Know About UNOs

We've learned our way around the OOo Object model, and we now know our way around the online documentation:




However, what if you know which method you want to use, but you don't know which service it belongs to? For example, if you remember, we started this chapter by considering the printing process. Apart from trial-and-error, how can we learn about the print method; after all there must be one, mustn't there?

You'll be pleased to know that the answer is very simple: you can either use the 'Index' link from any of the documentation pages, or go directly to the OOo's global index:


Here you'll find a complete A-Z of every module, service, method, constant, enum, exception, and struct. So, if we look up 'Print' we find three references:

- **PRINT**: Constant in constants group `com.sun.star.awt.KeyFunction`
- **Print**: Property in service `com.sun.star.text.BaseFrameProperties`
- **print()**: Function in interface `com.sun.star.view.XPrintable`

So we now know that there is a `print` function in the `com.sun.star.view.XPrintable` interface. The next question is 'Where can this interface be used?'. The answer is, of course, to follow the link to the `XPrintable` interface documentation page, and then click on **Use**. From this we can learn that there are a number of services that support the interface:

- `.com.sun.star.text.AdvancedTextDocument`
- `.com.sun.star.drawing.DrawingDocument`
- `.com.sun.star.drawing.GenericDrawingDocument`
- `.com.sun.star.text.GenericTextDocument`
- `.com.sun.star.text.GlobalDocument`
- `.com.sun.star.text.HypertextDocument`
- `.com.sun.star.sdb.OfficeDatabaseDocument`
- `.com.sun.star.document.OfficeDocument`
- `.com.sun.star.presentation.PresentationDocument`
- `.com.sun.star.sheet.SpreadsheetDocument`
- `.com.sun.star.text.TextDocument`
- `.com.sun.star.text.WebDocument`

You can see for yourself that the `XPrintable` interface (and therefore the `print` function) is available to the whole suite of OOo documents (no surprise there then). And to you, as you write your macros, it means that you don't have to do anything complicated when printing a spreadsheet, you just print it.

And really, that's the whole point of the OpenOffice.org UNOs—they are there to make your life as easy as possible.

# Summary

In this chapter we've been introduced to OpenOffice.org's UNOs and the OOo object model. We've learned that UNOs are OOo's Universal Network Objects and are the components that give Calc all of its functionality. Each UNO consists of: Interfaces, Services, and Types. The four types associated with services are: Constants, Enums, Exceptions, and Structs.

We also learned where to find OOo's online documentation on UNOs, and how they allow you to explore the structure modules, services, interfaces, methods, and properties of each UNO:

"OK, that's enough objectivity for one day." Pygoscelis muttered under his breath, "I know what I've got to do now."

And so do you. Move on to Chapter 4, *Using Macros with Spreadsheets*.

# 4
# Using Macros with Spreadsheets

The dark street was devoid of life now, and the shadows held sway, filling the doorways and alleys with darkness. Joe Public was sensibly tucked up in bed, either fast asleep or listening to the rain hammer against the window panes. As Joe unconsciously snuggled closer to his warm wife, three patches of blackness sped from the door of Pygoscelis P. Ellsworthy's office to a waiting car. Immediately it sped off into the howling gale that was now building itself into a frenzy.

In a doorway across the street a shadow that was just a little darker stirred. For a brief moment a face was lit as a mobile phone flicked open.

"Korora, it's Pygoscelis here. Don't talk, just listen. I have the disk, but the office has been compromised. Make for the safe house as soon as you can. We're going to have to rewrite the macros from scratch."

With that a shadow walked quickly down the street and into the nearest alley. In the darkness of Pygoscelis' doorway another dark shape stirred, and spoke into its own phone.

"He's on the move"

OK. Admit it. This is why you're reading this book, isn't it? You want to be writing macros, macros that can manipulate your spreadsheets. In fact, you've probably skipped straight here if you're just wanting to get on and get coding. And quite right too. However, if you have been following the book chapter by chapter, then you should now be able to:

- Find your way around the OOo IDE
- Understand using libraries, modules, subroutines, and functions
- Understand the basics of using objects in OOo by making use of the UNO services

In this chapter, we'll be bringing all of these elements together as we start to build macros that can fully automate your spreadsheets. By the end of the chapter, you should be confident enough to do the following:

- Open and close files
- Work with multiple spreadsheets
- Manipulate the data within a spreadsheet
- Work with built-in OOo functions
- Work with cells and ranges of cells

And all this is done using the power of macros.

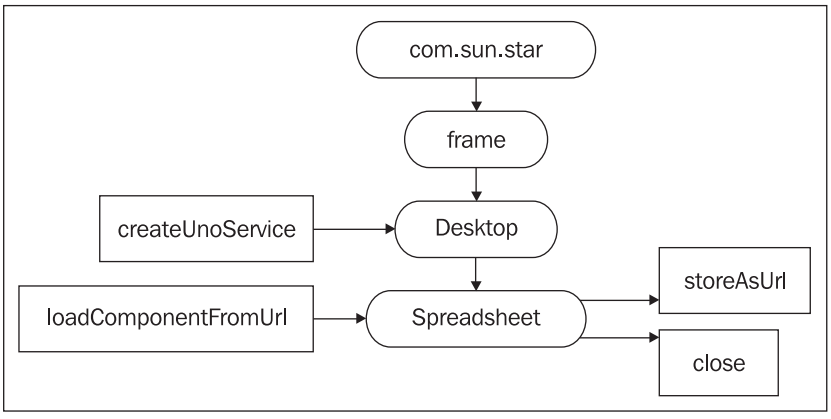
# Opening and Closing Spreadsheets

In Chapter 3 we saw just how easy it is to open and close spreadsheets:

1. Use the `starDeskTop` object to access the spreadsheet (since Calc automatically creates the Desktop UNO service for us).
2. Load the spreadsheet by using the `loadComponentFromURL` function.
3. Use the `close` subroutine to finish with the spreadsheet.

We also saw that we can easily control the spreadsheets themselves:

1. Use the `fileExists` function to decide whether to use an existing file or if it doesn't exist then open a blank file.
2. Use the `storeAsUrl` subroutine to save the spreadsheet.

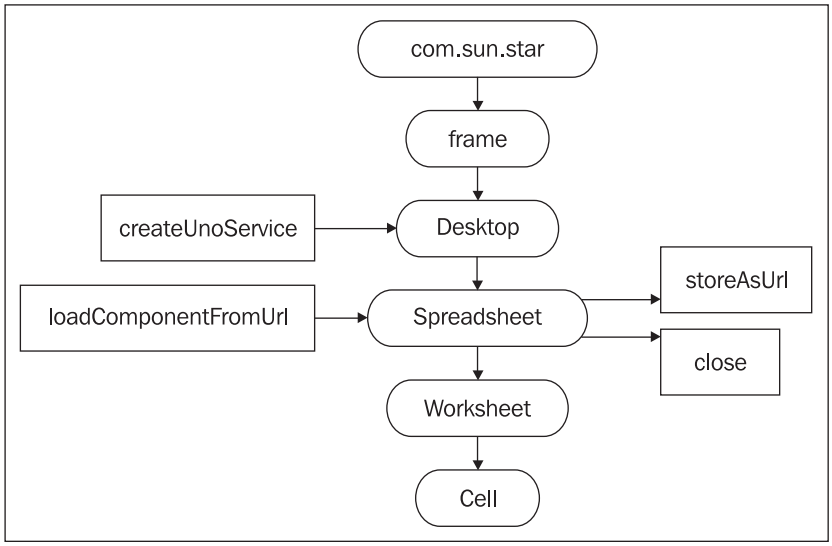


Having mastered working with the opening and closing of a spreadsheet (which is important), you're probably thinking that it's time to start manipulating the contents of the spreadsheet—and that's exactly what we'll do next.

# Manipulating Spreadsheet Cells

It would seem obvious to assume that we should be able to access the cells in the spreadsheet using some object or function, especially because of what we learned about OOo UNO objects in Chapter 3. And, as you would expect:

- Each spreadsheet that you create contains one or more worksheets (by default there will be three worksheets, and as a matter of interest there can be a maximum of 256).
- Each worksheet consists of a number of cells (8,192,000 of them arranged in 256 columns by 32,000 rows).




And, of course it is easy to access each worksheet and its cells:

- Each worksheet can be identified by an index number (0 to 255).
- Each cell can be identified by its position with a grid for which we can use the `getCellByPosition` method.

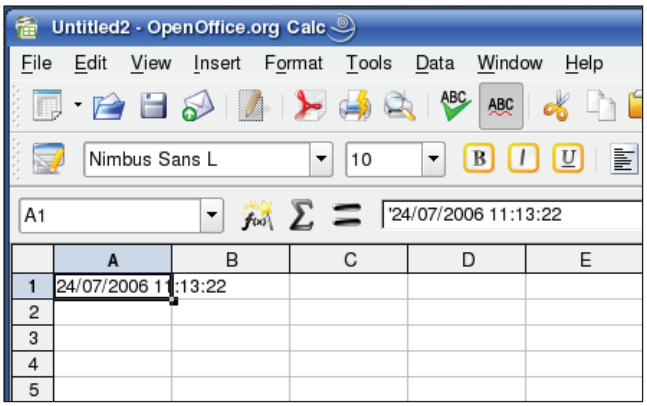
Let's start by looking at a simple subroutine that fills the contents of a cell in a worksheet:

```
Sub singleFile
  Dim oDoc as Object
  Dim oSheet as Object
  Dim oCell as Object

  oDoc = starDeskTop.loadComponentFromUrl _
    ("private:factory/scalc", "_blank",0,Array ())
  oSheet = oDoc.sheets (0)
  oCell = oSheet.getCellByPosition (0,0)
  oCell.String = now 'This function returns the current date and time
End Sub
```

You can see that the macro:

- Opens a blank spreadsheet
- Gets the first worksheet
- Gets the top left cell
- Writes information to the cell

So if you run this macro, you'll end up with a new spreadsheet with the current date and time entered into a cell:




There are a couple of things we've already mentioned, and that you may have deduced from the macro:

- The worksheets have indexes associated with them, allowing us to access them as if they were in an array. This means that in a default spreadsheet with three worksheets, sheet(0) is the first worksheet, sheet(1) is the second, and sheet(2) is the third.

- The cells are accessed by using the `getCellByPosition` function. This needs the column number and row number to be input (and in that order).

There's something else that you may have wondered about—the line:

```
oCell.String = now
```

The `now` part is obvious enough—it places the current date and time into the cell—however, it is the way that it puts it into the cell that's interesting. Information must be placed in a cell in one of three ways—Formula, String, Value.

This can be important. For example, let's consider these lines of code:

```
oCell = oSheet.getCellByPosition (0,1)
oCell.Value = 20
oCell = oSheet.getCellByPosition (0,2)
oCell.Value = 30
oCell = oSheet.getCellByPosition (1,1)
oCell.String = "=A2+A3"
oCell = oSheet.getCellByPosition (2,1)
oCell.Formula = "=A2+A3"
oCell = oSheet.getCellByPosition (3,1)
oCell.Value = "=A2+A3"
```

Although we've entered exactly the same information, the results are completely different:

| | | | | |
|---|---|---|---|---|
| 2 | 20 | =A2+A3 | 50 | 0 |
| 3 | 30 | | | |

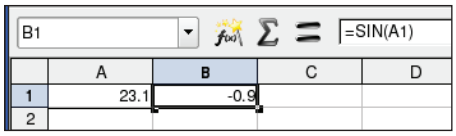
I'll leave you to work out why each of the cells behave differently (a clue: look at the different cell types).

And finally, if you are concerned with minimizing the number of lines of code, then you could re-write it as:

```
oSheet.getCellByPosition (0,1).Value = 20
oSheet.getCellByPosition (0,2).Value = 30
oSheet.getCellByPosition (1,1).String = "=A2+A3"
oSheet.getCellByPosition (2,1).Formula = "=A2+A3"
oSheet.getCellByPosition (3,1).Value = "=A2+A3"
```

# Using OOo's Built-in Functions

I'm sure that you've used OOo's built-in mathematical functions often enough in Calc:




Well, you'll be pleased (but probably not very surprised) to learn that you can use these in your macros as well. For example, you can try:

```
oCell1 = oSheet.getCellRangeByName ("A1")
oCell2 = oSheet.getCellRangeByName ("B1")
oCell1.value = 23.1
oCell2.value = SIN(oCell1.value)
```

You can also try:

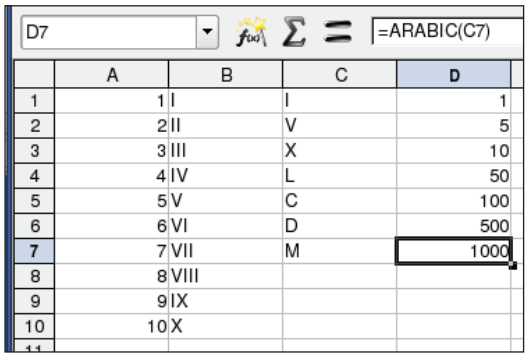
```
oCell2.formula = "=SIN(A1)"
```

What is the difference? The first version places the static value `-0.9` into cell B1. The second version places a formula into B1 and so changing the contents of A1 will cause the contents of B1 to change as well.

However, be warned. This only works for a subset of the functions. For example, the following will work:

```
For r = 0 to 9
  i = r + 1
  oCell1 = oSheet.getCellByPosition (0,r)
  oCell1.Value = i
  oCell2 = oSheet.getCellByPosition (1,r)
  oCell2.Formula = "=ROMAN(A" + i + ")"
Next r

oLetters = Array("I","V","X","L","C","D","M")
For r = 0 to ubound(oLetters)
  i = r + 1
  oCell1 = oSheet.getCellByPosition (2,r)
  oCell1.String = oLetters(r)
  oCell2 = oSheet.getCellByPosition (3,r)
  oCell2.Formula = "=ARABIC(C" + i + ")"
Next r
```

This will give you an interesting output (well, I think that it's interesting anyway):




| | A | B | C | D |
|---|---|---|---|---|
| 1 | 1 | I | I | 1 |
| 2 | 2 | II | V | 5 |
| 3 | 3 | III | X | 10 |
| 4 | 4 | IV | L | 50 |
| 5 | 5 | V | C | 100 |
| 6 | 6 | VI | D | 500 |
| 7 | 7 | VII | M | 1000 |
| 8 | 8 | VIII | | |
| 9 | 9 | IX | | |
| 10 | 10 | X | | |

But if you try the following:

```
oCell2.value =ARABIC(oCell1.String)
```

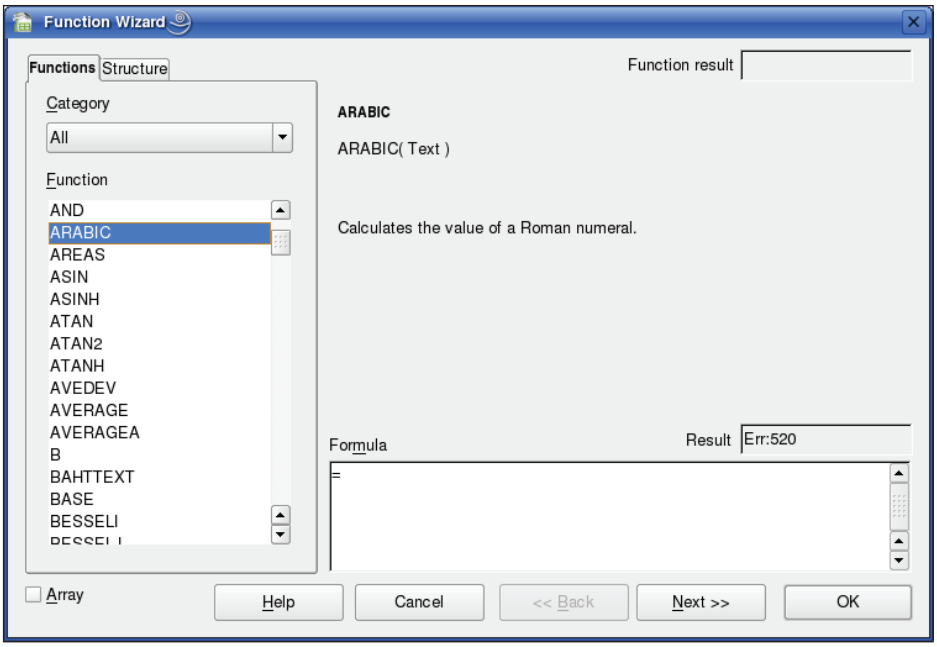
Then you'll get an error:


Don't worry. This doesn't mean that you've now got to write loads of functions because you can't access those built into OOo Calc. It just means that you have to tell OOo that you want access to all of the functions:

```
oFunction = createUnoService("com.sun.star.sheet.FunctionAccess")
oCell4.value = oFunction.callFunction("ARABIC", _
                                      Array(oCell3.String))
```

Notice that the function `callFunction` expects parameters to be passed in as an array, even if there is only a single variable.

If you want to see the full list of functions available to you, then you can, of course, see them by going to the Calc menu and clicking on **Insert | Function...**




# Named Worksheets and Cells

We now know the following about worksheets and cells:

- The worksheets can be accessed as if they are in an array called `sheets`
- The cells are accessed using the `getCellByPosition` function

It may have occurred to you that if you're accessing an existing spreadsheet, then there are some possible issues that may occur:

- Sheets may have been added or removed
- The order of the sheets may have changed

In this case, it may be better to access the sheets by using their names instead of their indexes.

# Accessing Existing Named Worksheets and Cells

You'll find it very straightforward to access sheets by name; rather than typing:

```
oSheet = oDoc.sheets (0)
```

you can type:

```
oSheet = oDoc.Sheets.getByName ("PPI Accounts")
```

Having accessed the correct sheet, you can also access a cell by its name rather than its position in the grid. So, instead of:

```
oCell = oSheet.getCellByPosition (0,1)
```

use:

```
oCell = oSheet.getCellRangeByName ("Daily Total")
```

# Creating New Named Worksheets and Cells

In our quest for automation, we've already seen that we can create any required spreadsheets as necessary. It's also a good idea to consider using a macro to name the sheets, add any additional ones, and set any named cells that we need.

Naming an existing sheet is simple. After having selected the sheet it's just a matter of adding the code:

```
oSheet.name = "PPI Client Details"
```

Adding a new sheet is nearly as easy:

```
oSheet = oDoc.createInstance ( "com.sun.star.sheet.Spreadsheet" )
oDoc.Sheets.insertByName ( "PPI Daily Tasks", oSheet )
```

What about applying a name to a cell? Well, a little more involved, but by no means complicated:

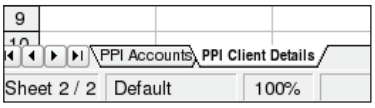
```
Dim oCellAddress As new com.sun.star.table.CellAddress
Dim oNamedRanges
oNamedRanges = oDoc.NamedRanges
oNamedRanges.addNewByName("Total", "$Sheet1.$A$8", oCellAddress, 0)
```

# Deleting Worksheets

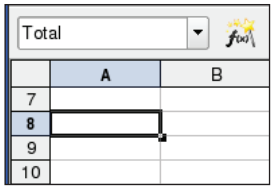
So, you've added all of the worksheets that you want and have renamed any that you need. However, if there is that extra one left over, then you may want to delete it. Easily done with:

```
oDoc.Sheets.removeByName("Sheet3")
```

With that you're now able to get your macro to load the spreadsheet that you want, or to modify one so that it contains the worksheets that you need:




You can name cells:




And you can fully control the contents of each cell in the spreadsheet.

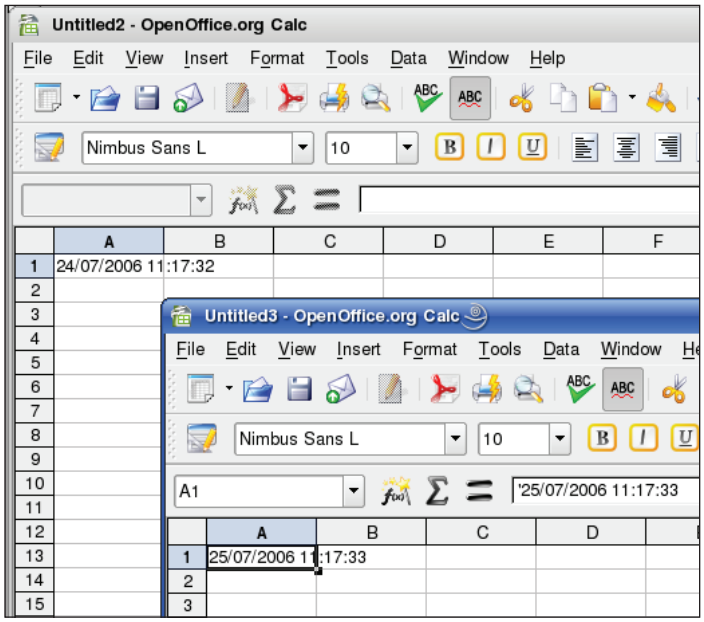
So, now that we can automate a single spreadsheet, it's time to start thinking about using macros with more that one of them.

# Working with Multiple Spreadsheets

You won't find working with multiple spreadsheets any more difficult than working with just one. It's just a matter of keeping the order right. For instance, if you're just wanting to change the details of a number of spreadsheets one at a time, then you can reuse objects:

```
Sub sequencialFiles
  Dim oDoc as Object
  Dim oDesk as Object
  Dim oSheet as Object
  Dim oCell as Object

  oDoc = starDeskTop.loadComponentFromUrl _
    ("private:factory/scalc", "_blank",0,Array())
  oSheet = thisComponent.sheets (0)
  oCell = oSheet.getCellByPosition (0,0)
  oCell.String = now

  oDoc = starDeskTop.loadComponentFromUrl _
    ("private:factory/scalc", "_blank",0,Array())
  oSheet = thisComponent.sheets (0)
  oCell = oSheet.getCellByPosition (0,0)
  oCell.String = now + 1
End Sub
```

If you run this, then you'll first see one spreadsheet opening, its contents changing, and then the second one opening and its contents changing:



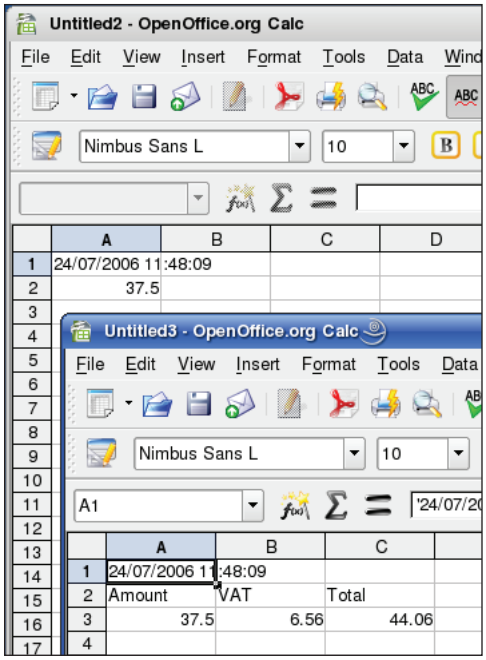

However, if you do reuse the objects, then you must be careful; you must make sure that you have completely finished with one spreadsheet before you move on to the next, and this must include saving and closing each file.

You will probably find it much more effective to use separate objects for each spreadsheet involved, giving you full control of everything:

```
Sub multiSheets
  Dim oURL1 as String
  Dim oURL2 as String
  Dim oDoc1 as Object
  Dim oDoc2 as Object
  Dim oCell1 as Object
  Dim OCell2 as Object

  oURL1 = "private:factory/scalc"
  oURL2 = "private:factory/scalc"

  oDoc1 = starDeskTop.loadComponentFromURL (oURL1, "_blank", 0, _
                                                    Array() )
  oDoc2 = starDeskTop.loadComponentFromURL (oURL2, "_blank", 0, _
                                                    Array() )
```

```
    oCell1 = oDoc1.Sheets (0).getCellByPosition (0,0)
    oCell1.String = now
    oCell1 = oDoc1.Sheets (0).getCellByPosition (0,1)
    oCell1.Value = 37.5

    oCell2 = oDoc2.Sheets (0).getCellByPosition (0,0)
    oCell2.String = now
    oCell2 = oDoc2.Sheets (0).getCellByPosition (0,1)
    oCell2.String = "Amount"
    oCell2 = oDoc2.Sheets (0).getCellByPosition (1,1)
    oCell2.String = "VAT"
    oCell2 = oDoc2.Sheets (0).getCellByPosition (2,1)
    oCell2.String = "Total"
    oCell2 = oDoc2.Sheets (0).getCellByPosition (0,2)
    oCell2.Value = oCell1.Value
    oCell2 = oDoc2.Sheets (0).getCellByPosition (1,2)
    oCell2.Value = oCell1.Value * 0.175
    oCell2 = oDoc2.Sheets (0).getCellByPosition (2,2)
    oCell2.Value = oCell1.Value * 1.175
End Sub
```

This time you can read from and write to each spreadsheet completely independently.



# Using Ranges of Cells

So far we've only worked with individual cells. However, sometimes it is useful to work with ranges of cells. For instance, let's just copy a range of cells from one spreadsheet to another:

```
oRange1 = oDoc1.Sheets (1).getCellRangeByName ("A1:A100")
oRange2 = oDoc2.Sheets (0).getCellRangeByName ("A1:A100")
oRange2.setDataArray (oRange1.getDataArray ())
```

However, if you prefer, you can use the cells' locations to obtain the range rather than the names:

```
oRange1 = oDoc1.Sheets (0).getCellRangeByPosition (0,0,0,100)
oRange2 = oDoc2.Sheets (0).getCellRangeByPosition (0,0,0,100)
```

You'll find ranges particularly useful when using some of the functions that we now have access to:

```
oCell1 = oSheet.getCellRangeByName ("A1")
oCell2 = oSheet.getCellRangeByName ("A2")
oCell3 = oSheet.getCellRangeByName ("A3")
oCell4 = oSheet.getCellRangeByName ("B3")
oCell1.Value = 36
oCell2.Value = 57
oCell3.Value = 42
oRange1 = oSheet.getCellRangeByName ("A1:A3")
'Remember to use callFunction
oCell4.Value = _
  oFunction.callFunction("STDEV", Array(oRange1.getDataArray ()))
```

The end result will be:

| | A | B |
|---|---|---|
| 1 | 36 | |
| 2 | 57 | |
| 3 | 42 | 10.82 |

# Summary

In Chapter 4 we've looked at how we can use macros to work with spreadsheets. You should now be confident enough to open any Calc document by making use of the starDeskTop UNO service, to create a new spreadsheet, and to work with each individual spreadsheet.

You have learned to add new worksheets, change a worksheet's name, and delete unwanted worksheets. You now know how to access the cells in worksheets, set your own cell names, and access the cells grouped into ranges by name or location.

So, back in the Penguin PI safe house:

A pale, wintry dawn threw its washed-out light onto the computer screen. Pygoscelis stretched and looked across the desk at Korora.

"I don't know what it is," he said, "but this just doesn't look right."

And so that's what we'll be looking at in Chapter 5—formatting spreadsheets automatically.

# 5
# Formatting your Spreadsheets

Korora stopped staring at her screen, and instead, stared at Pygoscelis.

"What do you mean that it doesn't look right?" she demanded, "The data is correct. You know it is!"

"Yes, I know that the data is correct," he said, "but it's the formatting. It doesn't matter how good the background information is, it is not going to be accepted unless it looks right, and people can read it."

"But we haven't got the time." said Korora, "They may already be on their way here."

"Don't worry," he answered, "we'll get a macro to do all of the work for us."

And you know, of course, that Pygoscelis has made a couple of important points here:

- The format of a printed report can be as important as the actual contents; what you can't read you can't understand.
- If there is a lot of formatting to do, then it is much faster to let a macro format the spreadsheet for us rather than doing it all by hand.

So, by the end of this chapter you should be able to use macros to:

- Change the look and feel of a worksheet
- Change the look and feel of cells and ranges of cells
- Automatically update the document information
- Prepare the document so that it's ready to be printed

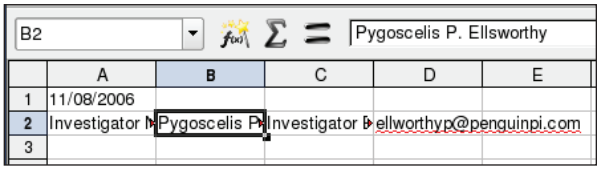
We'll get the macro to automatically do all of the things that you'd normally have to do to get a spreadsheet ready to print, and we'll set off simply by making cells fit the data that they hold.

# The Most Basic Formatting—Column and Row Dimensions

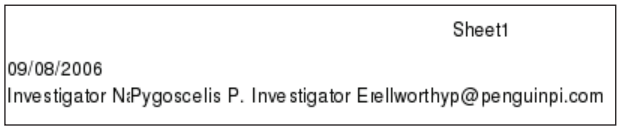
I'm sure that you've often put information into a cell and found that the column isn't wide enough:

```
oCell = oSheet.getCellByPosition (0,0)
oCell.String = date
oCell = oSheet.getCellByPosition (0,1)
oCell.String = "Investigator Name"
oCell = oSheet.getCellByPosition (1,1)
oCell.String = "Pygoscelis P.Ellsworthy"
oCell = oSheet.getCellByPosition (2,1)
oCell.String = "Client Email Address"
oCell = oSheet.getCellByPosition (3,1)
oCell.String = "ellworthyp@penguinpi.com"
```

You will get a similar screenshot; of course, you can click on each cell so that you can see what's actually displayed there




However, when you come to print the spreadsheet then you end up with garbled nonsense:




Obviously we need to get the macro to change the width of the cells to match the contents of the cell. Sounds complicated? No, it couldn't be simpler.

# Optimizing Column Widths

Optimizing the width of an individual column couldn't be easier:

```
oSheet.Columns(0).OptimalWidth = True
oSheet.Columns(1).OptimalWidth = True
oSheet.Columns(2).OptimalWidth = True
oSheet.Columns(3).OptimalWidth = True
oSheet.Columns(4).OptimalWidth = True
```

Now (once you've run the macro) everything fits nicely:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | 09/08/2006 | | | |
| 2 | Investigator Name | Pygoscelis P. Ellsworthy | Investigator Email Address | ellworthyp@penguinpi.com |

And the printed document also becomes intelligible:

Sheet1

11/08/2006
Investigator Name Pygoscelis P. Ellsworthy Investigator Email Address ellworthyp@penguinpi.com

At this point you're probably thinking that this is useful, but it's a little long winded. You don't want to have to add a line of code for every column that you're going to put data into. And, of course, you don't have to do that—a simple loop will do the job for you:

```
For c = 0 To 3
  oSheet.Columns(c).OptimalWidth = True
Next c
```

However, what if you don't know how many columns you're going to be using or the number of columns is going to be variable? In this case, you'll need a way of optimizing the width of every column in the worksheet.

# Optimizing Column Widths across a Whole Worksheet

We've just seen how to optimize the width of individual columns, but you may find it more practical to optimize all of the columns at the same time:

```
oSheet.getColumns.OptimalWidth = True
```

This time you'll find that all columns containing data will get to their optimal widths.

And as you've probably guessed, what you can do for columns you can also do for rows (except, of course, you'll use `OptimalHeight` instead of `OptimalWidth`).

## Setting Fixed Widths and Heights

Finally you may decide that you want to use a set width for each column, rather than just using the width of the text itself. Again, easily done:

```
oSheet.Columns(0).width = 10000
```

Now 10000 may seem very wide for a column, but not when you consider that it's actually measured in 100th of a millimeter; so 10000 is actually 100mm or 10 cm.

You can, as you might expect, set the height of the row as well:

```
oSheet.Rows(0).height = 1
```

We've looked at the most basic (and essential) formatting that you'll ever need—getting your information to fit nicely into the cells of a worksheet.

## Hiding Columns

Quite often there are columns that you want to be hidden (for example, ones containing formulae or maybe interim steps from one column to another). Hiding a column (or row) is just a matter of unsetting the `isVisible` property:

```
oSheet.Columns(4).isVisible = False
```

Next we'll just make sure that the page looks good when you print it.

# Formatting the Printed Page

You may decide that you don't need any more than the basic formatting that we've looked at so far. If that's the case, then you'll just want to print, and so we'll look a few of the simple ways that a macro can help you

## Adding a Page Break

Before printing your document you may decide that the default page breaks are not quite in the right places. For instance, one of them may split the data that you want to be kept together. A simple line of code allows you to add a page break exactly where you need it:

```
oSheet.Rows(1).IsStartOfNewPage = True
```

You can tell where page breaks are by looking for blue lines in the worksheet:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | 09/08/2006 | | | |
| 2 | Investigator Name | Pygoscelis P. Ellsworthy | Investigator Email Address | ellworthyp@penguinpi.com |

Or, of course, you can always click on **File | Page Preview**. And don't forget—you can create new page breaks on columns as well as rows.

# Defining a Print Area

We've seen how to add a page break, and so you may be wondering how to print just a portion of a worksheet instead of the whole of it. If you were to edit the spreadsheet manually, you would create a print area; and it's no different with a macro:

```
Dim oPrintArea(0) as new com.sun.star.table.CellRangeAddress
oPrintArea(0).StartColumn = 0
oPrintArea(0).StartRow = 1
oPrintArea(0).EndColumn = 3
oPrintArea(0).EndRow = 1
oDoc.Sheets(0).setPrintAreas(oPrintArea())
```

The end result of running the code is best seen when you print the page itself (although selecting **File | Page Preview** will save you money). However, the area to be printed will be outlined in the worksheet:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | 10/08/2006 | | | | |
| 2 | Investigator Name | Pygoscelis P. Ellsworthy | Investigator Email Address | ellworthyp@penguinpi.com | |
| 3 | | | | | |

Having decided what you want to print, you may want to consider how you want to print it.

# Setting the Header and Footer

You'll already know that when you come to print a Calc document then a header and footer are automatically added for you. By default this consists of the worksheet name at the top of the page, and the page number at the bottom (normally in the format 'Page n of nn'). You can easily add your own custom headers and footers using the following code:

```
oPageStyles = oDoc.StyleFamilies.getByName("PageStyles")
oDefault = oPageStyles.getByName("Default")
```

```
oDefault.HeaderIsOn = True
oHeader = oDefault.RightPageHeaderContent
oHeader.CenterText.String = "PPI Report"
oDefault.RightPageHeaderContent = oHeader

oDefault.FooterIsOn = True
oFooter = oDefault.RightPageFooterContent
oFooter.CenterText.String = "-- CONFIDENTIAL --"
oDefault.RightPageFooterContent = oFooter
```

This time the printout will have:

- PPI Report top center of the page
- **— CONFIDENTIAL —** bottom center of the page

If you're wondering why we've just overwritten the page number (after all that is rather useful), well, the answer is easy: just so that we can see how to put it back in...

# Adding Page Numbers

The page number and the page count are text fields that can be assigned to the document. You can create the page number by using:

```
oPageNumber = _
  oDoc.createInstance( "com.sun.star.text.TextField.PageNumber" )
```

You can add the page-count text-field in the same way

```
oPageCount = _
  oDoc.createInstance( "com.sun.star.text.TextField.PageCount" )
```

However, we can't add this directly to the footer (or the header if that is what you prefer). Instead we need to create a text cursor:

```
oTextCursor = oFooter.RightText.createTextCursor
```

Now we can build up the text for the cursor, and then add it to the footer:

```
oTextCursor.gotoEnd (False)
oTextCursor.String = "Page "
oTextCursor.gotoEnd (False)
oFooter.RightText.insertTextContent (oTextCursor, _
                                     oPageNumber, True)

oTextCursor.gotoEnd (False)
oTextCursor.String = " of "
oTextCursor.gotoEnd (False)
```

```
oFooter.RightText.insertTextContent(oTextCursor, _
                                    oPageCount, True)
```

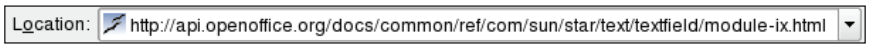
Remember, this must all go before the code:

```
oDefault.RightPageFooterContent = oFooter
```

If you add the code, re-run the macro, and have a look at the print preview you'll see:


We've only looked at two of the text fields here. After all, the page number and page count may be enough for you. However, there are quite a few others that you have access to. For example, some that you may also find useful are Author, Filename, URL, User, and Wordcount.

You can get the full list from the OpenOffice.org online documentation at:




Whether or not you decide to customize the header or footer, one thing that you will need to think about is the size of the page that you want to print.

# Setting the Page Size and Orientation

Chances are that you won't want to use the default printer settings when you come to print your spreadsheet. If your system is anything like mine, then the default is letter format with portrait orientation, useful, but I prefer A4, and sometimes I need to print a sheet in landscape mode. If, like me, you want to print a sheet in that way, then you'll need to add the following code:

```
oDefault.Width = 21000 'A4 Width in mm
oDefault.Height = 29700 'A4 Height in mm

Dim oPrintOptions(0) as new com.sun.star.beans.PropertyValue
oPrintOptions(0).Name = "PaperOrientation"
oPrintOptions(0).Value = _
                        com.sun.star.view.PaperOrientation.LANDSCAPE
odoc.Printer = oPrintOptions()
```

Do remember that we've already defined `oDefault` as:

```
oPageStyles = oDoc.StyleFamilies.getByName("PageStyles")
oDefault = oPageStyles.getByName("Default")
```

If you prefer to use differently sized pages, then you may need to consider encapsulating the code into a subroutine:

```
Sub setPaperSize (  iDOc as Object, _
                    optional iPaper as String, _
                    optional iOrient as String)
  Dim oPaperSize(5,2)
  Dim oPageStyles as Object
  Dim oDefault as Object
  Dim oPrintOptions(0) as new com.sun.star.beans.PropertyValue

  oPageStyles = iDoc.StyleFamilies.getByName("PageStyles")
  oDefault = oPageStyles.getByName("Default")

  If IsMissing (iPaper) Then
    iPaper = "A4"
  End If
  If IsMissing(iOrient) Then
    iOrient = "PORTRAIT"
  End If

  oPaperSize ("A4",0) = 21000 'Width in mm
  oPaperSize ("A4",1) = 29700 'Height in mm
  oPaperSize ("A5",0) = 14800
  oPaperSize ("A5",1) = 21000

  oDefault.Width = oPaperSize (iPaper,0)
  oDefault.Height = oPaperSize (iPaper,1)

  oPrintOptions(0).Name = "PaperOrientation"
  if iOrient = "PORTRAIT" Then
    oPrintOptions(0).Value = _
                         com.sun.star.view.PaperOrientation.PORTRAIT
  Else
    oPrintOptions(0).Value = _
                         com.sun.star.view.PaperOrientation.LANDSCAPE
  End If
  idoc.Printer = oPrintOptions()
End sub
```

However, you may decide that you just want to use all of the default settings when printing your document. This just leaves us with one problem—the sheet name.

# Customizing Worksheet Names

It's not that sheet names are actually a problem; it's just that if you use the default printing setup, then the worksheet name will be printed at the top of the sheet, and the default names are rather simplistic—Sheet1, Sheet2, and Sheet3. They don't really tell you anything about the worksheets do they? Surely descriptive names such as "Client Accounts" or "Investigator Time sheet" are much more useful—just reading the name tells you what's going on.

So, rather than changing the header and footer of your page, you may decide to customize the name of your worksheet instead. We'll want the macro to:

- Change worksheet names
- Add extra worksheets
- Remove any worksheets that we don't need

Of course, you'll remember how to do this from Chapter 4. So next we'll look at an area that is all too often overlooked—the document info.

# Updating the Document Information

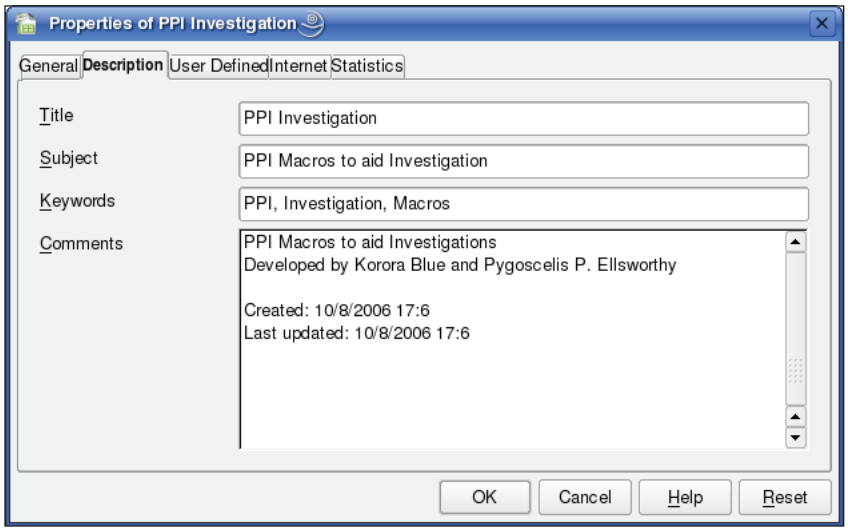
If you're anything like me (and any other programmer that I've ever met), then the last thing that you'll ever think about is the document information, and by that I mean Document author, Document title, Document subject, Document keywords.

Admit it: you just want to get on and do some programming, don't you? The obvious answer is to let the macro do all of the work for you, by making use of the `DocumentInfo` service:

```
oDoc.DocumentInfo.Author = "Pygoscelis P. Ellsworthy"
oDoc.DocumentInfo.Title = "PPI Investigation"
oDoc.DocumentInfo.Subject = "PPI Macros to aid Investigation"
oDoc.DocumentInfo.Keywords = "PPI, Investigation, Macros"

oURL = convertToUrl ("/home/bainm/bluek/ppi_investigation.ods")
oDoc.storeAsUrl (oUrl, Array())
oDoc.DocumentInfo.Description = "PPI Macros to aid Investigations" _
  & chr (10) & _
  "Developed by Korora Blue and " &   oDoc.DocumentInfo.Author _
  & chr (10) & chr (10) & "Created: " _
  & oDoc.DocumentInfo.CreationDate.Day _
  & "/" & oDoc.DocumentInfo.CreationDate.Month _
  & "/" & oDoc.DocumentInfo.CreationDate.Year _
  & " " & oDoc.DocumentInfo.CreationDate.Hours  _
```

```
    & ":" & oDoc.DocumentInfo.CreationDate.Minutes _
    & chr (10) & "Last updated: " _
    & oDoc.DocumentInfo.ModifyDate.Day _
    & "/" & oDoc.DocumentInfo.ModifyDate.Month _
    & "/" & oDoc.DocumentInfo.ModifyDate.Year _
    & " " & oDoc.DocumentInfo.ModifyDate.Hours  _
    & ":" & oDoc.DocumentInfo.ModifyDate.Minutes

  oCreated = oDoc.DocumentInfo.CreationDate
  oModified = oDoc.DocumentInfo.ModifyDate
  oDoc.DocumentInfo.Description = "PPI Macros to aid Investigations" _
    & chr (10) _
    & "Developed by Korora Blue and " & oDoc.DocumentInfo.Author _
    & chr (10) & chr (10) & "Created: " _
    & oCreated.Day & "/" & oCreated.Month & "/" & oCreated.Year _
    & " " & oCreated.Hours & ":" & oCreated.Minutes & chr (10) _
    & "Last updated: " _
    & oModified.Day & "/" & oModified.Month & "/" & oModified.Year _
    & " " & oModified.Hours & ":" & oModified.Minutes
```

To see what the code has done for you, just click on **File | Properties...**:




If you're interested in investigating the other document-information fields, then you can find the full list online at:

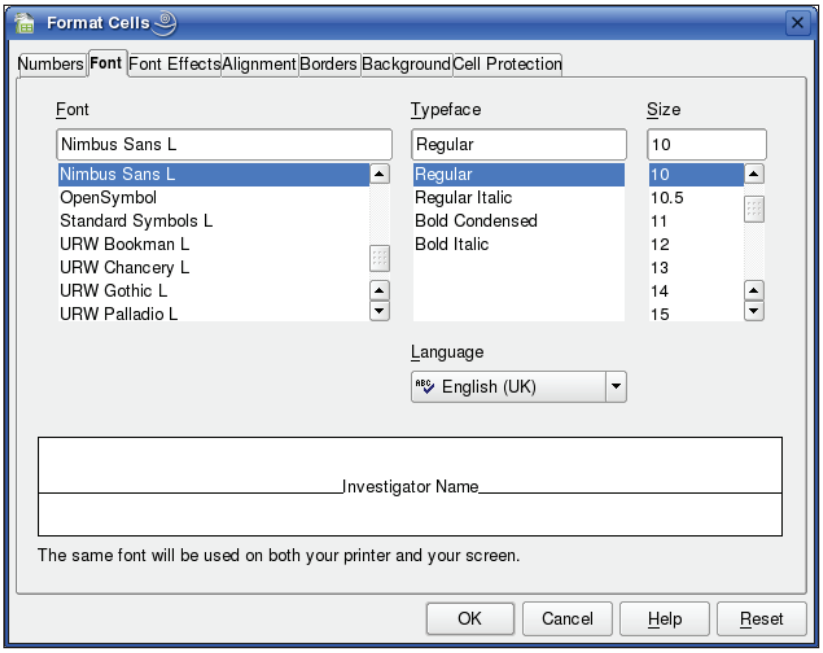
So far we've seen how to use macros to:

- Optimize row (and column) widths and heights
- Prepare the page to be printed
- Update the document information

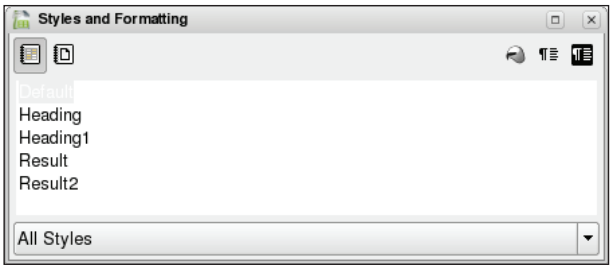
Next we'll see how to use a macro to format the cells within each worksheet.

# Formatting Cells and Ranges of Cells

If you want to format a cell (or a range of cells) manually, then you select the cells with the mouse and then click on **Format | Cell...**:




You may prefer to assign a style to the cell, this time using **Format | Styles and Formatting** (or by pressing *F11*):




We, of course, don't want to use either of those; we want the macro to do all of the work for us.

You may want to change individual settings: for example, the font face, the font size, the background, or you may want to add an underline or any other of the formatting options that you'd normally access through the **Format Cells** dialog box. On the other hand, you may find it easier just to apply a predefined style.

# Changing Cell Styles

You can choose any of the styles available to your spreadsheets, and this can be either one of the built-in styles or it can be a custom one that you've created yourself. When you've decided which style to use, the `CellStyle` method will do the work for you. For example, to apply the Heading style to a whole row of cells just use the following code:

```
ODoc.Sheets(0).Rows(0).CellStyle = "Heading"
```

This, as you can probably guess, applies the Heading style to the top row of the first worksheet. You can, if you prefer, apply a style to a single cell:

```
oDoc.Sheets.getByName _
  ("PPI Client Invoice").getCellRangeByName("A1"). _
                                              CellStyle = "Heading"
```

If that line seems too much of a mouthful, then you can break it down into more manageable chunks:

```
oSheet = oDoc.Sheets.getByName("PPI Client Invoice")
oCell = oSheet.getCellRangeByName("A1")
oCell.CellStyle = "Heading"
```

Once you've applied the style that you want, you can then think about any additional formats. For example, having set the style to Header, you may want to add an underline. On the other hand, you may just want to use the cell formats to create your own, custom styles.

# Changing Cell Formats

If you've looked at the **Format Cells** dialog box, then you'll already know that there are quite a number of things you can change. For example:

- Font
- Typeface
- Color
- Relief
- Alignment
- Background

We'll just look at a few useful ones just to get the general idea.

## Cell Background Colors

One simple way to highlight a cell is to change the background color. All you have to do is to use the `CellBackColor` property, and supply it with a long number representing the red, green, and blue components of the color. And how you get the identifying number? You can use OOo's `RGB` function to make life a little bit easier:

- `RGB (255, 0, 0)` returns the long number representing red
- `RGB (255, 255, 0)` returns the long number representing yellow
- `RGB (0, 255, 0)` returns the long number representing green
- `RGB (0, 255, 255)` returns the long number representing cyan
- `RGB (0, 0, 255)` returns the long number representing blue
- `RGB (255, 0, 255)` returns the long number representing magenta

So, to give a cell a green background you would use:

```
oCell.CellBackColor = RGB(0, 255, 0)
```

I'll leave you to experiment with all of the 16,777,216 possible colors.

## Text Colors

Along with setting the background color, you may also set the text (or character) color by using the `CharColor` property:

```
ocell.CharColor = RGB (255,0,0)
```

You'll now have a cell with nice red letters (or numbers).

## Cell Fonts

As you'd expect, the font is very easy to change. Just use the `CharFontName` property. If, for instance, you want to change the font of a whole worksheet, then you can use the code:

```
oSheet.CharFontName = "Courier"
```

One thing to remember, however, is that the choice of font will change the width of the cell that contains the text. So, don't forget to set the `OptimalWidth` and `OptimalHeight` properties after you've set the font name.

## Character Heights

Another way of highlighting cells is to change the character height. As with all of the other formatting, this can be done on the sheet, row, or column level. In this next example we change the character height for a whole row:

```
oDoc.Sheets(0).Rows(0).CharHeight = 10
```

As with the cell font the character height will affect the cell width and height, and you may need to change them to be able display the cell's contents correctly.

## The Underline

It may surprise you to learn that there are actually 18 different kinds of underline:

| ID | Name | OOo Constant |
|---|---|---|
| 0 | None | `com.sun.star.awt.FontUnderline.NONE` |
| 1 | Single | `com.sun.star.awt.FontUnderline.SINGLE` |
| 2 | Double | `com.sun.star.awt.FontUnderline.DOUBLE` |
| 3 | Dotted | `com.sun.star.awt.FontUnderline.DOTTED` |
| 4 | Don't know | `com.sun.star.awt.FontUnderline.DONTKNOW` |
| 5 | Dash | `com.sun.star.awt.FontUnderline.DASH` |
| 6 | Long dash | `com.sun.star.awt.FontUnderline.LONGDASH` |
| 7 | Dash dot | `com.sun.star.awt.FontUnderline.DASHDOT` |
| 8 | Dash dot dot | `com.sun.star.awt.FontUnderline.DASHDOTDOT` |
| 9 | Small wave | `com.sun.star.awt.FontUnderline.SMALLWAVE` |
| 10 | Wave | `com.sun.star.awt.FontUnderline.WAVE` |
| 11 | Double wave | `com.sun.star.awt.FontUnderline.DOUBLEWAVE` |
| 12 | Bold | `com.sun.star.awt.FontUnderline.BOLD` |
| 13 | Bold dotted | `com.sun.star.awt.FontUnderline.BOLDDOTTED` |

| ID | Name | OOo Constant |
|---|---|---|
| 14 | Bold dash | com.sun.star.awt.FontUnderline.BOLDDASH |
| 15 | Bold long dash | com.sun.star.awt.FontUnderline.BOLDLONGDASH |
| 16 | Bold dash dot | com.sun.star.awt.FontUnderline.BOLDDASHDOT |
| 17 | Bold dash dot dot | com.sun.star.awt.FontUnderline.BOLDDASHDOTDOT |
| 18 | Bold wave | com.sun.star.awt.FontUnderline.BOLDWAVE |

All you have to do is to set the property `CharUnderline` with the number for the underline that you want to use:

```
oCell.CharUnderline = 18
```

You can also use the OOo constant for underline:

```
oCell.CharUnderline = com.sun.star.awt.FontUnderline.BOLDWAVE
```

However, of the 18 underline types only 16 actually do anything:

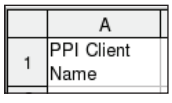
- Type 0 is "None" i.e. no underline.
- Type 4 is defined as "Don't Know" and this turns out to be the same as "None".

# Word Wrapping

We've discussed the fact that we can change the cell width to fit the contents or to have a fixed value, depending on the look and feel that we want to achieve. If you decide to use a fixed width, then it is worth considering setting the `IsTextWrapped` property to `True`. This will ensure that the information loaded into the cell will still be displayed correctly, even if the entered data is wider than the displayable width of the cell. So, for example, you could try:

```
oSheet = oDoc.Sheets.getByName ("PPI Client Invoice")
oCell = oSheet.getCellRangeByName("A1")
oCell.String = "PPI Client Name"
ocell.IsTextWrapped = True
```

The result is exactly as you would expect:




# Number Formats

Having looked at formatting the characters in a cell, you may now be wondering about formatting numbers.

As you've probably already worked out, you only have to set a property (`NumberFormat`) with a number representing the number format. However, that's where there can be problems:

- The ID for each number format may vary according to your country and language.
- Since an ID is required for each number format, how can you apply a custom number format to a cell?

Don't worry, the tools that you need come as components of the document:

1. Find a number format ID by using the `NumberFormats.queryKey` method.
2. If the required number format does not exist, then you can create a new one using the `NumberFormats.queryKey` method.

You'll probably want to set number formats quite regularly, so the best thing to do at this point is to write a function to keep things simple:

```
Function getNumberFormat (iDoc as Object, _
    iFormat as String, _
    optional iLang as String, _
    optional iCountry as String)
  Dim oFormatId As Long
  Dim oLocale As New com.sun.star.lang.Locale

  If IsMissing (iLang) Then
    iLang = "en"
  End If
  If IsMissing (iCountry) Then
    iCountry = "gb"
  End If
  oLocale.Language = iLang
  oLocale.Country = iCountry

  oFormatId = iDoc.NumberFormats.queryKey(iFormat, oLocale, True)
  If oFormatId = -1 Then
     oFormatId = iDoc.NumberFormats.addNew(iFormat, oLocale)
  End If
  getNumberFormat = oFormatId
End Function
```

Now the setting of cell formats becomes very easy:

```
  oCell = oSheet.getCellRangeByName("A1")
  oCell.Value = 23400.3523565

  oCell = oSheet.getCellRangeByName("A2")
```

```
oCell.Value = 23400.3523565
oCell.NumberFormat = getNumberFormat (oDoc, "£#,##0.00")

oCell = oSheet.getCellRangeByName("B2")
oCell.Value = -23400.3523565
oCell.NumberFormat = getNumberFormat (oDoc, "£#,##0.00")

oCell = oSheet.getCellRangeByName("A3")
oCell.Value = 23400.3523565
oCell.NumberFormat = _
  getNumberFormat (oDoc, "£#,##0.00;[RED]-£#,##0.00")

oCell = oSheet.getCellRangeByName("B3")
oCell.Value = -23400.3523565
oCell.NumberFormat = _
  getNumberFormat (oDoc, "£#,##0.00;[RED]-£#,##0.00")
```

When you run the code, you can see the effects of assigning different number formats to the cells:

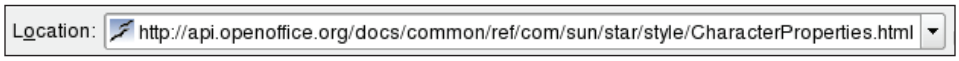
| | A | B |
|---|---|---|
| 1 | 23400.35 | |
| 2 | £23,400.35 | -£23,400.35 |
| 3 | £23,400.35 | -£23,400.35 |

You'll notice that the formatting for the last two cells (A3 and B3) is particularly useful because it introduces color coding that depends on the value of the cell—negative values are displayed in red.

# Online Reference Material

In this chapter we've only covered a small amount of the formatting that is available. If you're interested in all of the possibilities, then you can find more about cell properties at:


You'll find more about character properties at:



# Summary

In this chapter we've learned about formatting spreadsheets and how to customize the look and feel of: worksheets, pages prior to printing, Document Information, and Cells.

We have learned how to optimize column width, apply fixed widths to it, hide columns and rows for a worksheet. We also dealt with adding page breaks, creating print areas, custom headers and footers, changing page type and size, and so on for pages to be printed. We now know how to format cells with respect to style, background color, text color, font size, underlines, and many other formats.

And so back to the story...

Pygoscelis watched Korora as she read the newly printed sheet of paper.

"But this can't be true" she said, "I just can't believe that it's him"

"I'm sorry" said Pygoscelis, "but you've seen the data—and I don't like it any better than you."

He looked at her staring back at him, a horrified expression on her face. All too late he realized that she wasn't looking at him, she was looking over his shoulder at something else. As he turned, the butt of a pistol hit him on the temple. Immediately a sickening black pit opened up in front of him.

When he slowly came to, he became aware of two things. Something warm was oozing down the side of his face, and that he couldn't move. His eyes were blurred, but he tried to look around. "Korora?"

"It's OK Py. I'm here" said her voice from behind him "I'm tied to you"

As his vision cleared he saw the devastation around him. The PCs were smashed beyond any hope of repair. And then he became aware of a face leering at him.

"Surprised, Boss?" Sphen was sitting on one of the desks idly toying with the pistol in his hand. He grinned.

"Bit of a mess, isn't it? And all your hard work gone up in smoke. The disk and your printouts are in there". He pointed to a smoking waste paper bin. "All I was waiting for was for you to wake up. I want you to see the bullet coming."

Pygoscelis laughed.

"What's so funny"

"It doesn't matter what you do now. It's too late. All of the information has been uploaded into a database. Kill us and you've lost everything."

If you're interested in learning how Pygoscelis achieved this, then carry on to Chapter 6, *Working with Databases*.

# 6
# Working with Databases

Sphen's grin rapidly evaporated.

"What do you mean? What database?"

"You don't think that we'd be stupid enough to store all of the information here, do you?"

Sphen looked Pygoscelis up and down. The grin returned as he pointed the pistol at Korora and he pulled back the hammer.

"Well, I don't need both of you to get the data back."

"Actually, you do. I only have a password for some of the data. Korora has the password for the rest. You need both of us alive."

Sphen carefully released the hammer and put the gun into his pocket.

"OK. You've won this round. Now, about these databases..."

And so, as Sphen said, about these databases...

In Chapter 6 (as you may have guessed), we're going to be looking at how to use Calc macros with information stored in a database. By the end of the chapter, you should be able to:

- Use a macro to connect to connect to a database
- Use **SQL** (**Structured Query Language**) in a macro to extract information from a database and use it in a spreadsheet
- Use macros to add data to a database
- Use macros to update data in a database

What we *won't* be looking at is:

- How to design and build a database
- How to set up database connectivity using ODBC (Open DataBase Connectivity)
- The full extent of SQL—we'll only be looking at the basics, just enough for us to read and write to a database

So let's get started by accessing a database.

# Accessing Databases

Having decided that we want to use a database, it might be worth thinking about databases in general for a moment.

If you google "What is a database?", then you'll find a myriad of definitions, but they all boil down to the same thing—A database is a structured set of data. So, using this definition, a database could actually be a website or text files in a directory. However, there are limitations to this type of database:

- Searching can be difficult and often slow.
- It's easy for a number of people to be reading the data at the same time, but writing to the data at the same time can cause problems.
- The ways in which data can be extracted and analyzed are usually very limited.

And that, of course, is why most people use one of the readily available commercial databases or preferably one of the many free and open-source ones.

# Which Databases can We Use?

I can't advise you on which database to choose. You may be in an organization that limits you to the database that the company has already bought, for example Oracle, MS Access, and so on.

If you have a completely open choice, then you might choose from Kexi, MySQL, and PostgreSQL.

If you do use any of these databases, then you'll have to set up the database connectivity, and this depends on your system and your preferences. A common one is **Open DataBase Connectivity** (**ODBC**). You'll need to do two things:

1. Install the database-specific drivers for your system.

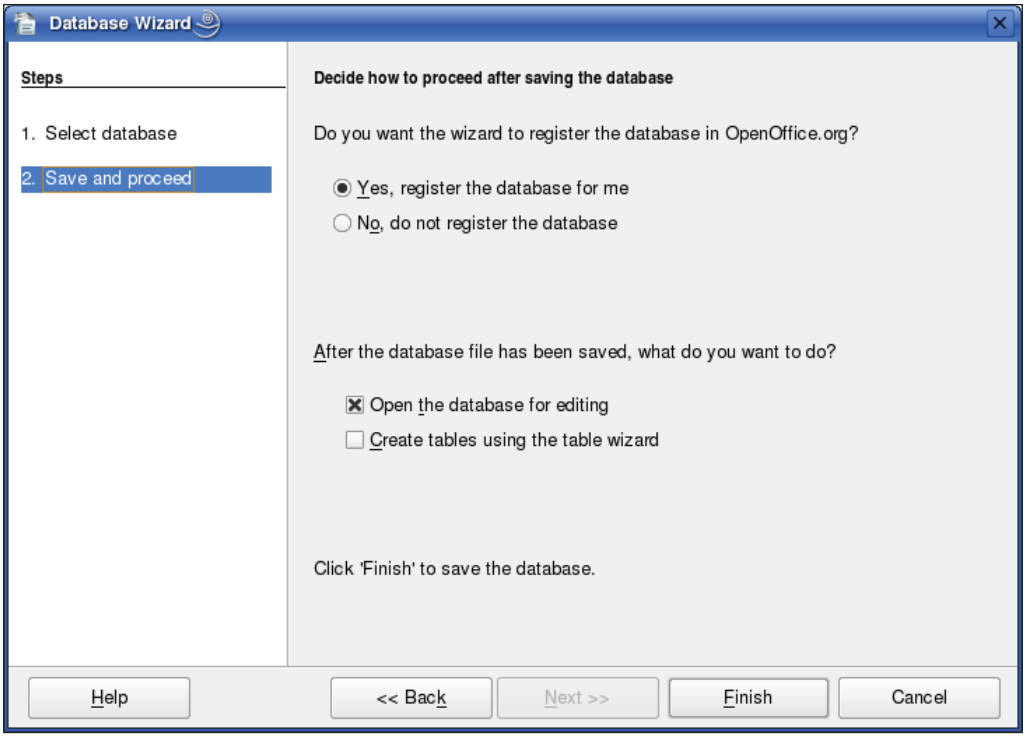
2. Use the database-connectivity software for your system to set up the connection. The software could be (for example) unixODBC for Linux or MS ODBC on Windows.

On the other hand, since you're already using OpenOffice.org, you could just stick with OpenOffice.org's Base database.

# Registering the Database as an OOo Data Source

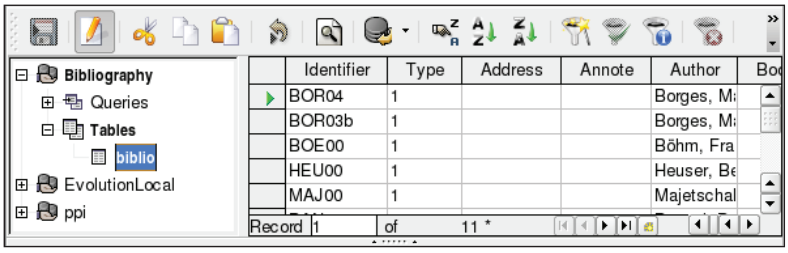
Once you've set up the database connectivity, you can register it as a data source with OpenOffice.org—this makes accessing the data just that little bit simpler. It's easily done by going to the **OOo** menu and selecting **File | New | Database**. You'll be given the choice of selecting an existing database (either an existing `OpenOffice.org Base` file or a database connection) or you can create a completely new database.

Once you've decided what you want to do, then OOo will register the database for you:



# Viewing Registered Data Sources

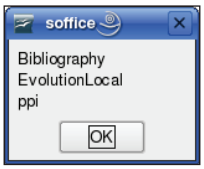
Undoubtedly, at some point you'll want to see a list of registered data sources. One way is to click on **View | Data Sources** (or press *F4*).




However, you may prefer to use a macro to get the list:

```
Sub Main
    list_available_databases
End Sub

Sub list_available_databases
    Dim dbContext as Object
    Dim dbNames
    Dim d as Integer
    Dim dbText as String

    dbContext = createUnoService("com.sun.star.sdb.DatabaseContext")
    dbNames = dbContext.getElementNames()
    For d = 0 To UBound(dbNames())
        dbText = dbText +  dbNames(d) + chr(10)
    Next d
    msgbox dbText
End Sub
```

The macro creates a `DatabaseContext` service and accesses its `getElementNames` method to create an array containing the list of data sources. The macro then loops through the array to create a string that can be displayed in a text box:




Having identified the data sources, you'll want to do something useful with them—this means connecting to the databases and then getting information from the tables contained in them.

# Connecting to a Database

We've already seen that we can use the `DatabaseContext` service to enable us to see the list of registered data sources. Next, we can use the service to connect to the database:

```
Sub Main
    Dim db as Object
    db = connect_to_database("ppi")

    'At the end of the session the connection should end
    'automatically
    'but it is best just to make sure of the job:
    disconnect_from_database (db)
End Sub

Sub disconnect_from_database (db as Object)
    db.close
    db.dispose()
End Sub

Function connect_to_database (dbName as String) as Object
    Dim dbContext As Object
    Dim oDataSource As Object

    dbContext = _
        createUnoService("com.sun.star.sdb.DatabaseContext")
    oDataSource = dbContext.getByName(dbName)
    connect_to_database = oDataSource.GetConnection("","")
End Function
```

You'll notice that the final line of the code contains two empty strings:

```
connect_to_database = oDataSource.GetConnection("","")
```

These are the spaces for a user name and a password in case you need these for connecting to the database. If your database does need a user name and a password (and obviously it's sensible to use them), then you would use something like:

```
db = oDataSource.GetConnection("bainm","nottelling")
```

So, that's easy enough. Next we can look at the actual tables in the database.

# Accessing Database Tables

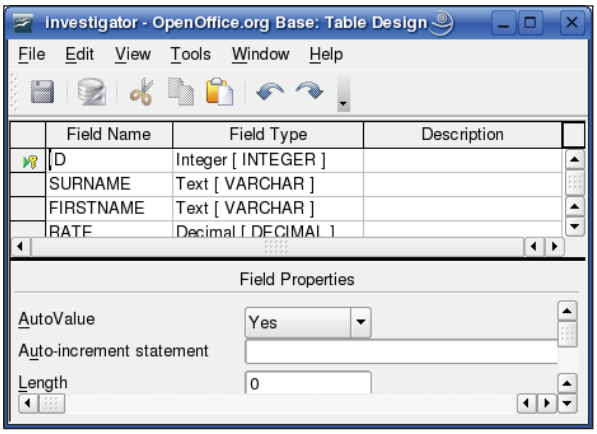
As I mentioned at the start of the chapter, we're not going to delve into the intricacies of designing and creating the database tables. That's because the techniques and tools that you'll have at your fingertips will vary according to the actual database that

you're using. For example, if you've got MySQL or PostgreSQL, then you could create the tables from the command line or use third-party tools to create the table visually.

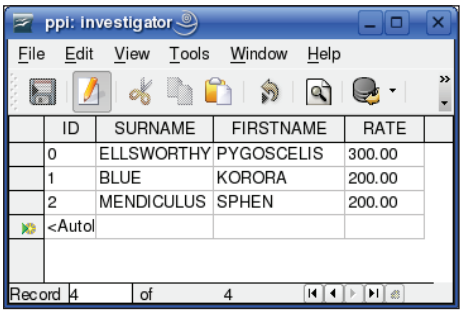
On the other hand, if you've got MS Access or you're using OpenOffice.org's Base application, then you will use their built-in table creation forms and wizards.

However, irrespective of the database you're using, it's worth spending the time constructing the data structure *before* you start creating macros. It's easier to build macros around well-structured data rather than trying to fit your data around the macros.

If you are using Base, then you can build tables easily using Base's **Table Design** dialog.




You can then use Base (with the table in **Edit** mode) to populate the tables manually (since we haven't written a macro to do it yet) as shown in the figure below:




Of course, for the time being, you may prefer to use the database that comes with Base—Biblio.

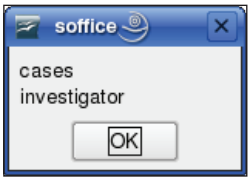
Earlier we saw that using the `DatabaseContext` service allowed us to list the available data sources and connect to any of the databases registered with OpenOffice.org.

Next we can use the service to obtain a list of the tables in the database:

```
Sub Main
    Dim db As Object
    db = connect_to_database ("ppi")
    list_tables (db)
    disconnect_from_database (db)
End Sub

Sub list_tables (db as Object)
    Dim dbTables as Object
    Dim dbTableNames
    Dim opText as String

    dbTables = db.getTables
    dbTableNames = dbTables.getElementNames
    opText = join ( dbTableNames , chr(10))
    msgbox opText
End Sub
```

You'll notice that the `list_tables` subroutine does not create the connection to the database; instead the `connect_to_database` function returns this as the variable `db`. This then represents the connection, and can be passed to any of the macros that you write.

When you run the `Main` macro, you'll see the list of tables displayed.




Of course, if you're using your own database or OOo's Biblio, then you'll get a different list—but you get the idea.

Don't forget you'll always have to run the `connect_to_database`; without this you have no connection to the database. If you do want to write any stand-alone macros that you can run independently and which use the database, then you'll have to include the statement

```
    db = connect_to_database ("ppi")
```

In our example `Main` does all of the work for us—calling the subroutine to connect to the database, and then calling a macro to list the tables. So having obtained a list of all of the tables in the database we can now start to think about extracting data from them.

## Running Queries on the Tables

We're going to use the following simple procedure:

- Connect to the database
- Send a SQL statement to the database
- Get a result from the database
- Make use of the results

Since we've already learned how to connect to the database, let's carry on and do the rest:

```
Sub Main
    Dim db As Object
    db = connect_to_database ("ppi")
    simple_query (db)
    disconnect_from_database (db)
End Sub

Function capitalize (iName as String) as String
    Dim wordStart as String
    Dim wordEnd as String

    wordStart = UCase (Mid (iName,1,1))
    wordEnd = LCase (Mid (iName,2))
    capitalize  = wordStart & wordEnd
End Function
Sub simple_query (db as Object)
    Dim oSql as String
    Dim i as Integer
    Dim oRowSet as Object
    Dim oResult as String

    oSql = _
    "SELECT SURNAME, FIRSTNAME, RATE" _
    & " FROM ""investigator"""

    oRowSet = createUnoService("com.sun.star.sdb.RowSet")

    oRowSet.activeConnection = db
    oRowSet.Command = oSql
    oRowSet.execute
```

```
        while oRowSet.Next
        oResult = oResult _
        & capitalize (oRowSet.getString(2)) & " " _
           & capitalize (oRowSet.getString(1)) & " " _
            & "£" & oRowSet.getFloat(3) _
            & " " & chr(13)
            wend
        msgbox oResult, ,"PPI Hourly Rate "
    End Sub
```

You'll see from the `simple_query` subroutine that we create a `RowSet` service, which then uses the database connection to send a SQL statement to the database and retrieve the results. Next, we just have to loop through the `RowSet` extracting information from it line by line.

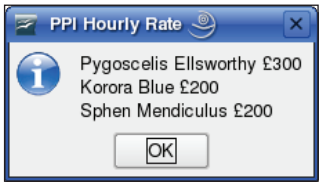
There are a couple of things to consider before we move on. The first one being the following three lines of the code:

```
        oRowSet.activeConnection = db
        oRowSet.Command = oSql
        oRowSet.execute
```

Personally, I'm perfectly happy with this, but you may prefer the `With` format. If so, then you could change the code to:

```
        With oRowSet
           .activeConnection = db
           .Command = oSql
           .execute
        End With
```

The other thing is the `capitalize` function. This is just used to format any names nicely (since they may have been saved as all lowercase, or all uppercase, or any mixture of the two). The end result is as follows:



# Putting it All into a Spreadsheet

We've seen just how easy it is to get data out of a database. We can now look at actually doing something useful with the information. The most obvious thing to do is to load the data into a spreadsheet:

```
Sub Main
    Dim db As Object
    db = connect_to_database ("ppi")
    load_investigator (db)
    disconnect_from_database (db)
End Sub

Sub load_investigator (db as Object)
    Dim oDoc as Object
    Dim oURL as String
    Dim oSheet as Object
    Dim oCell as Object
    Dim oRowSet as Object
    Dim i as Integer

    oURL = "private:factory/scalc"
        oDoc = starDeskTop.loadComponentFromURL _
                    (oURL, "_blank", 0, Array() )
    oSheet = oDoc.Sheets(0)
    oSheet.Name = "PPI Investigator"

    oRowSet = get_rowset (db, sql_select _
               ("investigator", Array("SURNAME","FIRSTNAME","RATE")))

    While oRowSet.Next
        i = i + 1
       oCell = oSheet.getCellByPosition(0,i)
       'Remember to use capitalize from page 8
       oCell.String = capitalize (oRowSet.getString(2))
        oCell = oSheet.getCellByPosition(1,i)
        oCell.String = capitalize (oRowSet.getString(1))
        oCell = oSheet.getCellByPosition(2,i)
        oCell.Value = capitalize (oRowSet.getString(3))
    Wend
End Sub
```

You'll realize, of course, that there's nothing new here. We are just using some of the techniques that we've learned in Chapter 4 (where we manipulated data in a spreadsheet), Chapter 5 (where we formatted the contents of spreadsheets), and this chapter (where we've extracted information from a database, and introduce functions such as `capitalize`).

However, there are a couple of functions that we call from `load_investigator` that you may be wondering about. The first is the function `sql_select`. If you do not prefer to create the SQL yourself, you can use this to do the job for you. All you have to do is feed it with the table name (as a string) and the list of fields (as an array):

```
Function sql_select (iTable as String, iFields())
    sql_select = _
    "SELECT " & join (iFields,",") _
    & " FROM """ & iTable + """"
End Function
```

The other function is `get_rowset`. If you don't want to remember how to create a `RowSet`, then you can use this to do the work for you. Just pass a SQL statement to it and it will return the `RowSet` as an object to you:

```
Function get_rowset (db as Object, iSql as String) as Object
    Dim oRowSet as Object

    oRowSet = createUnoService("com.sun.star.sdb.RowSet")
    oRowSet.activeConnection = db
    oRowSet.Command = iSql
    oRowSet.execute

    get_rowset = oRowSet
End Function
```

After all this, the result is a spreadsheet containing the result of the query that you've sent to the database:

| | A | B | C |
|---|---|---|---|
| 1 | | | |
| 2 | Pygoscelis | Ellsworthy | 300 |
| 3 | Korora | Blue | 200 |
| 4 | Sphen | Mendiculus | 200 |

I'm sure that you'll find these techniques very useful. However, it's rather limiting, isn't it? You'll need a completely new macro for every set of data that you want to load. Instead of that let's investigate a more generic approach.

# Loading Data into Custom Worksheets

I don't know about you, but I enjoy using code to come up with fresh solutions to problems. I don't particularly enjoy just typing the same old lines of code again, and again, and again. So let's look at changing `load_investigator` into a macro that will work for any data that we want to insert into the spreadsheet:

```
Sub Main
    Dim doc as Object
    Dim db As Object

    db = connect_to_database ("ppi")
    doc = open_spreadsheet
    ppi_sheets (db, doc)
    disconnect_from_database (db)
End Sub
```

We're still using the `connect_to_database` function (for obvious reasons); however, we've also added `open_spreadsheet` and `ppi_sheets`.

```
Function open_spreadsheet
    Dim oURL as String

    oURL = "private:factory/scalc"
    open_spreadsheet = starDeskTop.loadComponentFromURL (oURL, _
        "_blank", 0, Array() )
End Function
```

You'll realize immediately that all `open_spreadsheet` does is encapsulate code that we've already used often enough and open a blank spreadsheet. It's in `ppi_sheets` that all of the database work is done:

```
Sub ppi_sheets (db as Object, iDoc as Object)
    Dim sheetName as String
    Dim fields
    Dim oRowSet as Object
    Dim r as Integer

    sheetName = "PPI Investigator"
    oRowSet = get_rowset (db, sql_select _
        ("investigator", _
            Array("SURNAME","FIRSTNAME","RATE", "'END_OF_RECORD'")))
    r = oRowSet.RowCount + 1
    load_sheet (iDoc, sheetName, oRowSet)
    capitalize_column (iDoc, sheetName,0,r)
    capitalize_column (iDoc, sheetName,1,r)
    format_column (iDoc, sheetName,2,r,"£#,##0.00")
    deleteSheets (iDoc)
End Sub
```

OK, nothing contentious here. You can work out from the function name what's going on. However, you may be wondering about the `'END_OF_RECORD'` used in the SQL statement. It's not actually a field name; we're using it to mark the final record in each row. You can see it in use in the `load_sheet` subroutine:

```
Sub load_sheet (iDoc as Object, iName as String, iRowSet as Object)
    Dim oSheet as Object
    Dim oCell as Object
    Dim r as Integer
    Dim c as Integer
    Dim endMarker as String

    oSheet = iDoc.createInstance ( "com.sun.star.sheet.Spreadsheet" )
    iDoc.Sheets.insertByName ( iName, oSheet )

    If Not isNull (iRowSet) Then
        While iRowSet.Next
            r = r + 1
            c = 1
            endMarker = ""
            While endMarker <> "END_OF_RECORD"
               oCell = oSheet.getCellByPosition(c - 1,r)
             if isNumeric (iRowSet.getString(c)) Then
               oCell.Value = iRowSet.getString(c)
             Else
                 oCell.String = iRowSet.getString(c)
             End If
             c = c + 1
               endMarker = iRowSet.getString(c)
           Wend
       Wend
    End If
End Sub
```

The `load_sheet` subroutine is useful because you can use it to add a completely new worksheet, give it an appropriate name, and load it with the contents of a `RowSet`.

Having loaded the data that we require, the `ppi_sheets` subroutine then calls a few custom-build subroutines to carry out some formatting. The macro `capitalize_column` takes the `capitalize` function and applies it to a portion of a column:

```
Sub capitalize_column (iDoc as Object, _
                       iSheetName as String, _
                       iColumn as Integer, _
                       iRow as Integer)
    Dim oCell as Object
    Dim oSheet as Object
    Dim r as Integer
    oSheet = iDoc.Sheets.getByName (iSheetName)
    For r = 0 to iRow
       oCell = oSheet.getCellByPosition(iColumn,r)
```

```
            oCell.String = capitalize(oCell.String)
        Next r
    End Sub
```

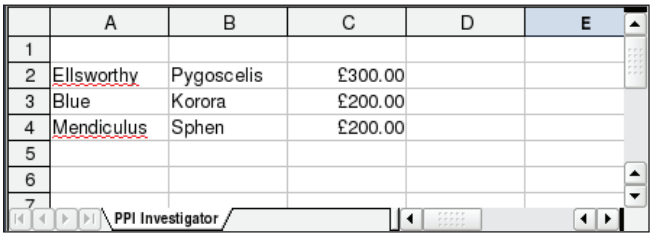
We can also apply number formats to any column that we want:

```
Sub format_column ( iDoc as Object, _
                    iSheetName as String, _
                    iColumn as Integer, _
                    iRow as Integer, _
                    iFormat as String)
    Dim oCell as Object
    Dim oSheet as Object
    Dim r as Integer

    oSheet = iDoc.Sheets.getByName (iSheetName)

    For r = 0 to iRow
        oCell = oSheet.getCellByPosition(iColumn,r)
        'The getNumberFormat function was created in chapter 5
        oCell.NumberFormat = getNumberFormat (iDoc, iFormat)
    Next r
End sub
```

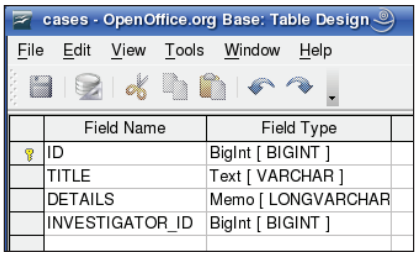
Finally, we remove any unused worksheets:

```
Sub deleteSheets (iDoc as Object)
    iDoc.Sheets.removeByName("Sheet1")
    iDoc.Sheets.removeByName("Sheet2")
    iDoc.Sheets.removeByName("Sheet3")
End Sub
```

At the end of the process we've got a formatted spreadsheet with only the necessary worksheets:



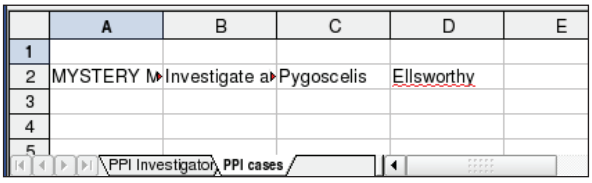

Of course, we may have other tables in the database, for example our friend Pygoscelis would have a table containing all the PPI cases:




And to use another table we only have to add a small amount of extra code:

```
sheetName = "PPI cases"
oSql =_
      "select TITLE,DETAILS,FIRSTNAME,SURNAME,'END_OF_RECORD' " _
      & " from ""investigator"" i, ""cases"" c "_
      & " where i.ID = c.INVESTIGATOR_ID" _
      & " order by c.ID "
oRowSet = get_rowset (db, oSql)
load_sheet (iDoc, sheetName, oRowSet)
r = oRowSet.RowCount + 1
capitalize_column (iDoc, sheetName,2,r)
capitalize_column (iDoc, sheetName,3,r)
```

The SQL statement is a little more complicated this time—rather than using the contents of a single table we're combining the contents of two tables (often referred to as a `Join` query). The end result is a worksheet containing details from both tables:




You'll notice that the column widths are not optimized, but I'm not going to do everything for you. All you have to do is create a new subroutine similar to `capitalize_column` and `format_column`, but this time using the column's `OptimalWidth` property (if you remember, we used that in Chapter 5).

# Adding New Records to the Database

So far we've extracted static data from the database, but we're not limited to that. It's easy to collect information from the spreadsheet or from manual input, and then use it with an `insert` query in order to create a new set of records in a table. In this example, we first need to select a row in the PPI Investigator worksheet.

| | A | B | C |
|---|---|---|---|
| | | | |
| | Ellsworthy | Pygoscelis | £300.00 |
| 3 | Blue | Korora | £200.00 |
| 4 | Mendiculus | Sphen | £200.00 |

We then call the `add_case` macro (by clicking on **Tools | Macros | Run Macro**...). The macro will obtain the investigator name from the selection and then ask the user (e.g. Pygoscelis) for the new case details.

```
Sub add_case
    Dim oRange as Object
    Dim oSheet as Object
    Dim oCell as Object
    Dim surname as string
    Dim firstname as string
    Dim title as string
    Dim details as string
    Dim r as Integer

    oRange =  thisComponent.getCurrentSelection.getRangeAddress
    r = oRange.startRow
    oSheet = thisComponent.CurrentSelection.getSpreadsheet
    surname = ucase(oSheet.getCellByPosition(0,r).String)
    firstname = ucase(oSheet.getCellByPosition(1,r).String)
    If surname = "" Or firstname = "" Then
        msgbox "Please select a row containing a name "
        stop
    End If
    While title = ""
         title = inputbox ("Enter case title (enter 0 to stop)")
    Wend
    If title = 0 Then
        stop
    End If
    While details = ""
        details = inputbox ("Enter case details (enter 0 to stop)")
    Wend
    If details = 0 Then
         stop
    End If
    create_case (surname, firstname, title, details)
End Sub
```

You'll notice that the subroutine collects some of the information from the spreadsheet itself.

```
surname = ucase(oSheet.getCellByPosition(0,r).String)
firstname = ucase(oSheet.getCellByPosition(1,r).String)
```

While the rest of the data is obtained from manual input as:

```
title = inputbox ("Enter case title (enter 0 to stop)")
details = inputbox ("Enter case details (enter 0 to stop)")
```

The collected information is then passed on to the `create_case` subroutine:

```
Sub create_case (iSurname as String, _
                 iFirstName as String, _
                 iTitle as String, _
                 iDetails as String)
    Dim id as Integer
    Dim oSql as String
    Dim oResult as Object
    Dim oStatement as Object
    Dim db as object

    db = connect_to_database ("ppi")

    id = investigator_id (db, iSurname, iFirstName)

    oSql = "insert into ""cases"" " _
    + "(""TITLE"",""DETAILS"",""INVESTIGATOR_ID"") values " _
    + "('" + iTitle + "','" + iDetails + "'," + id + ")"

    oStatement = db.createStatement
    oResult = oStatement.executeQuery (oSql)
    disconnect_from_database (db)
End Sub
```

You may wonder why the subroutine contains the line:

```
db - connect_to_database ("ppi")
```

If you remember, previously the connection to the database was created when you first ran `Main`. It was, of course, released when `Main` finished. However, this time `Main` is not being called, and so in order for `create_case` to run correctly it needs its own connection (and, of course, its own disconnection).

The subroutine finally sends an `insert` SQL statement to the database:

```
oSql = "insert into ""cases"" " _
+ "(""TITLE"",""DETAILS"",""INVESTIGATOR_ID"") values " _
+ "('" + iTitle + "','" + iDetails + "'," + id + ")"
```

However, before it carries out the `insert`, `create_case` obtains the field `investigator_id`—a number stored in the `investigator` table, and we use the `investigator_id` function:

```
Function investigator_id _
   (db as Object, iSurname as String, iFirstName as String)
    Dim oRowSet as Object
    Dim oSql
    oSql = "select ID from investigator " _
    + "where SURNAME ='" + iSurname + "'" _
    + " and FIRSTNAME = '" + iFirstName + "'"
    oRowSet = get_rowset (db, oSql)
    oRowSet.Next
    investigator_id = oRowSet.getInt(1)
End Function
```

You'll also notice that the function gets the information from a single table, but it is filtered to obtain a single ID number (by adding a *Where* clause to the SQL).

The final point to be aware of is that we don't use the `RowSet` when inserting into the database; we just create a statement and then execute the SQL:

```
    oStatement = db.createStatement
    oResult = oStatement.executeQuery (oSql)
```

# Updating the Database

So far we've learned how to:

- Use SQL to query the database and use the information obtained to fill a worksheet
- Use an `insert` SQL statement in a macro to create new records in the database

The next thing that we can do is to update data already in the database; again by deriving information from the spreadsheet. Let's say that Pygoscelis has changed some of the details in his "PPI Cases" worksheet (perhaps changing one of the case titles), then all he has to do is to click on **Tools | Macros | Run Macro...** to run the macro, which will scroll through the worksheet looking for any changes:

```
Sub update_case
    Dim oRange as Object
    Dim oSheet as object
    Dim oRowSet as Object
    Dim oCell as Object
    Dim oSql as String
```

```
    Dim r as Integer
    Dim c as Integer
    Dim oResult as Object
    Dim oStatement as Object
    Dim db as Object

    db = connect_to_database ("ppi")

    oRange =  thisComponent.getCurrentSelection.getRangeAddress
    oSheet = thisComponent.CurrentSelection.getSpreadsheet
    oRowSet = get_rowset (db, sql_select _
                ("case", Array("TITLE", "DETAILS")))
    r = 1
    oCell = oSheet.getCellByPosition(0,r)
    While oCell.String <> "" And Not oRowSet.isLast
        oRowSet.absolute(r)

        oCell = oSheet.getCellByPosition(0,r)
        If oCell.String <> oRowSet.getString(1) Then
            oSql = """TITLE"" = '" & oCell.String & "'"
        End If

        oCell = oSheet.getCellByPosition(1,r)
        If oCell.String <> oRowSet.getString(2) Then
           If oSql <> "" Then
                 oSql = oSql & ","
           End If
           oSql = oSql & """DETAILS"" = '" & oCell.String & "'"
        End If
        If oSql <> "" Then
         oSql =  "Update ""cases"" set " & oSql _
         & " where ""TITLE"" = '" & oRowSet.getString(1) & "'" _
         & " and ""DETAILS"" = '" & oRowSet.getString(2) & "'"
        End If
        oStatement = db.createStatement
        oResult = oStatement.executeQuery (oSql)
        r = r + 1
    Wend
    disconnect_from_database (db)
End Sub
```

The process of updating the database is similar to inserting new data:

- Information is obtained from the spreadsheet by making use of the range address.
- The appropriate SQL statement is built from the information.
- The SQL statement is passed to the database, but again no `RowSet` is created.

The main difference is that rather than taking an individual line of data from the spreadsheet, every line is checked. The macro then compares each line with the contents of the equivalent line in the `RowSet`. If there is a difference, then an update is carried out.

# Summary

In this chapter we've learned how to make use of the data stored in a database. And so we're now able to use a macro to obtain and use data in a database, create new records in a database, and update existing records in a database.

If you're using any database other that OpenOffice.org Base, then you'll have to install drivers for the database that you're going to be using, configure your ODBC software so that the database is available on your system, and register the database with OpenOffice.org as a Data Source. You can then connect to it using the `DatabaseContext` service.

With a connection in place you can extract information from the database, add new records to tables in the database, and update existing records in the database. You have also learned how to extract information from the database and also how to insert a new record or update an existing one.

Anyway...

Pygoscelis ignored Sphen and slumped back in his chair. All he was really aware of was the increasing pain in his head and Korora's heart beating through his back. He tried to concentrate on the contents of the report now smoldering in the bin.

At last he gave in and slipped back into oblivion.

In Chapter 7 we'll be looking at how Pygoscelis created that report when we look at '*Working with Other Documents*'.

# 7
# Working with Other Documents

Pygoscelis struggled for a moment and then fell silent. Sphen walked over to him, and slapped him. Nothing. He slapped him again. Still nothing. He raised his hand again, this time clenching his fist.

"Just leave him alone, will you!"

"Ah, Korora! I'd quite forgotten about you."

He relaxed his arm, and then moved so that he could face her.

"I must admit, I was impressed with that report. Very pretty. Almost a shame to burn it. Tell me how did you make it? You may as well tell me; it can't do any harm now."

She looked back at him. "No", she thought to herself, "it won't do any harm, but it may just delay you a little bit longer. Just long enough to..."

Well, we're not going to delay. We'll spend this chapter looking at how our Calc macros can make use of other documents. In the previous chapters, we've seen that we can:

- Open an existing spreadsheet and manipulate its contents
- Open a number of spreadsheets and make use of the content from one spreadsheet in another one
- Extract information from a database, manipulate the results with a spreadsheet, and then write information back to the database

By the end of the chapter, you should be able to:

- Use macros to make use of OpenOffice.org Chart documents within a spreadsheet
- Use macros to open other documents, importing their contents into a usable form, and then processing those contents

So far we've dealt with data in a tabular format. After all, that's what a spreadsheet is. However, although useful, this is not always the best way to view the information—very often a picture is much more effective. And that's why we're going to start making use of OpenOffice.org Charts within a spreadsheet (all done automatically with macros, of course).

In this chapter (and, infact, from now on) we''ll be dealing with some quite new functionality—so you must ensure that you're running the most current versions of OOo for all of the code to work.

# The OpenOffice.org Chart

Perhaps, the most useful document that you can make use of in conjunction with a spreadsheet is the chart. Why? Simply because—give someone ten pages of figures and they'll be able to understand the contents, eventually; give someone a chart representing ten pages of figures and they'll be able to understand the contents, immediately.

So, let's look at converting a set of raw data in a spreadsheet into a professional-looking chart.

# Inserting a Simple Chart into a Spreadsheet

Obviously, the first thing to do is to enter the information into a spreadsheet, either from another spreadsheet, from a database, or (if you really must) manually:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Date | Cases Opened | Cases Closed | |
| 2 | 01/10/06 | 3 | 10 | |
| 3 | 01/11/06 | 4 | 9 | |
| 4 | 01/12/06 | 5 | 7 | |
| 5 | 01/01/07 | 2 | 5 | |
| 6 | 01/02/07 | 1 | 9 | |
| 7 | 01/03/07 | 2 | 7 | |
| 8 | 01/04/07 | 3 | 8 | |
| 9 | 01/05/07 | 4 | 9 | |
| 10 | 01/06/07 | 5 | 0 | |
| 11 | 01/07/07 | 7 | 4 | |
| 12 | 01/08/07 | 8 | 5 | |
| 13 | 01/09/07 | 9 | 6 | |
| 14 | | | | |

Next, we need to select the data in the spreadsheet; and remember that after selecting the data with the mouse, you can either click on **Tools | Macros | Run Macro...** or go back to the Macro editor and click on the Run button, then run your macro:

```
Sub Main
  simple_chart
End Sub

Sub simple_chart
  Dim oRange as Object
  Dim oSheet as Object
  Dim oCharts as Object
  Dim cTitle as String
  Dim oRect As New com.sun.star.awt.Rectangle
  Dim oRangeAddress(1) As New com.sun.star.table.CellRangeAddress

  cTitle = "PPI Cases"

  oRange =  thisComponent.getCurrentSelection.getRangeAddress
  oSheet = thisComponent.CurrentSelection.getSpreadsheet
  oCharts = oSheet.Charts

  'Set Y azis Data
  oRangeAddress(1).Sheet = oRange.Sheet
  oRangeAddress(1).StartColumn = oRange.StartColumn
  oRangeAddress(1).EndColumn = oRange.StartColumn
  oRangeAddress(1).StartRow = oRange.StartRow
  oRangeAddress(1).EndRow = oRange.EndRow

  'Set X axis Data
  oRangeAddress(0).Sheet = oRange.Sheet
  oRangeAddress(0).StartColumn = oRange.StartColumn + 1
  oRangeAddress(0).EndColumn = oRange.EndColumn
  oRangeAddress(0).StartRow = oRange.StartRow
   oRangeAddress(0).EndRow = oRange.EndRow

  oCharts.addNewByName(cTitle,oRect,oRangeAddress(),TRUE, TRUE)
End Sub
```

If you read through the macro, then you'll soon realize that the key line is:

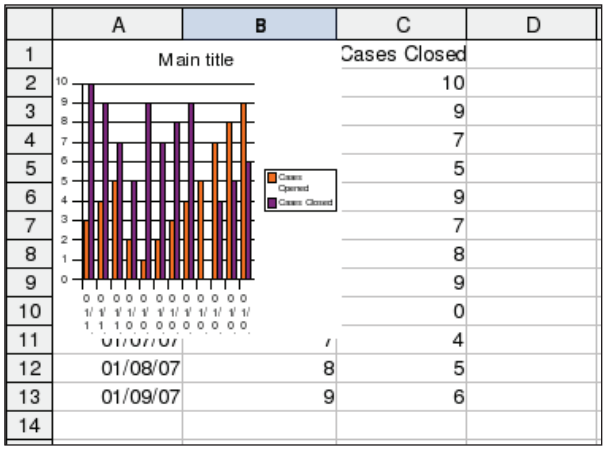
```
oCharts.addNewByName(cTitle,oRect,oRangeAddress(),TRUE, TRUE)
```

This is the line of code that actually creates the chart, and all of the previous lines are involved in setting up the required inputs to the chart creation. The inputs that you need to be aware of are:

- A title for the chart—this is just used to reference the chart, it isn't displayed.

- A `com.sun.star.awt.Rectangle`—this is used to define the size and position of the chart within the spreadsheet.
- A `com.sun.star.table.CellRangeAddress`—to define the ranges of data to be used for the Y axis and the X axis. As you can see the ranges are defined in exactly the same way that we've defined ranges before, by setting the start and end rows and columns.

Before long you'll have an OpenOffice.org chart inserted into the spreadsheet:




| | A | B | C | D |
|---|---|---|---|---|
| 1 | | | Cases Closed | |
| 2 | | | 10 | |
| 3 | | | 9 | |
| 4 | | | 7 | |
| 5 | | | 5 | |
| 6 | | | 9 | |
| 7 | | | 7 | |
| 8 | | | 8 | |
| 9 | | | 9 | |
| 10 | | | 0 | |
| 11 | | 7 | 4 | |
| 12 | 01/08/07 | 8 | 5 | |
| 13 | 01/09/07 | 9 | 6 | |
| 14 | | | | |

OK. I admit it; it doesn't look very impressive. But it does show you how easy it is to create a chart using the data in a spreadsheet. Next, we need to start looking at making the chart look a little bit nicer.

# Formatting OpenOffice.org Charts

If you were creating this chart manually you'd probably:

- Use the mouse to increase the area of the chart to make it more readable
- Add an appropriate title
- Put labels on the X axis and the Y axis
- Change the format of the dates (on the Y axis), so that you can actually tell what the date is

And so, that's what we'll do, but by using a macro.

## Chart Size

The chart that we've just created looks a little small, doesn't it? Fortunately, we can easily change this by making use of the `com.sun.star.awt.Rectangle` used in the creation of the chart:

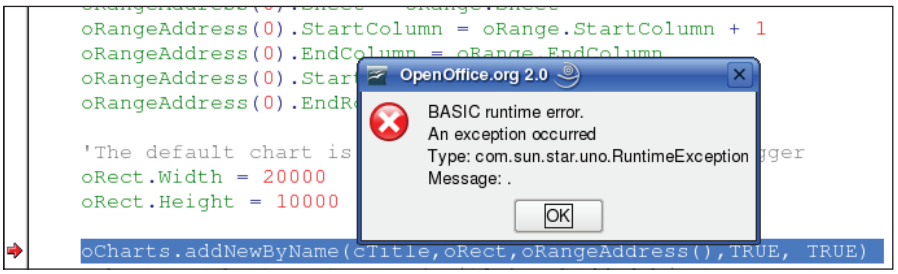
```
oRect.Width = 20000
oRect.Height = 10000
```

Note that the width and height are both defined in 1/100th of an mm, so the settings above give a width of 20cm and a height of 10cm.

This must, of course, be placed before the line:

```
oCharts.addNewByName(cTitle,oRect,oRangeAddress(),TRUE, TRUE)
```

Now, if you try this without removing the first chart that we created, then you're going to get an error:




This is, of course, because we create the chart with a name and we must always have uniquely named objects. The following is an easy answer to get the macro to remove the existing chart (if it exists) before we create a new one:

```
if oCharts.hasByName(cTitle) Then
  oCharts.RemoveByname(cTitle)
 end if
```

If you now re-run the macro (after having ensured that the correct cells have been selected in the spreadsheet), then a new chart will be displayed—whether or not you've already got one displayed.

## Chart Title

Setting the chart title (the displayed text) is maybe more involved than you might expect, but is by no means difficult:

```
Dim oChart As Object

oChart = oCharts.getByName(cTitle).embeddedObject
oChart.HasMainTitle = True
oChart.Title.string = cTitle
```

Also remember to add the above code after the line:

```
oCharts.addNewByName(cTitle, oRect, oRangeAddress(), TRUE, TRUE)
```

Remembering that we've already set `cTitle`:

```
cTitle = "PPI Cases"
```

This time rather than seeing 'Main title' at the top of the chart you will see 'PPI Cases'.

## Adding Chart Axis Labels

The labels for the X axis and the Y Axis are at least as important as the Chart Title; without these, the divisions on the chart are meaningless. After all, what does 0, 1, 2, 3, etc., actually mean? Number of elephants? Number of apples? So, to remove the possibility of confusion, just add the labels, especially since it is so easy to do:

```
oChart.diagram.HasXAxisTitle = True
oChart.diagram.XAxisTitle.string = "Date"
oChart.diagram.HasYAxisTitle = True
oChart.diagram.YAxisTitle.string = "Number of Cases"
```

## Y Axis Text Orientation

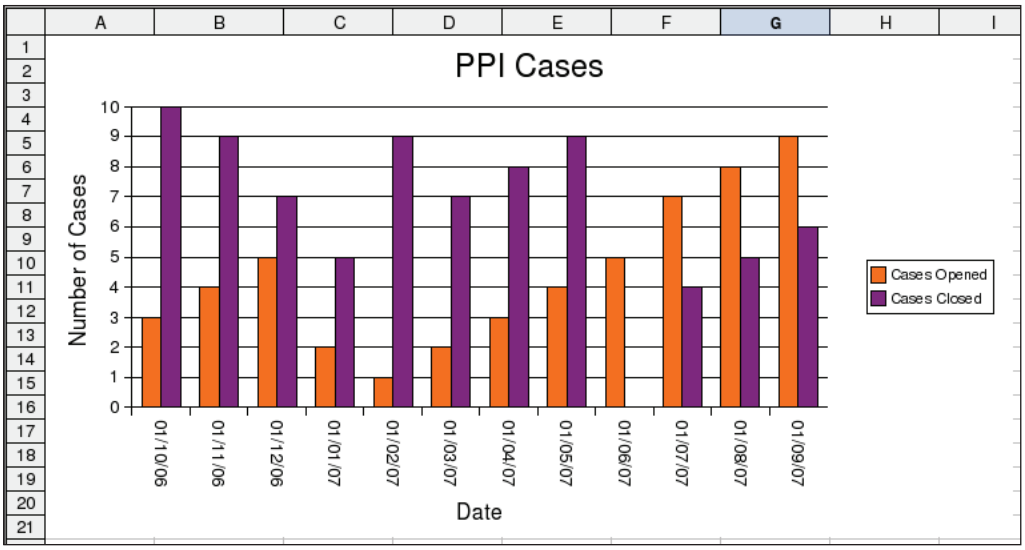
The text orientation for text on the Y axis can make a big difference to understanding the information, and dates look better if they are kept to a single line:

```
oChart.diagram.XAxis.TextBreak = False
Chart.diagram.XAxis.TextRotation = 27000
```

And as you've probably guessed, the rotation is measured in 1/100th of a degree.

## A Fully Formatted Bar Chart

After all this, you have a nicely formatted, professional looking chart in your spreadsheet:




Also, just to make sure that you know the order in which things need to be done, here is the complete code for the macro:

```
Sub nicer_chart
  Dim oRange as Object
  Dim oSheet as Object
  Dim oCharts as Object
  Dim oChart As Object
  Dim cTitle as String
  Dim oRect As New com.sun.star.awt.Rectangle
  Dim oRangeAddress(1) As New com.sun.star.table.CellRangeAddress

  cTitle = "PPI Cases"

  oRange =  thisComponent.getCurrentSelection.getRangeAddress
  oSheet = thisComponent.CurrentSelection.getSpreadsheet
  oCharts = oSheet.Charts

  if oCharts.hasByName(cTitle) Then
    oCharts.RemoveByname(cTitle)
  end if

  'Set Y azis Data
  oRangeAddress(1).Sheet = oRange.Sheet
  oRangeAddress(1).StartColumn = oRange.StartColumn
  oRangeAddress(1).EndColumn = oRange.StartColumn
  oRangeAddress(1).StartRow = oRange.StartRow
  oRangeAddress(1).EndRow = oRange.EndRow
```

```
    'Set X axis Data
    oRangeAddress(0).Sheet = oRange.Sheet
    oRangeAddress(0).StartColumn = oRange.StartColumn + 1
    oRangeAddress(0).EndColumn = oRange.EndColumn
    oRangeAddress(0).StartRow = oRange.StartRow
    oRangeAddress(0).EndRow = oRange.EndRow

    'The default chart is a little small - so make it bigger
    oRect.Width = 20000
    oRect.Height = 10000

    oCharts.addNewByName(cTitle,oRect,oRangeAddress(),TRUE, TRUE)
    oChart = oCharts.getByName(cTitle).embeddedObject
    oChart.HasMainTitle = True
    oChart.Title.string = cTitle

    'More formating
    oChart.diagram.XAxis.TextRotation = 27000
    oChart.diagram.XAxis.TextBreak = False
    oChart.diagram.HasXAxisTitle = True
    oChart.diagram.XAxisTitle.string = "Date"
    oChart.diagram.HasYAxisTitle = True
    oChart.diagram.YAxisTitle.string = "Number of Cases"
End Sub
```

Of course, that's not to say that you can't improve on this macro, for example:

- The data is limited to the cells that you manually select; instead you could have the range as an input to the subroutine.
- The labels and titles are all hard-coded, again these may be better as string inputs.
- The size of the rectangle for the chart could also be an input rather than fixed.

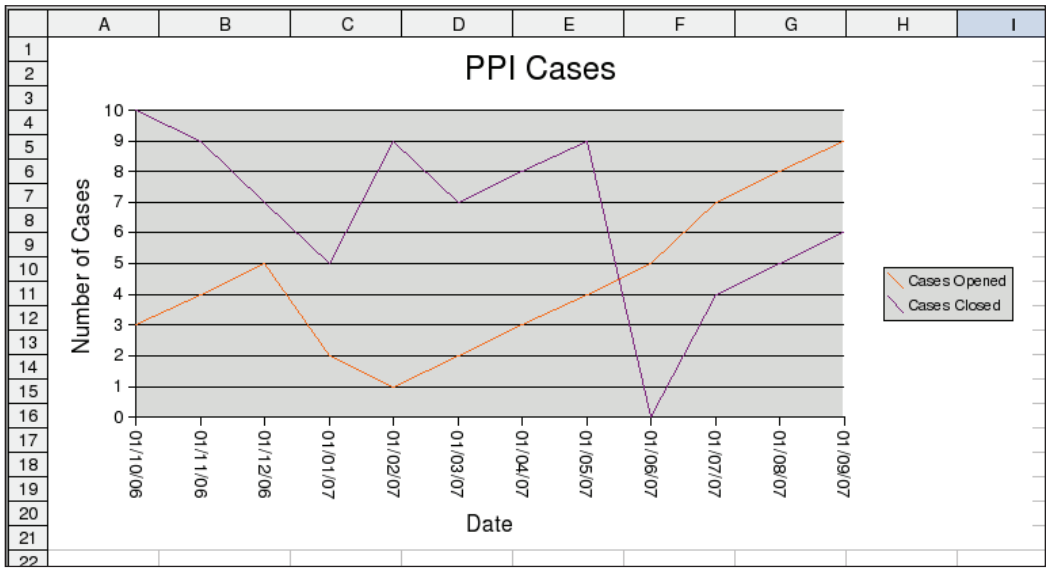
## Other Chart Types

At this stage, you're probably wondering about other chart types. After all, there's nothing in the code about them at all. Is the bar chart the only one available? Well, of course not; that's just the default type.

If you want to use a different type, then you just state which of the chart's diagram services you want to use. For example:

```
oChart.diagram = _
  oChart.createInstance("com.sun.star.chart.LineDiagram")
```

This code results in the chart being presented as a line diagram:




You've actually got a choice from:

- AreaDiagram
- BarDiagram (the default)
- DonutDigram
- LineDiagram
- NetDiagram
- PieDiagram
- StackableDiagram
- StockDiagram
- XYDiagram

Again, you can improve the macro (if you wish) by inputting the diagram type.

# Using Documents from Other Sources

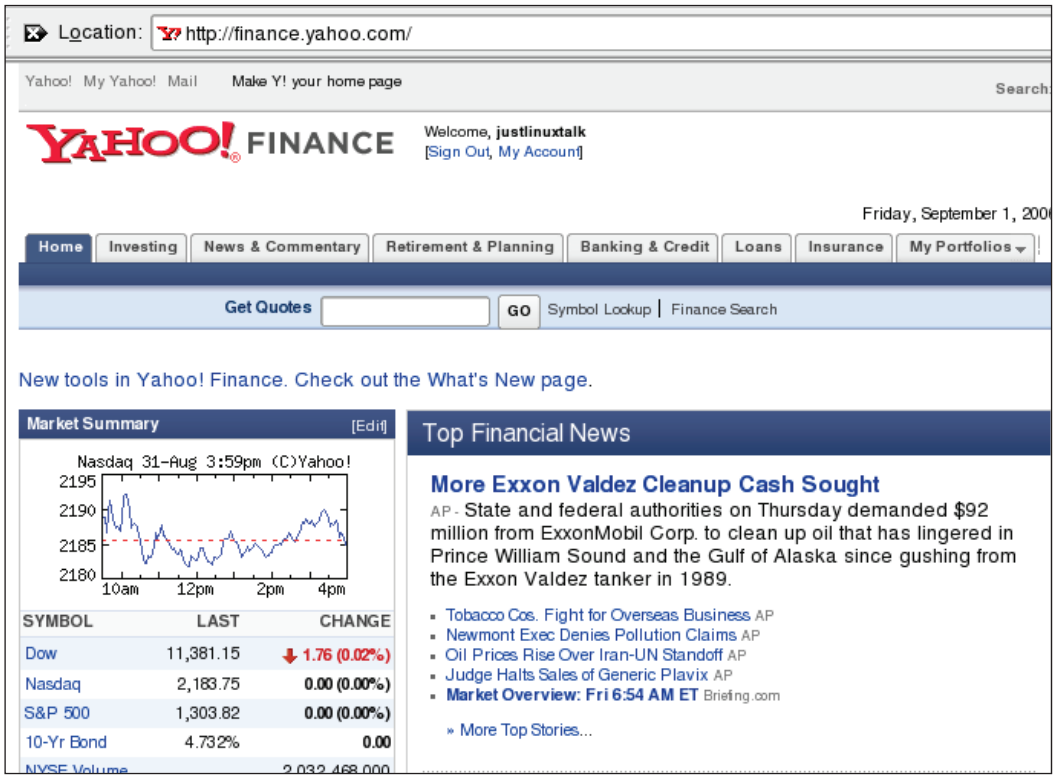
We've already learned how to use data stored in:

- A spreadsheet
- Other spreadsheets on the system
- A database

We've also learned how to make use of a chart document with our data. Next we'll look at data from other sources and make use of that.

# Stock Market Analysis—Yahoo! Finance

So what data shall we use? Let's see if we can make a fortune on the stock markets:




Are there any shares that you're particularly interested in? How about looking at the performance of the London Stock Exchange:


Along with the daily data, the service allows you to view historical data:

Now, how about doing all of that automatically?

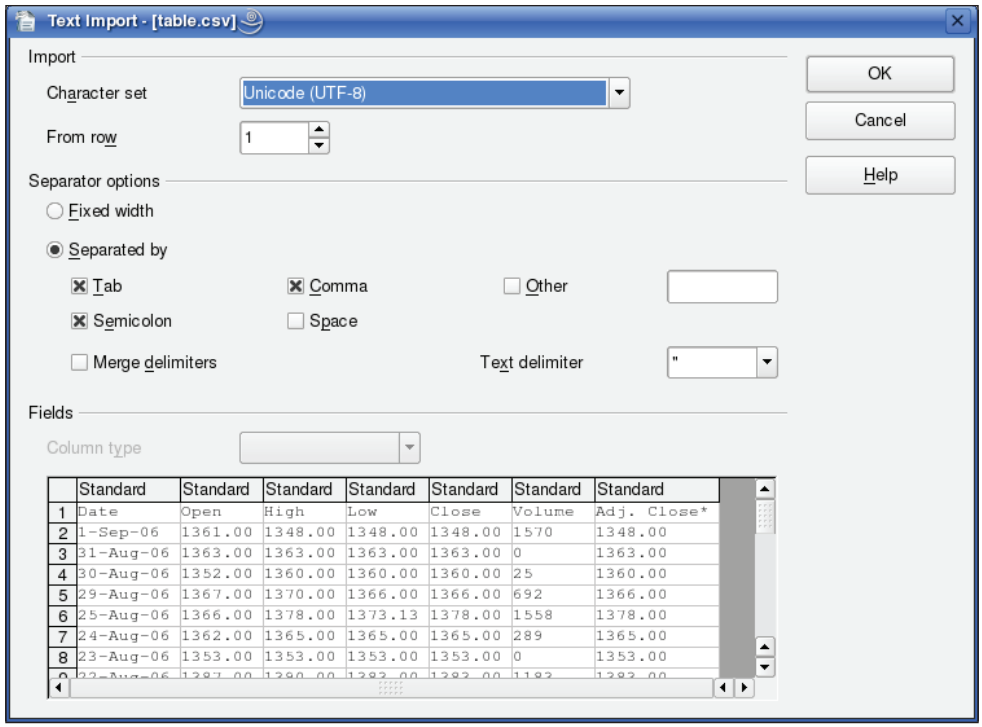
## Importing an Historical CSV File from Yahoo! Finance

Before importing the `.csv` file automatically, we first need to understand how to do it manually. If you spend some time looking at the Yahoo! Finance website, then you find that the historical `.csv` file for a company can be accessed using the same URL. All you have to do is to change the `s` input to the code for the company that you're interested in:


You'll find that you now have a choice:

- Save the file to disk
- Open the file using an application

If you choose OpenOffice.org Calc for the latter, then you'll be presented with the Text Import Wizard:



Next, you need to tell the wizard about the delimiter that the file is using to separate the columns (in this case it is a CSV file), OpenOffice.org Calc can then interpret the contents of the file correctly. Once you'v set the delimiters (or just left the defaults), then you can press **OK**, and a few moments later you'll have a spreadsheet containing the stock market details for the company that you've chosen.

Now that we've seen how to import a CSV file *manually*, it seems appropriate to creat a macro to do all that for us. Remember—for the following code to operate correctly, you must be using the most currect version of OpenOffice.org (2.0.4 on Windows or 2.0.2 on Linux). And don't worry when you run it—it does take a while to process all of the data:

```
Sub Main
  get_stock_price_history "MSF.L"
End Sub

Sub get_stock_price_history (iCompany_symbol as String)
  Dim oUrl as String
  Dim oDoc as Object
  Dim oPropertyValue(0) As New com.sun.star.beans.PropertyValue

  oUrl = "http://ichart.finance.yahoo.com/table.csv" _
                                  & "?s="& iCompany_symbol &"&e=.csv"

  oPropertyValue(0).Name = "FilterOptions"
  oPropertyValue(0).Value = "44"
  oDoc = starDeskTop.loadComponentFromURL( oUrl, "_blank", 0, _
    oPropertyValue)
End Sub
```

I'm sure that you can follow the macro quite easily as you've seen most of the code used in other subroutines and functions. There are just two lines that you may have to think about:

```
  oPropertyValue(0).Name = "FilterOptions"
  oPropertyValue(0).Value = "44"
```

You're probably wondering what the `44` stands for. Quite simply—it's the ASCII code for a comma.

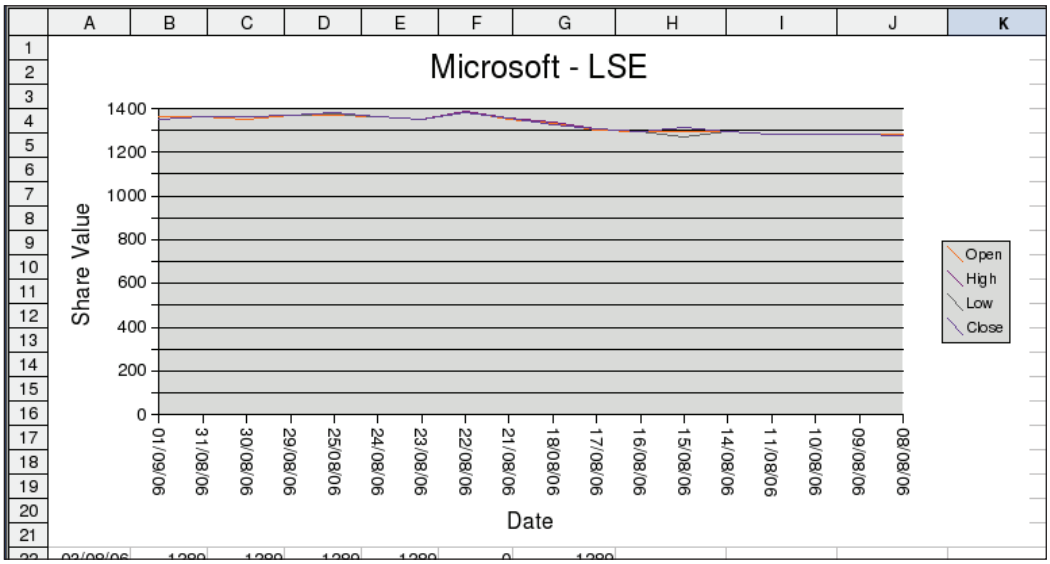
Now if you run the macro, you'll end up with something like:

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | Date | Open | High | Low | Close | Volume | Adj. Close* | |
| 2 | 01/09/06 | 1361 | 1348 | 1348 | 1348 | 1570 | 1348 | |
| 3 | 31/08/06 | 1363 | 1363 | 1363 | 1363 | 0 | 1363 | |
| 4 | 30/08/06 | 1352 | 1360 | 1360 | 1360 | 25 | 1360 | |
| 5 | 29/08/06 | 1367 | 1370 | 1366 | 1366 | 692 | 1366 | |
| 6 | 25/08/06 | 1366 | 1378 | 1373.13 | 1378 | 1558 | 1378 | |
| 7 | 24/08/06 | 1362 | 1365 | 1365 | 1365 | 289 | 1365 | |
| 8 | 23/08/06 | 1353 | 1353 | 1353 | 1353 | 0 | 1353 | |
| 9 | 22/08/06 | 1387 | 1390 | 1383 | 1383 | 1183 | 1383 | |
| 10 | 21/08/06 | 1351 | 1355 | 1355 | 1355 | 368 | 1355 | |
| 11 | 18/08/06 | 1331 | 1337 | 1327.34 | 1327.34 | 2311 | 1327.34 | |
| 12 | 17/08/06 | 1303 | 1307 | 1307 | 1307 | 24 | 1307 | |
| 13 | 16/08/06 | 1294 | 1294 | 1294 | 1294 | 3000 | 1294 | |
| 14 | 15/08/06 | 1292 | 1315 | 1270 | 1315 | 4800 | 1315 | |
| 15 | 14/08/06 | 1295 | 1292.5 | 1292 | 1292.5 | 5750 | 1292.5 | |
| 16 | 11/08/06 | 1280 | 1280 | 1280 | 1280 | 0 | 1280 | |
| 17 | 10/08/06 | 1280 | 1280 | 1280 | 1280 | 0 | 1280 | |
| 18 | 09/08/06 | 1281 | 1281 | 1281 | 1281 | 0 | 1281 | |
| 19 | 08/08/06 | 1280 | 1278 | 1278 | 1278 | 197 | 1278 | |
| 20 | 07/08/06 | 1275 | 1278 | 1277 | 1277 | 53 | 1277 | |
| 21 | 04/08/06 | 1281 | 1284 | 1284 | 1284 | 52 | 1284 | |
| 22 | 03/08/06 | 1289 | 1289 | 1289 | 1289 | 0 | 1289 | |
| 23 | 02/08/06 | 1285 | 1285 | 1285 | 1285 | 0 | 1285 | |

With the data correctly imported, you can either manually select an area of data, or if you've modified the `nicer_chart` macro to accept X and Y inputs, then you can run:

```
Sub Main
  yColumn = 0
  yStartRow= 0
  yEndRow = 18
  xStartColumn = 1
  xEndColumn = 4
  xStartRow= 0
  xEndRow = 18
  Nicer_chart( yColumn,  yStartRow,  yEndRow,  _
                    xStartColumn,  xEndColumn,  xStartRow,  xEndRow)
End Sub
```

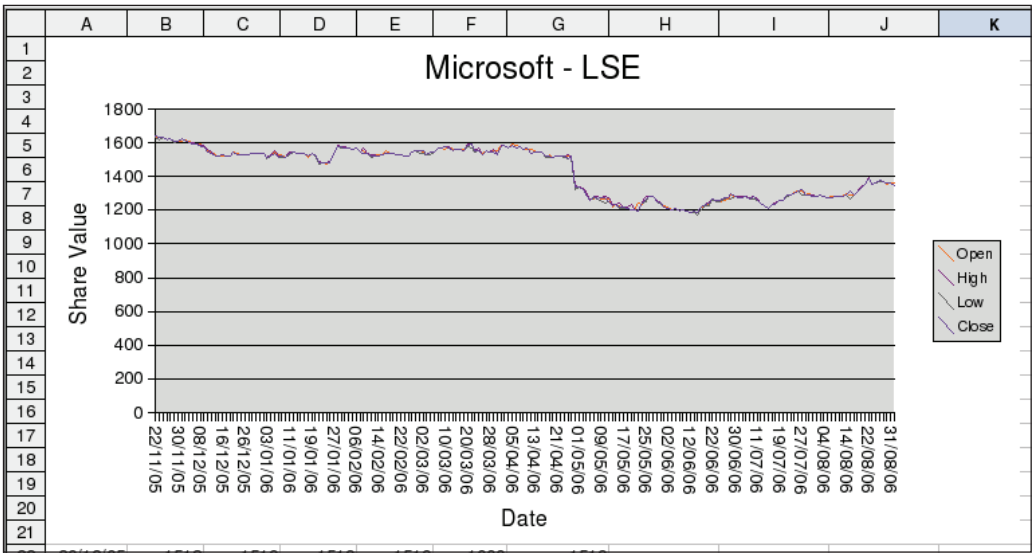
Now you're able to import the historical `.csv` file and display the results in a chart:




However, there's just one problem with the chart—it is reversed, i.e. the dates go from right to left instead of left to right. This is because the dates are in descending order in the spreadsheet. So all you have to do is to add a macro to sort the data for you:

```
Sub Main
  get_stock_price_history "MSF.L"
  range_sort "A1:G201"
End Sub

Sub range_sort(iSortArea as String)
  Dim oSortField(0) As New com.sun.star.table.TableSortField
  Dim oPropertyValue(1) As New com.sun.star.beans.PropertyValue
  Dim oRange as Object

  oRange = ThisComponent.Sheets(0).getCellRangeByName(iSortArea)

  oSortField(0).Field = 0
  oSortField(0).IsAscending = True
  oSortField(0).IsCaseSensitive = False
  oPropertyValue(0).Name = "SortFields"
  oPropertyValue(0).Value = oSortField
  oPropertyValue(1).Name = "ContainsHeader"
  oPropertyValue(1).Value = True

  oRange.sort(oPropertyValue)
End Sub
```

When you put it all together, you've got a nice picture of how Microsoft has been doing in the London Stock Exchange for the past few months:




Now can anyone tell me—does that mean we should be buying or selling?

## Comparing Companies within Yahoo! Finance

We've seen how to import a `.csv` file from Yahoo! Finance to view the historical performance of a company, but what about looking at how all of our shares are doing currently?

Just like the historical `.csv`, you just need to use the correct URL to access the data:


As before, we need to enter a code (or symbol) for each company of interest, for example:

- MSFT: Microsoft on the Nasdaq Market
- RHAT: Red Hat on the Nasdaq Market
- NOVL: Novell on Nasdaq

If you're not sure of the code, then there is a look-up facility on the website.

You may also be wondering about a portion of the URL—`f=sl1d1`. These are some of the fields that will be included in the `.csv` file:

- `s`: company symbol
- `l1`: Last trade price
- `d1`: Last trade date

There are a few different fields that you can use; refer to the Yahoo! Finance website to learn about all of the others.

So, the only thing that you need to do now is to it add a macro to import the information:

```
Sub Main
  get_stock_price( array("MSFT","RHAT","NOVL"))
End Sub

Sub get_stock_price (iCompany_symbols)
  Dim oUrl as String
  Dim oDoc as Object
  Dim oSymbols as String
  Dim oFields as String
  Dim oPropertyValue(0) As New com.sun.star.beans.PropertyValue

  oSymbols = join (iCompany_symbols,"&s=")
  oFields = "sl1d1"

  oUrl = "http://finance.yahoo.com/d/quotes.csv" _
           & "?s="& oSymbols & "&f=" & oFields &"&e=.csv"

  oPropertyValue(0).Name = "FilterOptions"
  oPropertyValue(0).Value = "44" 'Use a comma as the field separator
  oDoc = starDeskTop.loadComponentFromURL( _
                                    oUrl, "_blank", 0, oPropertyValue)
End Sub
```

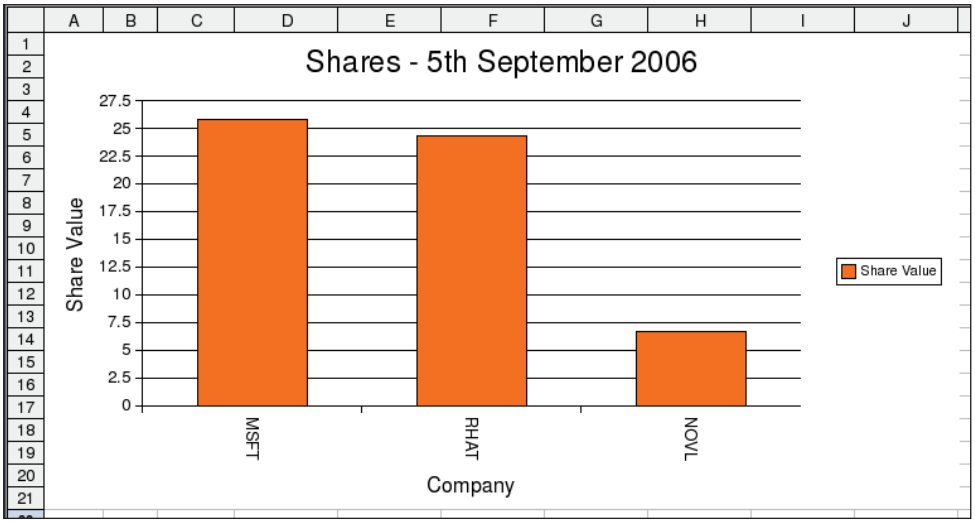
The new, current data will now be loaded into a spreadsheet:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | MSFT | 25.69 | 09/05/06 | |
| 2 | RHAT | 24.26 | 09/05/06 | |
| 3 | NOVL | 6.7 | 09/05/06 | |
| 4 | | | | |

I know you can see the problem with the data already. You can't? It's quite simple. There are no headers on the columns, which means that you won't be able to show the information in a chart. Fortunately, it just takes a couple of lines of code to do the job:

```
oDoc.Sheets(0).getRows.insertByIndex(0,1)
oDoc.Sheets(0).getCellByPosition ( 1, 0 ).String = "Share Value"
```

Now you can use a macro to display the data in a chart:




Of course, you may prefer to combine the `.csv` importation subroutine with the one for the historical `.csv` file. If you do, then you'll need to remember that:

- The historical `.csv` can only handle one company at a time, whereas the current `.csv` can handle as many companies as you need.
- The historical `.csv` is downloaded from `http://ichart.finance.yahoo.com`, whereas the current .csv is downloaded from `http://finance.yahoo.com`.
- The historical `.csv` is imported with column headers, whereas the current `.csv` is imported without column headers.

# Processing Web Pages

To start with a question—how do you get a macro to open a blank Calc document? Hopefully, you'll answer that you use the function that we wrote in Chapter 6:

```
Function open_spreadsheet
  Dim oURL as String
```

```
    oURL = "private:factory/scalc"
    open_spreadsheet = starDeskTop.loadComponentFromURL (oURL, _
                                                "_blank", 0, Array() )
End Function
```

So here's another question—How do you get a macro to open a blank Writer document? Not sure? Let me give you a clue—you can use the same subroutine, but you have to change one word. Give up? Just change:

```
    oURL = "private:factory/scalc"
```

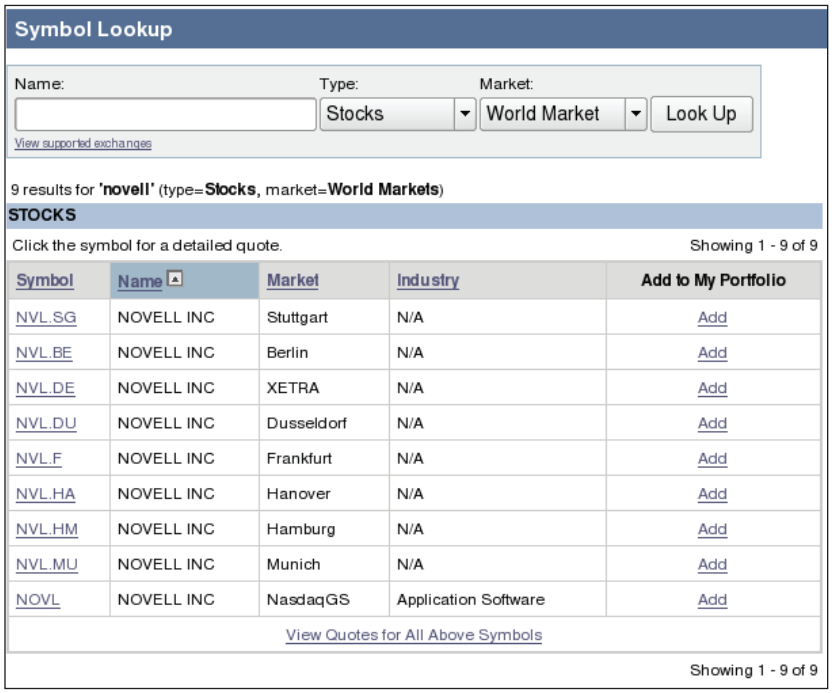
To:

```
    oURL = "private:factory/swriter"
```

From this I'm sure you can deduce that it's the URL that determines how a file is opened; this is true for both new, blank files, and existing documents.

So what happens when you try process a web page, for example:


which (when you look at it through a browser) gives you:



**Symbol Lookup**

Name: | Type: Stocks | Market: World Market | Look Up

View supported exchanges

9 results for **'novell'** (type=**Stocks**, market=**World Markets**)

**STOCKS**

Click the symbol for a detailed quote. Showing 1 - 9 of 9

| Symbol | Name | Market | Industry | Add to My Portfolio |
|---|---|---|---|---|
| NVL.SG | NOVELL INC | Stuttgart | N/A | Add |
| NVL.BE | NOVELL INC | Berlin | N/A | Add |
| NVL.DE | NOVELL INC | XETRA | N/A | Add |
| NVL.DU | NOVELL INC | Dusseldorf | N/A | Add |
| NVL.F | NOVELL INC | Frankfurt | N/A | Add |
| NVL.HA | NOVELL INC | Hanover | N/A | Add |
| NVL.HM | NOVELL INC | Hamburg | N/A | Add |
| NVL.MU | NOVELL INC | Munich | N/A | Add |
| NOVL | NOVELL INC | NasdaqGS | Application Software | Add |

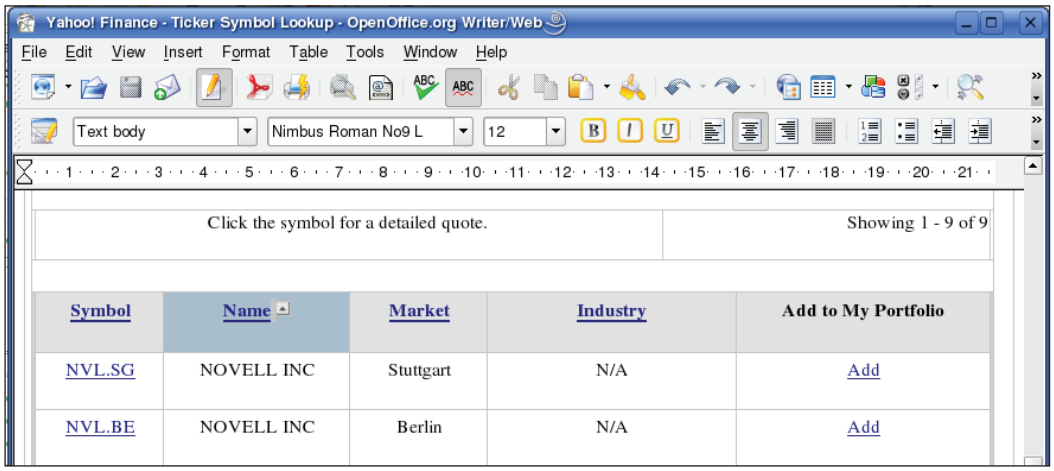
View Quotes for All Above Symbols

Showing 1 - 9 of 9

If you try to open a web page through a macro, for instance:

```
oUrl = "http://finance.yahoo.com/lookup?s=novell&t=S&m=ALL"
oDoc = oDesk.loadComponentFromURL( oUrl, "_blank", 0, Array())
```

This will be opened in the OpenOffice.org Web Writer application:




Unfortunately, this doesn't really do us much good, especially if we want to extract any information from the web page (for example, the list of symbols). Of course, we could go off and learn all about the web writer, but since we're already working with Calc, it seems sensible to import the page into Calc, and then process it. So that is what we'll do.

We've already seen that we can import a `.csv` file by using the `FilterOption` setting:

```
oPropertyValue(0).Name = "FilterOptions"
oPropertyValue(0).Value = "44" 'Use a comma as the field separator
oDoc = oDesk.loadComponentFromURL( oUrl, "_blank", 0, _
                                                    oPropertyValue)
```

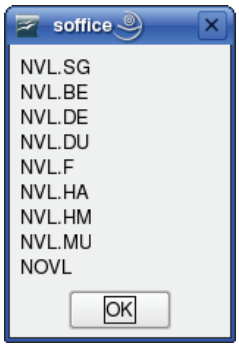
However, just doing that won't help with a web page, due to the `.html` suffix, OpenOffice.org automatically interprets HTML documents as a web page and not a spreadsheet. We, therefore, need to tell the macro specifically how to open the document:

```
oPropertyValue(1).Name  = "FilterName"
oPropertyValue(1).Value = "Text - txt - csv (StarCalc)"
```

Now we can write a macro that can take any web page and process the data contained with in it:

```
Sub Main
  get_symbols "novell"
End Sub

Sub get_symbols (icompany as String)
  Dim oUrl as String
  Dim oDoc as Object
  Dim oSheet as Object
  Dim oCell as Object
  Dim i as Integer
  Dim r as Integer
  Dim rows()
  Dim fields()
  Dim field()
  Dim result as String
  Dim oPropertyValue(1) As New com.sun.star.beans.PropertyValue

  oUrl = "http://finance.yahoo.com/lookup?s=" & icompany & _
                                                    "&t=S&m=ALL"
  oPropertyValue(0).Name = "FilterOptions"
  oPropertyValue(0).Value = "94" 'use a caret as a field separator
  oPropertyValue(1).Name  = "FilterName"
  oPropertyValue(1).Value = "Text - txt - csv (StarCalc)"

  oDoc = starDeskTop.loadComponentFromURL( _
                                  oUrl, "_blank", 0, oPropertyValue)

  'Get the cell that we need to process
  oCell =  oDoc.Sheets(0).getCellByPosition (0,54)

  'Get the number of symobls
  rows = split (ocell.String,"Showing 1 - ")
  fields = split (rows(1),"of")

  Dim symbol(fields(0))

  'Get the symbols for the company
  rows = split (ocell.String,"</tr>")
  For i = 0 To cInt (fields(0)) - 1
    fields = split (rows(i+3),"s=")
    field = Split( fields(1),"""")
    symbol(i) = field(0)
  Next i

  oDoc.Close(True) 'Edit this line out to view the preprocessed data

   msgbox join(symbol, chr(10))
End Sub
```

The end result is a message box containing the list of symbols for a company:




You'll notice that the macro:

- Specifically loads the web page as a Calc document
- Processes cells within the new spreadsheet by splitting the data into arrays
- Each array is created according to the HTML structure in particular cells

This is one of those cases where you can't write a generic macro. You need to understand the structure of the web page that you want to import and then build the macro around that structure.

# Summary

In this chapter we've seen that documents such as the OOo Chart can be used within Calc using the chart service that comes with every spreadsheet. The charts are bar graphs by default, but you can change this by setting the diagram object

CSV files can be imported by making use of the `FilterOptions` parameter when opening a document. Other document types can be loaded, but you must specify this by setting the `FilterName` property.

"Well, that interesting", said Sphen, "Thank you for the information, but I must love you and leave you. Don't worry though, I'll be back to finish the conversation."

He went back around to Pygoscelis, and slapped him again. "Still out cold", Sphen observed "And I thought you were so tough".

With that he turned and left the room. Immediately Korora felt a stirring behind her.

"I think you should be able to free your hands now; I've been working on the ropes while you two have been chatting. Then when you've done that have a look behind the filing cabinet."

Korora found that indeed her hands were free, and she was able to untie her legs herself. Then she looked where Pygoscelis had directed her, and found a laptop. This she placed on the desk, and then helped her boss over to it.

"Open up `emergency.ods`, it'll open a dialog box. Just click on the button that you see.'

And in Chapter 8 we'll see just how we can create forms in Calc as well.

# 8
# Developing Dialogs

Korora opened up the spreadsheet just as Pygoscelis had instructed. She expected to see a mass of figures or maybe a chart. Instead she saw a dialog. It contained a single button called **Run Emergency Macros**.

She clicked the button, and watched as spreadsheets were opened, populated with data, and then closed again. After a few minutes, a single message was displayed on the screen:

**All macros have been run. Have a nice day.**

And that's what Chapter 8 is all about—learning how to create dialogs.

At this stage you should be confident in running macros by either calling the macro by clicking on **Tools | Macros | Run Macro...** or calling a macro from the `Main` subroutine.

By the end of Chapter 8, you will be able to:

- Create custom dialogs
- Run macros from your dialogs
- Use the results of macros in your dialogs
- Customize OpenOffice.org's built-in dialogs

We'll start with the easiest of these: OpenOffice.org's own dialogs.

## Using OpenOffice.org's Built-In Dialogs

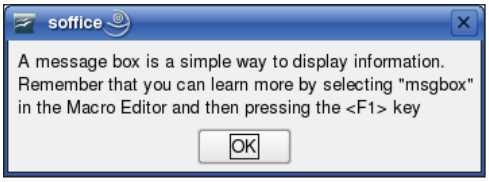
As you probably already know OpenOffice.org has two dialogs for you to use—message boxes and input boxes.

In fact we've used both of these in various macros already, but it's worth just reviewing them and looking at some aspects of using them that we haven't already covered.

# Customizing Message Boxes

The message box can, of course, be used to relay a piece of information to the users of your macros:

```
msgbox _
  "A message box is a simple way to display information." _
  & chr (10) _
  & "Remember that you can learn more by selecting ""msgbox""" _
  & chr (10) _
  & "in the Macro Editor and then pressing the <F1> key"
```



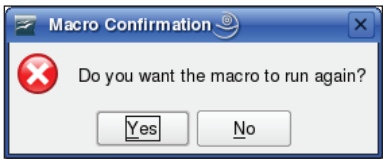

I'm sure that you've used message box in that way plenty of times. But you may not know that you can easily make the message box even more informative:

```
msgbox _
  "You can change the way that the message box is displayed," _
  & chr (10) _
  & "as well as adding your own title.", 48, "Macro Message Box"
```


In both of these examples we've used the message box to display something that we want the user to know about. However, did you know that you can get a message box to return data to your macro as well?

```
Sub message_box_return
  Dim msg_text as String
  Dim msg_return as Integer

  msg_text = "Do you want the macro to run again?"
```

```
  msg_return = msgbox (msg_text, 4 + 16, "Macro Confirmation")
  If msg_return = 6 Then
    message_box_return
  End If
End Sub
```




You'll notice that along with getting a return value from message box, we're combining box types; we've used `4 + 16` which translates as a Yes/No box with a Stop icon (you may also notice that your computer beeps as well).

For the full list of possible return values and box types, have a look at OpenOffice.org Calc's built-in help (remember you can type 'msgbox' into the basic editor, use the mouse to select it, and then press the *F1* key).

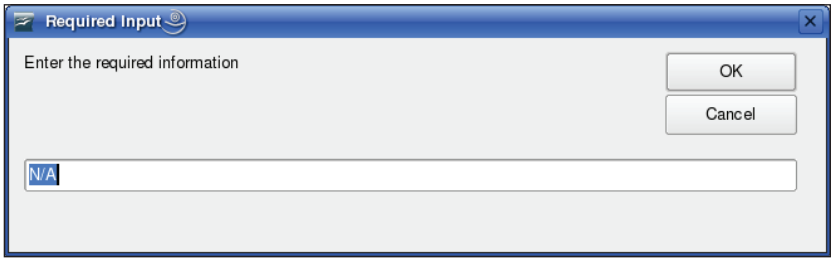
# Customizing Input Boxes

Unlike the message box, the input box is only a function (obvious really, when you think about it). Basically the customization that you're limited to is:

- Setting a default value for the text
- Changing the title of the box
- Changing the displayed text

For instance:

```
msg_text = "Enter the required information "
msg_title = "Required Input"
msg_default = "N/A"
input_text = inputbox (msg_text, msg_title, msg_default)
```



Now you may find that the input box and the message box are the only inputs that you'll ever need for your macros. However, you may decide that you need additional controls such as:

- Combo-boxes
- List boxes
- Buttons to carry out custom functions

If that's the case, then you need to consider creating your own custom dialogs.

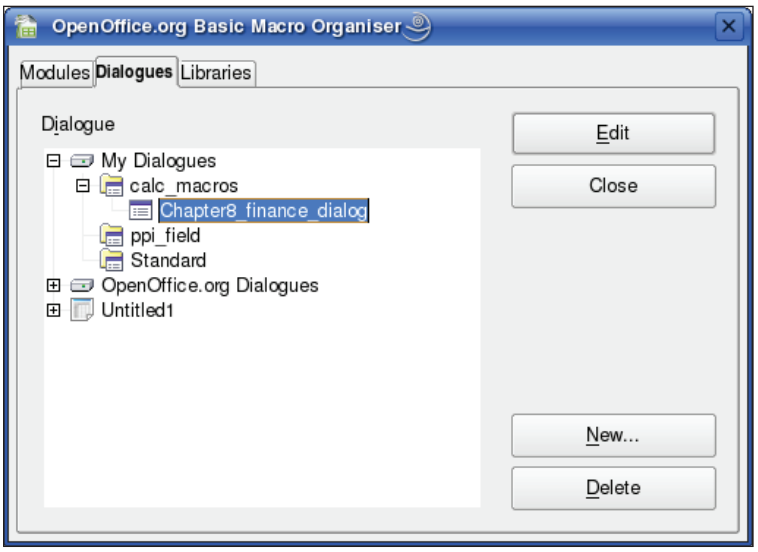
# Developing your Own Dialogs

You may remember that we first came across dialogs way back in Chapter 1, when we started getting used to OpenOffice.org's IDE. (Refer to Chapter 1, section *Designing Dialogs with the IDE*.)

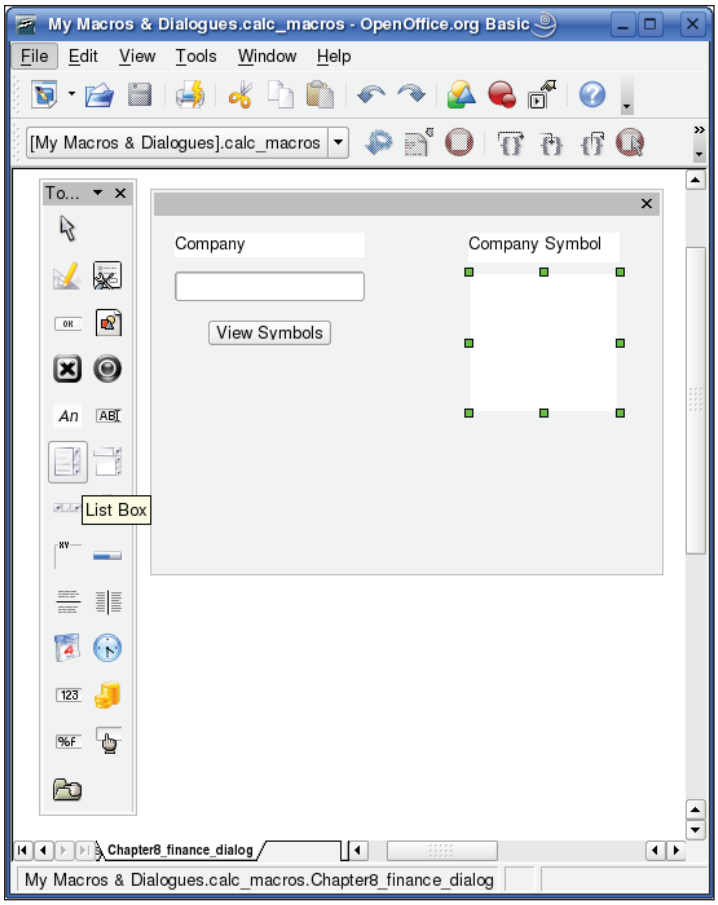
## Creating a Dialog

In Chapter 1 we saw how to create a dialog. Here we'll create a dialog named `Chapter8_finance_dialog`:

1. Create a new dialog by clicking on **Tools | Macros | Organize Dialogues...**




2. Edit this new dialog, and add any controls that we require from the IDE's toolbox:




So, we can build a dialog, but what can we actually do with it? Well, for a start let's call up the dialog and see what happens when we click a button.

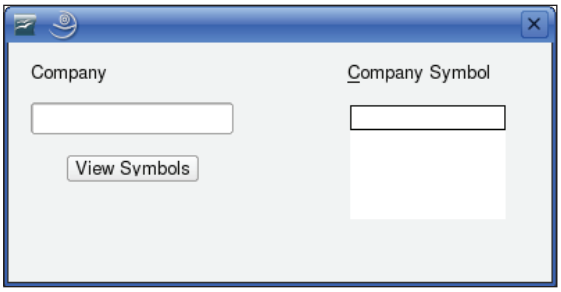
# Loading a Dialog

Having created your brand-new dialog, you're going to want to see it action. To do this you're going to have to write a macro to load the dialog:

```
Option Explicit

Dim oFinance_dialog as Object

Sub Main
  show_finance_dialog
End Sub

Sub show_finance_dialog
```

```
    basicLibraries.loadLibrary("Tools")
    'loadDailog needs the library and dialog names
    oFinance_dialog = loadDialog("calc_macros", _
    "Chapter8_finance_dialog")
    oFinance_dialog.execute
End Sub
```

And if you run this code it will, of course, display your new dialog:




OK. You can display the dialog, and you can click any of the buttons that you've added, but they won't do anything yet. But that's only because we haven't told them what to do.

Well, that's not altogether true—the dialog does one thing—if you press *Esc*, then the dialog will close.

So, let's start making the dialog more interesting.

# Assigning Actions to a Dialog

Just as Korora found, the most common thing you'll do with a dialog is to click on a button to run something; we'll have a look at how to do this.

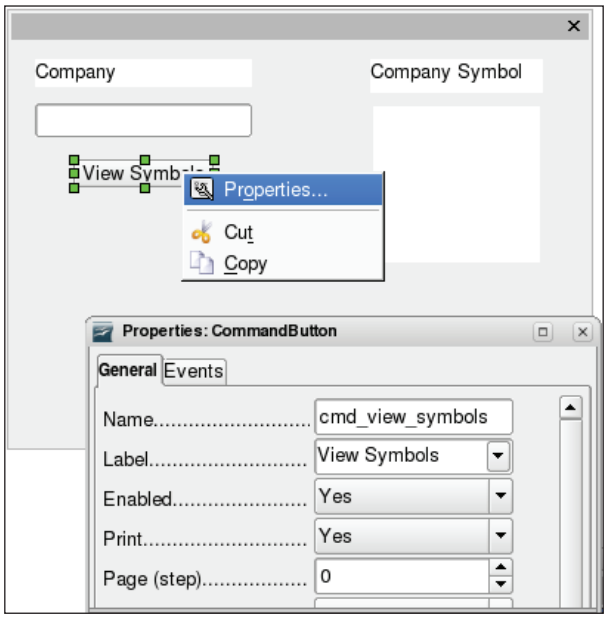
First we'll need to go back to writing some more code. Why? Because there are two stages to assign an action to one of the buttons on your dialog:

1. Create the macro that you want to run when you click on the button
2. Assign the macro to your button

By now you should be building quite a body of different macros—both ones that you've copied from the book and ones that you've come up with by yourself. You just need one more macro, the one that is called by your button, and which itself calls the macro that you actually want. In this case we're using the `get_symbols` subroutine from Chapter 7:

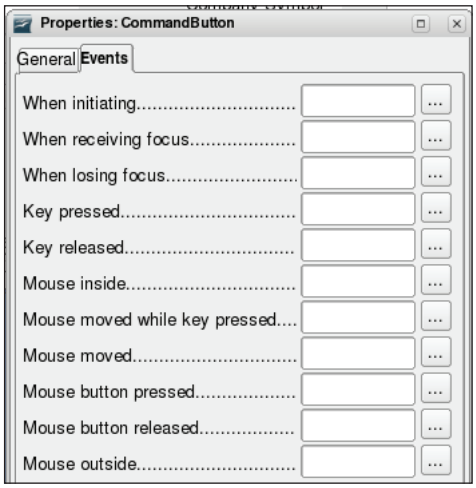
```
Sub click_cmd_view_symbols
  get_symbols "google"
End Sub
```

Next we need to give the button a suitable name. Do this via the button's property dialog:

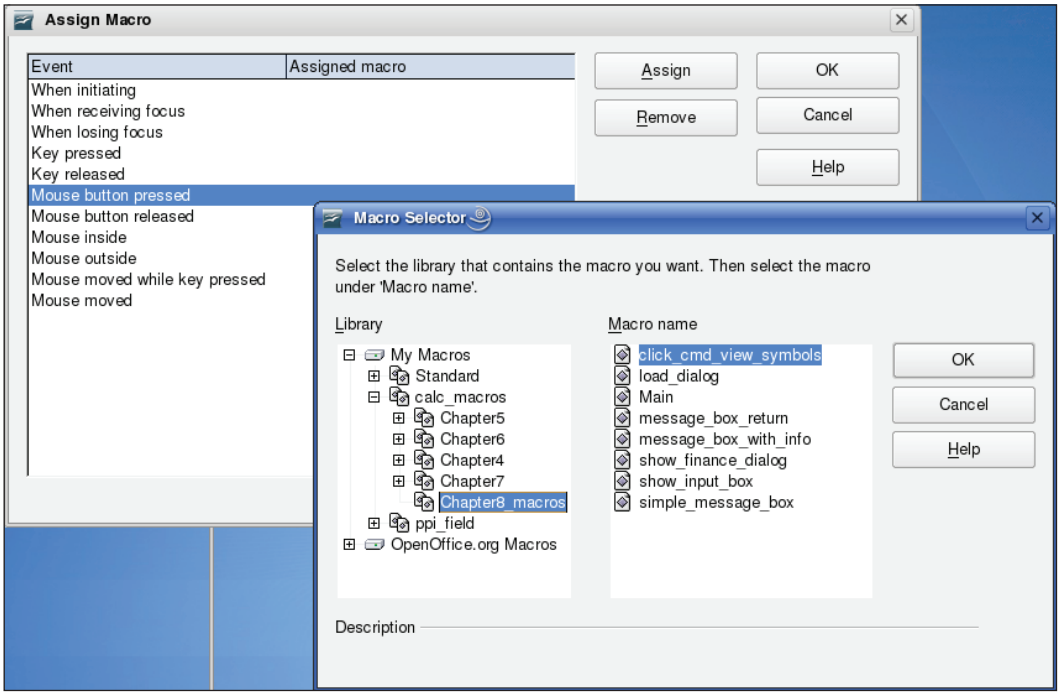



You'll have noticed that macro and the button have similar names. There's no law that says that you have do this. It's just that you'll find it much easier to manage your work if you name a macro to show that it's associated with an event of a button e.g. `click_cmd_view_symbols` (the macro) and `cmd_view_symbols` (the button). You may, of course, use your own naming system.

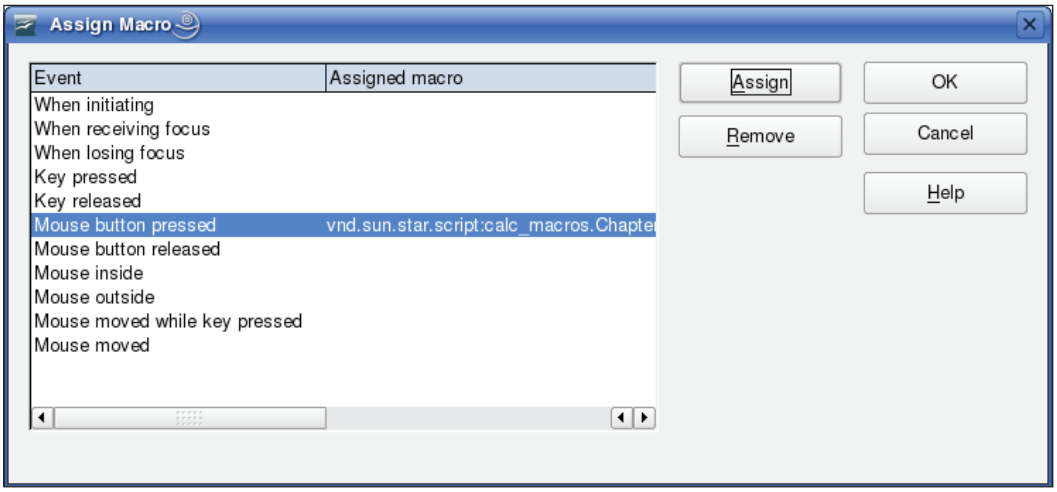
Now, keep the properties window open and move to its **Events** tab:



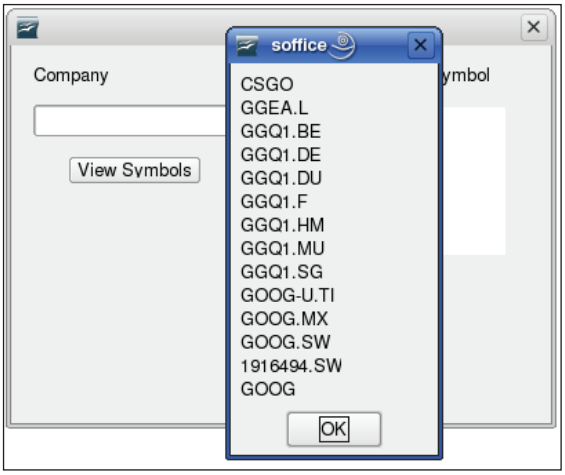

You now need to decide which event is going to trigger the macro. Typically for a button to be used with the mouse, this will be **Mouse button pressed**. Click on the 'three dots' button and you can then assign your macro to your button:



Once you've selected the appropriate macro, and clicked **OK**, then you'll be able to see the new details in the **Assign Macro** window:




And to prove that it works, all you have to do is run the macro that loads your dialog, click the button that you've just assigned the macro to, and see if you get the expected result:

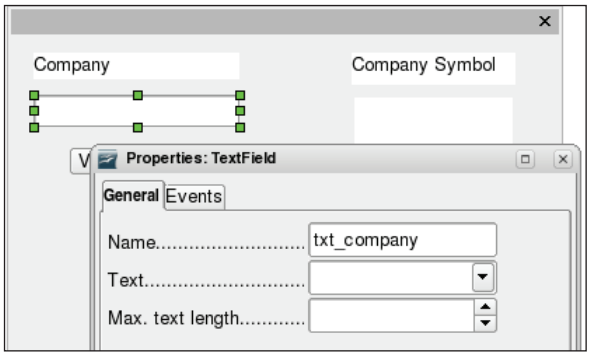



I'm sure that you'll find this very useful. You can now run any of your macros at the touch of a button. However, I'm also sure that you can see one major limitation—what happens if you need to see the symbols for any other company than Google? At the moment you will need to go back to the Basic Editor and amend the macro—which rather defeats the whole purpose of using a dialog.

The answer is, of course, to read information in from the dialog itself and then use this in your macro rather than use hard-coded text.

# Using Information in a Dialog

Probably the most common way to enter information in the dialog is via a TextField:




Now you can't just add a TextField to your dialog, and then use it directly in your code. For example, chances are the first thing that you'll try is:

```
Sub click_cmd_view_symbols
  get_symbols txt_company.Text
End Sub
```

And all you'll end up with is some error messages and a general feeling of frustration.

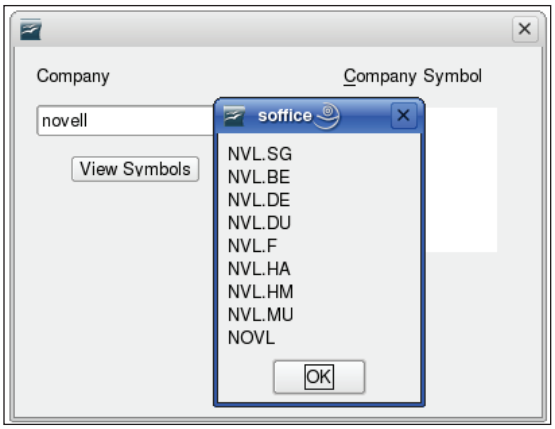
In order to make use of dialog controls from within your macros, you need to:

- Define a global variable that will represent the dialog (as we've already done)
- Access the contents of the control from the macro

So, the code for the button click now becomes:

```
Sub click_cmd_view_symbols
  Dim oTxt_company as Object
  oTxt_company = oFinance_dialog.getControl("txt_company")
  get_symbols oTxt_company.Text
End Sub
```

This time, when you load the macro and click the button, you'll see the output that you originally expected:

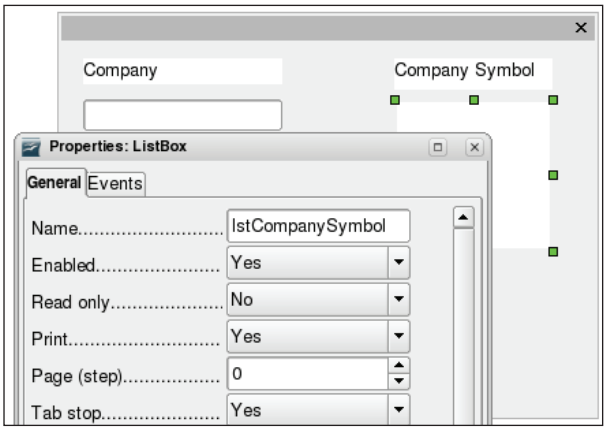



I think that you'll agree that this is now starting to get the kind of behavior that you would expect from any typical dialog. However, it's still rather limited at present. For example, the message-box output is useful, but it would be even more useful if the results were actually used in the dialog itself.

# Populating Controls in a Dialog

We've seen how easy it is to obtain information from a dialog and then make use of it in a macro. You'll find that this can work both ways—information from a macro can also be used in a dialog. It's just a matter of deciding how you want to use it.

If you're extracting a list of items from your macro (such as the Yahoo! Finance company symbols), then you may want to consider loading them into a ListBox:



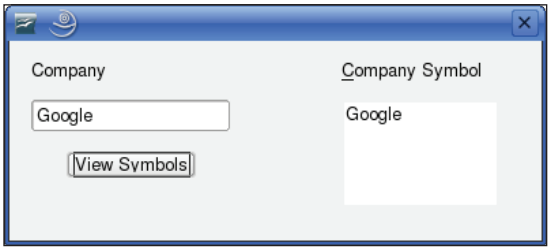
You'll remember from using the TextField that we will need to:

- Define a global variable that will represent the dialog
- Use the `getControl` method to access the dialogs

So, if you do want to see the ListBox in action then just try:

```
Sub click_cmd_view_symbols
  Dim oTxt_company as Object
  Dim oLstCompanySymbol as Object
  oTxt_company = oFinance_dialog.getControl("txt_company")
  oLstCompanySymbol = oFinance_dialog.getControl("lstCompanySymbol")
  oLstCompanySymbol.AddItem( oTxt_company.Text ,0)
End Sub
```

If you click on the button, then whatever is in the TextField gets added to the ListBox:




From that you can see how to load information into a Listbox making use of the `AddItem` method:

```
  lstCompanySymbol.AddItem( txt_company.Text ,0)
```

And for our next trick we can consider loading the ListBox with the output from the `get_symbols` macro. At the moment that's not possible because the output is just a message box. So the first thing to do is to change `get_symbols` from a subroutine to a function:

```
Function get_symbols (icompany as String) as Array
```

Also, rather than the message box we'll return an array of company symbols:

```
  get_symbols  = symbol
End Function
```

There's one other change that you may wish to make. I'm sure that you've noticed (and how could you miss it) that the `get_symbols` macro opens up a spreadsheet in order to collect the list of company symbols. Doesn't look very professional, does it? We can improve on this by hiding the spreadsheet when it's loaded.

You'll need to amend `get_symbols` so that it is able to use another `PropertyValue`:

```
Dim oPropertyValue(2) As New com.sun.star.beans.PropertyValue
```

And then set the `PropertyValue` so that it hides the document.

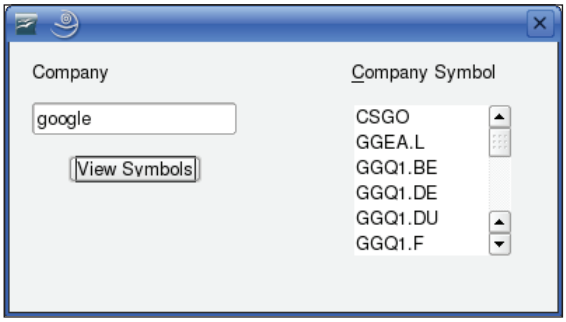
```
oPropertyValue(2).Name  = "Hidden"
oPropertyValue(2).Value = True
```

With that done we can change the button-click macro so that it loads the array from `get_symbols` directly into the ListBox:

```
oLstCompanySymbol = oFinance_dialog.getControl("1stCompanySymbol")
oLstCompanySymbol.AddItems (get_symbols (oTxt_company.Text), 0)
```

You'll notice here that instead of `AddItem` (which is used for loading individual items) we're using `AddItems` (used for loading sequences).

Our end result is a dialog that allows us to view the Yahoo! Finance symbols for any company listed with them:



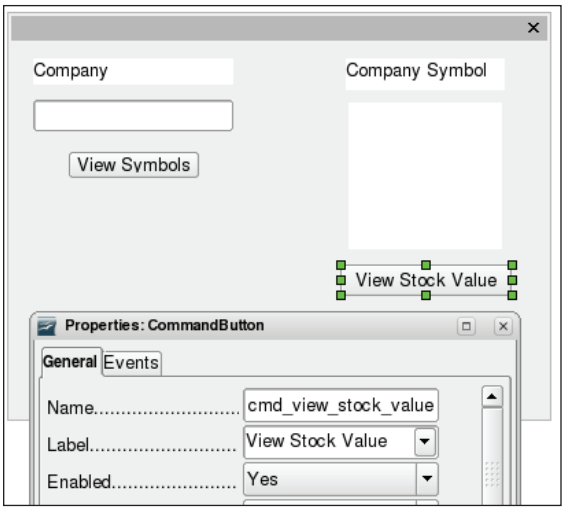

# The Finished Dialog

So far we've seen how to:

- Create a TextField on a dialog, and use information entered in it as an input variable for a macro
- Use the results from a macro to populate a ListBox in a dialog

We'll finish by adding another button; this will run a macro that uses data from the ListBox that we've just seen how to populate.

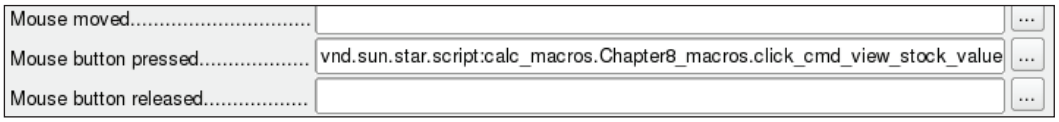
Obviously the first thing to do is to add a button to the dialog, give it a suitable name, and set its label:




Next it's time for us to create the macro that will be assigned to the button, again giving it a name that tells us which action and which button it's associated with:

```
Sub click_cmd_view_stock_value

End Sub
```

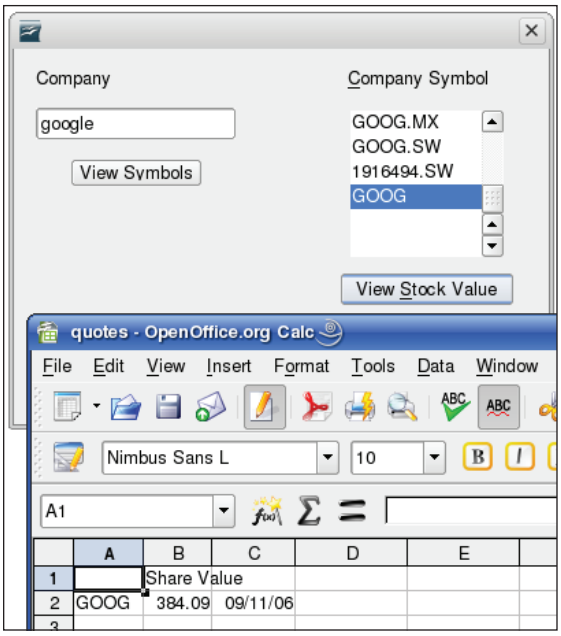
With the framework for the macro in place, we can assign it to the button (refer to the section *Assigning Actions to a Dialog*):




Then we can go back to completing the macro itself:

```
Sub click_cmd_view_stock_value
  Dim oLstCompanySymbol as Object
  oLstCompanySymbol = oFinance_dialog.getControl("lstCompanySymbol")
  get_stock_price Array(oLstCompanySymbol.getSelectedItem)
End Sub
```

You'll no doubt remember `get_stock_price` from Chapter 7, and using it in conjunction with the dialog you get:



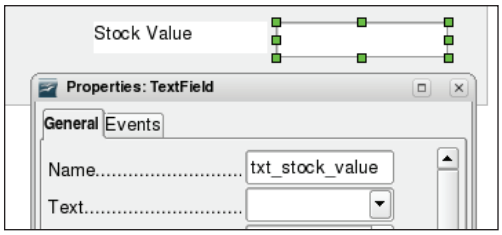

Of course, to really complete the dialog you could always place the share value back into the dialog itself.

The first thing to do is to amend the `get_stock_price` macro to change it to a function and to hide the spreadsheet:

```
Function get_stock_price (iCompany_symbols) as Double
  Dim oUrl as String
  Dim oDoc as Object
  Dim oCell as Object
  Dim oSymbols as String
  Dim oFields as String

  Dim oPropertyValue(1) As New com.sun.star.beans.PropertyValue
  oSymbols = join (iCompany_symbols,"&s=")
  oFields = "sl1d1"

  oUrl = "http://finance.yahoo.com/d/quotes.csv" _
    & "?s="& oSymbols & "&f=" & oFields &"&e=.csv"

  oPropertyValue(0).Name = "FilterOptions"
  oPropertyValue(0).Value = "44"
  oPropertyValue(1).Name  = "Hidden"
  oPropertyValue(1).Value = True

  oDoc = starDeskTop.loadComponentFromURL( _
```

```
                    oUrl, "_blank", 0, oPropertyValue)
    oCell = oDoc.Sheets(0).getCellByPosition (1,0)
    get_stock_price = oCell.Value
    oDoc.Close(True)
End Function
```

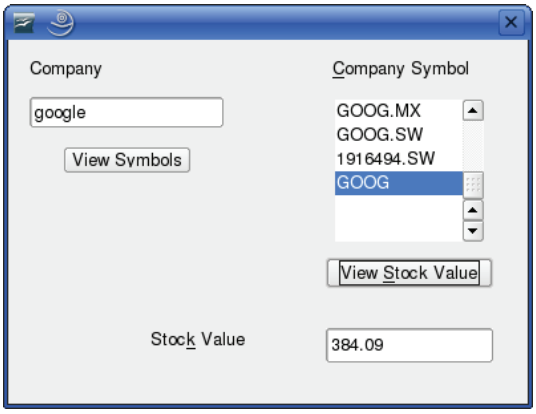
Next you'll need to put the result somewhere—for instance another TextField:




Now you'll need to change `click_cmd_view_stock_value` so that the result of the new function is written to your TextField:

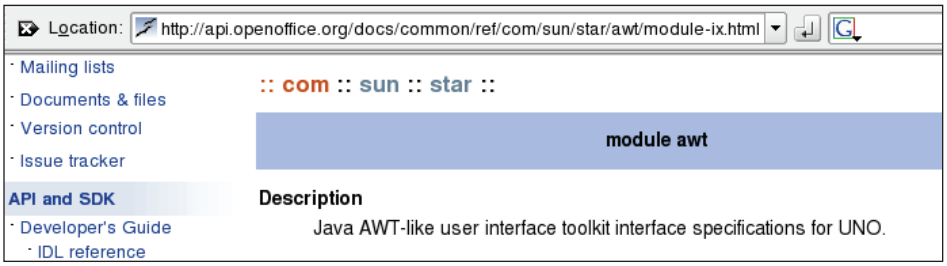
```
oTxt_stock_value.Text = _
get_stock_price(Array(oLstCompanySymbol.getSelectedItem))
```

Don't forget that you'll need use `oFinance_dialog`'s `getControl` method to access `txt_stock_value` from the macro (just as we did with the previous TextField and the ListBox). Once you've done that, then you can use the dialog with it's complete functionality:



# Finding Further Information

Although we've created a complete and fully working dialog, we haven't covered all of the controls that are available to us. In fact we haven't even covered all of the functionality of the controls that we have used. As always, you can investigate further with OpenOffice.org's online documentation:




# Summary

In this chapter we've seen how to customize OpenOffice.org's built-in dialogs: message box and input box.

Although the message and input boxes are very useful, you will probably find that they do not give you enough flexibility—you'll find that often the best results are achieved by creating your own custom dialogs. We began with creating a new dialog and adding controls from the toolbox, and then assigned macros to the controls. We are now able to obtain information from dialog and put it in a macro or we can also use information from a macro in a dialog.

"It's done" said Korora, "What do we do now?"

"Well, I don't know about you—but I'm getting out of here, and then we need to pull all of this together."

And that's what we'll be doing in Chapter 9; we'll pull everything together to produce a single application—one that you could roll out to anyone.

# 9

# Creating a Complete Application

In Chapter 8 we saw Korora opening up a spreadsheet that contained a dialog. She pressed a single button in the dialog, and this ran a series of macros that opened, processed, and closed other spreadsheets.

And we've ourselves seen that we can create our own custom dialogs, making the control of our macros much easier for anyone who isn't an expert in Calc Macros.

Let's face it: would you really trust one of your directors with the macros that you've so carefully created?

So, in Chapter 9 we'll be looking at how to create an application within OpenOffice.org Calc that anyone can use. By the end of this chapter, you'll be able to:

- Make your Calc macros available to other users within your organization
- Customize the OpenOffice.Org Calc menu to enable access to your macros and dialog boxes
- Call macros and dialog boxes direct from the command line
- Do any processing of spreadsheets in the background

In other words, we'll be creating a complete, working application.

## Making Macros and Dialogs Available to Everyone

If you throw your mind way back to Chapter 2, then you'll remember that we saw that libraries can be in one of three locations:

- My Macros and Dialogs for personal libraries
- OpenOffice.org Macros and Dialogs for global libraries
- Each individual spreadsheet

Now, if you want to distribute your macros and dialogs to other people, then the most obvious solution may seem to be to include the library in the spreadsheet. And, indeed, if your potential user doesn't have access to your network, then this may be your only solution.

However, there is a major drawback in including your library in the spreadsheet. By doing this you effectively loose control of the code. In fact, all control of the code is lost by doing this. Imagine this scenario:

1. You email the spreadsheet (plus library) to Jill.
2. Jill changes some of the code and emails the spreadsheet to Bob.
3. Bob changes some of the code and emails the spreadsheet to Jane.
4. Jane runs macros, but they give incorrect answers and cost the company £100,000.
5. Jane complains to your boss that your macros are rubbish.
6. You get the sacked.

Not really what we want to happen, is it? Instead, let's imagine this scenario:

1. You instruct Jill on how to run the centrally stored macros.
2. Jill tells you about some changes that are needed. You carry out the changes and tell Bob.
3. Bob tells you about some more changes that are needed. You carry out the changes and tell Jane.
4. Jane runs the macros, makes use of the information provided and saves the company £100,000.
5. Jane extols your value to your boss.
6. You get a pay rise.

All because the changes to the macros were properly controlled. So that, of course, brings us back to OpenOffice.org Macros and Dialogs.

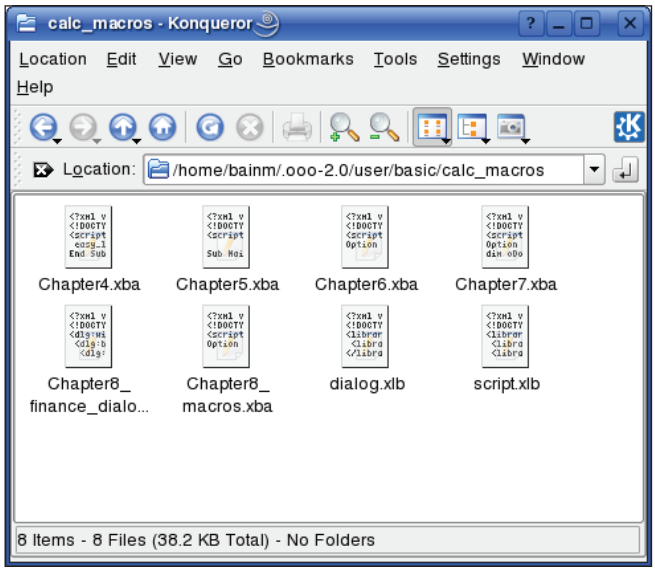
Let us just remind ourselves how to go about moving a library from My Macros and Dialogs to OpenOffice.org Macros and Dialogs.

# Creating a Global Library

Hopefully, you'll remember from Chapter 2 that a library is just a directory, a directory that contains a set of files:

- `script.xlb`: An index of all of your macro modules
- one or more `xba` files: One for each macro module
- `dialog.xlb`: An index of all of your dialog boxes
- one or more `xdl` files: One for each dialog box that you've got.

So, if you were to look into a library directory, then you'd see something like this:




Stage one of creating a global library is to move this directory into your system's OpenOffice.org Macros and Dialogs area. If you remember, this will depend on your operating system (Linux or Windows) and the particular distribution/version that you're using.

We learned in Chapter 2 that with that done you just need to update your user's `script.xlc` file (remember its location will depend on your particular system). Since we're also dealing with dialog boxes now we'll also have to update the `dialog.xlc` file. And just to remind you, you should add a line to each file. To `script.xlc` you need to add something like (but remember to make a copy before you edit the file):

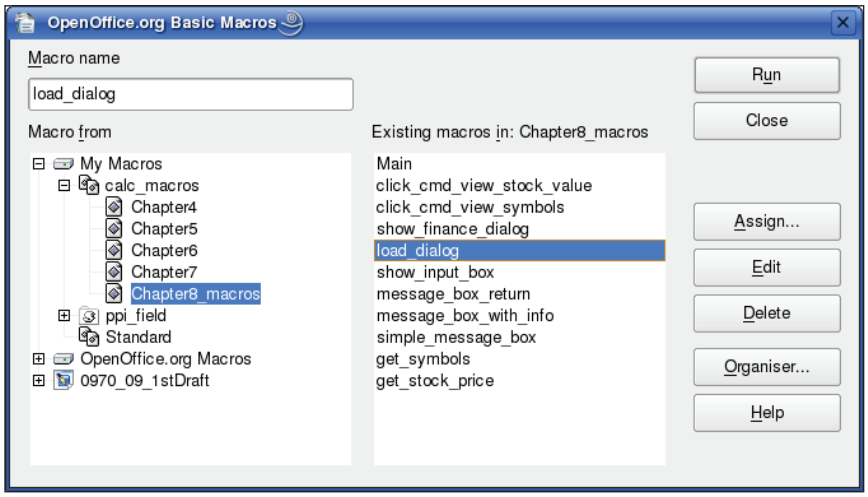
```
<library:library library:name="calc_macros"
xlink:href="$(INST)/share/basic/calc_macros/script.xlb/"
xlink:type="simple" library:link="true" library:readonly="false"/>
```

To `dialog.xlc` you need to add something like:

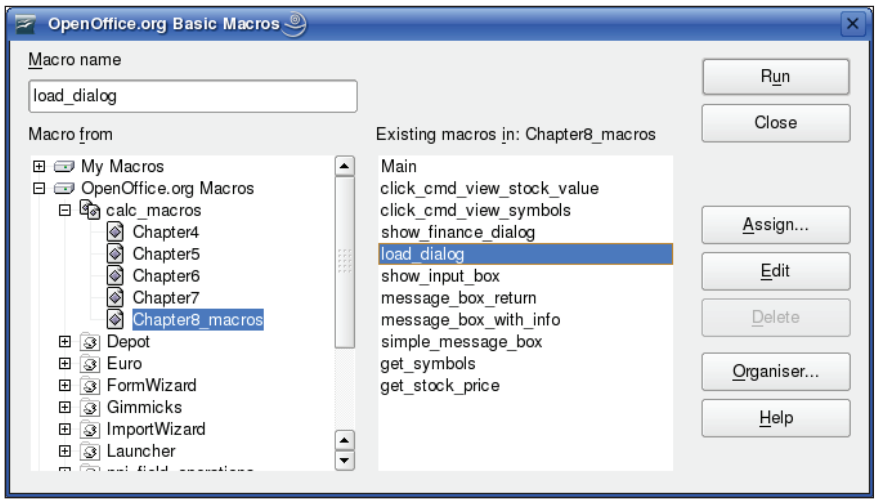
```
<library:library library:name="calc_macros"
xlink:href="$(INST)/share/basic/calc_macros/dialog.xlb/"
xlink:type="simple" library:link="true" library:readonly="false"/>
```

Whether you do this manually or whether you write a script to do it depends on the number of users that you've got and, of course, your personal preferences.

The end result? You'll have changed this:




To this:



More importantly, any user whose `.xlc` files you update will have access to your global macros and dialog boxes—and don't forget that they won't be able to see the macros and dialog boxes until you've updated their `.xlc` files.

# Using a Global Library to Automate OOo Calc

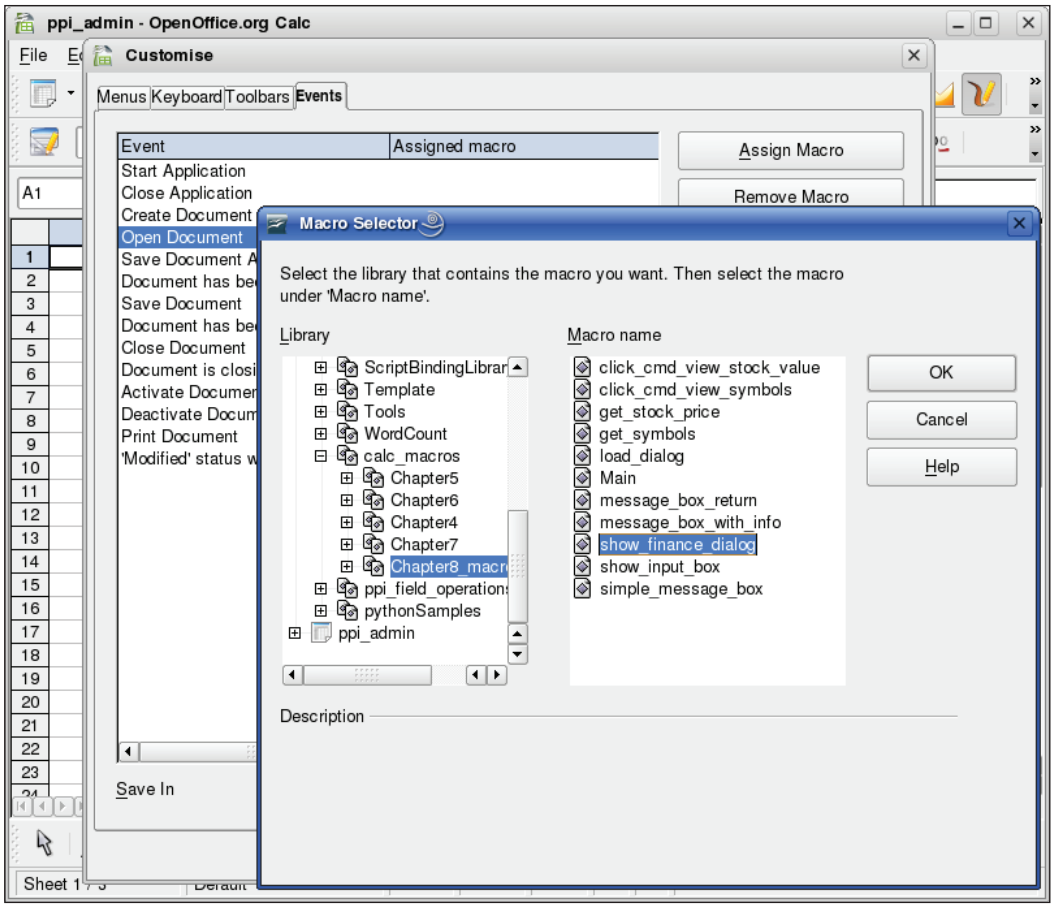
In every example that we've looked at so far, we've run the macros from one of two places:

- The Basic Editor by pressing the Run button
- OOo Calc by clicking **Tools | Run Macro...**

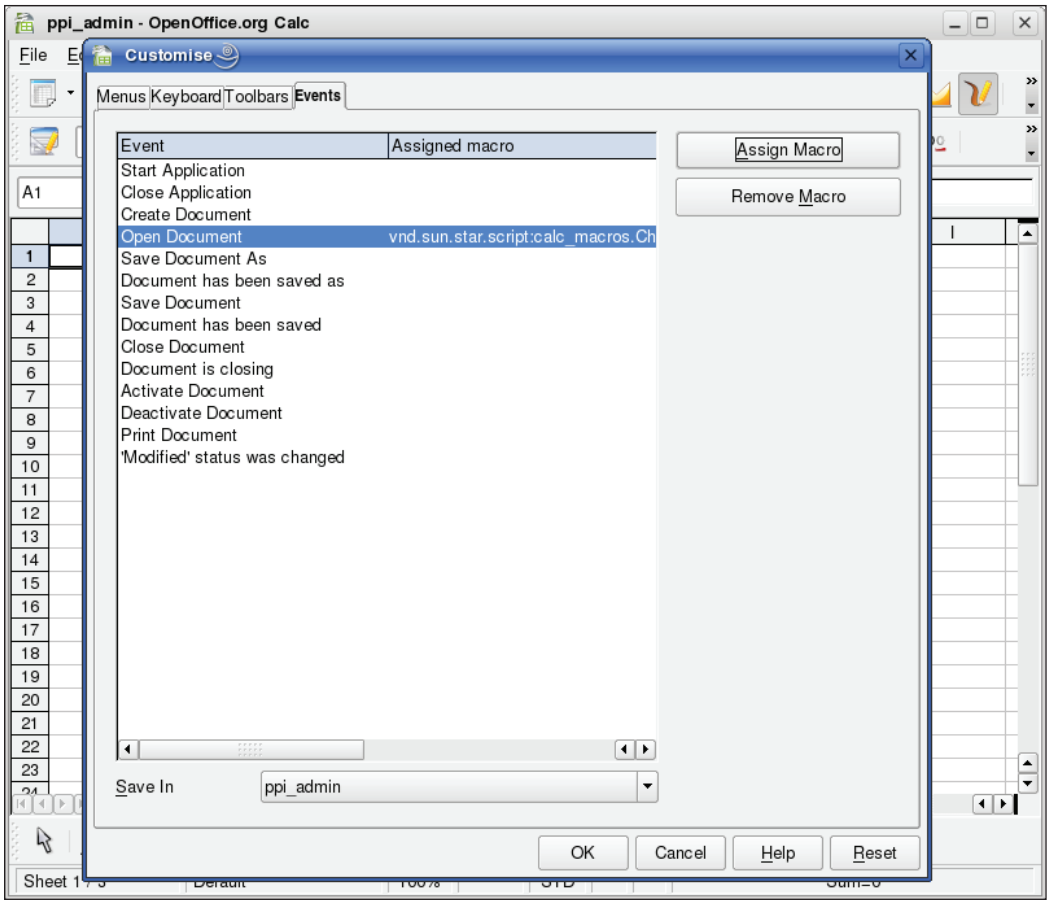
This is useful, saving you a lot of time by manipulating any spreadsheets for you. However, we can still make life easier by running macros automatically.

# Running Macros Automatically when Calc Opens

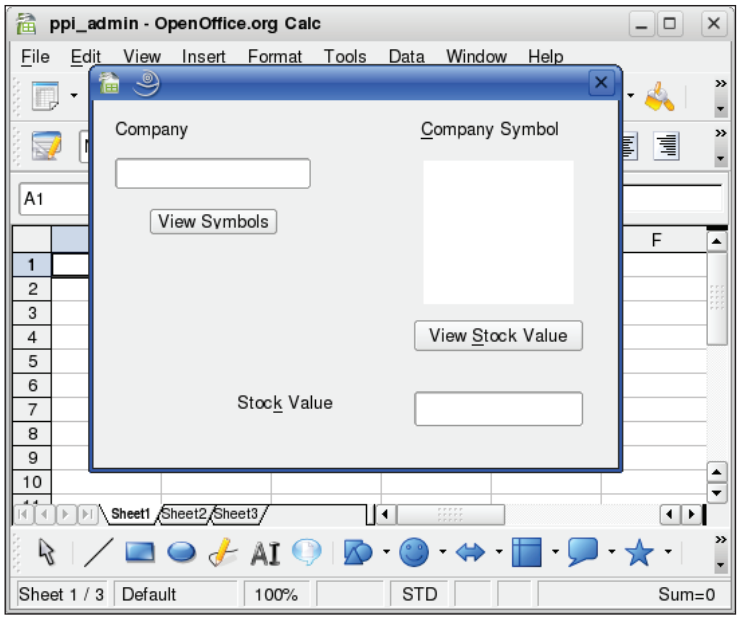
We've already seen that we can assign macros to events such as using a button on a dialog, so it shouldn't come as any surprise to find that we can also assign macros to events such as the opening of a document. Just open your Calc document and click on **Tools | Customize...**, then go to the **Events** tab and click on **Assign Macro**:




Once you've selected which macro you want to run when you open the document, then you need to use the **OK** button in the **Customize** dialog:




What next? After you click **OK** nothing obvious happens, you just return to the Calc spreadsheet. That is, of course, because the macro only runs when you open the document. So, close the spreadsheet, and then open it up again:




Now your users don't even need to know how to run a macro, all they have to do is open the spreadsheet and the macro will run itself, assuming of course that:

- Each user's `dialog.xlc` and `script.xlc` files have been amended to include any required global libraries.
- Each user (on Linux) has read access to the Calc spreadsheet that you've set to run the macro.

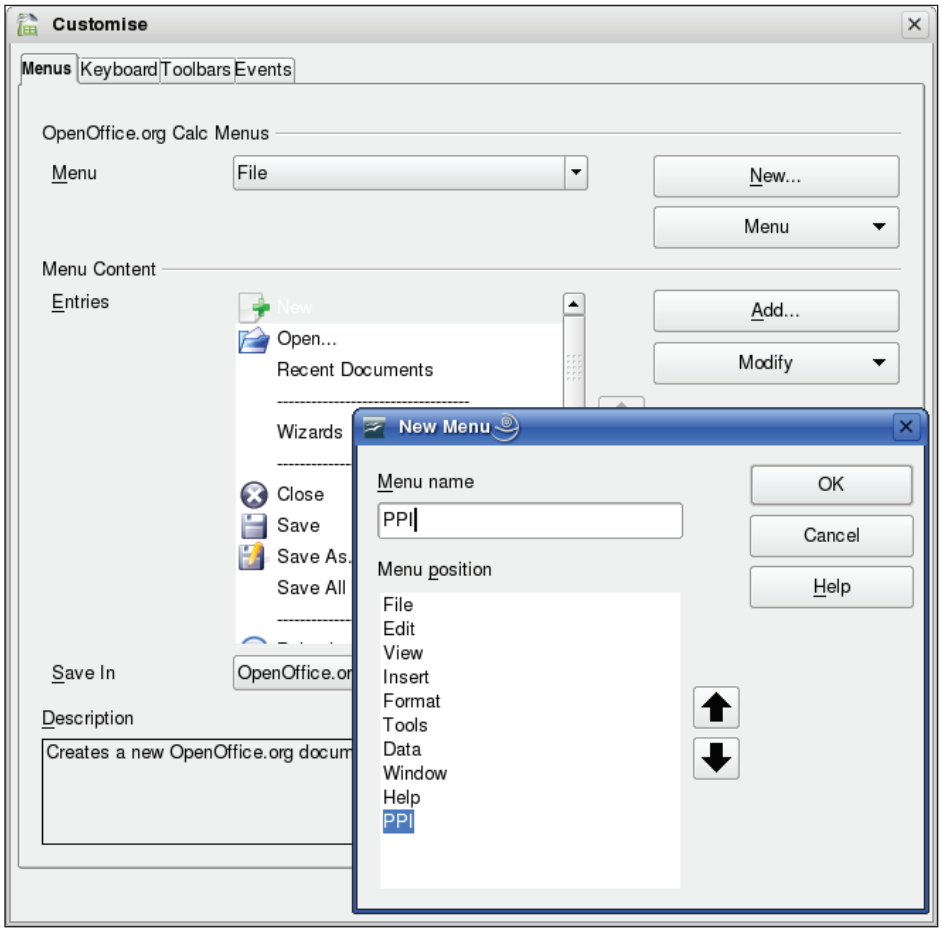
# Adding Macros to the OpenOffice.org Calc Menu

You may, of course, decide that you prefer to provide an easy way of running macros. If that's the case, then you might consider adding the macros to OpenOffice.org Calc menus.

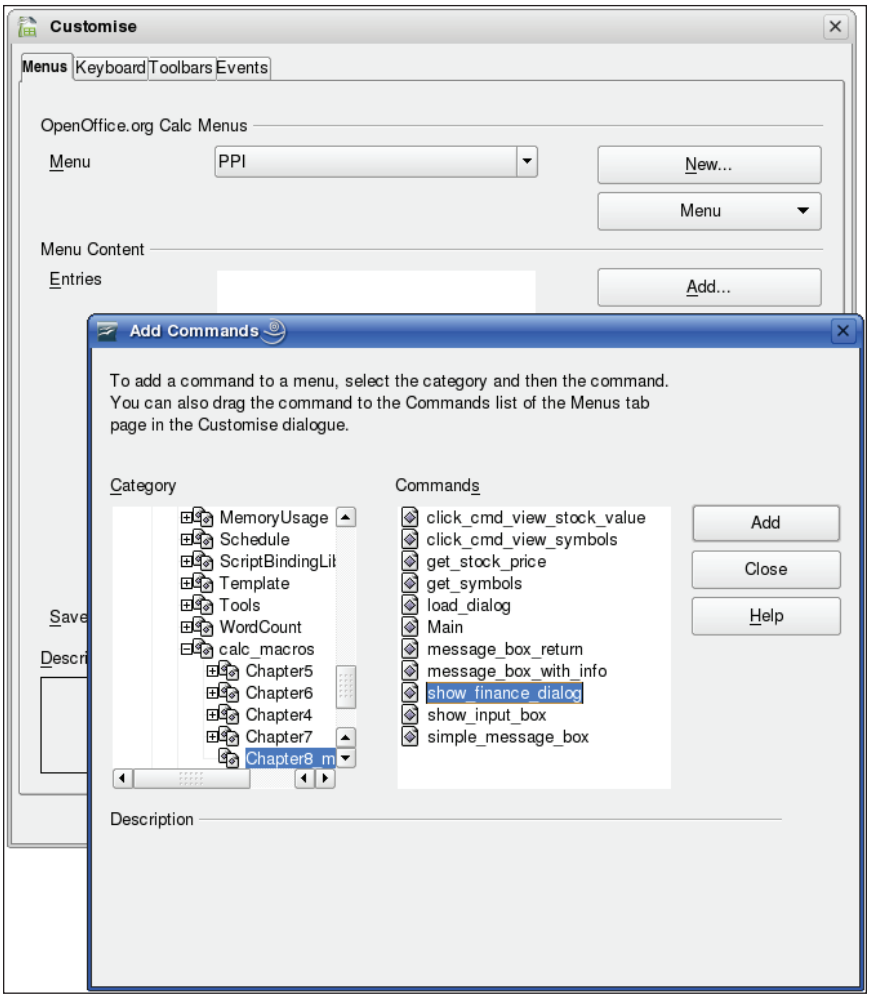
## Adding a Macro to the Menu Manually

We've seen that a macro can be run as soon as you open a spreadsheet, but that's only good for a single macro. However, if you want control of a number of macros, then you can add the macro to one of OpenOffice.org Calc's menus. Even better you can add a completely new menu.

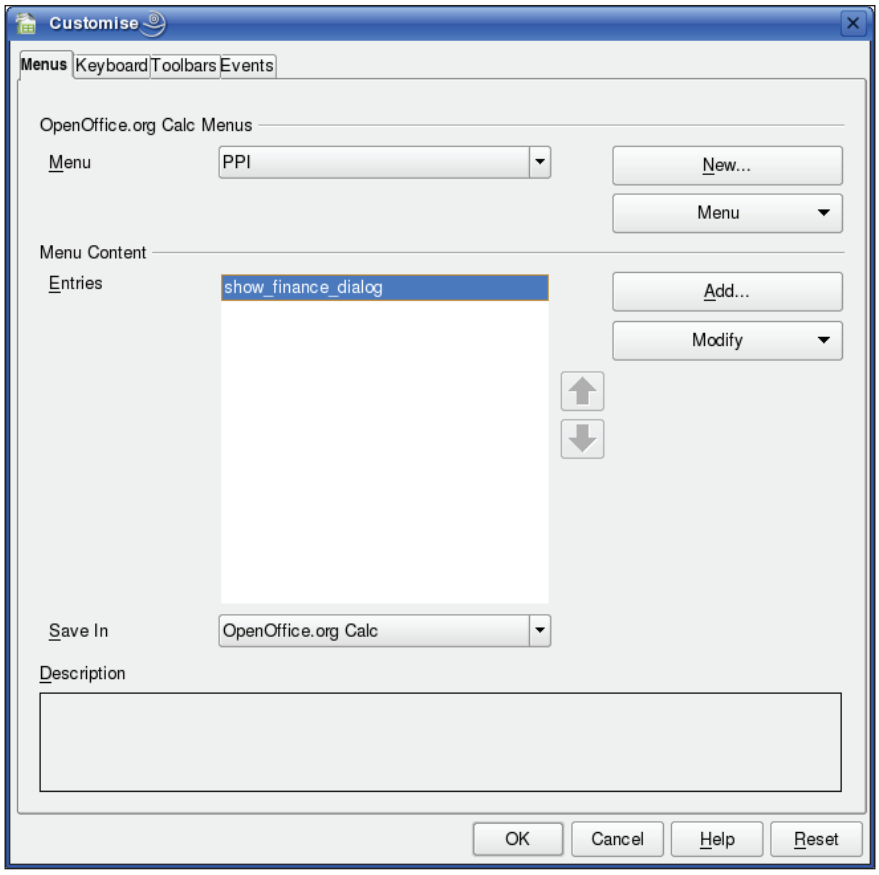
Again you need to click on **Tools | Customize...**; however, this time you'll need the **Menus** tab, then use the **New..**; button to open the **New Menu** dialog :



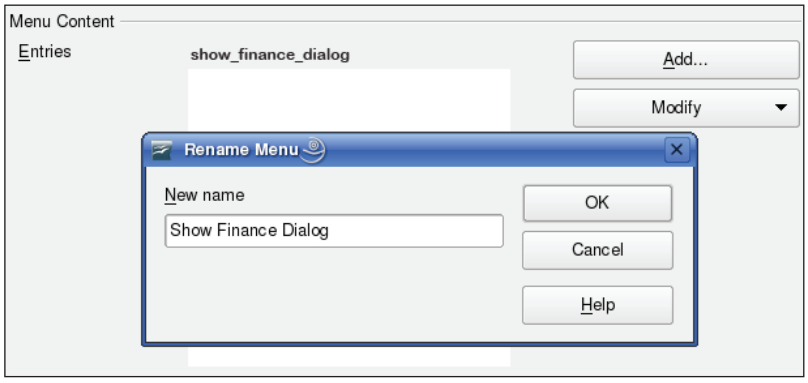

Once you've created your new menu (by pressing **OK**), you can start adding your macros to it. You'll need to make sure that the new menu is shown in the **Menu** combo box, and then you can click on **Add**. You can then add macros by using the **Add Commands** dialog:



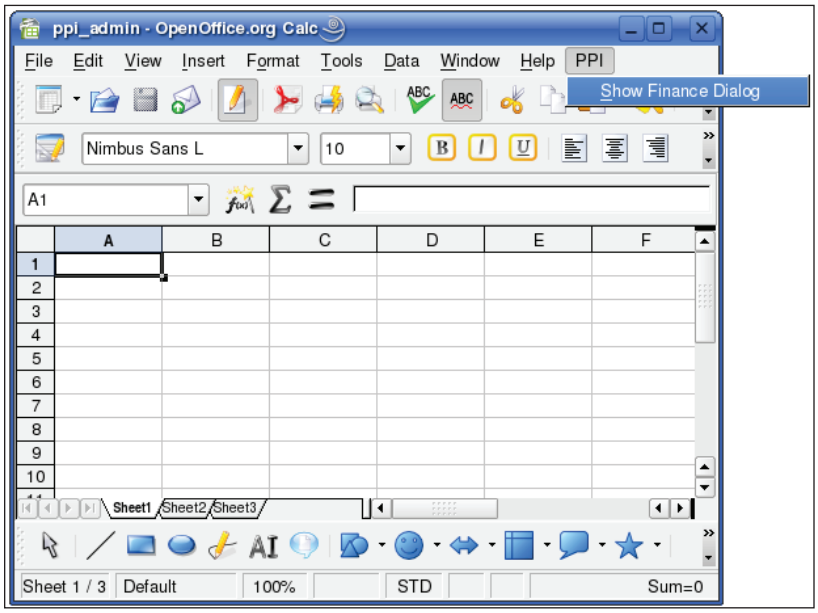

However, you'll notice that the element added to the menu has the same name as the macro. So in my example the menu will display `show_finance_dialog`. Obviously this doesn't look particularly nice (or professional for that matter):




Changing this is simple—just select the menu entry from the list and then click on **Modify**. You can then change the name to something more meaningful (or maybe just better formatted):



When you've finished you'll have a custom menu from which you can call your macros:




You will find, however, that there is a disadvantage with managing your macros in this way. You'll have to follow this process for everyone else who also wants to have a similar menu. Fine for a small number, but not really suitable for a larger organization.

# Distributing a Menu

Having created (and, of course, tested) your menu, you may want to make it available to other people in your organization. We've seen that it is easy to add custom menus from which we can run macros. So an obvious answer is to go to each person that you want to be able to use your menus and add the menus manually; but it is a bit time consuming.

If that seems a bit long winded, then a second option may be to write a macro to do the job for us. We've already learned how to run a macro on opening a spreadsheet. This macro could check to see if the custom menu already exists, if it doesn't, then the macro could add it.

However, there is a still simpler solution (and that's the kind that I like).

When you add your custom menu, you might expect it to be saved in some complex, unintelligible format. Well, you'd be wrong. The information is stored in a simple XML file. All you have to do is find the file.

If you're using Linux then look in:

```
~/.ooo-2.0/user/config/soffice.cfg/modules/scalc/menubar
```

If you're using Windows, then you'll need to look in someplace like:

```
C:\Documents and Settings\<username>\Application Data\ OpenOffice.
org2\user\config\soffice.cfg\modules\swriter\menubar
```

In both cases you'll find a file called `menubar.xml`. If you scroll through this file, then towards the end you'll find something like:

```
<menu:menu menu:id="vnd.openoffice.org:CustomMenu1" menu:label="PPI">
 <menu:menupopup>
  <menu:menuitem
   menu:id=
     "vnd.sun.star.script:calc_macros.Chapter8_macros.
     show_finance_dialog?language=Basic&amp;location=application"
   menu:helpid=
     "vnd.sun.star.script:calc_macros.Chapter8_macros.
     show_finance_dialog?language=Basic&amp;location=application"
   menu:label="Show Finance Dialog"/>
 </menu:menupopup>
</menu:menu>
```

Having got to this point, you'll realize that we can now do a few things:

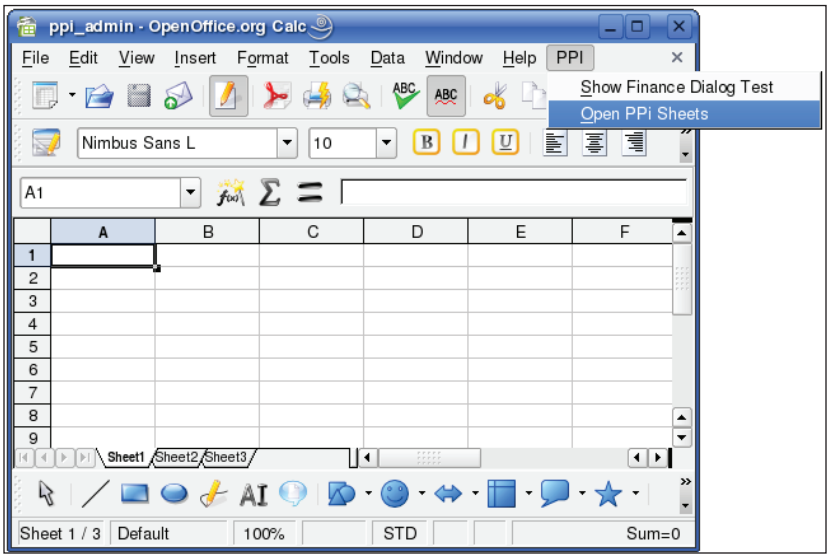
- A `menubar.xml` containing your custom menus can be copied into anyone's menubar folder.
- On Linux you can create a link to a central `menubar.xml` file.

It also means that you can bypass the manual menu creation by editing this file directly. For instance you could change the above code to:

```
<menu:menu menu:id="vnd.openoffice.org:CustomMenu1" menu:label="PPI">
  <menu:menupopup>
   <menu:menuitem
    menu:id=
"vnd.sun.star.script:calc_macros.Chapter8_macros.show_finance_dialog?
    language=Basic&amp;location=application"
    menu:helpid=
"vnd.sun.star.script:calc_macros.Chapter8_macros.show_finance_dialog?
    language=Basic&amp;location=application"
```

```
        menu:label="Show Finance Dialog"/>
      <menu:menuitem
        menu:id=
"vnd.sun.star.script:calc_macros.Chapter6.Main?
        language=Basic&amp;location=application"
        menu:helpid=
"vnd.sun.star.script:calc_macros.Chapter6.Main?
        language=Basic&amp;location=application"
        menu:label="Open PPi Sheets"/>
    </menu:menupopup>
  </menu:menu>
```

If you restart OpenOffice.org Calc, then you'll find that the custom menu has changed:




Now you're able to provide a complete application for yourself and for any other users of your system. Your macros can be run without the user having to have any knowledge of the background works of OpenOffice.org Calc.

# Keeping It All Hidden

In Chapter 8 we saw some macros that hid spreadsheets in the background while they were being processed. I'm sure that you'll agree that this is incredibly useful. It means that you (and more importantly any of your users) don't get to see the spreadsheet opening and closing while the macro does it work. What we want to do is to be able to get the macro to do its processing invisibly, just leaving us with the updated spreadsheet at the end. And so we'll have look at this in a bit more depth.

You'll remember that we load a document using the `loadComponentFromURL` function. You may also have noticed that we use an empty array for the last argument:

```
openSpreadSheet = oDesk.loadComponentFromURL _
  (oUrl, "_blank", 0, Array())
```

This empty array is actually a placeholder for any parameters that we want to pass, and which govern how the spreadsheet is opened. In this case we're going to set the `Hidden` property to `True` so that nothing is actually displayed, and everything is done in the background:

```
Function openSpreadSheet (oFile as String, _
    Optional oInvisible as boolean) as Object
  Dim oUrl as String
  Dim oPropertyValue As New com.sun.star.beans.PropertyValue
  If fileExists (oFile) Then
    oUrl = convertToUrl (oFile)
  Else
    oUrl = "private:factory/scalc"
  End If

  If not IsMissing(oInvisible) and oInvisible = true Then
    oPropertyValue.Name = "Hidden"
    oPropertyValue.Value = true
  End If

  openSpreadSheet = starDeskTop.loadComponentFromURL _
    (oUrl, "_blank", 0, Array(oPropertyValue ))
End Function
```

All you have to do it to amend any macros that call the modified function:

```
oDoc1 = openSpreadSheet(oFile1, true)
```

If you now call the macro from the command line (which we learn to do in the following section, *Running Macros from the Command Line*), then nothing will happen—visibly anyway. However, if you view your spreadsheet, then you'll find that it has actually been updated. For example:

```
bluek@aeneas:~> ls -l ~/ppi
-rw-r--r-- 1 bluek users 6077 2006-07-27 14:24 ppi_current.ods
bluek@aeneas:~> oocalc "macro:///ppi_general.ppi_accounts.main"
bluek@aeneas:~> ls -l ~/ppi
-rw-r--r-- 1 bluek users 6076 2006-07-27 14:36 ppi_current.ods
```

# Running Macros from the Command Line

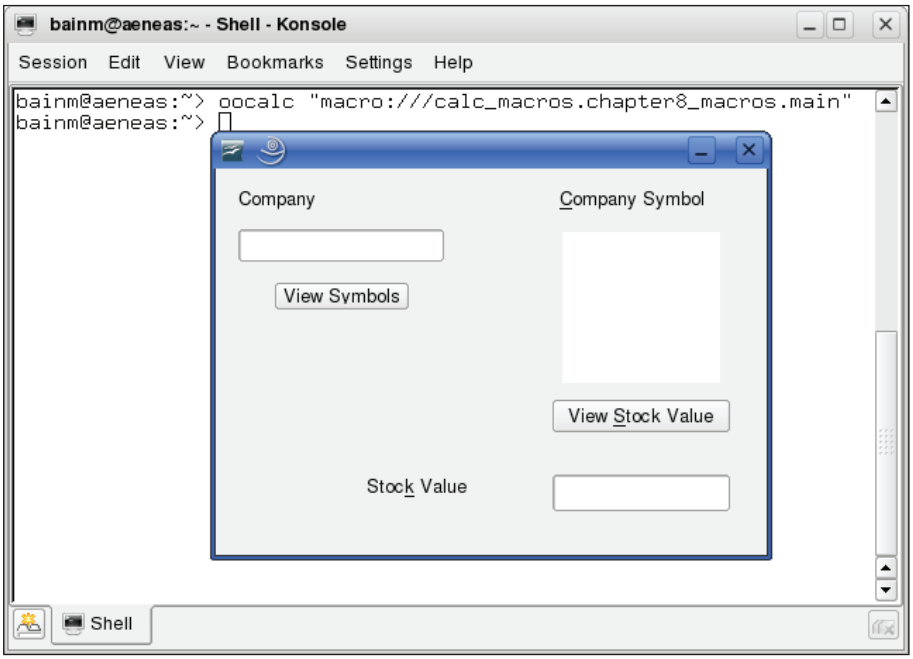
We've seen that we can hide a spreadsheet; and at this point it may occur to you that it may be useful to bypass Calc altogether to run your macros. In other words, you might want to run your macros directly from the command line.

## Running Macros in Linux

If you're using Linux, then you can run the `main` subroutine in module admin in library `ppi` by typing:

```
oocalc "macro:///ppi.admin.main"
```

When your run the macro, you should see exactly the same output as when you run the macro from within Calc.




## Running Macros in MS Windows

If you're using Windows, then you'll have to find out where OOo is installed, but the command should look something like:

```
C:"Program Files"\"OpenOffice.Org2"\program\scalc
macro:///ppi.admin.main
```

# Creating Background or Batch Processes

Now that we can call macros from the command line, and we can carry out the processing without any display being shown, we can start thinking about running OpenOffice.org Calc macros automatically. This means that you won't even have to do anything yourself (once the processes are running), you just need to leave your PC on and doing all of the work itself.

# Running Background Processes on Linux

If you're using Linux, then you're going to be interested in the commands `at` and `crontab`:

- `at`: This carries out a task at a predefined time,
- `crontab`: This carries out tasks at regular times: every day at 3:00PM, the first Monday of every month, etc.

Let's say, for example, that you've had a grueling day (a bit like Pygoscelis) and you're desperate for sleep, but you need that report for first thing tomorrow morning. Don't worry, just use `at` and then head off to bed:

```
ellsworthyp@aeneas:~> at 8:00 tomorrow
warning: commands will be executed using /bin/sh
at> ellsworthyp@aeneas:~> at 8:00 tomorrow
warning: commands will be executed using /bin/sh
at> oocalc "macro:///ppi_general.ppi_accounts.main"
at> <EOT>
job 8 at 2006-07-28 08:00
```

The `<EOT>` is used to tell `at` that you've entered all the commands that you need and is generated by entering *Ctrl+D*.

If you want to see what is due to run, then use the command `at -l`:

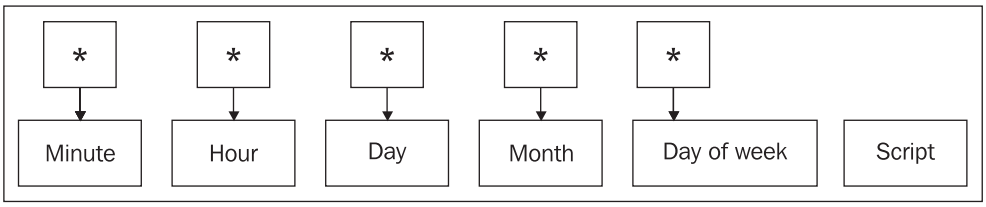
```
bluek@aeneas:~> at -l
10      2006-07-31 08:00 a bluek
```

After all that, if you decide that you don't want the command to run, then you can remove it from the queue:

```
bluek@aeneas:~> at -r 10
```

Where `10` is, of course, the job number.

And what if you need this report running every morning? Fine, just use `crontab`. And the hardest thing about `crontab` is remembering the order of the fields that it requires:




So to run the script at 8:00 every morning you'll need to type:

```
 ellsworthyp@aeneas:~> crontab -e
0 8 * * * oocalc "macro:///ppi_general.ppi_accounts.main"
```

This runs at minute zero, 8 AM, every day of the month, every month, and every day of the week. The star (*) is a placeholder and means 'every', i.e. every day, every month, etc.

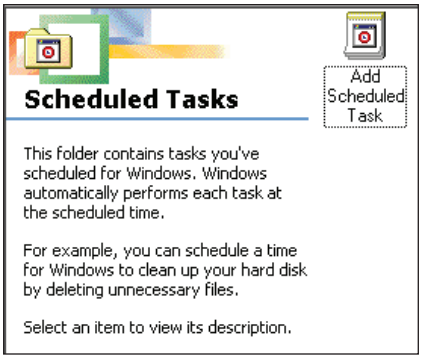
Just like `at`, you can see what's due to run:

```
bluek@aeneas:~> crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.XXXXgdXvC5 installed on Fri Jul 28 10:45:44 2006)
# (Cron version V5.0 -- $Id: crontab.c,v 1.12 2004/01/23 18:56:42
vixie Exp $)
0 8 * * * oocalc "macro:///ppi_general.ppi_accounts.main"
```

And for removing a job (once you don't need it any more), you'll have to used `crontab -e` again, and remove the entry manually.

# Running Background Processes on Windows

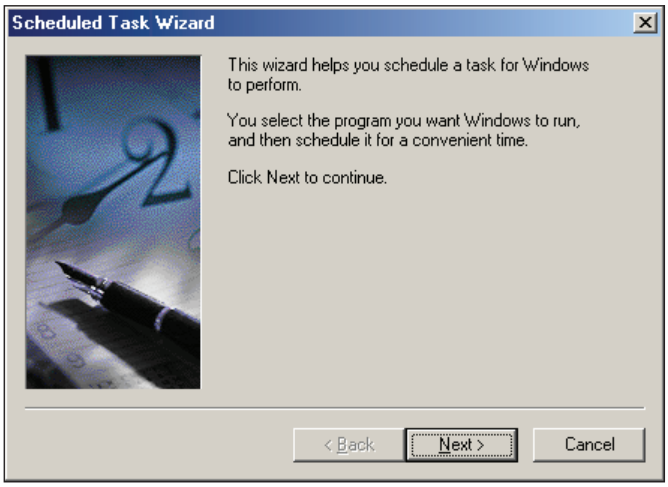
If you want to set up a background process and you're using Windows, then there's a wizard for you to use. You'll need to go to the **Control panel** and then **Scheduled Tasks**:



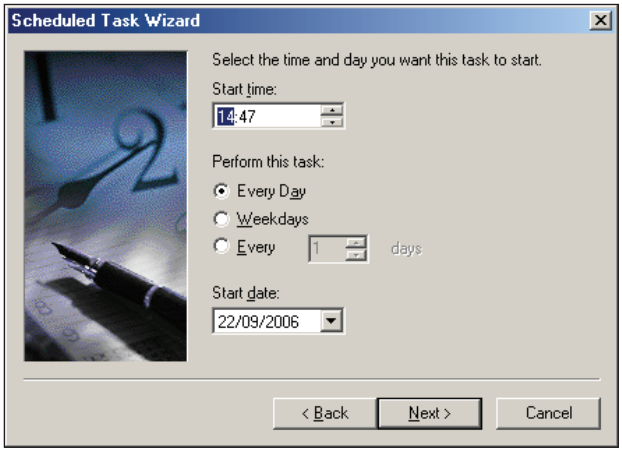
You'll find that the folder contains:

- An icon for every scheduled task that you create; clicking on these allows you to modify details such as the time to run the task
- An icon entitled **Add Scheduled Task**

If you click on the **Add Scheduled Task**, then this will start the wizard:




All you have to do is to enter the appropriate details as needed:



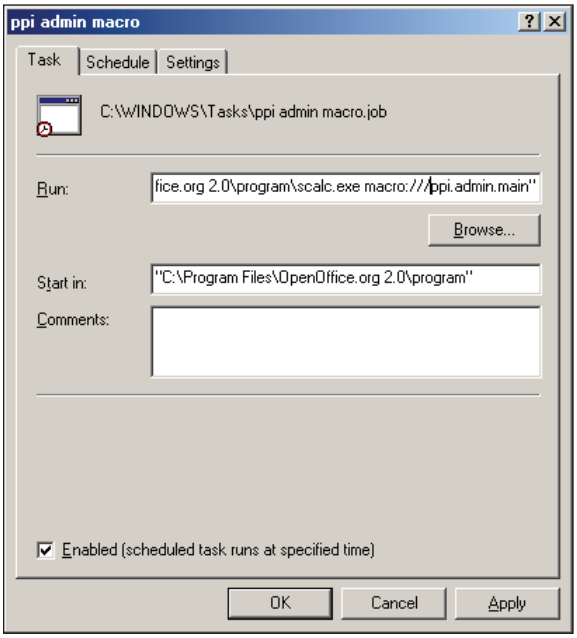

If you remember, the Windows command for running a macro directly is:

```
C:"Program Files"\"OpenOffice.Org2"\program\scalc
macro:///ppi.admin.main
```

However, the wizard will only allow you to select the `scalc` application itself, without the macro details. The answer is quite simple:

- Select `scalc` as the application to use and then carry on through the wizard until you've entered all of the information that is required.
- Once you've finished with the wizard, click on the new icon that will have been created for you.

Now you can enter the details for the macro that you want the scheduler to run:




Having created a useful application, we'll have a quick look at one of the things that can turn a good application into a great one.

# Sending Emails

No self-respecting application can really do without the ability to email a file:

```
Sub send_email
  Dim eMailAddress as String
  Dim eSubject as String
  Dim eMailer as Object
  Dim eMailClient as Object
  Dim eMessage as Object
```

```
    eMailAddress = "mark.bain@linuxtalk.co.uk"
    eSubject = "Test email"


    eMailer = createUnoService("com.sun.star.system.SimpleCommandMail")

    eMailClient = eMailer.querySimpleMailClient()

    eMessage = eMailClient.createSimpleMailMessage()

    eMessage.setRecipient( eMailAddress )
    eMessage.setSubject( eSubject )
    eMessage.setAttachement _
      (Array(convertToUrl("/home/bainm/bluek/ppi_current.ods")))
    eMailClient.sendSimpleMailMessage ( eMessage, _
      com.sun.star.system.SimpleMailClientFlags.NO_USER_INTERFACE )
End Sub
```

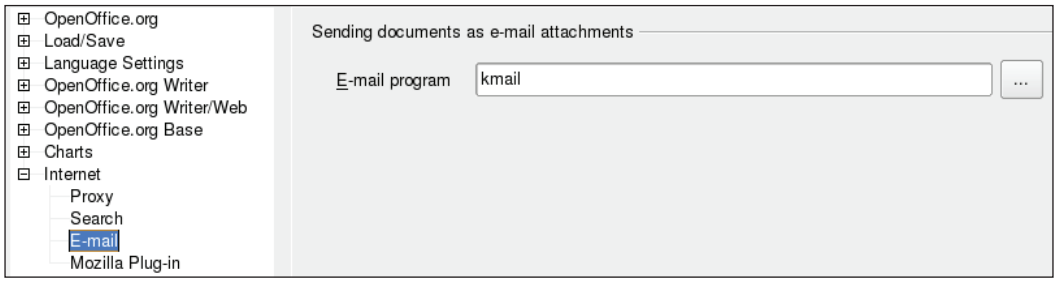
Of course, you realize that this macro won't work in Windows. Fortunately you just need to change one line:

```
    eMailer = createUnoService("com.sun.star.system.SimpleCommandMail")
```

The above line needs to be changed to:

```
    eMailer = createUnoService( "com.sun.star.system.SimpleSystemMail" )
```

Oh, and one other thing, if you're using Linux, then you'll need to set the default email client. Just click on **Tools | Options... | Internet | Email** and tell OpenOffice.org which email application you're using:



# Summary

In Chapter 9 we've looked at how to turn you're individual macros into complete applications that you can even distribute to other people. A key step in creating a complete application that others can use is to make both your macros and your dialogs global.

Rather than running macros manually, we can now assign then to events such as opening a document. We can also assign macros to menus, and distribute these menus. We can run macros from the command line and automate running macros using `at` and `crontab`. Finally, we also learned how to use macros to send emails.

Korora pushed the door open and the rising sun blazed in blinding them both. As their eyes became accustomed to the bright light, they became aware of figures in the car park, one of them sprawled on the tarmac.

"Well, Py, you look a bit worse for wear."

Pygoscelis squinted at the policeman leaning against a squad car.

"Hi Rocky. I take it that you got my note then."

"Yeah, nice piece of work that, not like this one. Take him away."

The spread-eagled body of Sphen was hoisted up and shoved into one of the black and whites.

Korora looked at her boss "I'm sure that you can see into the future."

And that's what we're going to do in our final chapter. We're going to have a look at the future of OpenOffice.org Calc.

# 10
# Using Excel VBA

> "One of the biggest barriers to OpenOffice adoption is lack of macro interoperability. In the enterprise space generally the most mission-critical macros exist in Excel spreadsheets. To eliminate or lower this barrier, clearly will ease adoption of OpenOffice.
>
> MS's marketing would have you believe that OO.o is only useful for a few staff, for taking basic notes. One of [Novell]'s clients using OO.o on their bank, told us "There's only a tiny minority of people who can't use OO.o". And when pressed on that tiny minority — the sticking point was Excel macros, and having re-tested with Novell's VBA macro support, even that tiny minority looks like disappearing."
>
> *Email from Noel Power, OpenOffice Developer, Novell, June 2006*

In all of the chapters so far, we've seen what you can do with a current version of OpenOffice.org. However, in this chapter we're going to have a look into the future of OpenOffice.org Calc, and we're going to see how to bring that future onto your own desktop. In that future is OpenOffice.org's Excel VBA support.

By the end of this chapter, you will:

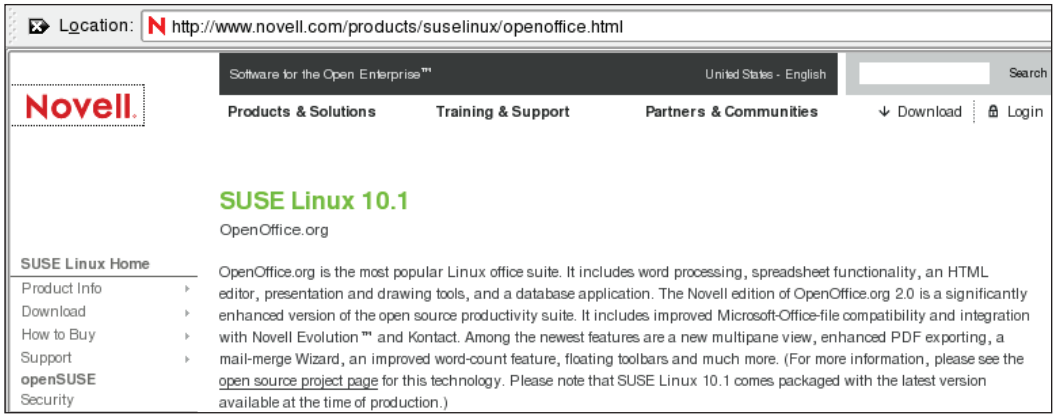
- Understand the requirements for making use of OpenOffice.org's Excel VBA support
- Be able to import an Excel spreadsheet and make use of any macros contained in the spreadsheet
- Be able to create your own macros using the Excel VBA language structure

# The Current State

You'll be pleased to know that you're at the cutting edge of technology here—right at the edge—and that makes the picture a little variable.

Why is the picture variable? Because, of course, OpenOffice.org is Open Source, which means that anyone (even you) can obtain the source code and make their own changes (and hopefully improvements).

And that's exactly what Novell has done for its SUSE Linux 10.1 product:




At this point you're probably thinking that this is all very interesting, but also wondering how it helps you exactly.

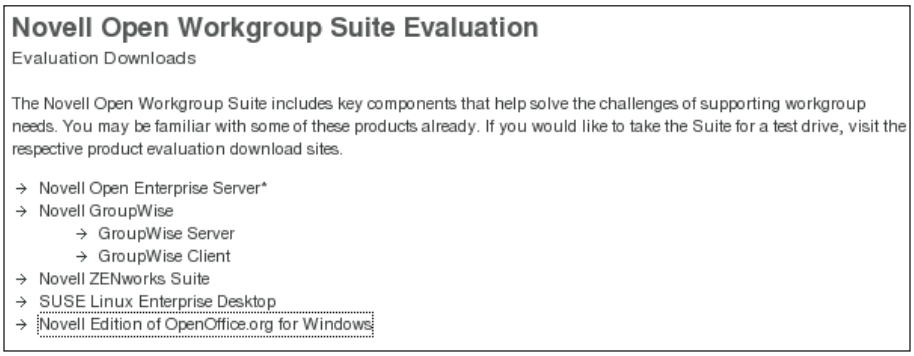
# OpenOffice.org's Excel VBA Support under MS Windows

Unfortunately, as I'm writing this, Excel VBA support is not yet available in the Windows version of OpenOffice.org. However, to quote Noel Power, OpenOffice.org developer at Novell:

"This year at OOoCon I had some frank discussions with some of the Sun developers and there at least seems to be some willingness to align their solution and ours. We hope to increase the pace of our upstreaming efforts and aim to have the initial effort completed in the next couple of months."

So, by the time you read this there is every change that you should be able to download a fully released version of OpenOffice.org with VBA support. If that's the case, then all you have to do is install the latest version from the OpenOffice.org website.

However, if that's not the case, then you can still download Novell's version:




# OpenOffice.org's Excel VBA Support under Linux

If you're a Linux user, then you'll find that the situation is very similar to that of a MS Windows user. So, it means that you can't just go to the OpenOffice.org website and download the installation files.

Just like Windows, the current mainstream Linux version of OpenOffice.org doesn't include Excel VBA support. Fortunately that's not the end of the story.

As we've already found out, the OpenOffice.org version with Excel VBA support has been produced by Novell, and Novell includes this in its SUSE Linux 10.1 product. Therefore, if (like me) you already use SUSE Linux 10.1, then you can just jump straight on to the section *Importing an Excel Spreadsheet containing Macros*. If not then you may still have some work to do. However, some Linux distros like Red Hat, Debian, Madriva, Gentoo, Arl Linux, DroplineGNOME, Frugalware, QiLinux, and Ubuntu, already make use of the Novell code:

So, what if your distro doesn't use the Novell code? How do you go about being able to use VBA support?

You've got four choices:

1. Migrate to a version of Linux that supports VBA. If you do this, then the correct version of OpenOffice.org automatically comes with it. If you're new to Linux or want to migrate, then read the section *Installing SUSE Linux 10.1*.
2. If you don't want to migrate and you're confident in working with Linux, then download Novell's source code and build it yourself. This is outlined in the section *Building OpenOffice.org from Source*.

3. Try to convince the producers of your Linux distro to incorporate Novell's version of OpenOffice.org into their own.
4. Like Windows users try to convince Sun to incorporate Novell's version of OpenOffice.org into the mainstream version and jump on to the section *Support your Local OpenOffice.org Issue*.

There are, of course, pros and cons to each of the solutions:

- Installing another version of Linux may be the most immediate and straightforward choice, but if you've already invested a large amount of time installing a particular brand of Linux on one or more computers, then you may not be too happy to start it all from scratch again.
- Building OpenOffice.org from source is a good option if you have experience in doing this type of thing before, or if you're wanting to learn more about Linux. Don't forget that this version has been built and tested on SUSE Linux. There's no guarantee that it will automatically work with yours.
- Convincing the owners of a distro (or for that matter, Sun) to incorporate the Novell code may be a bit of an uphill struggle, although things are progressing well at the moment.

Anyway, we'll now have a look at the different options in a little more depth.

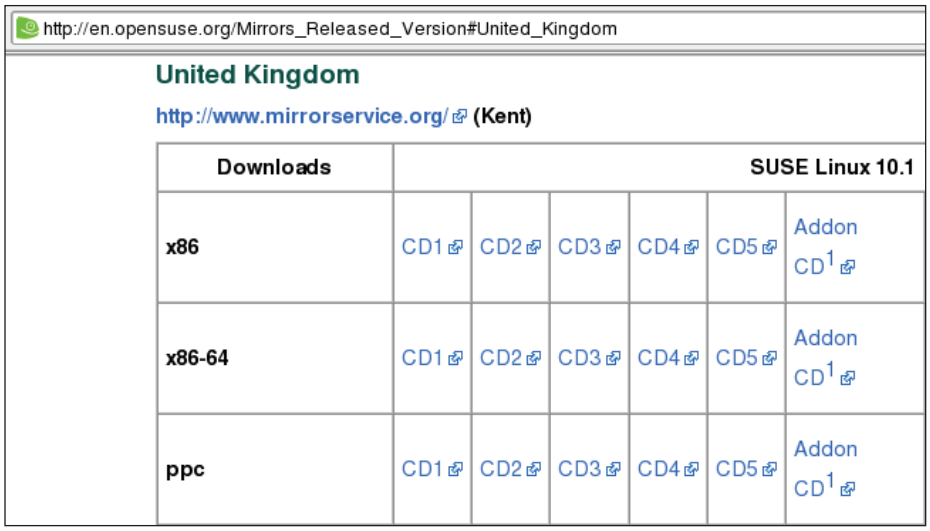
# Installing SUSE Linux 10.1

Now, if you're not committed to any particular Linux distribution, or if you want to migrate from Windows to Linux, then by far the easiest thing for you to do is to install a Linux version that contains OpenOffice.org with VBA support.

You'll find that installing Linux is very easy and will not take a great amount of time (whichever one you choose). Some of the distros even have a 'Live disk' available, meaning that you're able to test out Linux without having to fully install it on your PC.

However, since we know that SUSE Linux 10.1 definitely supports Excel VBA, that's the one we'll look at for now.

SUSE Linux 10.1 is a mature and stable rendition of Linux, and so you'll find installation very straightforward. First go to the SUSE download site and select the mirror site nearest your location:

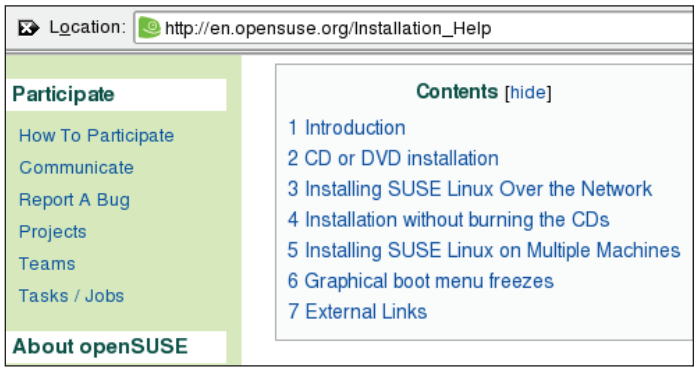

http://en.opensuse.org/Mirrors_Released_Version#United_Kingdom

**United Kingdom**

http://www.mirrorservice.org/ **(Kent)**

| Downloads | SUSE Linux 10.1 | | | | | |
|---|---|---|---|---|---|---|
| **x86** | CD1 | CD2 | CD3 | CD4 | CD5 | Addon CD1 |
| **x86-64** | CD1 | CD2 | CD3 | CD4 | CD5 | Addon CD1 |
| **ppc** | CD1 | CD2 | CD3 | CD4 | CD5 | Addon CD1 |

Having downloaded your installation CDs, you can then just follow the online installation instructions and have your SUSE Linux installed:


Location: http://en.opensuse.org/Installation_Help

**Participate**

How To Participate
Communicate
Report A Bug
Projects
Teams
Tasks / Jobs

**About openSUSE**

**Contents** [hide]

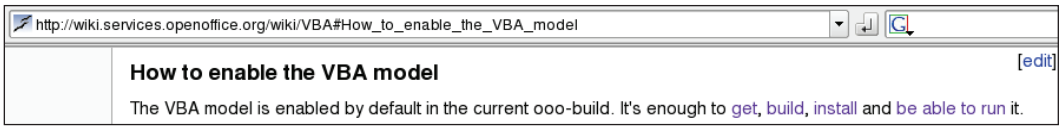
1 Introduction
2 CD or DVD installation
3 Installing SUSE Linux Over the Network
4 Installation without burning the CDs
5 Installing SUSE Linux on Multiple Machines
6 Graphical boot menu freezes
7 External Links

# Building OpenOffice.org from Source

If you're feeling confident, and if you've got the time to do it, then you may consider building your own version of OpenOffice.org.

## Building on Linux

When you come to build OpenOffice.org on Linux, you'll be pleased to know that the instructions are all online and just waiting for you to follow.




How successful you are, will depend on (unsurprisingly) the OpenOffice.org dependencies. If your distro matches SUSE fairly well, or if you're able to install any extra files that OpenOffice.org requires, then the process should be fairly painless.

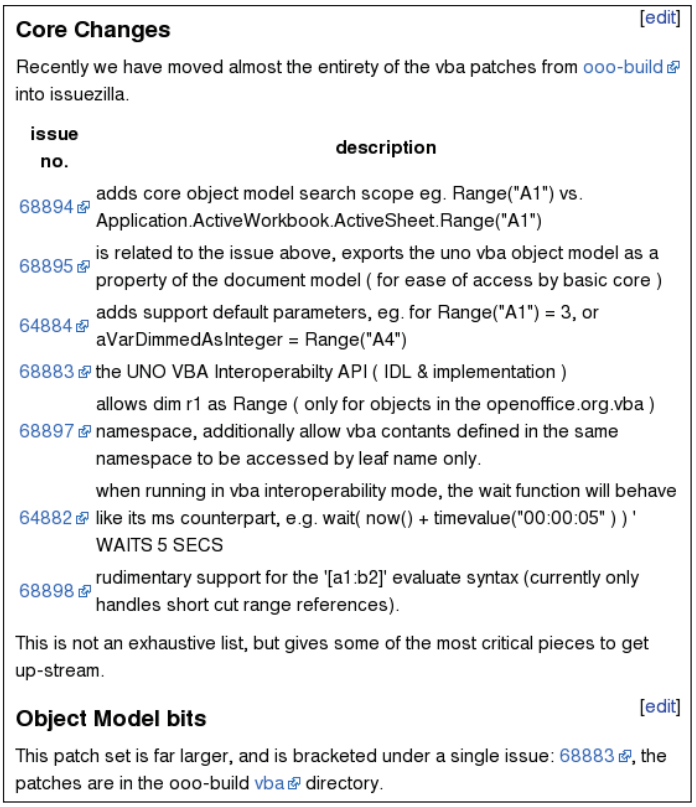
Of course, once you have managed to get this version of OpenOffice.org running, don't forget to tell the rest of the Linux community about it—whether that be via your distro forums, the Novell developers, or your own website.

# Support your Local OpenOffice.org Issue

Now, you may have gotten to this stage and decided that:

- You're a Windows user and you want (or have) to stay that way.
- You're a Linux user but you don't want to (or can't) migrate to SUSE Linux 10.1.
- You can't build your own version of OpenOffice.org.

If that's the case then you may prefer to go online and vote for the issues that Novell have raised with OpenOffice.org at `http://wiki.services.openoffice.org/wiki/VBA`:



**Core Changes** [edit]

Recently we have moved almost the entirety of the vba patches from ooo-build into issuezilla.

| issue no. | description |
|---|---|
| 68894 | adds core object model search scope eg. Range("A1") vs. Application.ActiveWorkbook.ActiveSheet.Range("A1") |
| 68895 | is related to the issue above, exports the uno vba object model as a property of the document model ( for ease of access by basic core ) |
| 64884 | adds support default parameters, eg. for Range("A1") = 3, or aVarDimmedAsInteger = Range("A4") |
| 68883 | the UNO VBA Interoperabilty API ( IDL & implementation ) |
| 68897 | allows dim r1 as Range ( only for objects in the openoffice.org.vba ) namespace, additionally allow vba contants defined in the same namespace to be accessed by leaf name only. |
| 64882 | when running in vba interoperability mode, the wait function will behave like its ms counterpart, e.g. wait( now() + timevalue("00:00:05" ) ) ' WAITS 5 SECS |
| 68898 | rudimentary support for the '[a1:b2]' evaluate syntax (currently only handles short cut range references). |

This is not an exhaustive list, but gives some of the most critical pieces to get up-stream.

**Object Model bits** [edit]

This patch set is far larger, and is bracketed under a single issue: 68883, the patches are in the ooo-build vba directory.

The more support that the issues receive, the more likely it is that the VBA support will be brought into the mainstream OpenOffice.org.

# Importing an Excel Spreadsheet that Contains Macros

Hopefully by now you've either:

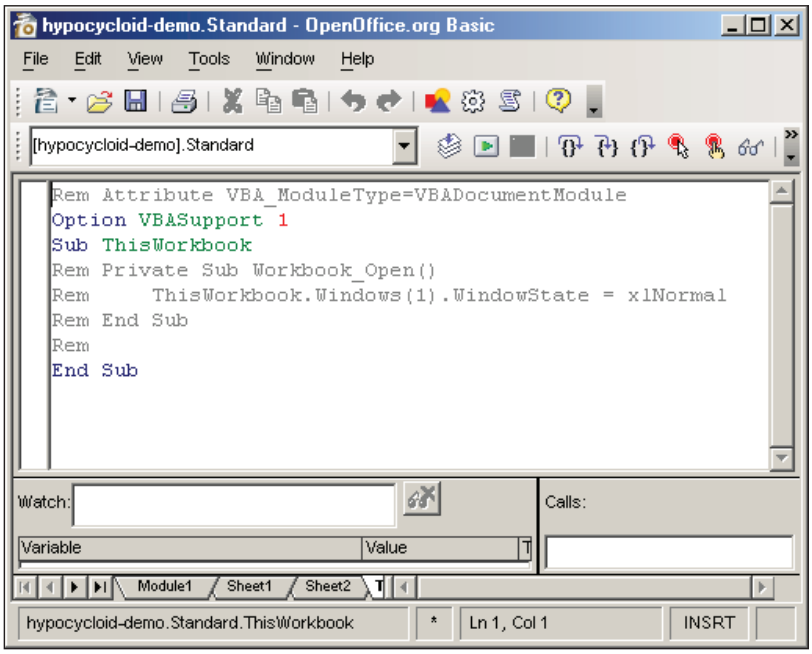
- Migrated to a suitable Linux distro
- Managed to compile the Novell source code
- Started using Sun's mainstream OpenOffice.org containing Excel VBA support

# Opening Up an Excel Spreadsheet

Obviously the first thing that you'll want to do is to carry out the difficult task of importing an Excel spreadsheet that contains working macros. Difficult? No, just use **File | Open...** as you would with any other Excel file that you wanted to use in OpenOffice.org Calc.

# Viewing Code without VBA Support

If you haven't been able to enable VBA support then don't despair. You can still load the Excel spreadsheet; you won't get any errors from the code trying to run. Why? All becomes clear if you view the code in the Basic Editor:



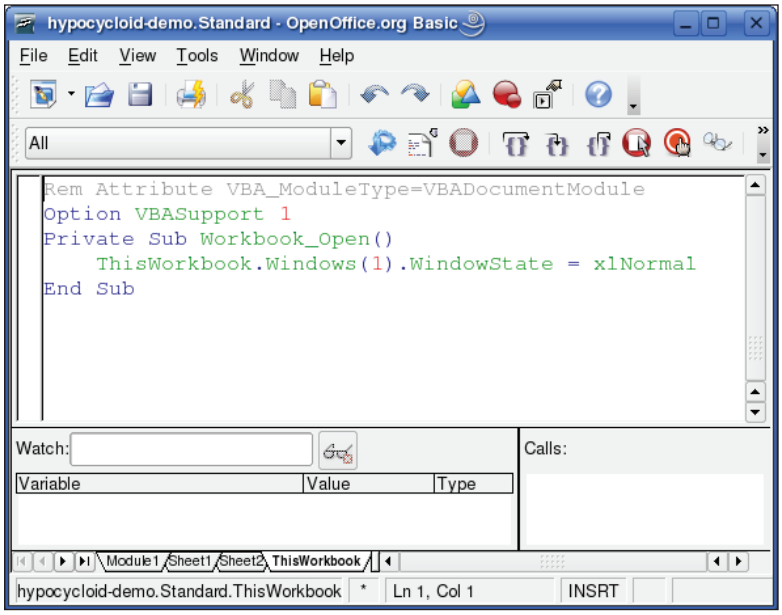

You'll notice that all of the VBA code has been turned into comments (as denoted by the `Rem` at the start of each line). You may also notice that the whole module has been turned into a single subroutine.

It's now down to you to manually convert the VBA code into OpenOffice.org Basic.

# Viewing Code with VBA Support

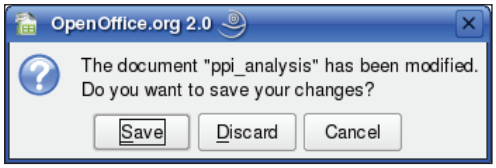
If you have a version of OpenOffice.org with Excel VBA support, then you'll notice something immediately—any macros set to run at load time will run straightaway

and without any errors. You'll also notice a big difference when you look at the code in the Basic Editor—all of the code is treated as normal OpenOffice.org Basic code, and the only commented-out code is the code that's meant to be commented out:




# Closing your Spreadsheet

When you come to close the Excel spreadsheet you may be surprised to be told that the file has been modified, even if you know that no changes have been made:




You may not have changed the file, but Calc has. Calc adds a line of code to every module that it recognizes as having VBA code:

```
Option VBASupport 1
```

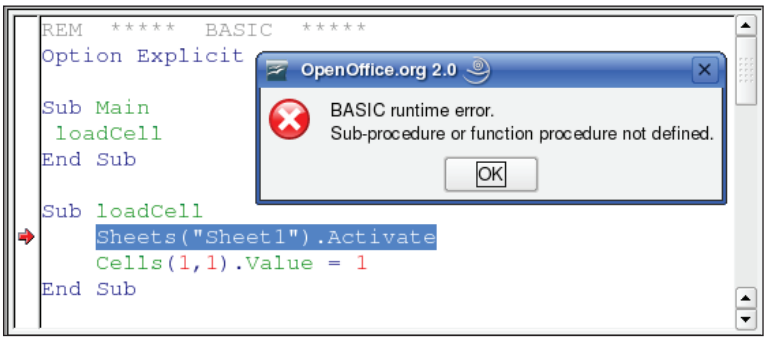
This line of code is essential; without it no VBA code will be run by Calc.

# Starting to Code with Excel VBA in Calc

We've spent a lot of time looking at:

- What you need to do in order to enable VBA support in OpenOffice.org Calc
- How to import an Excel spreadsheet into OpenOffice.org Calc

If you're already experienced in Excel VBA, then you'll be champing at the bit to get on and start developing your own macros. In fact, you've probably received your first error message:




Don't panic. We've already learned that Calc automatically adds a line of code to every module that it imports and which contains VBA code. In order to write VBA code yourself, you need to do the same. So, at the start of every module (where you want to use VBA code) you need to add:

```
Option VBASupport 1
```

# Combining VBA Code and OOo Basic Code

Of course, a question may occur to you at this point—how will this affect all of your existing OpenOffice.org Basic code? The answer is that it won't. In fact, once you've set the VBASupport option, OpenOffice.org Basic code and VBA code will co-exist quite happily. For example:

```
Option Explicit
Option VBASupport 1

Sub Main
  loadCells
End Sub

Sub loadCells
  'Standard OpenOffice.org Basic
```

```
    Dim Sheet as Object
    Dim Cell as Object

    Sheet = thisComponent.Sheets("Sheet1")
    Cell = Sheet.getCellByPosition(1,1)
    Cell.String = "B2"
    Cell = Sheet.getCellByPosition(1,2)
    Cell.String = "B3"

    'VBA code
    Sheets("Sheet1").Activate
    Cells(1,1).Value = "A1"
    Cells(1,2).Value = "B1"
End Sub
```

If you run the code from an open spreadsheet, then you'll see:

| | A | B |
|---|---|---|
| 1 | A1 | B1 |
| 2 | | B2 |
| 3 | | B3 |

As you can see the combinations of VBA and OOo Basic code will run quite happily, but there are differences in the way that that they run.

# Comparing VBA and OOo Basic Code

If you look through the VBA code and OOo Basic code we've seen so far, you'll see that there are a number of obvious differences:

- There appear to be more lines of OOo Basic code required to do the same job.
- In VBA, a cell can contain either values or formulas, but not strings.
- The cell positioning is different.

# Simplifying Code

In actual fact, you can re-write the OOo Basic code so that it uses (almost) the same number of lines as VBA:

```
'Standard OpenOffice.org Basic
Dim Sheet as Object

Sheet = thisComponent.Sheets("Sheet1")
Sheet.getCellByPosition(1,1).String = "B2"
Sheet.getCellByPosition(1,2).String = "B3"
```

```
'VBA code
Sheets("Sheet1").Activate
Cells(1,1).Value = "=2*3"
Cells(1,2).Value = "B1"
```

You can even re-write it so that it uses less lines of code than the VBA:

```
'Standard OpenOffice.org Basic
thisComponent.Sheets("Sheet1").getCellByPosition(1,1).String = "B2"
thisComponent.Sheets("Sheet1").getCellByPosition(1,2).String = "B3"
```

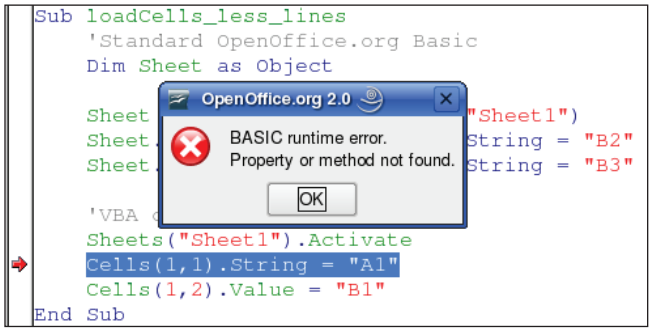
```
'VBA code
Sheets("Sheet1").Activate
Cells(1,1).value = "A1"
Cells(1,2).Value = "B1"
```

However, I'm sure that you'll agree that each of the lines is made much more complex. VBA support induces some extra objects for us to play with and (as we've just seen) these can help us simplify the code.

# VBA—No Strings Attached

We've already learned that if we're using OOo Basic, then we can assign one of three types to a cell: formula, string, and value. However, if we're using VBA then we can only assign one of two types: formula and value (this does incorporate strings).

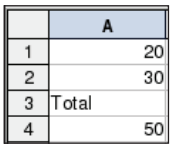
If you do try to assign cell as a string, then you'll just get an error when you try to run the code:




So, to write to cells in VBA use:

```
Cells(1,1).value = 20
Cells(2,1).Value = 30
Cells(3,1).Value = "Total"
Cells(4,1).Formula = "=A1+A2"
```

Which will produce:



| | A |
|---|---|
| 1 | 20 |
| 2 | 30 |
| 3 | Total |
| 4 | 50 |

And you might be interested to know that you can even dispense with `value`:

```
Cells(1,1) = 20
Cells(2,1) = 30
Cells(3,1) = "Total"
Cells(4,1).Formula = "=A1+A2"
```

# Getting the Right Cell Position

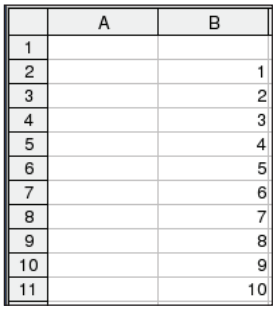
I'm sure you remember that we can write to a cell by using `getCellByPosition`:

```
thisComponent.Sheets("Sheet1").getCellByPosition(1,1).Value = 100
```

Also, you need to enter the co-ordinates as the column and then the row. This means that in order to write to a single column you'd use:

```
Dim r as Integer
For r = 1 to 10
  thisComponent.Sheets("Sheet1").getCellByPosition(1,r).Value = r
Next r
```

This would, of course, be seen as the following:



| | A | B |
|---|---|---|
| 1 | | |
| 2 | | 1 |
| 3 | | 2 |
| 4 | | 3 |
| 5 | | 4 |
| 6 | | 5 |
| 7 | | 6 |
| 8 | | 7 |
| 9 | | 8 |
| 10 | | 9 |
| 11 | | 10 |

You'll have realized by now that the standard OOo `getCellByPosition` requires the column and row numbers to be passed to it (and in that order). However, when you use the VBA `Cells` object, you'll find that it requires the row number first followed by the column number (i.e. the order is reversed). For example::

```
Dim r as Integer
Sheets("Sheet1").Activate
For r = 1 to 10
  Cells(r,1) = r
Next r
```

This time you'll see:

| | A | B |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | |
| 5 | 5 | |
| 6 | 6 | |
| 7 | 7 | |
| 8 | 8 | |
| 9 | 9 | |
| 10 | 10 | |

You'll also have noticed something else here: In the VBA code column A is 1, and in the OOo Basic code column A is 0.

## Using Named Cells and Ranges

We've just seen how to access a cell by its position, and, of course, we can use standard OOo Basic code to make use of a cell's name:

```
thisComponent.Sheets("Sheet1").getCellRangeByName("C9").Value = 20
```

You'll find that VBA is no different, and is actually a little simpler:

```
Range("C10") = 10
```

On top of the `Range` object, VBA support allows you to do some quite interesting things. For example, you can write directly to a whole range of cells with a single line of code:

```
Range("A1:D10") = 20
```

The display on the screen will be as shown below:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | 20 | 20 | 20 | 20 | |
| 2 | 20 | 20 | 20 | 20 | |
| 3 | 20 | 20 | 20 | 20 | |
| 4 | 20 | 20 | 20 | 20 | |
| 5 | | | | | |

OK, that's nice, but actually you want to be writing to cells within the range, don't you? This next little bit of code shows you exactly how to do this:

```
Dim i as Integer
Sheets("Sheet1").Activate
For i = 1 to 16
  Range("A1:D4").cells(i) = i
Next i
```

This will lead to the following display:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | |
| 2 | 5 | 6 | 7 | 8 | |
| 3 | 9 | 10 | 11 | 12 | |
| 4 | 13 | 14 | 15 | 16 | |
| 5 | | | | | |

# Further VBA Examples

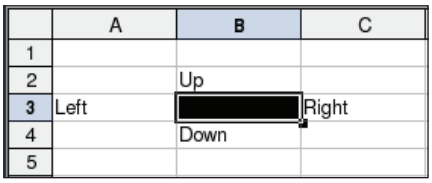
So far we've looked at writing into cells in a spreadsheet using standard OOo Basic and using the objects that VBA support introduces into OpenOffice.org. We can now look at some of the other objects that can make our lives much easier.

# Using Active Cells and Cell Offsets

We've already seen how to activate a cell, but we can also make a cell the main center of attention and then we can use cells relative to the selected one by using the cell offset:

```
Sheets("Sheet1").Activate
Range("B3").Select
ActiveCell.Offset(0,-1) = "Left"
ActiveCell.Offset(0,1) = "Right"
ActiveCell.Offset(-1,0) = "Up"
ActiveCell.Offset(1,0) = "Down"
```

This will result in following screen being displayed:




# Using the Workbooks Object

A key object that you'll be able to use is `workbooks`—this represents a complete spreadsheet. So, what's the first thing that you'll want to do? Open up a spreadsheet, of course:

```
Workbooks.Open "/home/bluek/ppi_investigation.ods"
```

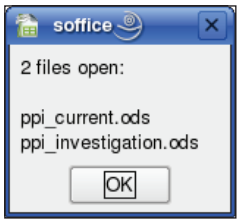
And, you can use the `workbooks` object to close a spreadsheet as well:

```
Workbooks.Close "/home/bluek/ppi_investigation.ods"
```

Interestingly the `workbooks` object contains an array of Workbooks.

```
Dim wBook as Workbook
Dim wList as String

For Each wBook In Workbooks
  wList = wList & wBook.Name & chr(13)
Next wBook
msgbox Workbooks.Count & " files open:" & chr(13) & chr(13) & wList
```

This will display the following message box:




# Using the Worksheets Object

Having used the `workbook` object (representing a complete spreadsheet), it won't be any surprise for you to learn that there are `worksheet` objects (representing individual sheets in the spreadsheet):

```
Workbooks("ppi_current.ods").Worksheets("Sheet2").Range("A1") = _
                                                                Now()
```

Of course, if you're just wanting to write to the most recently accessed spreadsheet, then you can just use:

```
Worksheets("Sheet2").Range("A1") = Now()
```

# Further Information

We're not going to do any more than dip our toes into the world of Excel VBA. After all, you'll probably fit into one of the following groups:

- You already know how to program using Excel VBA—if that's the case, then you don't need me to tell you anything more.
- You don't know how to program using Excel VBA, but from what you've seen it seems a good idea—if that's the case, then that's more than what we can cover in a single chapter.

The examples we've covered in this chapter should enable you to create your own macros. However, if you want to learn more, then there are number of Internet sites devoted to supplying Excel VBA tutorials and examples; just do a quick Google search and you'll see that you're spoiled for choice.

# Summary

In this chapter we've had a look at OpenOffice.org's Excel VBA support.

We've seen that (at the time of writing) VBA support is not yet incorporated into the released version of OpenOffice.org. However, we've also seen that through the Novell developers' efforts this is likely to change in the near future; in fact, by the time that you're reading this the situation may be completely different.

We have seen how Linux users can obtain the Novell OpenOffice.org source code and build it themselves to start using VBA support. They could even migrate to a version of Linux that uses OpenOffice.org with VBA support. Dedicated Windows users can check on the OpenOffice.org website for the current version or they can download and install Novell's own branded version of OpenOffice.org for Windows.

Finally, we have seen how to import Excel spreadsheets and use them exactly like Calc spreadsheets and how to use VBA in OpenOffice.

# Index

## O

## S



## U



## V